самоучитель

Максим Кузнецов, Игорь Симдянов

PHP 7

Новинки PHP 7

Шаблоны проектирования, итераторы и генераторы

Приемы работы с СУБД PostgreSQL

Взаимодействие с базами данных NoSQL (Redis и подобными)

100 заданий

Материалы на www.bhv.ru

bhv®

Максим Кузнецов
Игорь Симдянов

самоучитель

PHP 7

Санкт-Петербург
«БХВ-Петербург»
2018

УДК 004.438 PHP
ББК 32.973.26-018.1
К89

Кузнецов, М. В.

К89 Самоучитель PHP 7 / М. В. Кузнецов, И. В. Симдянов. — СПб.: БХВ-Петербург, 2018. — 448 с.: ил. — (Самоучитель)

ISBN 978-5-9775-3817-6

Книга опытных разработчиков описывает последнюю версию языка разработки серверных сценариев PHP 7. Рассмотрены все нововведения языка и связанные с ними изменения в разработке современных Web-сайтов. Изложение ведется с упором на объектно-ориентированное программирование, шаблоны проектирования, итераторы, генераторы, а также взаимодействие с современными базами данных (PostgreSQL и Redis).

В конце глав приведены более 100 заданий для закрепления материала и освоения не вошедших в книгу разделов языка. Электронный архив с исходными кодами доступен на сайтах издательства и GitHub.

Для Web-разработчиков

УДК 004.438 PHP
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Марины Дамбиевой*

Подписано в печать 30.03.18.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 36,12.
Тираж 1500 экз. Заказ № 6274.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

ISBN 978-5-9775-3817-6

© ООО "БХВ", 2018
© Оформление. ООО "БХВ-Петербург", 2018

Оглавление

Предисловие

Вы держите в руках четвертое, полностью переработанное издание популярной книги. Предыдущее издание было посвящено PHP 5 и вышло 8 лет назад. С тех пор язык обогатился большим количеством нововведений, все их мы осветим на страницах книги.

В связи с этим книгу пришлось полностью переписать, лишь 10 глав из 32-х основаны на материале предыдущей книги, хотя и их пришлось подвергнуть глубокой переработке. Остальные главы написаны с чистого листа.

Объектно-ориентированный подход

До версии PHP 5 поддержка объектно-ориентированного программирования (ООП) в языке была довольно скудна. В PHP 5 объектно-ориентированный подход долго оставался альтернативой для традиционного процедурного подхода. PHP 7 практически полностью предназначен для объектно-ориентированной разработки.

Без ООП невозможна разработка современных PHP-приложений: все больше расширений предполагают объектно-ориентированный интерфейс, компоненты оформляются в виде классов, PSR-стандарты и современные фреймворки диктуют разработку, полностью ориентированную на объектно-ориентированный подход.

Именно поэтому знакомиться с объектно-ориентированным подходом мы начинаем сразу с первых глав книги. Рассматриваем шаблоны проектирования и там, где это возможно, ориентируемся на новые расширения с объектно-ориентированным интерфейсом.

PostgreSQL и Redis

Традиционно PHP-приложения работают совместно с СУБД MySQL. В редкой книге по PHP не уделяется внимание этой популярной базе данных. Однако мы отступим от традиции.

С одной стороны, перепродажа компании AB MySQL корпорации Sun, которая в свою очередь была поглощена в 2009 г. Oracle, привела к тому, что наиболее по-

пулярная свободная СУБД оказалась в руках крупнейшего в мире производителя коммерческих баз данных. Развитие MySQL если и не находится в стагнации, то значительно уступает конкурирующим СУБД, например PostgreSQL, которая долгое время оставалась на вторых ролях.

С другой стороны, развитие объектно-ориентированного подхода и увеличение количества памяти на серверах привели к всплеску интереса к нереляционным базам данных, которые оформились в виде движения NoSQL. PostgreSQL — не совсем традиционная СУБД. Задуманная как объектно-ориентированная база данных, PostgreSQL за несколько лет предвосхитила NoSQL-движение. Поэтому уже сегодня можно наблюдать, как большинство современных Web-приложений переходят на PostgreSQL.

Кроме этого, вместо традиционного memcached мы рассмотрим NoSQL базу данных Redis. Как правило, современные Web-приложения не обходятся без одной или нескольких NoSQL баз данных, поэтому обойти их вниманием не представляется возможным. Redis выделяется среди них высокой производительностью (100 000 RPS — request per seconds), богатыми возможностями (коллекции, кластеризация, механизм Pub/Sub).

Задания

Мы не рассматриваем детальную установку полноценного окружения, включающего Web-сервер, настройку его связи с PHP, обеспечение безопасного доступа к удаленному серверу. Вместо этого используется встроенный PHP-сервер, сразу же доступный при установке PHP.

В силу ограничений по объему книги пришлось отказаться от описания многих особенностей PHP, например: детального рассмотрения сокетов, расширения curl, математических функций, преобразования изображений, управления буфером вывода, регулярных выражений и даже файловых функций. Все эти возможности выходят за рамки книги.

Вместо этого каждая глава снабжается заданиями, которые побуждают исследовать документацию, знакомиться с не вошедшими в книгу функциями и расширениями, исследовать альтернативные базы данных, читать статьи, искать решение. Всего 100 заданий.

Исходные коды

В начале каждой главы (за исключением 1-й и 29-й) указан каталог, в котором можно обнаружить примеры данной главы, названия файлов приведены в заголовках к листингам.

Исходные коды к книге можно найти на GitHub-аккаунте по адресу:

https://github.com/igorsimdyanov/phpworkshop

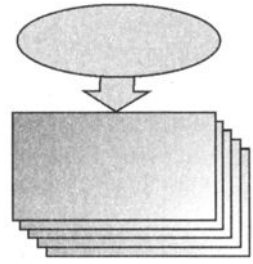
По ссылке **https://github.com/igorsimdyanov/phpworkshop/issues** можно адресовать вопросы авторам.

Электронный архив с исходными кодами к книге также можно скачать по ссылке **ftp://ftp.bhv.ru/9785977538176.zip**. Эта ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Благодарности

С Максимом Кузнецовым мы написали множество книг, и первое издание этой книги было стартом нашей совместной работы. Пять лет, как его с нами нет: жил быстро, ярко, помогал многим, казалось бы, в безвыходных ситуациях. Это издание книги посвящено ему.

ГЛАВА 1



Что представляет собой PHP?

Язык программирования PHP — серверный язык, при помощи которого можно создавать Web-сайты, причем как небольшие лендинги, состоящие из одной страницы, так и гигантские системы, использующие сотни и тысячи серверов. Электронная энциклопедия Wikipedia, социальные сети Facebook, "ВКонтакте", электронная площадка объявлений Avito созданы с использованием PHP.

Будучи одним из самых первых языков программирования, ориентированных на Web-разработку, PHP прошел длительный путь практически с самого начала зарождения Web. Поэтому в мире он остается одним из самых популярных и востребованных языков.

1.1. Достоинства и недостатки

В основе популярности PHP лежат следующие достоинства.

- Ориентация на Web-разработку — PHP создавался, развивался и поддерживается как язык для создания Web-сайтов. Многие конструкции и решения в нем созданы для удобства работы в Web-среде.
- Кроссплатформенность — PHP перенесен на все основные операционные системы: можно разрабатывать сайт в Windows, Mac OS X, а эксплуатировать на Linux-сервере. Сложности переноса будут минимальны и нивелироваться языком.
- Бесплатность — PHP является разработкой из мира свободного программного обеспечения, не потребуется платить ни за сам язык, ни за большинство сопутствующих программ (редакторы, Web-серверы, базы данных). Вдобавок большинство программных продуктов, с которыми придется иметь дело, будут иметь доступный для изучения и модификации исходный код. Вложения могут потребоваться при аренде доменного имени и сервера для публикации сайта в Интернете. Однако изучать PHP можно, не вкладывая ни копейки.
- Низкий порог входа — изучить PHP и начать создавать на нем готовые приложения много проще, чем с использованием конкурирующих технологий (.NET,

Python, Ruby, Go). Изучение PHP не закрывает для разработчика другие технологии, в Web сам язык — значительная, но меньшая часть используемых технологий. Знания, приемы работы, сопутствующие технологии (Web-серверы, базы данных, библиотеки, вспомогательные языки) пригодятся и в любой другой экосистеме, отличной от PHP. При создании собственного бизнеса собрать команду PHP-разработчиков зачастую проще и дешевле всего.

По закону сохранения, любая вещь, обладающая хоть каким-либо достоинством, имеет недостатки. Ими обладает и PHP.

- Отсутствие лидера — многие технологии и языки имеют лидера, архитектора, который определяет облик технологии, задает вектор развития, принимает решение о том, что должно быть обязательно, а чего не будет никогда (Linux, Python, Ruby и т. п). В PHP лидера нет, многие решения и конструкции — это компромисс заинтересованных групп и исторически сложившихся реалий.
- Непоследовательный синтаксис — при изучении языка PHP, особенно старой части, основанной на функциях, можно заметить, что часть функций имеет префиксы array_, str_, часть не имеет. Параметры функций могут быть расположены не совсем логично и не так, как в другой функции этой же группы.
- PHP — уже довольно долго живущий язык. Когда язык только появляется, он довольно элегантный и внутренне согласованный. По мере жизненного цикла язык обрастает дополнительными ключевыми словами, артефактами, устаревшими конструкциями, которые вроде есть, работают, но которыми не рекомендуется пользоваться. У PHP была довольно бурная молодость, в ходе которой была отменена масса директив и приемов, которые на первый взгляд должны были облегчать разработку, а на практике оборачивались серьезными проблемами безопасности. Сам PHP, стартовавший как необъектно-ориентированный язык, в настоящий момент стал полноценным объектно-ориентированным языком. Однако в нем полно старых процедурных артефактов, которыми придется пользоваться.
- Сообщество PHP-разработчиков велико и разъединено, т. к. PHP — это одна из первых технологий для разработки Web-проектов, половина Интернета создана с его участием. В PHP-разработку одновременно было вовлечено огромное количество программистов по всему миру. Все это породило большое число самых разных подходов, фреймворков и не совместимых друг с другом экосистем. Более того, благодаря усилиям мощных и влиятельных социальных сетей (в первую очередь Facebook, "ВКонтакте") появились альтернативные реализации PHP. Это плохо, т. к. многие экосистемы внутри PHP не совместимы, а сообщество раздроблено и тратит силы на создание одних и тех же библиотек в рамках разных групп. Ситуация исправляется и при помощи PSR-стандартов. Разработчики договариваются о единых правилах и интерфейсах, обеспечивающих совместимость фреймворков, но этот процесс еще в начале пути, в то время как конкурирующие технологии (.NET, Ruby) уже имеют единую платформу для всех фреймворков.

1.2. Структура PHP

Язык PHP имеет ядро и расширения языка. Между ядром и расширениями довольно трудно провести границу, т. к. многие расширения давно включены в состав ядра или распространяются в виде скомпилированных бинарных библиотек и легко устанавливаются.

Есть и другая часть — код, созданный на PHP, который условно можно поделить на следующие типы:

- компоненты — библиотеки на PHP, которые собираются при помощи менеджера пакетов Composer *(см. главу 26)*;
- фреймворки — готовые сборки, зачастую из компонентов, при помощи которых можно создавать сайты любой степени сложности. В книге, к сожалению, мы их не касаемся, однако если вы выберите PHP в качестве основного языка разработки, то не пройдете мимо них. На следующие PHP-фреймворки стоит обратить внимание: Symfony, Laravel, Zend, Yii. Всего их сотни, если не тысячи;
- готовые приложения — готовые к использованию разработки на PHP. Это системы управления контентом (Wordpress, Drupal), форумы (phpBB), Web-интерфейсы управления базами данных (phpMyAdmin, pgAdmin).

Книга, которую вы держите в руках, познакомит вас с языком; описанные выше системы созданы с использованием языка PHP, но требуют отдельного изучения, и, к сожалению, их обсуждение выходит за рамки нашей книги.

1.3. Сопутствующие технологии

При помощи PHP можно быстро разрабатывать Web-сайты, однако современные реализации PHP — это не самая быстрая и эффективная часть сайта. Поэтому для запуска сайта потребуются дополнительное программное обеспечение и технологии.

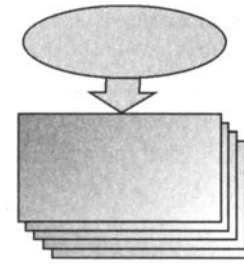
- Web-сервер — программа, которая обеспечивает взаимодействие клиента и вашего приложения посредством протокола HTTP. На протяжении всей книги мы используем встроенный PHP-сервер *(см. главу 3)*, хотя для эксплуатации настоящего сайта потребуется Web-сервер nginx или Apache.
- Сервер базы данных — данные нужно где-то хранить. В книге довольно подробно рассматриваются две базы данных: PostgreSQL *(см. главы 27–28)* и Redis *(см. главы 29–30)*. Однако это далеко не все базы данных, которые вам встретятся на практике, да и каждая из затронутых заслуживает отдельной книги.
- Система контроля версий Git, которая служит для хранения истории разработки, резервного копирования, доставки кода на сервер, организации командной работы. Работаете ли вы в коллективе или в одиночку — Git в настоящий момент превратился в основной инструмент современного программиста, какой бы язык программирования не был выбран в качестве базового.

Этот список можно продолжать и продолжать. Однако на самом деле можно начать даже без этого — опираясь просто на язык PHP. Если вам интересно, что вас ждет после изучения PHP, загляните в заключение.

Задание

Установите систему Git и загрузите исходные коды к книге с GitHub: **https://github.com/igorsimdyanov/phpworkshop**.

ГЛАВА 2



Установка PHP

Листинги данной главы можно найти в подкаталоге buildin.

В текущей главе будет рассмотрен порядок установки PHP в операционных системах Windows, Mac OS X и Linux (дистрибутив Ubuntu). Наличие PHP-интерпретатора позволит запускать PHP-программы, однако для запуска Web-приложений потребуется также Web-сервер. В ранних версиях PHP для этих целей использовался промышленный Web-сервер: Apache или nginx. Начиная с версии 5.4, дистрибутив PHP снабжается встроенным сервером, возможностей которого вполне хватает для локальной разработки и тестирования Web-приложений.

2.1. Установка 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:

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

Вариант **Thread Safe** предназначен для безопасного выполнения PHP в параллельных потоках в рамках одного системного процесса, например, если PHP устанавливается в качестве модуля Web-сервера Apache. Так как мы собираемся использовать встроенный сервер, не имеет значения, какой дистрибутив будет выбран, лучше всего воспользоваться вариантом **Non Thread Safe**. Последний вариант так же применяется при подключении PHP в качестве внешнего FastCGI-приложения, которое запускается на каждый внешний запрос.

Перед названием дистрибутива может быть помещена одна из аббревиатур VC11, VC14, означающих версии Visual Studio (2012 и 2015, соответственно), при помощи которой был скомпилирован дистрибутив. Для того чтобы успешно запустить проект, следует загрузить соответствующий распространяемый пакет Visual C++ для Visual Studio, который содержит необходимые динамические библиотеки:

- вариант VC11

  http://www.microsoft.com/en-us/download/details.aspx?id=30679;

- вариант VC14

  http://www.microsoft.com/en-us/download/details.aspx?id=48145.

> ***Внимание!***
>
> Необходимы библиотеки именно от английского варианта Visual Studio, русский вариант пакета не подойдет.

После загрузки zip-архива его следует распаковать в какую-нибудь папку, например C:\php.

Убедиться в том, что PHP доступен, можно, запустив командную строку, а затем перейти 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 7.0.0 (cli) (built: Dec  3 2015 09:31:54) ( NTS )
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
```

Для того чтобы команда PHP была доступна в любой точке файловой системы, путь к PHP-интерпретатору следует прописать в переменной окружения `PATH`.

Для доступа к переменным окружения нужно открыть Панель управления, перейти к разделу **Система**. Самый быстрый способ добраться до этого пункта — это щелкнуть правой кнопкой мыши по кнопке **Пуск** и выбрать пункт **Система** из контекстного меню. В операционных системах, предшествующих Windows 8, следует выбрать в меню **Пуск** пункт **Компьютер** и в контекстном меню выбрать пункт **Свойства**. В открывшемся окне Панели управления с активным разделом **Система** слева щелкнуть по ссылке **Дополнительные параметры системы**. Затем в окне **Свойства системы** на вкладке **Дополнительно** необходимо нажать кнопку **Переменные среды**. В открывшемся диалоговом окне в разделе **Системные переменные** следует отыскать переменную окружения `PATH` и дополнить 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.

---

[1] Полужирным шрифтом будем выделять команды или текст, вводимый пользователем.

2.2. Установка в Mac OS X

Прежде чем устанавливать PHP, следует установить Command Line Tools for XCode из магазина AppStore. XCode — это интегрированная среда разработки приложений для Mac OS X и iOS. Полная загрузка XCode не обязательна, достаточно установить 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 выше пути выводится предложение установить Command Line Tools, следует установить этот пакет, выполнив команду

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

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

```
$ ruby -e "$(curl -fsSL
↳ https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Впрочем, точную команду всегда можно выяснить на официальном сайте **http://brew.sh**. После установки в командной строке будет доступна команда `brew`, при помощи которой можно загружать, удалять и обновлять пакеты с программным обеспечением.

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

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

Для того чтобы получить доступ к репозиториям с PHP-дистрибутивами, следует выполнить серию команд:

```
$ brew tap homebrew/dupes
$ brew tap homebrew/versions
$ brew tap homebrew/homebrew-php
```

По умолчанию команда `brew tap` предполагает, что ей передаются названия GitHub-репозиториев. Таким образом, предыдущие команды прописывают доступ к следующим репозиториям:

https://github.com/Homebrew/homebrew-dupes

https://github.com/Homebrew/homebrew-versions

https://github.com/Homebrew/homebrew-php

Можно самостоятельно найти на GitHub дистрибутив или альтернативный проект и добавить его в менеджер пакетов Homebrew. Если пакет больше не нужен, его можно исключить при помощи команды `brew untap`. Разумеется, по возможности лучше использовать официальные пакеты Homebrew.

После выполнения приведенных выше команд надо убедиться в том, что пакеты успешно добавлены. Для этого следует выполнить команду `brew tap` без параметров. В результате будет выведен список добавленных репозиториев.

```
$ brew tap
homebrew/dupeshomebrew/php
homebrew/versions
```

Теперь можно установить PHP, запустив команду установки:

```
$ brew install php71
```

После установки PHP готов к работе.

```
$ php -v
PHP 7.1.2 (cli) (built: Feb 17 2017 10:51:21) ( NTS )
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.1.0, Copyright (c) 1998-2017 Zend Technologies
```

Если PHP недоступен, следует убедиться, что путь к sbin-каталогу Homebrew прописан в переменной окружения `PATH` в файле ~/.bash_profile:

```
# Homebrew sbin path
PATH=/usr/local/sbin:$PATH
```

2.3. Установка в Linux (Ubuntu)

Для того чтобы установить PHP в Ubuntu, следует воспользоваться штатным менеджером пакетов `apt-get`. Предварительно нужно обновить сведения о репозиториях и текущие пакеты при помощи команд:

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

Затем можно приступить к установке:

```
$ sudo apt-get install php7.0
```

После успешной установки команда `php` должна быть доступна в любой точке компьютера:

```
$ php -v
PHP 7.0.15-0ubuntu0.16.04.4 (cli) ( NTS )
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2017 Zend Technologies
    with Zend OPcache v7.0.15-0ubuntu0.16.04.4, Copyright (c) 1999-2017,
by Zend Technologies
```

2.4. Встроенный сервер

Для проверки работоспособности интерпретатора PHP создадим скрипт index.php, который выводит традиционную для книг по программированию фразу "Hello, world!" (листинг 2.1).

Листинг 2.1. Проверочный скрипт Hello World. Файл index.php

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

Если у текущего пользователя отсутствуют права администратора, тогда в папке со скриптом index.php следует выполнить команду

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

Команда запустит на 4000-м порту Web-сервер. Обратившись в браузере по адресу **http://localhost:4000/**, можно увидеть фразу "Hello, world!". Если работа ведется из-под учетной записи системного администратора (Windows) или задействована команда `sudo` (Mac OS X или Linux), встроенный сервер можно запустить на стандартном 80-м порту:

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

В этом случае порт в адресе можно не указывать, браузер автоматически будет обращаться по 80-порту, закрепленному за протоколом HTTP: **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 buildin
```

Журнальные записи или, как еще говорят, логи сервера выводятся непосредственно в консоль, в которой он был запущен. Остановить сервер можно, нажав комбинацию клавиш <Ctrl>+<C>.

2.5. Файл hosts

В предыдущем разделе в качестве домена использовался 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 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 в качестве домена localhost. 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, например **http://site.dev:4000/**.

2.6. Вещание вовне

При использовании локальных IP-адресов из диапазона 127.*XXX* можно быть уверенным, что никто извне не сможет обратиться к серверу. Если же 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 Web-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, либо его придется 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. Разумеется, в этом случае при запуске в качестве хоста потребуется указать это доменное имя.

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

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

2.7. Настройка PHP

PHP имеет огромное количество разнообразных настроек, которые сосредоточены в файле php.ini. Сразу после установки вместо файла php.ini можно обнаружить 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. По умолчанию интерпретатор PHP последовательно ищет конфигурационный файл в следующих местах:

- по пути, указанному в переменной окружения PHPRC;
- в текущем каталоге (если скрипт выполняется под управлением Web-сервера);

- в каталоге C:\Windows\ в случае Windows, в каталоге /etc/ или /etc/php7/ в Linux, в каталоге /usr/local/etc/php7.0 в Mac OS X или в каталоге компиляции в случае любой другой операционной системы.

Если имеется несколько файлов php.ini и нет уверенности в том, какой из них используется в настоящий момент, выяснить путь к используемому файлу php.ini можно из отчета информационной функции `phpinfo()` (листинг 2.2).

Листинг 2.2. Информация о PHP. Файл phpinfo.php

```
<?php
phpinfo();
?>
```

Функция выводит подробную информацию обо всех параметрах PHP. Путь к файлу php.ini, который сейчас используется PHP, указывается в параметре `Loaded Configuration File`.

Можно явно указать PHP местоположение файла php.ini при помощи параметра `-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
```

Конфигурационный файл PHP является обычным текстовым файлом, который можно открыть при помощи любого редактора. Содержимое 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`.

Сразу после установки PHP следует отредактировать как минимум одну директиву — текущий часовой пояс. Для того чтобы выставить московский часовой пояс, необходимо найти директиву `date.timezone` и привести ее к виду:

```
date.timezone = 'Europe/Moscow'
```

Если встроенный сервер при этом оставался запущенным, потребуется его перезапуск для того, чтобы изменения в php.ini вступили в силу.

2.8. Расширения

Интерпретатор PHP строится по модульному принципу. Язык состоит из ядра, к которому при необходимости могут подключаться расширения — библиотеки, дополняющие язык новыми возможностями. Например, для обработки изображений можно подключить расширение GDLib, для связи с базами данных — расширение PDO, для поддержки многобайтовых строк (UTF-8) — mbstring. Часть расширений получили настолько широкую популярность, что их включили в состав ядра. Например, это строковые функции из расширения calendar или поддержка сессий из расширения session.

> ***Внимание!***
> Материал текущего раздела можно пропустить при первом чтении.

Каждое расширение увеличивает размер оперативной памяти, которую занимает интерпретатор PHP. Чем больше памяти потребляет интерпретатор, тем меньше процессов PHP можно запустить на сервере, тем меньше соединений сервер может обслужить в одну секунду. Поэтому расширения подключаются и отключаются по мере необходимости. Для того чтобы уменьшить размер исполняемого файла PHP, а следовательно, и объем потребляемой оперативной памяти, большинство расширений не включаются в состав ядра PHP. Вместо этого они компилируются в виде внешних динамических библиотек: *.dll для Windows и *.so для UNIX-подобных операционных систем.

Самый простой способ выяснить, подключено ли расширение, — это выполнить команду `php -m`, которая выводит список доступных расширений. Можно так же воспользоваться отчетом функции `phpinfo()` (см. листинг 2.2). В разделе `Configuration` указываются подключенные расширения, в таблицах, которые сопровождают каждое из расширений, приводится список параметров данного расширения. Большинство из этих параметров могут быть скорректированы в конфигурационном файле php.ini.

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
```

Для того 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 следует найти директиву `extension_dir` и указать в ней путь к папке с расширениями:

```
extension_dir = "ext"
```

После того как путь к папке с расширением указан, можно активировать сами расширения, воспользовавшись директивой `extension`. 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
```

В случае UNIX-подобных операционных систем установка расширений осуществляется еще проще. Как правило, они оформлены в виде отдельных пакетов.

Так в Mac OS X, в основном, используется менеджер пакетов Homebrew. В *разд. 2.2* PHP устанавливался при помощи команды

```
$ brew install php71
```

Точно так же устанавливается расширение, только название пакета `php71` дополняется именем расширения, например, для CURL команда может выглядеть следующим образом:

```
$ brew install php71-curl
```

В современных Linux-дистрибутивах установка расширений также осуществляется при помощи менеджера пакетов, в случае Ubuntu это `apt-get`. Напомним, что установка PHP осуществляется командой

```
$ sudo apt-get install php7.0
```

Точно так же устанавливаются расширения. Так, CURL можно установить при помощи команды

```
$ sudo apt-get install php7.0-curl
```

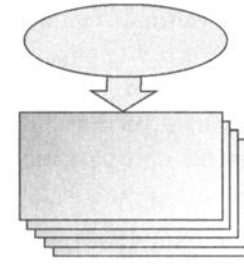
2.9. Документация

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 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.

Задания

1. При помощи `php -h` выведите справочную информацию о параметрах интерпретатора PHP. Найдите способ запуска интерактивного режима и выполните в нем фразу "Hello world".

2. Найдите в сети редактор Sublime Text и установите его в своей операционной системе. Создайте в нем проект Hello World с программой из листинга 2.1.
3. На сайте **http://php.net** воспользуйтесь поиском, найдите информацию о конструкции `echo` и функции `phpinfo()`.
4. Установите VirtualBox и запустите в нем альтернативную операционную систему: Ubuntu, если вы работаете в Windows или Mac OS X, или Windows, если ваша рабочая система — Ubuntu.
5. Установите Web-сервер nginx и запустите php-fpm, настройте их на совместную работу таким образом, чтобы они поддерживали обработку PHP-скриптов.



ГЛАВА 3

Быстрый старт

Листинги данной главы
можно найти в подкаталоге start.

Несмотря на то, что PHP является универсальным языком программирования и может применяться для разработки практически любого программного обеспечения, основная его специализация — Web-разработка.

В текущей главе будут показаны наиболее типичные приемы работы с PHP. Часть конструкций (`if`, `include`) будут освещены вскользь. В последующих главах мы остановимся на них более детально.

3.1. Скрипты

Язык программирования PHP считается скриптовым языком, поэтому программы, написанные на нем, называют *скриптами*. Главное отличие традиционных программ от скриптов заключается в том, что скрипты работают только в определенной среде и используют ресурсы данной среды.

Например, скриптовый язык программирования JavaScript работает преимущественно в Web-браузерах, Visual Basic for Applications — только в среде Microsoft Office. С использованием этих языков программирования невозможно создать программу, работающую без соответствующей среды.

В случае PHP в качестве такой среды выступает Web-окружение (Web-сервер, сервер базы данных, почтовый сервер и т. п.). Именно он принимает запросы от клиента, обеспечивает их параллельное выполнение и отправку данных. PHP-скрипт получает всю информацию о запросе и выполняет запрос, а затем отправляет данные обратно серверу.

> ***Замечание***
>
> Впрочем, язык программирования PHP допускает создание программ, работающих независимо от Web-сервера, однако в такой форме он не получил сколько бы то ни было широкого распространения.

Одной из главных особенностей языка программирования PHP является тот факт, что его код может располагаться вперемешку с HTML-кодом. Для того чтобы интерпретатор PHP различал HTML- и PHP-коды, последний заключается в специальные теги `<?php` и `?>`, между которыми располагаются конструкции и операторы языка программирования PHP.

В листинге 3.1 приводится классический пример, выводящий в окно браузера при помощи конструкции `echo` фразу "Hello, world!" ("Привет, мир!"). Содержимое листинга 3.1 следует поместить в файл с расширением php, например в файл index.php.

> ***Замечание***
>
> Первоначальная версия PHP разрабатывалась как шаблонизатор — система, встраиваемая в HTML-код для выполнения операций, которые не поддерживаются статическим HTML. По мере того, как PHP трансформировался в полноценный язык, стала приобретать популярность обратная тенденция — отделение PHP и HTML-кода.

Листинг 3.1. Простейший PHP-скрипт. Файл index.php

```
<!DOCTYPE html>
<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 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>
```

3.2. Начальные и конечные теги

Как было указано в предыдущем разделе, PHP-скрипт должен быть размещен между начальным тегом <?php и конечным тегом ?> для того, чтобы интерпретатор мог разделить HTML- и PHP-коды. 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. Например, скрипт из листинга 3.1 можно было бы переписать так, как это продемонстрировано в листинге 3.2.

Листинг 3.2. 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 видно из примера выше, 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 3.3 приводится пример, который содержит две вставки: одна задает название страницы (в HTML-теге <title>), а вторая определяет содержимое страницы (в HTML-теге <body>).

Листинг 3.3. Допускается несколько PHP-вставок в HTML-код. Файл few.php

```
<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 3.4).

Листинг 3.4. Завершающий тег ?> можно опускать. Файл missing.php

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

Более того, стандарт кодирования PSR-2, который определяет форматирование PHP-кода для распространяемых компонентов, 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.

Стандарты PSR определяют правила разработки компонентов PHP, их цель — унифицировать фреймворки и компоненты, распространяемые через менеджер Composer *(см. главу 26)*, чтобы компоненты одного фреймворка могли использоваться в другом.

На момент написания книги было принято 8 стандартов и 9 находилось в процессе подготовки. Следить за процессом принятия стандартов можно на странице **http://www.php-fig.org/psr/**. Стандарты PSR-1 и PSR-2 определяют форматирование и стилевое оформление PHP-кода. Мы будем придерживаться этих стандартов и сообщать о правилах форматирования по мере изучения языка.

3.3. Использование точки с запятой

Совокупность конструкций языка программирования, завершающуюся точкой с запятой, будем называть *выражением*.

Как видно из листинга 3.3, после строки "Вывод текущей даты" не указывается точка с запятой. Выражение одно, и надобность отделять его от других выражений отсутствует. Однако, как можно видеть во второй вставке, в конце каждой из конструкций echo имеется 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. В результате будет сгенерировано сообщение об ошибке "Parse error: syntax error, unexpected 'echo' (T_ECHO), expecting ',' or ';'" ("Ошибка разбора: синтаксическая ошибка, неожиданно встречена конструкция echo, ожидается либо запятая ',', либо точка с запятой ';'").

Последнее выражение перед завершающим тегом ?> можно не снабжать точкой с запятой. Например, в листинге 3.3 после выражения `echo date(DATE_RSS)` точку с запятой можно не указывать. Однако настоятельно рекомендуется не пользовать-

ся этой особенностью и помещать точки с запятой после каждого выражения, т. к. добавление новых операторов может привести к появлению трудноулавливаемых ошибок.

Переводы строк никак не влияют на интерпретацию скрипта, выражение может быть разбито на несколько строк — интерпретатор PHP будет считать, что выражение закончено лишь после того, как обнаружит точку с запятой или завершающий тег ?>. В листингах 3.5 и 3.6 представлены два скрипта, аналогичные по своей функциональности.

Листинг 3.5. Использование точки с запятой. Файл semicolumn.php

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

Листинг 3.6. Альтернативная запись скрипта из листинга 3.5. Файл mech.php

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

Следует избегать конструкций, подобной той, которая приведена в листинге 3.6. Чем более понятно и ожидаемо написан код, тем проще и быстрее его отлаживать.

3.4. Составные выражения. Фигурные скобки

Фигурные скобки позволяют объединить несколько выражений в группу, которую обычно называют *составным выражением* (листинг 3.7).

Листинг 3.7. Составное выражение. Файл curly.php

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

Как видно из примера выше, выражения внутри фигурных скобок располагаются с отступом. Такой отступ не обязателен, однако он повышает читаемость программы. Стандарт кодирования PSR-2 требует, чтобы отступ оформлялся 4 пробелами.

Если вы привыкли к использованию символа табуляции, следует настроить свой редактор на замену символа табуляции пробелами.

Само по себе составное выражение практически никогда не используется, основное его предназначение — совместная работа с условными операторами, операторами цикла и т. п., которые мы рассмотрим в последующих главах.

Составное выражение может быть расположено в нескольких PHP-вставках. В листинге 3.8 приводится пример двух составных выражений, которые разбиты несколькими PHP-вставками. Задача скрипта сводится к случайному выводу в окно браузера либо зеленого слова "Истина", либо красного слова "Ложь". Без использования фигурных скобок оператор `if` распространял бы свое действие только на одно выражение, использование составного выражения позволяет распространить его действие на несколько простых выражений.

> ***Замечание***
>
> Условный оператор `if` рассматривается в *главе 8*.

Листинг 3.8. Составное выражение в нескольких PHP-вставках. Файл fews.php

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

Как видно из листинга 3.8, составное выражение в любой момент может быть прервано тегами `<?php` и `?>`, а затем продолжено. Впрочем, существуют исключения, например, составное выражение, применяемое для формирования класса нельзя разбивать тегами `<?php` и `?>`.

3.5. 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 достаточно сложных построений, которые нуждаются в объяснении на обычном языке. Для вставки таких пояснений в код предназначены *комментарии*.

PHP предоставляет несколько способов для вставки комментариев, варианты которых представлены в табл. 3.1.

Таблица 3.1. *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 3.9 демонстрируется использование всех трех видов комментариев из табл. 3.1.

Листинг 3.9. Комментарии. Файл comments.php

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

Естественно, что комментарии PHP действуют только внутри тегов-ограничителей `<?php ... ?>`. То есть, если символы комментариев будут находиться вне тегов-ограничителей, то они, как и любой текст, будут отображены браузером (листинг 3.10).

Листинг 3.10. 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 -->
```

Комментарии можно вставлять не только после точки с запятой, но и в середине выражения (листинг 3.11).

Листинг 3.11. Комментарий в списке аргументов функции. Файл position.php

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

3.6. Включение PHP-файла

До этого момента мы имели дело лишь с одним PHP-скриптом. Однако PHP-скрипты можно подключать к другим PHP-скриптам при помощи двух конструкций: `include` и `require`. Обе 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 3.12).

Листинг 3.12. 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 3.13.

Листинг 3.13. Файл included.php

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

В результате запуска скрипта из листинга 3.12 в браузер будут выведены следующие строки

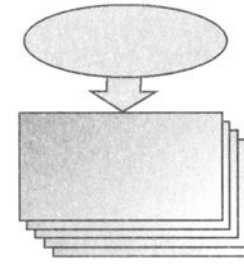
```
Основной скрипт
Включаемый файл
Текст не обязательно должен выводиться оператором echo
Основной скрипт
```

Различие `include` и `require` заключается в их реакции на отсутствие включаемого файла. В случае `include` выводится предупреждение, весь последующий код продолжает выполняться; в случае `require`, если нельзя найти файл, работа скрипта останавливается.

Задания

1. Найдите в документации на сайте **http://php.net** функцию `date()`, которая занимается форматированием даты, изучите приемы работы с функцией.
2. Найдите в документации функции для работы с датой и временем `time()` и `mktime()`, ознакомьтесь с возможностями, которые предоставляют эти функции. С использованием календарных функций выведите текущие дату и время в следующем формате: `'10.07.2018 14:10'`.
3. Найдите в документации функцию генерации случайных чисел `mt_rand()`, ознакомьтесь с ее возможностями. Создайте скрипт, который генерировал бы случайное число от 0 до 1000.

ГЛАВА 4



Переменные и типы данных

Листинги данной главы
можно найти в подкаталоге variables.

Ключевым объектом практически любого языка программирования является переменная. Под *переменной* в общем случае понимается именованная область памяти. В этой области может храниться либо строка, либо число, либо сложный объект. Манипулировать этим значением можно при помощи имени переменной (его мы далее для простоты будем называть просто переменной). То, что хранится в области памяти, будем называть *значением переменной.*

4.1. Объявление переменной. Оператор =

В PHP переменные начинаются со знака доллара (`$`), за которым может следовать любое количество буквенно-цифровых символов и символов подчеркивания, но первый символ не может быть цифрой. Таким образом, допустимы следующие имена переменных: `$n`, `$n1`, `$user_func_5` и т. д. В отличие от ключевых слов, имена переменных в PHP *чувствительны к регистру*, т. е. переменные `$user`, `$User` и `$USER` являются различными (листинг 4.1).

Листинг 4.1. Зависимость переменных от регистра. Файл case_sensitive.php

```
<?php
$user = "Владимир";
$User = "Дмитрий";
$USER = "Юрий";
echo $user; // Владимир
echo $User; // Дмитрий
echo $USER; // Юрий
```

Как видно из листинга 4.1, для присвоения значения переменной необходимо воспользоваться оператором присваивания `=`, который позволяет инициализировать переменную.

При объявлении числовых значений в качестве разделителя целого значения и дробной части выступает точка (листинг 4.2).

Листинг 4.2. Объявление чисел. 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 4.3 переменным `$num`, `$number` и `$var` присваивается значение 1 в одну строку за счет использования оператора = в цепочке.

Листинг 4.3. Инициализация переменных одним значением. Файл multi_set.php

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

4.2. Типы данных

Язык PHP является слаботипизированным, в большинстве случаев переменные языка не требуют строгого задания типа при их объявлении, а в ходе выполнения программы тип переменной может быть практически всегда изменен неявным образом без специальных преобразований.

Например, переменная, объявленная строкой, может использоваться далее в арифметических операциях, 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.

Тем не менее некоторые типы, такие как `null`, `object`, `array` или `resource`, настолько специфичны, что без учета их особенностей невозможна успешная разработка.

Более того, при использовании функций и методов вы сможете явно указывать тип их параметров и возвращаемых значений, которые будут рассмотрены более подробно в *главе 11*.

В табл. 4.1 представлены типы данных, которые поддерживаются PHP.

> ***Замечание***
>
> В документации и в нашей книге помимо типов данных, представленных в табл. 4.1, вам будут встречаться *псевдотипы* — условные обозначения одного или нескольких типов. В основном они используются совместно функциями и методами (*см. главы 11* и *16*). Например, `mixed` — любой тип, `number` — либо `integer`, либо `double`, `iterable` — итерируемый объект. Особняком стоит ключевое слово `void`, обозначающее отсутствие переменной. Иногда несколько типов могут быть разделены вертикальной чертой | для обозначения того факта, что переменная может принимать один из указанных типов. Например, псевдотип `number` можно было бы записать, используя следующую комбинацию базовых типов: `integer|double`.

Таблица 4.1. 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
double (или 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, максимальный размер которой составляет 2 Гбайт
array	Массив. Это объединение нескольких переменных под одним именем. Обращаться к отдельным переменным можно при помощи индекса массива. Более подробно массивы обсуждаются в *главе 10*
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. д. Более подробно дескрипторы будут рассмотрены в соответствующих главах
null	Специальный тип, который сигнализирует о том, что переменная не была инициализирована
callable (или callback)	Некоторые функции PHP могут принимать в качестве аргументов другие функции, которые называются *функциями обратного вызова*. Переменные данного типа содержат ссылки на такие функции

В текущей главе мы изучим лишь целые, вещественные числа, логический тип, строки и тип null. Остальные типы будут рассмотрены в следующих главах.

4.3. Целые числа

Целые числа являются наиболее распространенными в программировании. Это связано с тем, что большинство прикладных задач носит арифметический характер, а также с тем, что это наиболее быстродействующий тип данных.

> ***Замечание***
>
> В отличие от других языков программирования переполнения (т. е. выхода значения за допустимые границы) в PHP не бывает, если полученное значение не убирается в integer, для него автоматически выбирается больший тип данных (double).

Целочисленная переменная может быть объявлена несколькими способами (листинг 4.4).

Листинг 4.4. Объявление целочисленных переменных. Файл integer.php

```
<?php
$num = 1234; // десятичное число
$num = +123; // десятичное число
$num = -123; // отрицательное число
$num = 0123; // восьмеричное число (эквивалентно 83)
$num = 0x1A; // шестнадцатеричное число (эквивалентно 26)
```

Целое положительное число объявляется, как правило, без указания плюса перед ним (впрочем, его использование не приводит к ошибке). Для объявления отрицательного числа перед ним размещается символ минуса -.

По умолчанию числа задаются в десятичной системе счисления, однако PHP позволяет объявлять переменные в восьмеричной и шестнадцатеричной системах счисления.

В восьмеричном числе основанием служит 8, а для выражения всех чисел используются цифры от 0 до 7. Число 8 в восьмеричной системе счисления играет ту же роль, что число 10 в десятичной.

Шестнадцатеричная система счисления (основанием служит число 16) использует цифры от 0 до 9 и буквы английского алфавита от A до F, означающие шестнадцатеричные "цифры" 10, 11, 12, 13, 14 и 15. Числа, объявленные в восьмеричной системе счисления, предваряются префиксом 0, в шестнадцатеричной системе — префиксом 0x.

> ***Замечание***
>
> Может показаться, что восьмеричные и шестнадцатеричные числа являются избыточным наследием языка C, однако они применяются и в Web-разработке, например, при задании прав доступа к файлам и папкам, кодировании цвета и т. д.

4.4. Вещественные числа

Вещественные числа (`float` или `double`) позволяют сохранять данные в огромном интервале, за пределы которого прикладные программы не выходят практически никогда. Различают две формы записи вещественного числа: стандартную и экспоненциальную (листинг 4.5).

> ***Замечание***
>
> При выводе под число с плавающей точкой отводится 12 символов, это значение может быть изменено при помощи директивы `precision` в конфигурационном файле php.ini.

Листинг 4.5. Объявление вещественных чисел. Файл double.php

```
<?php
// Объявление положительного вещественного числа
// Стандартная запись
$var = 1.23456;
```

```
// Объявление отрицательного вещественного числа
// Стандартная запись
$var = +1.23456;
// Объявление положительного вещественного числа
// больше единицы. Научная запись.
$var = 1.23456E2; // 123.456
// Объявление положительного вещественного числа
// больше единицы. Научная запись.
$var = 1.23456E+2; // 123.456
// Объявление положительного вещественного числа
// меньше единицы. Научная запись.
$var = 1.23456E-3; // 0.00123456
// Объявление отрицательного вещественного числа
// Научная запись
$var = -1.23456e-2; // -0.0123456
```

Экспоненциальная запись помимо мантиссы (целой и дробной части) содержит порядок, который начинается со строчной или прописной буквы E и целого положительного (или отрицательного) числа. Так запись 1.2e–3 эквивалентна произведению 1.2×10^{-3} (или 0.0012). Вещественное число 1.1e+2 (или 1.1e2) эквивалентно 1.1×10^{2} (или 110.0).

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 7)*.

4.5. Логический тип

Переменные логического типа `boolean` принимают только два значения: `true` (истина) или `false` (ложь). В листинге 4.6 приводится объявление логической переменной, принимающей значение `true`.

> ***Замечание***
>
> Константы `true` и `false` не зависят от регистра. В коде допускается использование форм `True` и `TRUE`. Однако стандарт PSR-2 требует записи констант строчными буквами: `true` и `false`. Более подробно константы обсуждаются в *главе 6*.

Листинг 4.6. Объявление переменной логического типа. Файл boolean.php

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

Переменные логического типа интенсивно используются совместно с операторами сравнения и цикла, которые более подробно освещаются в *главах 8* и *9*.

4.6. Строки

Строки предназначены для хранения текстовой информации и могут достигать размера 2 Гбайт. Однако на максимальный объем строки так же влияет ограничение объема памяти, отводимой под PHP-скрипт. Этот размер, равный по умолчанию 128 Мбайт, определяется при помощи директивы `memory_limit` конфигурационного файла php.ini. Объявление строк осуществляется при помощи кавычек (листинг 4.7).

Листинг 4.7. Объявление строки. Файл string.php

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

4.7. Кавычки

Как уже было продемонстрировано ранее, строки и строковые переменные формируются путем заключения той или иной фразы в кавычки. До сих пор использовались только двойные кавычки, однако в PHP имеется возможность применять несколько типов кавычек, каждый вид которых имеет собственные особенности (табл. 4.2).

Таблица 4.2. *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 4.8).

Листинг 4.8. 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 4.9). Такая подстановка называется *интерполяцией*.

Листинг 4.9. 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 4.10.

Листинг 4.10. 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 4.11).

Листинг 4.11. 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 символов и соответствующие им значения представлены в табл. 4.3.

***Таблица 4.3.** Специальные символы и их значения*

Значение	Описание
\n	Перевод строки
\r	Возврат каретки
\t	Символ табуляции
\\	Обратный слеш
\"	Двойная кавычка
\'	Одинарная кавычка

Размещение переменных и специальных символов (за исключением \') в одиночных кавычках не приводит к их специальной интерпретации (листинг 4.12).

Листинг 4.12. 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 4.13).

Листинг 4.13. Файл interpolation_quotes.php

```
<?php
// Подавляем вывод замечаний
error_reporting(E_ALL & ~E_NOTICE);
// Подставляем переменную $text в строки,
// обрамленные двойными кавычками
$text = "Паро";
echo "Едет $textвоз<br />";      // Едет
echo "Плывет $textход<br />";    // Плывет
echo "Едет {$text}воз<br />";    // Едет Паровоз
echo "Плывет {$text}ход<br />"; // Плывет Пароход
```

В первом случае PHP, встретив знак доллара `$`, будет искать окончание переменной и посчитает переменными последовательности `$textвоз` и `$textход`, во втором случае мы явно указываем границы переменных при помощи фигурных скобок. Для того чтобы подавить замечания о неинициированных переменных, в начале скрипта вызывается функция `error_reporting()`, которая более подробно освещается в *главе 22*.

В PHP существует 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 4.14).

Листинг 4.14. 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

25.03.2004  20:41    <DIR>          .
25.03.2004  20:41    <DIR>          ..
```

```
23.10.2004  00:19               411 config.php
24.07.2004  12:43                54 index.php
03.07.2004  12:55             1 815 main.php
18.09.2003  19:43             2 789 top.php
25.03.2004  20:42    <DIR>          images
11.04.2004  16:30    <DIR>          Projects
25.03.2004  20:43    <DIR>          admin
30.03.2004  22:17    <DIR>          scripts
04.04.2004  12:27    <DIR>          Books
25.04.2004  12:52    <DIR>          tools
12.06.2004  09:46    <DIR>          test
26.09.2004  12:53    <DIR>          Site
               4  файлов          5 069 байт
               10 папок  18 157 846 528 байт свободно
```

4.8. Оператор <<<

Объявить строку можно также, не прибегая к кавычкам. Для этого используется оператор <<<, который еще называют *встроенным документом* (here-документ). Сразу после данной последовательности размещается метка, конец оператора обозначается повторным вхождением метки (листинг 4.15).

> **Замечание**
>
> Метка является зависимой от регистра. Традиционно вводится при помощи заглавных букв, несмотря на то, что использование символов в нижнем регистре не запрещается.

Листинг 4.15. Использование последовательности <<<. Файл here.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 одинарные кавычки, которые сообщают интерпретатору PHP, что переменные внутри такой строки не интерполируются.

```
<?php
$name = "Имя пользователя";
$text = <<<'MARKER'
Переменные PHP не будут интерполироваться: $name
MARKER;
```

4.9. Обращение к неинициализированной переменной. Замечания (Notice)

В отличие от большинства языков программирования, в PHP переменная создается сразу же при первом обращении. В ранних версиях языка это приводило к возникновению множества трудноулавливаемых ошибок и брешей безопасности. Поэтому требования к переменным ужесточились. Переменная по-прежнему создается при первом обращении к ней, однако, если она не была инициализирована при помощи оператора присваивания =, генерируется замечание (Notice). В листинге 4.16 показано обращение к неинициализированной переменной `$user`, которое приводит к генерации замечания.

Листинг 4.16. Обращение к несуществующей переменной. Файл notice.php

```
<?php
echo $user;
```

В результате обращения к неинициализированной переменной `$user` (см. листинг 4.16) генерируется замечание "Notice: Undefined variable: user" ("Замечание: неопределенная переменная user").

Такое поведение скрипта можно подавить, настроив чувствительность PHP к ошибкам. Для этого в конфигурационном файле php.ini предназначена специальная директива `error_reporting`.

```
error_reporting  =  E_ALL & ~E_NOTICE
```

Кроме этого можно настроить значение директивы `error_reporting` при помощи функции `error_reporting()` (см. листинг 4.13). В *главе 22* будет рассмотрен еще один способ подавления вывода сообщений об ошибках — при помощи оператора `@`. Однако на практике лучше избегать как подавления сообщений при помощи `error_reporting`, так и использования оператора `@`. Если отображение сообщений интерпретатора в браузере не желательно, лучше отключить вывод сообщений при помощи директивы `display_errors` и выводить их в журнальный файл. Таким образом, они не будут видны в браузере, однако сообщение всегда можно будет посмотреть в файле журнала для "разбора полетов".

4.10. Специальный тип *null*

Специальный тип `null` предназначен для пометки неинициализированной переменной. Если интерпретатор PHP встречает в выражении неинициализированную переменную (как это описывается в *разд. 4.9*), она получает тип `null`. Переменная также получает данный тип, если она инициализируется константой типа `null` (листинг 4.17) или уничтожается при помощи конструкции `unset()` *(см. разд. 4.11)*.

ЗАМЕЧАНИЕ

Константа `null` не зависит от регистра. Однако, так же как и в случае констант `true` и `false`, стандарт PSR-2 предписывает записывать ее в строчном виде.

Листинг 4.17. Использование константы `null`. Файл null.php

```
<?php
$user = null;
```

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

4.11. Уничтожение переменной. Конструкция *unset()*

Переменная может быть уничтожена при помощи конструкции `unset()`, которая имеет следующий синтаксис:

```
void unset(mixed $var [, mixed $var1...])
```

В круглых скобках конструкции `unset()` указывается либо одна, либо несколько разделенных запятыми переменных, которые подлежат уничтожению.

Конструкция `unset()` описывается при помощи синтаксиса функций. Он используется в официальной документации и далее в книге. Перед названием конструкции указывается возвращаемый тип *(см. разд. 4.2)*, в данном случае это псевдотип `void`. Это означает, что `unset()` не возвращает никакого значения и конструкцию нельзя применять для инициализации переменной `$result = unset($var)`. В круглых скобках перечисляются параметры, перед которыми указывается их тип, в данном случае `mixed`, т. е. допускаются переменные любого типа. В квадратных скобках указываются необязательные параметры, а многоточие (...) сигнализирует о том, что их может быть произвольное количество.

После вызова конструкции `unset()` память, выделенная под значение переменной, возвращается системе, а самой переменной присваивается значение `null` (листинг 4.18).

Листинг 4.18. Использование конструкции `unset()`. Файл unset.php

```
<?php
// Объявляем переменные
$user = "Юрий";
$number = 123;
// Уничтожаем переменные
unset($user, $number);
// Попытка обращения к несуществующей переменной
echo $user; // Генерируется "Notice: Undefined variable"
```

Уничтожение ненужных переменных может быть полезно, когда скрипт оперирует объемными данными (например, содержимым файлов), и их размер грозит превысить объем памяти, выделяемой скрипту.

4.12. Проверка существования переменной. Конструкции *isset()* и *empty()*

Как видно из предыдущих разделов, в произвольной точке скрипта переменная может оказаться неинициализированной или быть уничтоженной к этому времени. Для проверки существования переменной используется конструкция `isset()`, которая имеет следующий синтаксис:

```
bool isset(mixed $var [, mixed $... ])
```

В круглых скобках конструкции `isset()` указывается либо одна, либо несколько разделенных запятыми переменных. В отличие от конструкции `unset()`, здесь возвращается значение логического типа. Если все переменные существуют, конструкция возвращает `true`, если хотя бы одна переменная не существует — возвращается `false`.

В листинге 4.19 демонстрируется использование конструкции `isset()`.

> ***Замечание***
>
> В листинге 4.19 используется оператор сравнения `if`, который выполняет следующие за ним операторы, если ему передано значение `true`, и игнорирует, если значение `false`. Более подробно синтаксис оператора `if` обсуждается в *главе 8*.

Листинг 4.19. Использование конструкции `isset()`. 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 />';
}
```

Как видно из листинга 4.19, пустая строка не эквивалентна неинициализированной переменной (т. е. переменной, принимающей значение `null`). Для проверки, является ли строка пустой или нет, предназначена специальная конструкция `empty()`, которая имеет следующий синтаксис:

`bool empty(mixed $var)`

В отличие от конструкции `isset()`, конструкция `empty()` принимает в качестве параметра лишь одну переменную *`$var`* и возвращает `true`, если переменная равна пустой строке `""`, нулю 0, символу нуля в строке `"0"`, `null`, `false`, пустому массиву `array()`, неинициализированной переменной (случай, впрочем, аналогичный `null`). Во всех остальных случаях возвращается `false` (листинг 4.20).

Листинг 4.20. 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 />';
}
```

4.13. Определение типа переменной

PHP предоставляет гибкие средства, позволяющие определить, к какому типу принадлежит переменная. Это бывает полезно, учитывая, что некоторые функции (особенно функции для работы с массивами и объектами) крайне чувствительны к типу данных.

Универсальной функцией, позволяющей определить тип переменной, является функция `gettype()`, которая имеет следующий синтаксис:

```
string gettype(mixed $var)
```

Функция возвращает тип переменной *$var*. Возвращаемое значение может принимать одно из значений: `"boolean"`, `"integer"`, `"double"` (и для `double`, и для `float`), `"string"`, `"array"`, `"object"`, `"resource"`, `"null"` и `"unknown type"` (листинг 4.21).

Листинг 4.21. Использование функции `gettype()`. Файл gettype.php

```
<?php
// Объявляем целую переменную
$number = 123;
echo gettype($number); // integer

// Объявляем логическую переменную
$flag = true;
echo gettype($flag); // boolean

// Объявляем строковую переменную
$str = '';
echo gettype($str); // string

// Объявляем вещественную переменную
$var = 1.7;
echo gettype($var); // double

// Уничтожаем переменную $var
unset($var);
echo gettype($var); // null
```

Для каждого из типов, представленных в табл. 4.1, предназначена функция, начинающаяся с префикса `is_`, которая возвращает `true`, если переменная принадлежит заданному типу. В листинге 4.22 демонстрируется использование функции `is_int()`, определяющей, принадлежит переменная целому типу `int` или нет.

Листинг 4.22. Использование функции `is_int()`. Файл is_int.php

```
<?php
// Объявляем целую переменную
$number = 123;
```

```
if(is_int($number)) { // true
    echo "Переменная $number является целочисленной<br />";
}

// Объявляем строковую переменную
$str = '123';
if(is_int($str)) { // false
    echo "Переменная $str является целочисленной<br />";
}
```

Результатом работы скрипта из листинга 4.22 будет лишь одна строка, сообщающая, что только переменная `$number` является целочисленной. Переменная `$str`, несмотря на то что содержит число, имеет строковый тип, поэтому функция `is_int()` возвращает для нее `false`.

В отличие от других языков программирования, где типы `float` и `double` имеют разный размер допустимых значений, в PHP между ними не существует различий, в чем можно легко убедиться при помощи скрипта, представленного в листинге 4.23.

Листинг 4.23. Использование функций `is_float()` и `is_double()`. Файл is_float.php

```
<?php
// Объявляем переменную типа "float"
$float = 123.24;
if(is_float($float)) { // true
    echo "Переменная $float имеет тип float<br />";
}
if(is_double($float)) { // true
    echo "Переменная $float имеет тип double<br />";
}

// Объявляем переменную типа "double"
$double = 123.24e307;
if(is_float($double)) { // true
    echo "Переменная $double имеет тип float<br />";
}
if(is_double($double)) { // true
    echo "Переменная $double имеет тип double<br />";
}
```

Все проверки из листинга 4.23 возвращают `true`. Такой же особенностью обладает функция `gettype()`, которая возвращает значение `"double"` для всех вещественных чисел, не зависимо от того, имеют они тип `float` или `double` (листинг 4.24).

Листинг 4.24. Файл gettype_float.php

```
<?php
// Объявляем переменную типа "float"
$float = 123.24;
echo gettype($float); // double

// Объявляем переменную типа "double"
$double = 123.24e307;
echo gettype($double); // double
```

4.14. Неявное приведение типов

В сильно типизированных языках программирования при объявлении переменной, как правило, следует указывать ее тип, а использование переменной неправильного типа приводит к ошибке. В слабо типизированных языках программирования (к ним относится и PHP) не требуется явное указание типа переменной, а попытка использования переменной в контексте, где ожидается переменная другого типа, приведет к тому, что PHP попытается автоматически (неявно) преобразовать переменную к нужному типу.

Например, если строка содержит число и используется в арифметическом выражении, то она автоматически будет приведена к числовому типу (листинг 4.25).

Листинг 4.25. Автоматическое преобразование строки в число. Файл cast.php

```
<?php
$str = '12.6';
$number = 3 + $str;
echo $number; // 15.6
```

Строка может содержать помимо числа любые другие символы, интерпретатор PHP постарается извлечь из начала строки наиболее полное значение, соответствующее числу. Если извлечь число из строки не удается, ее значение рассматривается как нулевое (листинг 4.26).

Листинг 4.26. 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
```

Аналогичным образом число автоматически преобразуется в строку там, где ожидается строковая переменная (например, при выводе в окно браузера).

Если ожидается логический тип (например, в условных операторах), 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 true (листинг 4.27).

Листинг 4.27. Преобразование к логическому типу. Файл cast_boolean.php

```
<?php
// Объявляем нулевое вещественное число
$float = 0.0;
if($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:

```
<?php
echo true;  // "1"
echo false; // ""
```

4.15. Явное приведение типов

Помимо неявного преобразования типов, разработчику может явно потребоваться 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 4.28).

Листинг 4.28. Преобразование типа float к int. Файл float_to_int.php

```
<?php
// Объявляем вещественную переменную
$float = 4.863;
// Преобразуем переменную $float к
// целому типу
$number = (int)$float;

echo $number; // 4
```

В табл. 4.4 приводятся все возможные варианты использования оператора круглых скобок.

Таблица 4.4. *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 = (real) $var;`	Приведение к вещественному типу `double`
`$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 4.29 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`.

> ***Замечание***
>
> В листинге 4.29 совместно с оператором `if` используется конструкция `else`, код в которой выполняется в том случае, если условие, переданное в `if`, вычисляется как `false` *(см. главу 8)*.

Листинг 4.29. 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.

bool settype(mixed $var, string $type)

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"`, `"float"`, `"string"`, `"array"`, `"object"` и `"null"`. Функция возвращает `true`, если преобразование было успешно осуществлено, и `false` в противном случае (листинг 4.30).

Листинг 4.30. 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 необходимости явного указания типа и того факта, что функция 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 4.31 на примере функции `intval()` демонстрируется использование функций данного класса.

> ***Замечание***
>
> Если функции `intval()` передается число, которое превышает размеры типа `int`, возвращается максимально возможное для данного типа значение — 9 223 372 036 854 775 807. Если в силу каких-либо обстоятельств переменная не может быть преобразована в целый тип, функция возвращает 0.

Листинг 4.31. 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()` имеет второй необязательный параметр, который позволяет задать основание числа в первом параметре функции. Так, если этот необязательный параметр 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. Функция `intval()` всегда возвращает результат в десятичной системе счисления.

4.16. Динамические переменные

В завершение главы рассмотрим еще один способ создания переменных. Он заключается в использовании двойного символа `$$`, позволяющего создавать переменные произвольного типа. Иногда название переменной невозможно определить заранее, и оно должно определяться по мере выполнения скрипта. В этом случае прибегают к *динамическим переменным* (листинг 4.32).

> ***Замечание***
>
> Не следует злоупотреблять динамическим формированием имени переменной, т. к. это приводит к снижению читабельности кода и значительно усложняет его отладку и сопровождение.

Листинг 4.32. Создание динамической переменной. Файл dynamic.php

```
<?php
$id_menu = 3;
$str = "active$id_menu"; // "active3"
$$str = 1;               // $active3 = 1;
if(isset($active3)) {
    echo "Переменная \${$str} существует и равна $active3";
}
```

Результатом работы скрипта из листинга 4.32 будет строка:

```
Переменная $active3 существует и равна 1
```

Значение переменной, следующей после первого знака $, воспринимается как имя новой переменной. В результате если строка $str имеет значение "active3", то имя новой переменной будет $active3.

Еще один способ создания динамических переменных заключается в использовании функции eval(). Она принимает в качестве параметра строку с PHP-кодом и выполняет ее (листинг 4.33).

Листинг 4.33. Использование функции eval(). Файл eval.php

```
<?php
$code = '$str = "Hello, world!<br />";
         echo $str;';
eval($code);
echo $str;
```

В результате выполнения скрипта из листинга 4.33 в окно браузера будут выведены следующие строки

```
Таким образом, переменная $str, которая определена в строке $code, доступна
не только в пределах данной строкиHello, world!
Hello, world!
```

но и после выполнения функции eval().

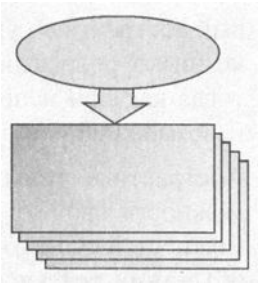
При помощи функции eval() можно динамически формировать не только значения переменных, но и сами переменные. Код в листинге 4.34 по результату полностью эквивалентен скрипту из листинга 4.33.

Листинг 4.34. Динамическое имя переменной. Файл eval_var.php

```
<?php
$id_menu = 3;
eval("\$active$id_menu = true;");
if(isset($active3)) {
    echo "Переменная \$active3 существует и равна $active3";
}
```

Задания

1. Найдите в документации на сайте **http://php.net** функцию is_numeric(). Чем она отличается от функций is_int() и is_float()?
2. Найдите в документации функции округления чисел round(), ceil() и floor(). Чем они отличаются друг от друга?
3. Создайте программу, которая округляет число 42.43752 до второго знака после точки — 42.44.
4. Найдите в документации функции decbin() и bindec(), осуществляющие преобразования десятичного числа в двоичное и наоборот. Переведите десятичные числа 4252 и 89080 в двоичное представление.



ГЛАВА 5

Классы и объекты

Листинги данной главы
можно найти в подкаталоге classes.

Переменные в PHP могут быть не только заранее заданного типа. При помощи объектно-ориентированного подхода можно формировать собственные типы переменных. Такие пользовательские типы называются *классами*. Экземпляры классов или переменные их типа называются *объектами*.

Тема ООП очень велика и находит отражения почти во всех аспектах языка. Традиционно, в книгах, посвященных PHP, описание объектов и классов выносится в самый конец книги. В результате может сложиться впечатление, что объектно-ориентированный подход представляет собой факультативный, необязательный раздел языка. Десять лет назад так оно и было. Однако в настоящий момент невозможно создание современных Web-приложений без понимания принципов ООП. Для востребуемой разработки необходимы компоненты и предопределенные классы, пространство имен и наследование, современные расширения имеют объектно-ориентированный интерфейс, популярные фреймворки построены по объектно-ориентированным принципам.

Поэтому в данном издании мы отказались от выделения объектно-ориентированных возможностей в отдельный цикл глав и рассматриваем их как неотъемлемую часть PHP, игнорировать или не знать которую не может позволить себе ни один разработчик.

5.1. Собственные типы данных

Одним из главных назначений объектно-ориентированного подхода является создание собственных *абстрактных типов данных*, позволяющих наряду с предопределенными типами (такими как `integer`, `bool`, `double`, `string`) вводить свои (классы) и объявлять "переменные" таких типов (объекты).

Создавая собственные типы данных, программист оперирует не машинными терминами (переменная), а объектами реального мира, поднимаясь тем самым на но-

вый абстрактный уровень. Яблоки и людей нельзя складывать друг с другом, однако низкоуровневый код запросто позволит совершить такую логическую ошибку, тогда как при использовании абстрактных типов данных эта операция становится невозможной или, по крайней мере, сильно затрудненной.

Абстрактные типы данных необходимы для того, чтобы дать программисту возможность вводить в программу переменные с желаемыми свойствами, т. к. возможностей существующих в языке типов данных зачастую не хватает. Связи между объектами реального мира, как правило, настолько сложны, что для их эффективного моделирования необходим отдельный язык программирования. Разрабатывать специализированный язык программирования для каждой прикладной задачи — очень дорогое удовольствие. Поэтому в языки программирования вводится объектно-ориентированный подход, который позволяет создавать свой мини-язык предметной области.

Переменными такого мини-языка программирования являются программные *объекты*, в качестве типа для которых выступает класс. *Класс* описывает состав объекта — переменные и функции, которые обрабатывают переменные и тем самым определяют поведение объекта (рис. 5.1).

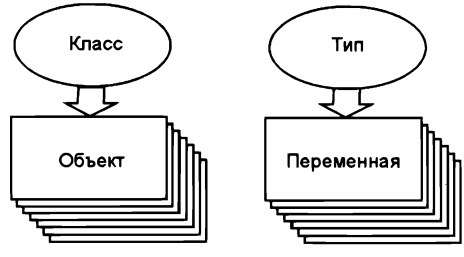


Рис. 5.1. Переменные объявляются при помощи типа, объекты — при помощи класса

5.2. Создание класса

Класс объявляется при помощи ключевого слова `class`, после которого следуют уникальное имя класса и тело класса в фигурных скобках. 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. Имена классов, в отличие от имен переменных и объектов, не зависят от регистра.

В сообществе PHP-разработчиков требования к именованию классов строже правил, устанавливаемых на уровне требований языка. Связано это с тем, что стандарт кодирования PSR требует, чтобы имя класса начиналось со строчной буквы, не содержало символов подчеркивания, а само имя было задано в CamelCase-стиле: название каждого составного слова начинается с прописной буквы. Как будет показано далее, эти правила позволяют автоматически загружать классы *(см. главу 24)*.

Замечание

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 5.1 приводится общий синтаксис объявления класса.

Листинг 5.1. Объявление класса. Файл my_class.php

```
<?php
class MyClass
{
    // Переменные класса
}
```

Важной особенностью PHP является то, что PHP-скрипты могут включаться в документ при помощи тегов `<?php` 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 "Parse error: syntax error, unexpected '?>', expecting function (T_FUNCTION) or const (T_CONST)".

Так как прерывать объявление класса недопустимо, его не удастся механически разбить и при помощи инструкции `include()` или `require()`.

В теле класса могут быть объявлены переменные, которые называются *переменными класса*. Например, для задания точки координат на плоскости можно создать класс `Point`, содержащий две координаты `$x` и `$y` (листинг 5.2).

Листинг 5.2. Класс точки `Point`. Файл point.php

```
<?php
class Point
{
    public $x;
    public $y;
}
```

Ключевое слов `public` используется для задания области видимости переменных и будет более подробно рассмотрено в *разд. 5.6*.

5.3. 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 5.3).

Листинг 5.3. Попытка повторного определения класса `Point`. Файл redeclare.php

```
<?php
class Point
{
    public $x;
    public $y;
}

// Fatal error: Cannot redeclare class Point
class Point
{
}
```

Ошибку в листинге 5.3 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. 24)*. 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. Это также повлечет генерацию ошибки (листинг 5.4).

Листинг 5.4. Файл redeclare_require.php

```
<?php
require 'point.php';
/*
...
Очень много кода
...
*/
require 'point.php'; // Fatal error: Cannot redeclare class Point
```

Ситуация может усложняться тем, что внутри включаемых файлов могут быть расположены другие конструкции `require`, включающие другие файлы. На практике может складываться довольно сложная система зависимых друг от друга файлов.

Чтобы исключить повторное определение классов, вместо инструкций `require` и `include` используются `require_once` и `include_once`. Их отличие от оригинальных инструкций состоит в том, что они включают файл только один раз, попытки повторного включения файла игнорируются (листинг 5.5).

Листинг 5.5. Файл redeclare_require_once.php

```
<?php
// Включение файла
require_once 'point.php';
```

```
// Все последующие попытки игнорируются
require_once 'point.php';
require_once 'point.php';
```

5.4. Создание объекта

Имея корректно объявленный класс, можно создать объект данного класса путем объявления при помощи ключевого слова `new`, за которым следует имя класса.

Можно определять класс и объявление объекта 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 *разд. 3.6* (листинг 5.6).

Листинг 5.6. 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 `$point` класса `Point` создается при помощи ключевого слова `new`. Для того чтобы обратиться к переменным объекта `$x` и `$y`, следует воспользоваться специальным оператором `->`. В отличие от традиционных переменных PHP, при обращении к переменным объекта символ `$` не указывается после `->`.

Объект `$point` является обычной переменной PHP, правда, со специфичными свойствами. Как и любой другой переменной, объекту можно присваивать новое значение. В листинге 5.7 приводится пример, в котором объекту `$point` присваивается числовое значение, и он становится переменной числового типа.

> ***Замечание***
>
> При создании объекта после имени класса могут следовать необязательные круглые скобки. Как будет показано в *главе 17*, в круглых скобках можно указывать параметры, инициализирующие объект.

Листинг 5.7. 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 5.8). Использование конструкции `unset()` может быть полезным, если объект занимает большой объем оперативной памяти, и ее следует освободить, чтобы не допустить переполнения.

Листинг 5.8. 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; // Notice: Undefined variable: point
```

Как видно из листинга, попытка обращения к объекту после его уничтожения при помощи конструкции `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.

5.5. Область видимости переменных класса

До текущего момента объявленные переменные были доступны во всех областях скрипта. Однако при введении класса мы вынуждены ввести понятия: *области видимости*. Не всегда переменные, объявленные в одной части программы, доступны в другой. Например, попытка обратиться к переменной `$x`, объявленной внутри класса, завершается неудачей (листинг 5.9).

Листинг 5.9. Файл class_scope.php

```
<?php
require 'point.php';
// class Point
// {
//     public $x;
//     public $y;
// }
$point = new Point;
echo $x; // Notice: Undefined variable: x in
```

Таким образом, переменные действуют лишь в своей определенной области видимости, между которыми существуют водоразделы. В языке PHP такими водораздe-

лами 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.

5.6. Спецификаторы доступа

Переменные класса объявляются при помощи одного из ключевых слов: `public`, `private` или `protected`. Эти ключевые слова называются *спецификаторами доступа* и позволяют указать, какие переменные доступны извне объекта, 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 члены класса объявляются при помощи спецификатора `private` и доступны только в рамках класса; обратиться к ним извне невозможно.

> ***Замечание***
>
> Спецификатор `protected` используется при наследовании и подробно рассматривается в *главе 18*.

В листинге 5.10 представлен модифицированный класс `PrivatePoint` (точка координат), который содержит два члена:

- `$x` — открытая переменная класса;
- `$y` — закрытая переменная класса;

Листинг 5.10. Класс `PrivatePoint`. Файл private_point.php

```
<?php
class PrivatePoint
{
    public $x;
    private $y;
}
```

Теперь можно создавать объект `$point` класса `PrivatePoint` и попробовать обратиться к его переменным (листинг 5.11).

Листинг 5.11. Файл private_point_use.php

```
<?php
// Подключаем объявление класса
require_once('private_point.php');

// Объявляем объект класса PrivatePoint
$point = new PrivatePoint;

// Присваиваем значения переменным объекта
$point->x = 3;
$point->y = 1; // Fatal error: Uncaught Error: Cannot access
```

Обращение к закрытому члену класса `$y` завершится ошибкой "Fatal error: Uncaught Error: Cannot access private property PrivatePoint::$y". На рис. 5.2 приводится схема процесса доступа к членам объекта, снабженным разными спецификаторами доступа.

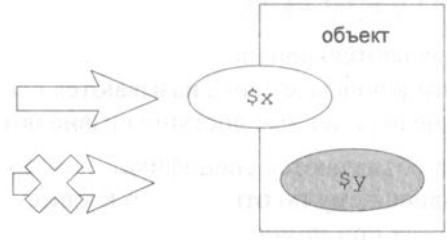


Рис. 5.2. Открытые и закрытые члены объекта

Вероятно, сейчас настройка доступа переменных классов не кажется ценной возможностью языка, однако по мере изучения классов и объектов в следующих главах вы измените свое мнение.

5.7. Статические переменные класса

До текущего 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 5.12).

Листинг 5.12. 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 таких переменных является возможность их инициализации прямо в классе при объявлении (листинг 5.13).

Листинг 5.13. Переменные объекта независимы. Файл my_static.php

```
<?php
class MyStatic
{
    public static $staticvar = 100;
}
```

Обращаться к таким переменным можно без создания объектов при помощи оператора разрешения области видимости :: (листинг 5.14).

Листинг 5.14. 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.

5.8. Ссылки на переменные

При присваивании переменным одинаковых значений при помощи оператора = получаются две независимые переменные (листинг 5.15).

Листинг 5.15. Оператор = с переменными. Файл var.php

```
<?php
$first = $second = 1;
$first = 3;
echo $second; // 1
```

Однако в случае объектов ситуация совершенно другая. Оператор присваивания = не приводит к созданию новой копии объекта: и старый, и новый объект указывают на одну и ту же область памяти (листинг 5.16).

Листинг 5.16. 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. 5.3).

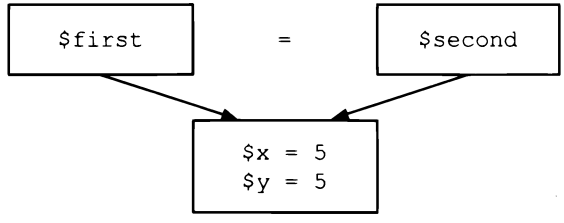


Рис. 5.3. Переменные лишь ссылаются на объект

Иными словами, вместо того чтобы каждый раз передавать объект, как это происходит в случае обычных переменных, передается *ссылка* на объект. Это позволяет значительно 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 этого используется оператор `&` (листинг 5.17).

Листинг 5.17. Создание ссылок для обычных переменных. Файл links.php

```
<?php
$first = 5;
$second = &$first;
$second = 1;
echo $first; // 1
```

Как видно из листинга 5.17, если перед именем переменной использовать оператор `&`, то другая переменная становится ссылкой. Оба названия переменных выступают как синонимы.

5.9. Клонирование объектов

Как уже было показано в предыдущем разделе, оператор присваивания `=` 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 5.18).

Листинг 5.18. 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. 5.4.

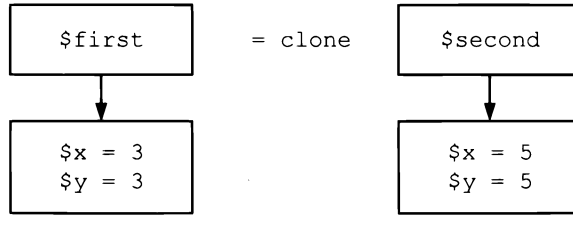
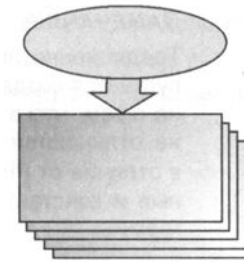


Рис. 5.4. Получение независимой копии объекта `$second` посредством клонирования

Задания

1. Создайте альтернативный класс `Point`, объект которого моделировал бы точку в трехмерном декартовом пространстве. В документации найдите и изучите функцию `sqrt()` для вычисления квадратного корня. Создайте скрипт, который, используя объекты класса `Point`, вычислял бы расстояние между двумя точками пространства.
2. Создайте класс для комплексных чисел.
3. Требуется промоделировать многоквартирный дом. Какие классы и объекты вы использовали бы для этого?

ГЛАВА 6



Константы

Листинги данной главы
можно найти в подкаталоге constants.

Константы отличаются от переменных тем, что их значение не может изменяться во время выполнения скрипта. Кроме того, константы, в отличие от переменных, не могут принимать неопределенное значение `null`.

6.1. Объявление константы. Функция *define()*

Объявление константы осуществляется при помощи функции `define()`, которая имеет следующий синтаксис:

```
bool define(
  string $name,
  mixed $value [,
  bool $case_insensitive = false])
```

Функция принимает в качестве первого параметра *`$name`* строку с именем константы, в качестве второго параметра *`$value`* передается значение константы. Третий необязательный параметр *`$case_insensitive`* определяет чувствительность константы к регистру. По умолчанию константы, точно так же как и переменные, зависят от регистра, т. е. константы с именами `CONSTANT` и `constant` считаются разными, однако если при создании константы в качестве третьего параметра *`$case_insensitive`* передать значение `true`, новая константа не будет зависеть от регистра. В случае успешного создания константы функция `define()` возвращает `true`, в противном случае возвращается значение `false`.

В листинге 6.1 приводится пример создания двух констант: `NUMBER`, принимающей значение 1, и `VALUE`, принимающей строковое значение "Hello, world!". В качестве значения константы может выступать логическое (`boolean`), целочисленное (`integer`), вещественное (`float`), строковое (`string`) значения либо массив (`array`). Константы не могут получать в качестве значения объекты (`object`) и дескрипторы (`resource`).

Замечание

Традиционно имена констант записываются символами верхнего регистра, хотя допускается создание констант с именами произвольного регистра. Эта традиция связана с тем, что в большинстве языков высокого уровня константы и переменные никак не отличаются друг от друга (переменные не снабжаются начальным символом `$`, в отличие от PHP), поэтому правила хорошего тона предписывают снабжать переменные и константы именами в разных регистрах, чтобы улучшить читабельность программы.

Листинг 6.1. Создание констант. Файл define.php

```
<?php
define('NUMBER', 1);
define('VALUE', 'Hello, world!');
echo NUMBER; // 1
echo VALUE;  // Hello, world!
echo Number; // Notice: Use of undefined
             // constant Number - assumed 'Number'
```

Как видно из листинга 6.1, обращаться к константам можно по их имени, при этом, в отличие от переменных, указывать символ `$` перед именем константы не требуется. Если предпринимается попытка обратиться к несуществующей константе, при включенном режиме отображения сообщений (Notice) генерируется замечание "Notice: Use of undefined constant".

Если необходима константа, не зависящая от регистра, то при ее создании функции `define()` в качестве третьего параметра необходимо передать значение `true` (листинг 6.2).

Замечание

Если в качестве третьего параметра функции `define()` передать значение `false`, то константы будут зависеть от регистра, однако обычно третий параметр просто не указывают, и он получает значение `false` по умолчанию (см. листинг 6.1).

Листинг 6.2. Создание констант, не зависящих от регистра. Файл define_ci.php

```
<?php
define('VALUE', 'Hello, world!', true);
echo VALUE;
echo Value;
echo VaLuE;
```

Следует отметить, что PHP имеет ряд предопределенных констант, с которыми мы уже встречались в предыдущих главах: `null`, `true` и `false`. Все они не зависят от регистра, поэтому в программах можно встретить различные их записи, например, `TRUE`, `true`, `True` и т. п. Однако стандарт кодирования PSR требует, чтобы они указывались в нижнем регистре.

Интересно отметить поведение функции `define()` при попытке переопределения уже существующей константы. В этом случае функция возвращает значение `false`, поэтому всегда можно проверить успешность объявления константы при помощи оператора `if` (листинг 6.3).

> ***Замечание***
>
> Условный оператор `if` более подробно рассматривается в *главе 8*.

Листинг 6.3. Попытка переопределения константы. Файл redefine.php

```
<?php
if (define('VALUE', 'Hello, world!')) {
    echo 'Константа VALUE получила значение "Hello, world!"<br />';
}
if (define("VALUE", 1)) {
    echo 'Константа VALUE получила значение 1<br />';
}
echo VALUE;
```

Если в конфигурационном файле php.ini директивой `error_reporting` разрешается вывод замечаний, то при попытке повторного определения константы генерируется замечание "Notice: Constant value already defined":

```
Константа VALUE получила значение "Hello, world!"
Notice: Constant VALUE already defined in redefine.php on line 4
Hello, world!
```

Как видно из результатов, при первом вызове функции `define()`, создающем константу `VALUE`, функция возвращает `true`. Повторный вызов функции для создания уже существующей константы возвращает `false`, при этом у константы остается старое значение. После определения константы изменить ее имя невозможно даже случайно.

6.2. Проверка существования константы

Функция `define()` не очень удобна для проверки существования константы, во-первых она создает константу, если она не существует, во-вторых, выводит замечание, если константа уже определена.

Для проверки существования константы имеется отдельная функция `defined()` со следующим синтаксисом:

```
bool defined(string $name)
```

В качестве единственного параметра функция принимает строку с именем константы *`$name`* и возвращает `true`, если константа существует, иначе возвращается `false`. В листинге 6.4 приводится пример использования функции.

Листинг 6.4. Использование функции `defined()`. Файл defined.php

```
<?php
// Определяем константу
define('VALUE', 1);

// Если константа существует, выводим ее значение
if (defined('VALUE')) echo VALUE; // 1
```

6.3. Динамическое имя константы

В *главе 5* мы сталкивались с динамическими именами переменных, которые можно получить при помощи конструкции `$$`. В отличие от переменных, для динамического формирования констант специальный синтаксис не предусмотрен, однако существует отдельная функция `constant()` со следующим синтаксисом:

```
mixed constant(string $name)
```

Функция принимает строку `$name` с именем константы и возвращает ее значение. Если константа с таким именем не обнаружена, возвращается `false` и выводится предупреждение "Warning: constant(): Couldn't find constant".

В листинге 6.5 обе строки, выводящие значение константы `VALUE` при помощи прямого обращения и функции `constant()`, эквивалентны.

Листинг 6.5. 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 особенно в том случае, если имя константы формируется на лету. В листинге 6.6 при помощи функции `mt_rand()` генерируется случайное число `$index` от 1 до 10, которое в дальнейшем используется для формирования константы. В результате имя константы всякий раз может принимать одно из десяти значений. Получить значения такой константы можно только при помощи функции `constant()`.

Листинг 6.6. Константа с динамическим именем. Файл constant_dynamic.php

```
<?php
// Формируем случайное число от 1 до 10
$index = mt_rand(1, 10);
```

```
// Формируем имя константы
$name = "VALUE{$index}";

// Определяем константу с динамическим именем
define($name, 1);

// Получаем значение константы
echo constant($name);
```

6.4. Предопределенные константы

Помимо констант, определяемых разработчиком, существует ряд предопределенных констант, значение которым выставляет интерпретатор PHP. Список предопределенных констант представлен в табл. 6.1.

***Таблица 6.1.** Предопределенные константы*

Константа	Описание
__LINE__	Текущая строка в файле
__FILE__	Полный путь и имя текущего файла
__FUNCTION__	Имя функции
__CLASS__	Имя класса
__METHOD__	Имя метода класса
__DIR__	Текущий каталог, в котором расположен скрипт
PHP_VERSION	Версия интерпретатора PHP
OS_VERSION	Имя операционной системы, под управлением которой работает PHP
PHP_EOL	Символ конца строки, используемый на текущей платформе: \n для Linux/Mac OS X и \r\n для Windows

Помимо представленных в табл. 6.1 констант, которые доступны всегда, существует огромное количество предопределенных констант, объявленных в различных расширениях. Привести их полный список не представляется возможным, кроме того, их набор зависит от подключенных в данный момент расширений и может меняться. В любом случае получить полный список доступных в текущий момент констант можно при помощи функции `get_defined_constants()`.

В листинге 6.7 приводится пример использования предопределенных констант. Константы `__LINE__` и `__FILE__` удобно использовать для вывода сообщений об ошибках.

Листинг 6.7. Использование предопределенных констант. Файл pre_const.php

```
<?php
echo 'Имя файла ' . __FILE__ . '<br />';
echo 'Строка ' . __LINE__;
```

6.5. Абсолютный и относительный пути к файлу

Предопределенную константу `__DIR__` довольно часто используют совместно с конструкциями `include` и `require` *(см. разд. 3.6)*. При подключении к скрипту файлов следует указать к ним путь. Причем путь может быть либо относительным

```
<?php
require_once 'utils/lib.php';
```

либо абсолютным, от корня диска

```
<?php
require_once '/var/www/project/utils/lib.php';
```

Подавляющее большинство PHP-разработчиков предпочитают относительные пути, как более короткие и не зависящие от сервера размещения или рабочей станции, где ведется разработка. Проект 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 довольно замысловатыми. Например, пусть имеется следующая структура каталогов:

```
include
  classes.php
utils
  lib.php
catatalogs
  index.php
```

Точкой входа служит catalogs/index.php. Для того 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 могут выглядеть следующим образом:

```
<?php
require_once '../include/classes.php';
require_once '../unils/index.php';
```

В случае, если в каталоге catalogs расположен подкаталог и требуется подключить файлы в скрипте catalogs/cars/index.php, 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. Тем более предпочтительны 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.

> **ЗАМЕЧАНИЕ**
>
> Оператор точка . используется для объединения или конкатенации строк. Более подробно оператор рассматривается в *главе 12*.

```
<?php
require_once __DIR__  . 'include/classes.php';
require_once __DIR___ . 'unils/index.php';
```

Кроме того, приложения стараются разрабатывать в объектно-ориентированном стиле, что позволяет воспользоваться механизмом пространства имен *(см. главу 23)* и автозагрузкой *(см. главу 24)*, не требующими явного включения файлов классов при помощи конструкций include и require.

6.6. Константы класса

Классы также могут содержать константы, которые определяются при помощи ключевого слова const. В листинге 6.8 приводится пример класса ConstantClass, включающего в свой состав константу NAME, которая содержит имя класса.

> **ЗАМЕЧАНИЕ**
>
> Следует обратить внимание, что ключевое слово const используется только в классах, для объявления констант вне классов предназначена функция define().

Листинг 6.8. 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 используется оператор разрешения области видимости ::.

Существование констант может быть проверено при помощи функции defined(), которая возвращает true, если константа существует, и false — в противном случае (листинг 6.9).

> **ЗАМЕЧАНИЕ**
>
> При проверке классовых констант следует в обязательном порядке использовать оператор разрешения области видимости :: и имя класса.

Листинг 6.9. 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 />';
}
```

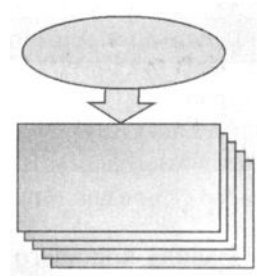
Результатом выполнения скрипта из листинга 6.9 будут следующие строки:

```
Константа определена
Константа не определена
```

Задания

1. Создайте скрипт, который выводил бы текущую версию PHP.
2. Используя константы и обслуживающие их функции, добейтесь от конструкции `require` поведения `require_once`. Иными словами, при многократном включении PHP-файла, например, с классом `Point`, код во включаемом файле должен выполниться только один раз.

ГЛАВА 7



Операторы

Листинги данной главы
можно найти в подкаталоге operators.

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 =, оператор создания строки <<<, мы познакомились в предыдущих главах.

В данной главе будут рассмотрены строковые, арифметические операторы и операторы сравнения. Остальные операторы будут обсуждаться в других главах книги по мере знакомства с языковыми конструкциями, совместно с которыми они используются.

7.1. Объединение строк. Оператор "точка"

В предыдущих главах уже упоминался оператор "точка", который позволяет объединять строки. В листинге 7.1 приводится пример, в котором при помощи точки объединяются две строки и число в одну целую строку. Число при этом автоматически приводится к строковому типу.

> ***Замечание***
>
> Во многих языках программирования для объединения строк используется оператор +. В PHP это не так, при использовании строк в выражении с оператором + строка автоматически приводится к числовому значению. Если строка не содержит ничего похожего на число, она приводится к нулевому значению.

Листинг 7.1. Использование оператора "точка". Файл concat.php

```
<?php
$number = 216;
echo 'Сегодня ' . $number . ' участников'; // Сегодня 216 участников
```

Скрипт из листинга 7.1 можно записать в альтернативной форме путем интерполяции переменной, а не объединения строк (листинг 7.2).

Листинг 7.2. Альтернативная запись. Файл interpolation.php

```
<?php
$number = 216;
echo "Сегодня $number участников";
echo "Сегодня {$number} участников";
```

Правила хорошего тона предписывают по возможности использовать одиночные кавычки для создания строк, в которых не используется интерполяция. В тех 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 7.1 с учетом оператора `.=` можно переписать так, как это представлено в листинге 7.3.

Листинг 7.3. Использование оператора .=. Файл concat_eq.php

```
<?php
$number = 216;
$str = 'Сегодня ';
$str .= $number;
$str .= ' участников';
echo $str;
```

7.2. Конструкция *echo*. Оператор "запятая"

Конструкция `echo` предназначена для вывода переменных в окно браузера и уже многократно использовалась в предыдущих главах. Конструкция имеет следующий синтаксис:

```
void echo(string $arg1 [, string $... ])
```

Может принимать один или более параметров, разделенных запятой, которые она выводит в окно браузера. В листинге 7.1 приводился пример, в котором три строки объединялись в одну перед выводом их конструкцией `echo`. Однако эту операцию можно осуществить средствами конструкции, просто перечислив все параметры через запятую после ключевого слова `echo` (листинг 7.4).

Листинг 7.4. Использование оператора "запятая". Файл comma.php

```
<?php
$number = 216;
echo 'Сегодня ', $number, ' участников'; // Сегодня 216 участников
```

Параметры конструкции `echo` могут помещаться в необязательные круглые скобки (листинг 7.5).

Листинг 7.5. Использование круглых скобок в `echo`. Файл echo_bracket.php

```
<?php
$number = 216;
echo('Сегодня ', $number, ' участников'); // Сегодня 216 участников
```

Часто вместо конструкции `echo` используют строковую функцию `print()`, которая также позволяет выводить переменные в окно браузера. Однако функция `print()` принимает только один параметр, поэтому в ней нельзя приводить параметры через запятую. Кроме того, функция возвращает `true` в случае успешного вывода переменной, иначе возвращается `false`. Конструкция `echo` не возвращает ничего (листинг 7.6).

Замечание

Строковые функции подробно рассматриваются в *главе 12*.

Листинг 7.6. Использование строковой функции `print()`. Файл print.php

```
<?php
$number = 216;
print('Сегодня ' . $number . ' участников'); // Сегодня 216 участников
echo print('Hello, world!');    // Hello, world!
// print echo "Hello, world!";  // Ошибка
```

7.3. Арифметические операторы

Выполнение операций над числами производится при помощи арифметических операторов, которые перечислены в табл. 7.1.

***Таблица 7.1.** 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. Для того чтобы изменить такой порядок, следует прибегать к группированию чисел при помощи круглых скобок (листинг 7.7).

> ***Замечание***
>
> Традиционно до и после арифметического оператора помещаются пробелы, т. к. это позволяет увеличить читабельность программы. Однако это необязательное требование. Так, выражение в листинге 7.7 может быть записано следующим образом: `(5+3)*(6+7)`.

Листинг 7.7. Использование арифметических операторов. Файл operators.php

```
<?php
echo (5 + 3) * (6 + 7); // 104
```

В отличие от многих других языков программирования, если при операции деления одного целого числа на другое получается дробное число, то результат автоматически приводится к вещественному типу (листинг 7.8).

> ***Замечание***
>
> 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 изменить значение до 4, то вместо дроби 1.66666666667 будет выведено число 1.667. Изменив значение директивы до 20, можно наблюдать ошибки вычисления, накапливающиеся в конце дроби — 1.6666666666666667407.

Листинг 7.8. Использование оператора деления /. Файл division.php

```
<?php
echo 5 / 3; // 1.66666666667
```

Для того чтобы получить целое значение, потребуется явно привести результат деления к целому типу либо при помощи конструкции `(int)`, либо при помощи функции `intval()` (листинг 7.9).

Листинг 7.9. Получение целочисленного результата деления. Файл int.php

```
<?php
echo (int)(5 / 3); // 1
```

Как видно из листинга 7.9, дробная часть при приведении к целому числу отбрасывается.

Для того чтобы выяснить, делится ли число без остатка на другое число, можно воспользоваться оператором деления по модулю `%` (листинг 7.10).

Листинг 7.10. 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 7.11).

Листинг 7.11. Проверка числа на четность. Файл even_odd.php

```
<?php
$number = 5317;
if ($number % 2) echo 'Число не четное';
else echo 'Число четное';
```

Помимо операторов, приведенных в табл. 7.1, можно также использовать "сокращенную запись" арифметических операторов (табл. 7.2). 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 приведенных выше операторов.

Таблица 7.2. *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;`

Замечание

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. разд. 8.3)*.

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 7.12).

Листинг 7.12. 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 7.13). Для удобства далее в примерах будет говориться только об операторе инкремента — все, что справедливо для него, справедливо также и для оператора декремента.

Листинг 7.13. 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 7.14.

Листинг 7.14. 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 7.15 приведен пример кода, в котором операция инкремента применяется к переменной типа `string`.

Листинг 7.15. Применение инкремента к строке. Файл increment_str.php

```
<?php
$var = "aaa";
echo ++$var;
```

Результатом выполнения этого кода будет строка `aab`, т. к. буква `b` является следующей за `a` буквой алфавита.

7.4. Поразрядные операторы

Данная группа операторов предназначена для выполнения манипуляций над отдельными битами числовых значений (табл. 7.3). Поразрядные операторы часто также называют *битовыми операторами*.

***Таблица 7.3.** Поразрядные операторы*

Оператор	Описание
`&`	Поразрядное пересечение — И (AND)
`|`	Поразрядное объединение — ИЛИ (OR)
`^`	Поразрядное исключающее ИЛИ (XOR)
`~`	Поразрядное отрицание (NOT)
`<<`	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда
`>>`	Сдвиг вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда

Замечание

Поразрядные операторы не только интенсивно используются в криптографии и компьютерной графике, но и применяются для кодирования констант в директивах php.ini и многих функциях PHP. Когда вы видите на страницах книги или в документации конструкцию вида `E_ALL & ~E_NOTICE`, помните, что это две числовые константы, соединенные побитовым И, при этом к константе `E_NOTICE` применено отрицание.

Рассмотрим поразрядные операторы более подробно. Оператор `&` соответствует побитовому И (пересечение). При использовании битовых операторов следует помнить, что все операции выполняются над числами в двоичном представлении. Пусть имеются два числа — 6 и 10, применение к ним оператора `&` дает 2 (листинг 7.16).

Листинг 7.16. Использование оператора &. Файл bit_and.php

```
<?php
echo 6 & 10; // 2
```

Этот, казалось бы, парадоксальный результат можно легко объяснить, если перевести числа 6 и 10 в двоичное представление, в котором они равны 0110 и 1010 соответственно. Сложение разрядов при использовании оператора & происходит согласно правилам, представленным в табл. 7.4; учитывая их, можно легко получить результат, который равен 0010 (в десятичной системе — 2).

```
0110 (6)
1010 (10)
0010 (2)
```

Таблица 7.4. *Правила сложения разрядов при использовании оператора &*

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	0	0	0

В листинге 7.17 приведен более сложный пример — использование оператора & для чисел 113 и 45.

Листинг 7.17. Использование оператора &. Файл bit_and_use.php

```
<?php
echo 113 & 45; // 33
```

Для наглядности представим складываемые числа в двоичном представлении:

```
1110001 (113)
0101101 (45)
0100001 (33)
```

Оператор | соответствует побитовому ИЛИ (пересечение). В табл. 7.5 приведены правила сложения разрядов при использовании оператора |.

Таблица 7.5. *Правила сложения разрядов при использовании оператора |*

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	1	1	0

Применяя операцию | к числам, рассмотренным ранее, получим следующий результат (листинг 7.18).

Листинг 7.18. Использование оператора |. Файл bit_or.php

```
<?php
echo 6 | 10;     // 14
echo '<br />';   // Перевод строки
echo 113 | 45;   // 125
```

Распишем числа в двоичном представлении, чтобы действие оператора было более понятно:

```
0110 (6)
1010 (10)
1110 (14)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1111101 (125)
```

Оператор ^ соответствует побитовому исключающему ИЛИ (XOR). В табл. 7.6 приведены правила сложения разрядов при использовании оператора ^.

Таблица 7.6. *Правила сложения разрядов при использовании оператора* ^

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	0	1	1	0

В листинге 7.19 приведен пример использования данного оператора с числами 6, 10 и 113, 45.

Листинг 7.19. Использование оператора ^. Файл bit_xor.php

```
<?php
echo 6 ^ 10;     // 12
echo '<br />';
echo 113 ^ 45;   // 92
```

Распишем числа в двоичном представлении, чтобы действие оператора было более понятно:

```
0110 (6)
1010 (10)
1100 (12)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1011100 (92)
```

Оператор отрицания ~ является побитовой инверсией и заменяет все 0 на 1 и наоборот. В листинге 7.20 демонстрируется использование данного оператора.

Листинг 7.20. Использование оператора ~. Файл bit_not.php

```
<?php
echo ~45;     // -46
echo '<br />';
echo ~92;     // -93
```

Отношение бинарного представления отрицательных и положительных чисел можно выразить при помощи соотношения ~(45 + 1) = –45.

Все побитовые операции производятся над целыми числами, знак в них кодируется первым разрядом: если он равен 0, число положительное, если он равен 1 — отрицательное. В двоичных числах ведущие нули обычно не указывают, но т. к. операция ~ производит их инвертирование — они заменяются единицами, приводя к отрицательным значениям. Так, операция `~45` в двоичном представлении для 32-битного числа выглядит следующим образом:

```
00000000000000000000000000101101 (45)
11111111111111111111111111010010 (-46)
```

Помимо этого, бинарное представление отрицательных чисел инвертировано относительно положительных таким образом, чтобы их объединение при помощи & давало 0. Получающаяся при этом ведущая единица вытесняется за левую границу числа. Если вычисления осуществляются на 32-битном процессоре, позиций в бинарном представлении тоже будет 32, единица на 33 позиции отбрасывается, и получается число, все бинарные позиции которого равны 0, что эквивалентно нулю в любой системе счисления.

```
00000000000000000000000000101110 (46)
11111111111111111111111111010010 (-46)
00000000000000000000000000000000 (0)
```

Для `~92`:

```
00000000000000000000000001011100 (92)
11111111111111111111111110100011 (-93)
```

Оператор << сдвигает влево все биты первого операнда на число позиций, указанное во втором операнде (листинг 7.21). Сдвиг на отрицательное значение приводит к нулевому результату.

Листинг 7.21. Использование оператора <<. Файл bit_left_shift.php

```
<?php
echo 6 << 1;   // 12
echo '<br />';
echo 6 << 2;   // 24
echo '<br />';
echo 6 << 10;  // 6144
echo '<br />';
echo 6 << -1;  // 0
echo '<br />';
echo 6 << 31;  // 0
```

В двоичном представлении эти операции выглядят следующим образом:

- 6 << 1:
  0110 (6)
  1100 (12)
- 6 << 2:
  00110 (6)
  11000 (24)
- 6 << 10:
  0000000000110 (6)
  1100000000000 (6144)

Как видно из листинга 7.21, к нулевому результату приводит не только сдвиг на отрицательное число позиций, но и сдвиг на 31 позицию, т. к. это приводит к тому, что все 1 уходят за границу 32-битного числа и все разряды принимают значение 0.

```
00000000000000000000000000000000 (0)
```

Оператор >> сдвигает вправо все биты первого операнда на число позиций, указанных во втором операнде. Сдвиг на отрицательное значение приводит к нулевому результату (листинг 7.22).

Листинг 7.22. Использование оператора >>. Файл bit_right_shift.php

```
<?php
echo 6144 >> 10; // 6
echo '<br />';
echo 45 >> 2;    // 11
echo '<br />';
echo 45 > 2;     // 1
```

В двоичном представлении эти операции выглядят следующим образом:

- 6144 >> 10:
  1100000000000 (6144)
  0000000000110 (6)

❒ `45 >> 2:`
```
101101 (45)
001011 (11)
```

Как видно из листинга 7.22, при использовании операторов сдвига очень легко ошибиться, указав только один знак > (оператор "больше", который рассматривается в *разд. 7.5*). В этом случае интерпретатор PHP вернет либо `true`, либо `false`, которые будут приведены к 1 и 0 соответственно. За такими опечатками следует следить очень внимательно, т. к. PHP не генерирует предупреждений, в то же время результат будет отличаться от ожидаемого.

Также как и арифметические операторы, поразрядные операторы поддерживают сокращенную запись, позволяющую сократить выражения вида `$number = $number & $var` до `$number &= $var`. В табл. 7.7 приводятся сокращенные формы побитовых операторов.

Таблица 7.7. *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;`

7.5. Операторы сравнения

PHP предоставляет разработчику операторы сравнения, полный список которых представлен в табл. 7.8. Все эти операторы сравнивают заданные значения и возвращают `true` (истина) или `false` (ложь). Исключение составляет лишь оператор <=>, который возвращает целое число.

Таблица 7.8. *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`

Таблица 7.8 (окончание)

Оператор	Описание
`$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

Пример использования операторов сравнения приведен в листинге 7.23.

Замечание

Сами по себе операторы сравнения практически не используются, главное их предназначение — это совместная работа с оператором `if`, который подробно рассматривается в *главе 8*.

Листинг 7.23. Использование операторов сравнения. Файл compare.php

```
<?php
echo 1 > 0;              // true
echo 1 > 1;              // false
echo 1 >= 1;             // true
echo 1 < 0;              // false
echo 1 < 1;              // false
echo 1 <= 1;             // true
echo 1 == 0;             // false
echo 1 == 1;             // true
echo 1 != 0;             // true
echo 1 != 1;             // false
echo 0 == '4bfj';        // false
echo 0 == '';            // true
echo 0 == 'hello world'; // true
echo 0 == null;          // true
```

Замечание

Ранее уже отмечалось, что строки автоматически приводятся интерпретатором PHP к числам. Если строки содержат что-то похожее на число, например `'5'`, `'3.14'` или `'4bfj'`, то они автоматически приводятся к числу `5`, `3.14` или `4` соответственно. Строки, не содержащие чисел, например `'Hello, world'`, автоматически приводятся к 0.

В комментариях листинга 7.23 приводятся значения `true` или `false`, реально в окно браузера выводится либо значение 1 для `true`, либо пустая строка для `false`.

Последние примеры в листинге 7.23 показывают, как ведут себя операторы, если одним из операндов выступает строка. Даже если значение 0 сравнивается с `null` при помощи оператора равенства ==, возвращается значение истины (`true`). Однако такое поведение является неудобным в ряде ситуаций — иногда требуется проверить, равно ли значение переменной пустой строке `''`, `0` или `null`. Для решения этой задачи предназначены операторы эквивалентности === и неэквивалентности !==, они возвращают `true` только в том случае, если значения операндов в точности соответствуют друг другу (листинг 7.24).

Листинг 7.24. Использование операторов эквивалентности. Файл eq.php

```
<?php
echo 0 === 0;     // true
echo 0 === '';    // false
echo 0 === null; // false
echo 1 != '1';    // false
echo 1 !== '1';   // true
```

Операции == и != имеют более низкий приоритет по сравнению с остальными. В силу этого, к примеру, выражение (`$x < $a == $b < $x`) является истинным только тогда, когда `$x` принадлежит интервалу от `$a` до `$b`. Это происходит потому, что сначала вычисляется (`$x < $a`), потом (`$x < $b`), и только затем к результатам применяется операция сравнения на равенство ==.

В заключение раздела приведем таблицы сравнения различных величин при помощи операторов равенства == (табл. 7.9) и эквивалентности === (табл. 7.10). В таблицах "T" обозначает `true`, а "F", соответственно, `false`.

Таблица 7.9. *Сравнение величин при помощи оператора равенства ==*

	TRUE	FALSE	1	0	–1	"1"	"0"	"–1"	NULL	array	"php"
TRUE	T	F	T	F	T	T	F	T	F	F	T
FALSE	F	T	F	T	F	F	T	F	T	T	F
1	T	F	T	F	F	T	F	F	F	F	F
0	F	T	F	T	F	F	T	F	T	F	F
–1	T	F	F	F	T	F	F	T	F	F	F
"1"	T	F	T	F	F	T	F	F	F	F	F
"0"	F	T	F	T	F	F	T	F	F	F	F
"–1"	T	F	F	F	T	F	F	T	F	F	F
NULL	F	T	F	T	F	F	F	F	T	T	F
array	F	N	F	F	F	F	F	F	F	T	F
"php"	T	F	F	T	F	F	F	F	F	F	T

Таблица 7.10. *Сравнение величин при помощи оператора эквивалентности ===*

	TRUE	FALSE	1	0	–1	"1"	"0"	"–1"	NULL	array	"php"
TRUE	T	F	F	F	F	F	F	F	F	F	F
FALSE	F	T	F	F	F	F	F	F	F	F	F
1	F	F	T	F	F	F	F	F	F	F	F
0	F	F	F	T	F	F	F	F	F	F	F
–1	F	F	F	F	T	F	F	F	F	F	F
"1"	F	F	F	F	F	T	F	F	F	F	F
"0"	F	F	F	F	F	F	T	F	F	F	F
"–1"	F	F	F	F	F	F	F	T	F	F	F
NULL	F	F	F	F	F	F	F	F	T	F	F
array	F	F	F	F	F	F	F	F	F	T	F
"php"	F	F	F	F	F	F	F	F	F	F	T

7.6. Приоритет выполнения операторов

В заключение приведем таблицу приоритета выполнения операторов (табл. 7.11). Операторы с более низким приоритетом расположены в таблице ниже, с более высоким — выше. Операторы, расположенные на одной строке, имеют одинаковый приоритет, и выполняется в первую очередь тот из операторов, который встречается в выражении первым.

Замечание

В данной главе были рассмотрены не все операторы, остальные операторы будут представлены в последующих главах.

Таблица 7.11. *Приоритет выполнения операторов*

Оператор
clone new
[]
**
++ -- ~ (int) (float) (string) (array) (object) @
instanceof
!
* / %
+ - .
<< >>

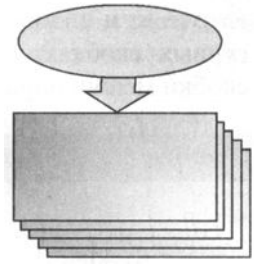
Таблица 7.11 (окончание)

Оператор
< <= > >=
== != === !== <> <=>
&
^
|
&&
||
??
? :
= += -= *= /= .= %= &= |= ^= <<= >>=
and
xor
or
,

Задания

1. Создайте скрипт, который возводил бы одно целое число в другое, не прибегая к оператору `**` или встроенной функции `pow()`.
2. Создайте скрипт, который определял бы четность или нечетность числа только при помощи поразрядных операторов и конструкции `if`.

ГЛАВА 8



Условия

Листинги данной главы
можно найти в подкаталоге conditions.

Ветвление является одной из базовых возможностей всех современных языков программирования. В зависимости от выполнения или не выполнения условий, ветвление позволяет выполнить тот или иной набор выражений, объединенных составным оператором. Для реализации ветвления PHP предоставляет несколько операторов и конструкций, которым будет посвящена текущая глава.

8.1. Условный оператор *if*

Условный оператор `if` имеет следующий синтаксис:

```
if (условие) оператор1 else оператор2
```

В качестве аргумента *условие* оператор `if` принимает логическую переменную или выражение, возвращающее логическое значение. Если оно истинно, то выполняется *оператор1*. В противном случае выполняется *оператор2* (листинг 8.1).

Листинг 8.1. Использование условного оператора. Файл ifelse.php

```
<?php
$flag = true; // Истина
if ($flag) {
    echo '<p>Переменная flag имеет значение true</p>';
} else {
    echo '<p>Переменная flag имеет значение false</p>';
}
```

Оператор `if` проверяет условие `$flag`, и если его значение `true`, выполняется код в фигурных скобках, следующий за `if`, а если `false` — код после ключевого слова `else`. Если блок `else` не требуется, его можно опустить (листинг 8.2). Кроме этого,

оператор1 и *оператор2* необязательно должны быть составными операторами в фигурных скобках; если необходимо выполнить только один оператор, фигурные скобки можно опустить.

Листинг 8.2. Сокращенная запись оператора `if`. Файл if.php

```
<?php
$number = 4;
if ($number == 4) echo 'Число равно 4';
```

Однако стандарт кодирования PSR запрещает использовать оператор `if` без фигурных скобок. Поэтому далее в книге они будут везде указываться.

Проверка дополнительных условий возможна также при помощи оператора `elseif`:

```
if (условие) {
  операторы;
}
elseif (условие) {
  операторы;
} else {
  операторы;
}
```

Оператор `if` может включать сколько угодно блоков `elseif`, но ключевое слово `else` в каждом `if` может присутствовать только в одном экземпляре. Как правило, в конструкциях `if...elseif...else` оператор `else` определяет, что нужно делать, если никакие другие условия не являются истинными.

В листинге 8.3 приводится пример использования конструкции `if...elseif` для проверки значения переменной.

Листинг 8.3. Использование конструкции `if ... elseif`. Файл elseif.php

```
<?php
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';
}
```

PHP предоставляет также возможность альтернативного синтаксиса условного оператора — без фигурных скобок. В этом случае операторы `if`, `else` и `elseif` заканчиваются двоеточием, а сама конструкция `if` завершается обязательным ключевым словом `endif`.

В следующем примере h1-заголовок помещается на страницу в зависимости от значения переменной $hdd. Наличие оператора endif в этом случае обязательно, т. к. фигурная скобка, обозначающая конец блока if, отсутствует (листинг 8.4).

Листинг 8.4. Альтернативный синтаксис оператора if. Файл alterif.php

```
<?php
if ($hdd == 'Western Digital'):
?>
<h1> Western Digital </h1>
<?php
elseif ($hdd == 'Seagate'):
?>
<h1> Seagate </h1>
<?php
else :
?>
<h1> Неизвестный производитель </h1>
<?php
endif;
```

Впрочем, использование альтернативного синтаксиса оператора if, как и смешение PHP- и HTML-кода, в современной разработке не поощряется и напрямую запрещается стандартом кодирования PSR. Поэтому вы вряд ли встретите такую конструкцию на практике.

8.2. Логические операторы

В предыдущих главах и *разд. 8.1* в операторе if фигурировало лишь одно условие. Однако в повседневной практике зачастую требуется использовать несколько условий, объединенных логическими операторами. Логические операторы очень похожи на поразрядные операторы (см. *главу 7*), только вместо чисел они оперируют переменными логического типа. В табл. 8.1 приводится список логических операторов.

***Таблица 8.1.** 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
$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 записывается как `&&`. Оператор возвращает `true`, если оба операнда равны `true`, и `false` в любом другом случае. Пример использования оператора `&&` приведен в листинге 8.5.

> ***Замечание***
>
> Переменную типа `boolean`, принимающую только два значения `true` и `false` (или 1 и 0), часто называют *флагом*. Причем флаг считается установленным, если переменная принимает значение `true`, и сброшенным, если переменная принимает значение `false`.

Листинг 8.5. Использование оператора && (логическое И). Файл and.php

```
<?php
$flag1 = true; // Истина
$flag2 = true; // Истина
if ($flag1 && $flag2) {
    echo '<p>Оба флага истинны</p>';
}
```

Код, представленный в листинге 8.5, эквивалентен по результату коду из листинга 8.6.

Листинг 8.6. Альтернативное представление двойного условия. Файл fewif.php

```
<?php
$flag1 = true; // Истина
$flag2 = true; // Истина
if ($flag1) {
    if ($flag2) {
        echo '<p>Оба флага истинны</p>';
    }
}
```

Зачастую проще прибегнуть к двум операторам `if`, чем к двойному условию (см. листинг 8.5), т. к. совершить ошибку в нем проще. Однако некоторые конструкции, например с участием блока `else`, наиболее эффективны только при использовании двойного условия (листинг 8.7).

Листинг 8.7. Двойное условие совместно с блоком else. Файл twocond.php

```
<?php
$flag1 = true; // Истина
$flag2 = true; // Истина
if ($flag1 && $flag2) { // и $flag1, и $flag2 равны true
    echo '<p>Условие: true (Оба флага истинны)</p>';
```

```
} else {
    echo '<p>Условие: false (Один или оба флага ложны)</p>';
}
```

Скрипт, представленный в листинге 8.7, выведет в окно браузера фразу:

```
Условие: true (Оба флага истинны)
```

Для того чтобы воспроизвести этот результат при помощи двух блоков `if`, потребуется после каждого из них поместить блок `else`, в котором будет продублирована вторая фраза (листинг 8.8), что чрезвычайно неудобно, особенно если по мере разработки приложения потребуется изменить ее.

> ***Замечание***
>
> Следует всеми силами стараться избегать дублирующего кода: обнаруженная в нем ошибка или потребность модернизации потребует искать дублирующие блоки по всему коду — пара из них рано или поздно останется не поправленной, приводя к трудно локализуемым ошибкам.

Листинг 8.8. Неудачное дублирование кода. Файл double.php

```
<?php
$flag1 = true; // Истина
$flag2 = true; // Истина
if ($flag1) {
    if ($flag2) {
        echo '<p>Оба флага истинны</p>';
    } else {
        echo '<p>Условие: false (Один из флагов ложен)</p>';
    }
} else {
    echo '<p>Условие: false (Один из флагов ложен)</p>';
}
```

По аналогии с поразрядным оператором ИЛИ (`|`) существует логическое ИЛИ, которое обозначается двумя вертикальными чертами — `||`. Заменим в листинге 8.6 оператор `&&` (И) оператором `||` (ИЛИ) (листинг 8.9).

Листинг 8.9. Использование оператора || (логическое ИЛИ). Файл or.php

```
<?php
$flag1 = true;  // Истина
$flag2 = false; // Ложь
if ($flag1 || $flag2) { // true
    echo '<p>Условие: true (Один из флагов истинен)</p>';
} else {
    echo '<p>Условие: false (Оба флага ложны)</p>';
}
```

Оператор || возвращает true, если хотя бы один из его операндов равен true, и false, если оба операнда равны false. В результате работы скрипта из листинга 8.9 получаем:

```
Условие: true (Один из флагов истинен)
```

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 8.10 демонстрируется использование оператора or.

Листинг 8.10. Использование оператора or (логическое ИЛИ). Файл oralt.php

```
<?php
$flag1 = true;          // Истина
$flag2 = false;         // Ложь
if ($flag1 or $flag2) { // true
    echo '<p>Условие: true (Один из флагов истинен)</p>';
} else {
    echo '<p>Условие: false (Оба флага ложны)</p>';
}
```

Особенность логических операторов || и or заключается в том, что если первый операнд равен true, то в вычислении второго операнда отсутствует надобность — не зависимо от того, будет он равен true или false, логическое выражение все равно примет значение true. Поэтому существует практика для функций, возвращающих в качестве значения true или значение, которое автоматически приводится к true, осуществлять проверку корректности выполнения при помощи операторов || или or (листинг 8.11).

> ***Замечание***
>
> Функция exit() останавливает работу скрипта и выводит сообщение, переданное ей в качестве параметра, в окно браузера. Функция file_get_contents() читает содержимое файла, в том числе по сети.

Листинг 8.11. Проверка выполнения функции. Файл checkor.php

```
<?php
file_get_contents('http://php.net') or exit('Ошибка');
// Дальнейшие операторы
// ...
```

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 — 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(), сообщающая о возникшей ошибке.

Следует отметить, что функция file_get_contents() возвращает строку, и если содержимое запрашиваемой страницы требуется для дальнейших действий, выражение проверки может выглядеть так, как это демонстрируется в листинге 8.12.

Листинг 8.12. 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.

Если бы вместо оператора or использовался оператор ||, имеющий больший приоритет, чем =, сначала бы осуществлялось вычисление оператора ||, а затем результат этого присваивался бы переменной $content, в итоге она имела бы не содержимое страницы, а значение true. Таким образом, если потребовалась бы проверка корректности выполнения функции, необходимо было бы воспользоваться дополнительными кавычками (листинг 8.13).

Листинг 8.13. Файл file_get_contents_or.php

```
<?php
($content = file_get_contents('http://php.net')) || exit('Ошибка');
echo $content;
```

В любом случае старайтесь избегать проверок, основанных на операторах || и or. Вместо этого лучше использовать оператор if, тогда скрипты получаются пусть и несколько более объемными, зато более читабельными.

Стиль программирования, приведенный в листингах 8.11–8.13, был заимствован из языка программирования Perl. В нем такие конструкции выглядят естественно и гармонично увязаны с остальными конструкциями языка. В PHP, который чаще используется для промышленной разработки, стиль программирования, основанный на эксплуатации приоритета операторов, не приветствуется.

Иногда бывает необходимо проверить условие безальтернативно на ложность или истинность. В этом случае в применении ключевого слова else и следующих за ним операторов нет надобности. Например, если требуется вывести сообщение только в случае удачного выполнения функции file_get_contents('http://php.net'), можно воспользоваться простейшей формой оператора if (листинг 8.14).

Листинг 8.14. Файл file_get_contents_success.php

```
<?php
if (file_get_contents('http://php.net')) {
    echo '<p>Имеется сетевой доступ<p>';
}
```

Если же необходимо вывести сообщение только в том случае, если работа функции завершилась неудачно, код может выглядеть так, как это представлено в листинге 8.15.

Листинг 8.15. Файл file_get_contents_wrong.php

```
<?php
if (file_get_contents('http://php.net')) {

} else {
    echo '<p>Отсутствует сетевой доступ<p>';
}
```

Пустой составной оператор `{}` вполне допускается синтаксисом языка. Однако способ, продемонстрированный в листинге 8.15, не является элегантным. Здесь лучше воспользоваться оператором отрицания `!`, применение которого к переменной меняет ее значение с `true` на `false` и наоборот (листинг 8.16).

Листинг 8.16. Использование оператора отрицания !. Файл not.php

```
<?php
if (!file_get_contents('http://php.net')) {
    echo '<p>Отсутствует сетевой доступ<p>';
}
```

8.3. Условный оператор *x ? y : z*

Оператор `if` в PHP можно заменить *условным оператором (тернарным оператором)*, который имеет следующий синтаксис:

выражение1* ? *выражение2* : *выражение3

Первым вычисляется значение *выражение1*. Если оно истинно, то вычисляется значение *выражение2*, которое и становится результатом. Если *выражение1* ложно, то в качестве результата берется *выражение3*. Классическим примером условной операции является получение абсолютного значения переменной (листинг 8.17).

Листинг 8.17. 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.

8.4. Оператор ??

В PHP 7 введен специальный оператор `??`, который позволяет проинициализировать переменную только в том случае, если переменная не была ранее проинициализирована или ей присвоено значение `null` (листинг 8.18).

Листинг 8.18. Файл ternary_alt.php

```
<?php
$y = null;
$z = 'hello';
$x = $x ?? 1; // 1
$y = $y ?? 2; // 2
$z = $z ?? 'world'; // '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.

8.5. Переключатель *switch*

Переключатель `switch` предоставляет более удобные средства для организации множественного выбора, чем оператор `elseif`. Оператор имеет следующий синтаксис:

```
switch(выражение)  // переключающее выражение
{
  case значение1:  // выражение 1
    операторы;     // блок операторов
    break;
  case значение2:   // выражение 2
    операторы;
    break;
```

```
  default:
    операторы;
}
```

Управляющая структура `switch` передает управление тому из помеченных операторов `case`, для которого значение константного выражения совпадает со значением переключающего выражения.

Сначала анализируется переключающее *выражение* и осуществляется переход к той ветви программы, для которой его значение совпадает с выражением в операторе `case`. Далее следует выполнение оператора или группы операторов 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 оператору, помеченному 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 8.19 в зависимости от того, какое значение принимает переменная `$answer`, в окно браузера выводится та или иная фраза. Если переменная `$answer` принимает значение "yes", выводится фраза "Продолжаем работу!", если переменная принимает значение "no", выводится фраза "Завершаем работу". Если `$answer` принимает любое другое значение, управление передается блоку `default` и выводится фраза "Некорректный ввод".

Листинг 8.19. Пример использования оператора `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';
}
```

Операторы в блоке `case` могут быть заключены в необязательные фигурные скобки.

Если пропущен оператор `break`, то скрипт выполняет операторы следующего блока до тех пор, пока не закончится `switch` или не встретится `break`. В листинге 8.20 выводятся названия нечетных целых десятичных чисел от 1 до 9, не меньше введенного.

Листинг 8.20. 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` примет значение 2, то результат работы скрипта может выглядеть следующим образом:

```
три пять семь девять
```

Оператор `switch` переключает скрипт на позицию `case 2`, пропуская позицию `case 1`, и далее скрипт выполняется до ближайшего оператора `break`.

Помимо синтаксиса, представленного в начале раздела, PHP позволяет использовать альтернативный синтаксис без фигурных скобок с применением ключевых слов `switch:` и `endswitch`. В листинге 8.21 представлена альтернативная запись оператора `switch` для скрипта из листинга 8.19.

Листинг 8.21. Альтернативная форма оператора `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 представлена при помощи оператора `if...elseif`. Так, скрипт из листинга 8.21 можно переписать следующим образом (листинг 8.22).

Листинг 8.22. 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 оператор `if...elseif` более гибкий, т. к. позволяет оперировать сложными условиями с применением логических операторов, например, так, как это демонстрируется в листинге 8.23.

Листинг 8.23. 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 оператор `switch` позволяет реализовывать точно такие же построения благодаря тому, что логическое выражение можно сравнивать со значением `true` или `false`. В листинге 8.24 приводится скрипт, аналогичный скрипту 8.25 по функциональности за счет того, что переменная `$number` может использоваться для формирования условий в конструкциях `case`.

Листинг 8.24. 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;
}
```

8.6. Оператор *goto*

Оператор `goto` позволяет осуществлять безусловный переход на метку, название которой указывается в качестве единственного аргумента.

```
goto метка;
...
метка:
```

В листинге 8.25 приводится пример организации цикла при помощи двух операторов `goto`.

Листинг 8.25. Использование оператора `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`, 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`.

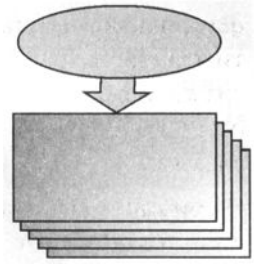
Оператор `goto` в языках высокого уровня возник из-за соображений производительности. Когда появлялись языки высокого уровня, они не могли конкурировать с ассемблером, способным молниеносно перемещаться к инструкции по любому адресу в программе. Чтобы дополнить первые языки высокого уровня схожей функциональностью, их снабжали оператором `goto`, который позволяет 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 запутанных программ с использованием goto было немало. После критической статьи Э. Дейкстры, в которой было показано, что при разработке программ можно обойтись без оператора goto, началось массовое движение по отказу от этого оператора. Современное поколение разработчиков практически с ним не знакомо.

В действительности же ключевое слово было зарезервировано с самых первых версий языка PHP, но оператор не вводился до версии 5.3 из-за его плохой репутации. В текущей реализации его возможности сильно ограничены по сравнению с goto прошлых лет. Вы не сможете перемещаться при помощи goto в другой файл, в функцию или из функции. Нельзя перейти и внутрь цикла или оператора switch. Фактически goto — это более удобная замена break.

Задания

1. В документации на сайте **http://php.net** найдите описание функций file_get_contents() и file_put_contents(), которые позволяют читать и записывать содержимое файла. Создайте два скрипта, первый должен записывать строку "Hello, world!" в файл hello.txt, а второй — считывать содержимое файла hello.txt и выводить его в окно браузера.
2. Напишите скрипт, который при вызове создает в текущем каталоге файл с именем, отражающем текущие дату и время в формате "год-месяц-число-часминута-секунда", например 2017-04-16-13-10-13.txt. В файл запишите случайное число от 0 до максимально возможного целого числа в PHP.
3. Напишите скрипт, который при вызове создает два файла: со списком всех возможных расширений extensions.txt и со списком всех предопределенных констант PHP constants.txt.



ГЛАВА 9

Циклы

Листинги данной главы
можно найти в подкаталоге loops.

Циклы — фундаментальные конструкции, предназначенные для программирования повторяющихся действий. В том или ином виде циклы можно обнаружить практически во всех современных языках программирования.

В данной главе не рассматривается оператор цикла `foreach()`, обсуждение которого отложено до *главы 10*, посвященной массивам.

9.1. Цикл *while*

Оператор `while` называется *оператором цикла с предусловием* (листинг 9.1). Синтаксис оператора таков:

```
while (условие) {
    операторы;
}
```

При входе в цикл вычисляется *условие*, и, если его значение истинно (`true`), выполняется тело цикла. После завершения выполнения операторов выражение-условие вычисляется повторно, и, если оно по-прежнему равно `true`, операторы тела цикла выполняются повторно. Это происходит до тех пор, пока значение выражения-условия не станет ложным (`false`). Тело цикла не обязательно должно быть заключено в фигурные скобки; если требуется выполнить только один оператор, они могут быть опущены. Однако стандарт кодирования PSR требует обязательного их присутствия.

> ***ЗАМЕЧАНИЕ***
>
> Необходимо заботиться о том, чтобы рано или поздно *условие* принимало значение `false`, иначе произойдет образование *бесконечного цикла*, и приложение зависнет.

Листинг 9.1. Использование оператора `while`. Файл while.php

```
<?php
$i = 1;
while ($i <= 5) {
    echo "$i<br />";
    $i++;
}
```

Результатом выполнения кода из листинга 9.1 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 при помощи оператора инкремента `$i++`. Как только значение переменной `$i` достигает 6 и становится больше 5, выражение `$i <= 5` принимает значение `false` и на следующей итерации оператор `while` прекращает свою работу.

Достижение условия `$i <= 5` не приводит к немедленному выходу из цикла, цикл прекращает свою работу только после того, как начинается новая итерация (листинг 9.2).

Листинг 9.2. Файл while_false.php

```
<?php
$i = 0;
while ($i <= 5) {
    $i++;
    echo "$i<br />";
}
```

Результатом работы скрипта из листинга 9.2 будет колонка цифр от 1 до 6:

```
1
2
3
4
5
6
```

Таким образом, до того как условие `$i <= 5` будет проверено в начале очередной итерации, значение `$i` с "некорректным" значением выводится в стандартный поток вывода.

Однако существует способ прекратить выполнение цикла досрочно. Для этого применяется оператор `break`. При обнаружении этого оператора цикл прекращает свою работу немедленно (листинг 9.3).

Листинг 9.3. Использование оператора `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 9.3 является колонка цифр от 1 до 5. Когда `$i` становится больше 5, последний оператор `echo` в цикле просто не выполняется, т. к. `break` переводит текущий поток программы на первый оператор после цикла `while`.

Иногда требуется досрочно прекратить текущую итерацию и перейти сразу к началу следующей. Для этого применяется оператор `continue` (листинг 9.4).

> ***Замечание***
>
> **Оператор `switch` можно рассматривать как цикл, всегда выполняющийся один раз, поэтому в нем операторы `break` и `continue` эквивалентны по своему действию.**

Листинг 9.4. Пример оператора `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 операторы `break` по умолчанию действуют каждый на своем уровне вложения (листинг 9.5).

Листинг 9.5. 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 9.5 выводит пять раз подряд последовательность из пяти чисел от 1 до 5.

Оператор `break` может управлять не только своим циклом, но и внешним циклом, для этого следует указать его номер. По умолчанию, если передать `break` значение 1, его действия будут относиться к текущему циклу. В листинге 9.6 действия операторов `break` аналогичны скрипту из листинга 9.5.

Листинг 9.6. Указания номера цикла в операторе `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 оператору `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 9.7 приводится пример тройного вложенного цикла, который прерывается из самого внутреннего цикла.

Листинг 9.7. 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 `while` может иметь более замысловатую условную конструкцию. Например, цикл, представленный в листинге 9.8, выводит столбик цифр от 4 до 1.

Листинг 9.8. Файл while_decrement.php

```
<?php
$i = 5;
while (--$i) {
     echo "$i<br />";
}
```

Если использовать постфиксную форму оператора декремента `--`, как это представлено в листинге 9.9, выводится столбик цифр от 4 до 0.

Листинг 9.9. Файл while_post_decrement.php

```
<?php
$i = 5;
while ($i--) {
     echo "$i<br />";
}
```

Работа циклов из листингов 9.8 и 9.9 основана на том факте, что любое число больше нуля преобразуется к значению `true` (истина), а значение 0 к `false` (ложь). Как только значение `$i` принимает значение 0 — цикл прекращает свою работу. В листинге 9.8 используется префиксная форма декремента, в которой от числа сначала вычитается единица, и лишь затем возвращается значение, поэтому цифра 0 не выводится. В листинге 9.9 по достижении `$i` значения 1 в операторе `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 операторов `if` и `switch`, оператор `while` поддерживает синтаксис без фигурных скобок, с использованием ключевых слов `while:` и `endwhile`. В листинге 9.10 приводится скрипт из листинга 9.9, модифицированный в соответствии альтернативным синтаксисом.

Листинг 9.10. Альтернативный синтаксис оператора `while`. Файл while_alter.php

```
<?php
$i = 5;
while ($i--):
     echo "$i<br />";
endwhile;
```

9.2. 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 {
    операторы;
} while (условие);
```

Такой цикл всегда будет выполнен хотя бы один раз. После выполнения тела цикла вычисляется выражение *условие*, и, если оно истинно (`true`), вновь выполняется тело цикла. В листинге 9.11 ноль всегда будет добавлен в список, независимо от значения условия `(++$i <= 5)`.

> ***Замечание***
>
> При использовании цикла `do...while` также допускается применение операторов `break` и `continue`.

Листинг 9.11. Использование оператора `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 операторов цикла, а также операторов `if` и `switch`, данный вид цикла не поддерживает альтернативный синтаксис.

9.3. Цикл *for*

Итерационный цикл `for` имеет следующий синтаксис:

```
for (начало; условие; тело) {
    операторы;
}
```

Здесь оператор *начало* (инициализация цикла) — последовательность определений и выражений, разделяемая запятыми. Все выражения, входящие в инициализацию, вычисляются только один раз при входе в цикл. Как правило, здесь устанавливаются начальные значения счетчиков и параметров цикла. Смысл выражения-условия (*условие*) такой же, как и у циклов с пред- и постусловиями — цикл выполняется до тех пор, пока *условие* 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.

> ***Замечание***
>
> При использовании цикла `for` также допускается применение операторов `break` и `continue`.

В листинге 9.12 при помощи цикла выводятся цифры от 0 до 4.

Листинг 9.12. 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 `$i` увеличивается на единицу при помощи оператора инкремента (`$i++`). Цикл выполняет свою работу до тех пор, пока значение `$i` меньше 5. Результат работы скрипта из листинга 9.12 выглядит следующим образом:

```
0
1
2
3
4
```

В листинге 9.13 приводится пример вывода цифр от 5 до 1 при помощи цикла с декрементом.

Листинг 9.13. Использование декремента в цикле `for`. Файл for_decrement.php

```
<?php
for ($i = 5; $i; $i--) {
    echo "$i<br />";
}
```

Результат работы цикла из листинга 9.13 выглядит следующим образом:

```
5
4
3
2
1
```

Переменная `$i` инициируется значением 5, на каждой итерации цикла это значение уменьшается на единицу при помощи оператора декремента (`$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 `body`. Цикл из листинга 9.13 может быть переписан так, как это представлено в листинге 9.14.

Замечание

Использование конструкции `echo` внутри круглых скобок `for` не допускается.

Листинг 9.14. 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 операторов, разделенных запятой, — `print $i, print "<br />", $i--`. Такая форма записи вполне допускается синтаксисом PHP, но не приветствуется, т. к. сильно уменьшает читабельность кода.

Разрешено использование нескольких операторов и в блоке инициализации (листинг 9.15).

Замечание

Код в листинге 9.15 является ярким примером того, как не следует программировать. Здесь он приведен лишь для демонстрации особенностей работы цикла `for`.

Листинг 9.15. Файл for_complex.php

```
<?php
for ($i = 6, $j = 0;
     $i != $j;
     print "$i - $j", print "<br />", $i--, $j++);
```

Результат работы цикла из листинга 9.15 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` 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 9.16, выводятся значения от 0 до 100 с интервалом в 5.

Листинг 9.16. Использование шага, равного 5. Файл for_step_five.php

```
<?php
for ($i = 0; $i <= 100; $i += 5) {
    echo "$i<br />";
}
```

Выражения *начало*, *условие* и *тело* не обязательно должны присутствовать. Так, в листинге 9.17 приводится пример цикла `for`, в котором отсутствуют выражения *начало* и *тело*.

Листинг 9.17. 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 в листинге 9.17 может быть переписан (листинг 9.18).

Замечание

Такое использование цикла `for` не является типичным; если число циклов заранее не известно, традиционно используется цикл `while`.

Листинг 9.18. 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 на вложенные циклы является программа для нахождения простых чисел (листинг 9.19).

Замечание

Напомним, что *простыми* называются числа, которые делятся только на 1 и на самих себя.

Листинг 9.19. 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 9.19 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 оператору `continue` осуществляется продолжение

внутреннего цикла, поскольку этот оператор предписывает программе перейти на следующую итерацию цикла. Если же остаток от деления равен нулю, то происходит выход из внутреннего цикла по оператору `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 операторов `if` и `switch`, оператор `for` поддерживает синтаксис без фигурных скобок с использованием ключевых слов `for` и `endfor`. В листинге 9.20 приводится скрипт из листинга 9.12, модифицированный в соответствии альтернативным синтаксисом.

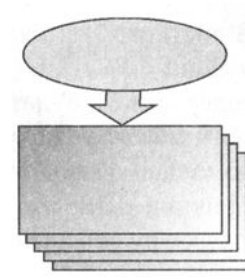
Листинг 9.20. Альтернативный синтаксис оператора `for`. Файл for_alter.php

```
<?php
for ($i = 0; $i < 5; $i++):
    echo "$i<br />";
endfor;
```

Задания

1. Числа Фибоначчи — это последовательность вида 0, 1, 1, 2, 3, 5, ... каждое число является суммой двух предыдущих чисел. Создайте скрипт, который бы вычислял любое наперед заданное число Фибоначчи, скажем, число с порядковым номером 200.
2. Создайте скрипт, который выводил бы календарь на текущий месяц в виде таблицы. Столбцы таблицы должны представлять дни недели от понедельника до воскресенья, а в ячейках таблицы выводиться числа месяца.

ГЛАВА 10



Массивы

Листинги данной главы
можно найти в подкаталоге arrays.

Массивы являются одной из основных и часто встречающихся структур для хранения данных. По определению, массив представляет собой индексированную совокупность переменных одного типа. Каждая переменная или элемент массива имеет свой *индекс*, т. е. все элементы массива последовательно пронумерованы от 0 до $N-1$, где N — *размер массива*. Имена массивов, так же как и переменных, начинаются с символа `$`. Для того чтобы обратиться к отдельному элементу массива, необходимо поместить после его имени квадратные скобки, в которых указать индекс элемента массива. Так, обращение к `$arr[0]` запрашивает первый элемент массива `$str`, `$arr[1]` — второй и т. д. В качестве элементов массива могут выступать другие массивы, в этом случае говорят о *многомерных массивах*, а для обращения к 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.

10.1. Создание массива

Существует несколько способов создания массивов. Первый из них заключается в использовании конструкции `array()`, в круглых скобках которой через запятую перечисляются элементы массива. Конструкция `array()` в качестве результата возвращает массив. В листинге 10.1 создается массив `$arr`, состоящий из трех элементов, пронумерованных от 0 до 2.

Листинг 10.1. Создание массива конструкцией `array()`. Файл array.php

```
<?php
$arr = array('Hello, ', 'world', '!');
echo $arr[0]; // Hello,
echo $arr[1]; // world
echo $arr[2]; // !
```

В листинге 10.1 каждый элемент массива выводится при помощи отдельной конструкции `echo`. При попытке вывода всего массива `$arr` в окно браузера вместо его содержимого будет выведена строка "Array". Для просмотра структуры и содержимого массива предусмотрена специальная функция `print_r()`. В листинге 10.2 с ее помощью выводится структура массива `$arr`. Для удобства восприятия вызов функции `print_r()` обрамляется HTML-тегами `<pre>` и `</pre>`, которые сохраняют структуру переносов и отступов при отображении результата в браузере.

Листинг 10.2. Вывод структуры массива. Файл print_r.php

```
<?php
$arr = array('Hello, ', 'world', '!');
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.2 будет следующий дамп массива `$arr`:

```
Array
(
    [0] => Hello,
    [1] => world
    [2] => !
)
```

Создать массив можно при помощи пары квадратных скобок (листинг 10.3).

Листинг 10.3. 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()`, т. к. исторически она появилась первой.

Как видно из листингов 10.1 и 10.2, 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 оператора `=>`. В листинге 10.4 первому элементу массива `$arr` назначается индекс 10, последующие элементы автоматически получают номера 11 и 12.

Листинг 10.4. Назначение индекса первому элементу массива. Файл index.php

```
<?php
$arr = [10 => 'Hello, ', 'world', '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.4 будет следующий дамп массива `$arr`:

```
Array
(
    [10] => Hello,
    [11] => world
    [12] => !
)
```

> ***Замечание***
>
> Следует отметить, что элементы с индексами 0, 1, ..., 9 и 13, ... просто не существуют, 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 10.5).

Листинг 10.5. Назначение индексов всем элементам массива. Файл indexes.php

```
<?php
$arr = [10 => 'Hello, ', 9 => 'world', 8 => '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.5 будет следующий дамп массива `$arr`:

```
Array
(
    [10] => Hello,
    [9] => world
    [8] => !
)
```

Следует отметить, что все непронумерованные элементы будут автоматически получать значение, равное максимальному и увеличенному на единицу (листинг 10.6).

Листинг 10.6. Файл indexes_default.php

```
<?php
$arr = [10 => 'Hello, ', 9 => 'world', '!'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.6 будет следующий дамп массива `$arr`:

```
Array
(
    [10] => Hello,
    [9] => world
    [11] => !
)
```

Еще одним способом создания массива является присвоение неинициализированным элементам массива новых значений (листинг 10.7).

Листинг 10.7. Файл square.php

```
<?php
$arr[10] = 'Hello, ';
$arr[11] = 'world';
$arr[12] = '!';
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Скрипт из листинга 10.7 по результату выполнения полностью эквивалентен листингу 10.4. Допускается и автоматическое назначение индексов элементам, для этого значение индекса просто не указывается в квадратных скобках (листинг 10.8).

Листинг 10.8. 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 10.9. Приведение переменной к массиву. Файл array_cast.php

```
<?php
$var = 'Hello, world';
$arr = (array) $var;
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.9 будет следующий дамп:

```
Array
(
    [0] => Hello, world
)
```

10.2. Ассоциативные и индексные массивы

В качестве индексов массивов могут выступать не только числа, но и строки. В последнем случае массив называют *ассоциативным*, а индексы — *ключами*. Если в качестве индексов массива выступают числа, он называется *индексным*. В листинге 10.10 приводится пример создания ассоциативного массива с двумя элементами, имеющими в качестве ключей строки 'one' и 'two'.

> ***Замечание***
>
> Один массив может иметь как числовые индексы, так и строковые ключи. В этом случае говорят о *смешанном массиве*.

Листинг 10.10. Создание ассоциативного массива. Файл assoc.php

```
<?php
$arr = ['one' => '1', 'two' => '2'];
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Результатом выполнения скрипта из листинга 10.10 будет следующий дамп массива `$arr`:

```
Array
(
  [one] => 1
  [two] => 2
)
```

При обращении к элементам ассоциативного массива вместо индексов указываются соответствующие ключи (листинг 10.11).

> ***Замечание***
>
> Ключи ассоциативного массива являются обычными строками, поэтому для обращения к элементам можно использовать как двойные, так и одиночные кавычки.

Листинг 10.11. 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 10.12).

Листинг 10.12. Файл assoc_add.php

```
<?php
$arr['one'] = '1';
$arr['two'] = '2';
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Если осуществляется попытка создания двух элементов массива с одинаковыми ключами, создается один элемент с последним значением (листинг 10.13).

> ***Замечание***
>
> Ключи ассоциативных массивов, как и обычные строки, зависят от регистра: 'two' и 'Two' считаются разными ключами.

Листинг 10.13. Файл assoc_same_keys.php

```
<?php
$arr = ['one' => '1', 'two' => '2', 'two' => '3'];
echo $arr['one']; // 1
echo '<br />';    // Перевод строки
echo $arr['two']; // 3
```

10.3. 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 10.14 демонстрируется создание многомерного ассоциативного массива `$ships`.

Листинг 10.14. Создание многомерного массива. Файл array_multi.php

```
<?php
$ships = [
    'Пассажирские корабли' => ['Лайнер', 'Яхта', 'Паром'],
    'Военные корабли' => ['Авианосец', 'Линкор', 'Эсминец'],
    'Грузовые корабли' => ['Сормовский', 'Волго-Дон', 'Окский']
];
echo '<pre>';
print_r($ship);
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, каждый из которых содержит в себе 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".

10.4. Интерполяция элементов массива в строки

Интересно подробнее остановиться на интерполяции элементов массива в строки. Вместо переменных в строках с двойными кавычками подставляется их значение. Точно так же ведут себя элементы массива (листинг 10.15).

Листинг 10.15. 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 10.16), в противном случае скрипт вернет ошибку разбора.

Листинг 10.16. 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 продемонстрировано в листинге 10.16, уже не получится. Для интерполяции потребуется либо заключить элемент в фигурные кавычки, либо использовать оператор "точка" (листинг 10.17).

Листинг 10.17. Файл interpolate_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 10.18).

Листинг 10.18. Использование фигурных скобок. Файл curly_brackets.php

```
<?php
$arr['one'] = 14;
echo "Событие произошло {$arr['one']} дней назад";
// Событие произошло 14 дней назад
```

10.5. Конструкция *list()*

Если конструкции `array()` и `[]` позволяют создавать массивы, то конструкция `list()` решает обратную задачу — преобразует элементы массива в обычные переменные (листинг 10.19).

Листинг 10.19. Использование конструкции `list()`. 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. В листинге 10.20 осуществляется попытка преобразования элементов ассоциативного массива в переменные `$one`, `$two` и `$three`, которые, однако, остаются неинициализированными.

Листинг 10.20. Файл list_assoc.php

```
<?php
$arr = ['one' => 1, 'two' => 2, 'three' => 3];
list($one, $two, $three) = $arr;
echo $one;   // Notice: Undefined offset
echo $two;   // Notice: Undefined offset
echo $three; // Notice: Undefined offset
```

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 10.21).

Листинг 10.21. 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 10.22).

Листинг 10.22. 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
```

10.6. Обход массива

При работе с массивами часто возникающей задачей является обход их элементов в цикле. В случае индексных массивов для их обхода можно воспользоваться операторами `for` или `while` *(см. главу 9)*, для ассоциативных массивов предназначен специализированный оператор `foreach` *(см. разд. 10.7)*. В листинге 10.23 демонстрируется обход индексного массива при помощи цикла `for`.

Листинг 10.23. 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 10.23 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 10.23 для массива `$number` будет возвращено значение 3.

10.7. 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) {
    операторы;
}
```

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 10.24).

> **Замечание**
>
> В качестве первого аргумента конструкции `foreach` обязательно должен выступать массив, иначе конструкция выводит предупреждение.

Листинг 10.24. 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 листинга 10.24 будут следующие строки:

```
first = 1
second = 2
third = 3
```

Переменная `$index` для ключа массива необязательна и может быть опущена (листинг 10.25).

Листинг 10.25. Вариант обхода массива в цикле `foreach`. Файл foreach_alter.php

```
<?php
$arr = [
    'first' => '1',
    'second' => '2',
    'third' => '3'
];
foreach ($number as $val) {
    echo $val; // выведет 123
}
```

Обход многомерных массивов проводится с помощью вложенных циклов `foreach`, при этом число вложенных циклов соответствует размерности массива (листинг 10.26).

Листинг 10.26. 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 10.26 представлен на рис. 10.1.

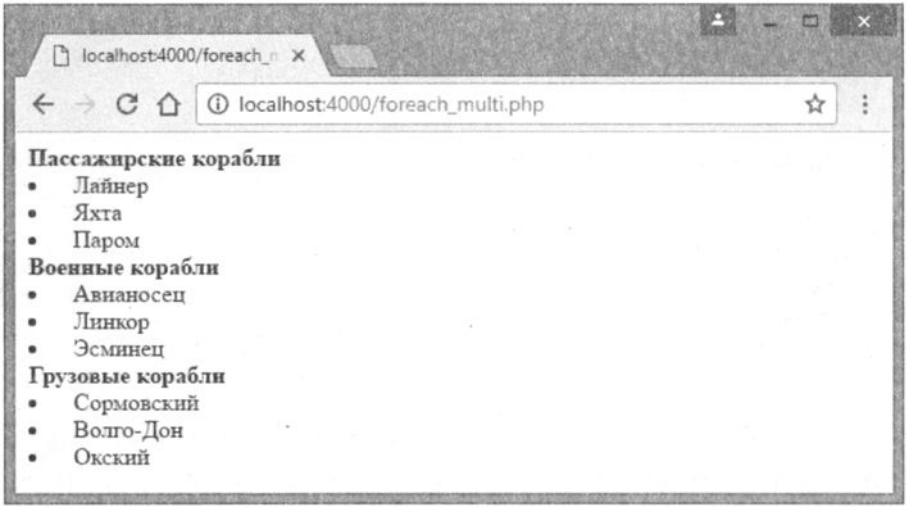


Рис. 10.1. Вывод многомерного массива

10.8. Слияние массивов

Оператор "плюс" (+), позволяющий складывать значения двух переменных, применим и для массивов (листинг 10.27).

> ***Замечание***
>
> Наряду с оператором "плюс" (+) допускается использование оператора +=.

Листинг 10.27. 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 10.27 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 10.28).

Листинг 10.28. Файл plus_alter.php

```
<?php
$fst = ['one', 'two'];
$snd = ['three', 'four', 'five'];
$sum = $fst + $snd;
echo '<pre>';
print_r($sum);
echo '</pre>';
```

В результате выполнения скрипта из листинга 10.28 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 array_merge(array $array1 [, array $... ])
```

В листинге 10.29 приводится пример использования функции `array_merge()`.

> ***Замечание***
>
> Функция `array_merge()` может принимать в качестве параметров более двух массивов.

Листинг 10.29. 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 10.29 в окно браузера будет выведен следующий дамп нового массива `$sum`:

```
Array
(
    [0] => one
    [1] => two
    [2] => three
    [3] => four
    [4] => five
)
```

10.9. Сравнение массивов

Как и любые две переменные, массивы можно сравнивать при помощи операторов равенства == и неравенства != (листинг 10.30), которые более подробно рассматриваются в *главе 7*. Два массива считаются *равными*, если количество и значения их ключей и значений совпадают.

Листинг 10.30. 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 10.30 выводит следующие строки:

```
Массивы равны
Массивы не равны
Массивы не равны
```

Операторы эквивалентности `===` и неэквивалентности `!==` соответственно 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 10.31).

Листинг 10.31. 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 10.31 выводит следующие строки:

```
Массивы равны
Массивы не эквивалентны
```

Интересно отметить, что массивы, в которых используются ключи разного типа, не считаются неэквивалентными (листинг 10.32).

Листинг 10.32. Файл keys_types_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 10.32 выводит следующие строки:

```
Массивы равны
Массивы эквивалентны
```

10.10. Проверка существования элементов массива

Проверить, существует тот или иной элемент массива, можно при помощи конструкции `isset()` (листинг 10.33).

Листинг 10.33. 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 10.33 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 10.34).

Листинг 10.34. 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 10.34 выводит следующие строки:

```
Это массив
Это не массив
```

Функция `in_array()` осуществляет поиск значения в массиве и имеет следующий синтаксис:

```
bool in_array(mixed $value, array $arr [, bool $strict = false ])
```

Функция ищет в массиве `$arr` значение `$value` и возвращает `true`, если значение найдено, и `false` — в противном случае (листинг 10.35).

Листинг 10.35. 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 выведена фраза:

```
Значение 21.5 найдено
```

Найденный элемент массива взят в одиночные кавычки, т. е. относится к строковому типу, в отличие от других элементов этого же массива. 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 10.36).

Листинг 10.36. 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 выведена фраза:

```
Ничего не найдено
```

В этом случае элемент обнаружен не будет, т. к. тип первого аргумента функции in_array (float) отличается от типа элемента в массиве (string). При таком вызове функции она найдет элемент 21.5 только в том случае, 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:

```
bool array_key_exists(mixed $key, array $array)
```

Функция возвращает true, если ключ $key найден в массиве $arr (листинг 10.37).

Листинг 10.37. 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:

```
mixed array_search(mixed $value, array $arr [, bool $strict = false])
```

Функция ищет значение $value в массиве $arr 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 $value будет использоваться оператор эквивалентности ===,

в противном случае будет использован оператор равенства ==. Пример использования функции приводится в листинге 10.38.

Листинг 10.38. Использование функции `array_search()`. Файл array_search.php

```
<?php
$array = [0 => 'blue', 1 => 'red', 2 => 'green', 3 => 'red'];

$key = array_search('green', $array); // $key = 2;
$key = array_search('red', $array);   // $key = 1;
```

10.11. Удаление элементов массива

Удаление элемента массива, как и любой другой переменной, осуществляется при помощи конструкции `unset()` (листинг 10.39).

Листинг 10.39. 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 10.39 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 10.40).

Листинг 10.40. Удаление массива. Файл array_unset.php

```
<?php
$arr = [1, 2, 3, 4, 5];
```

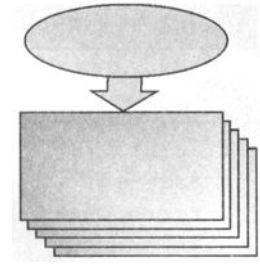
```
// Удаляем третий элемент массива
unset($arr);

// Выводим дамп массива
echo '<pre>';
print_r($arr);
echo '</pre>';
```

Обращение к несуществующему массиву `$arr` вызовет генерацию замечания "Notice: Undefined variable: arr".

Задания

1. Пусть имеется массив `['fst', 'snd', 'thd', 'fth']`. Выведите случайный элемент массива.
2. Пусть имеется массив `['fst' => 1, 'snd' => 2, 'thd' => 3, 'fth' => 4]`. Получите на основании его новый массив с ключами его элементов `['fst', 'snd', 'thd', 'fth']`.
3. Пусть имеется массив `['fst', 'snd', 'thd', 'fth', 'snd', 'thd']`. Получите из него новый массив, содержащий только уникальные элементы `['fst', 'snd', 'thd', 'fth']`.
4. Решите задачу обмена значений двух целочисленных переменных, не прибегая к конструкции `list()` и использованию третьей промежуточной переменной.
5. Создайте массив со случайным количеством элементов от 5 до 10, элементы которого принимают случайное значение от 0 до 100. Отсортируйте элементы массива в порядке от наименьших к наибольшим значениям.
6. Создайте текстовый файл с названиями месяцев. В документации PHP найдите функцию `file()`, изучите приемы ее использования и создайте массив с названиями месяцев из содержимого текстового файла.

ГЛАВА 11



Функции

Листинги данной главы
можно найти в подкаталоге functions.

В предыдущих главах встроенные функции уже неоднократно описывались и применялись для самых разнообразных задач.

Помимо встроенных функций PHP позволяет разработчикам создавать собственные пользовательские функции. Пользовательская функция представляет собой фрагмент программы, предназначенный для реализации определенных действий, выполнение которых достигается вызовом функции в требуемом месте программы. Это позволяет исключить дублирующие фрагменты кода и добиться повторного его использования. Функция может принимать несколько аргументов и, если нужно, возвращает полученное значение.

11.1. Объявление и вызов функции

Функция объявляется при помощи ключевого слова `function`, после которого следует имя функции, в круглых скобках параметры функции и в фигурных скобках записываются различные операторы, составляющие тело функции:

```
function myFunction()
{
  // операторы
}
```

Согласно стандарту кодирования PSR имя функции задается в CamelCase-стиле, однако первая буква задается строчной, чтобы отличать функции от классов.

Если функция возвращает какое-либо значение, в теле функции обязательно должна присутствовать конструкция `return`:

```
function myFunction()
{
  // Вычисления
  return 0; // возвращается значение 0
}
```

Листинг 11.1. Пример простой функции. Файл function.php

```
<?php
function getSum()
{
    $sum = 10 + 5;
    return $sum;
}
echo getSum(); // 15
```

В листинге 11.1 демонстрируется функция, вычисляющая сумму двух чисел. Эта функция не принимает ни одного аргумента, а просто вычисляет сумму и возвращает полученный результат. После этого она вызывается в теле оператора `echo` для вывода результата в браузер.

Модифицируем эту функцию так, чтобы она не возвращала полученный результат, а выводила его в браузер. Для этого достаточно внести оператор `echo` в тело функции (листинг 11.2).

Листинг 11.2. Вывод в браузер внутри функции. Файл function_echo.php

```
<?php
function getSum()
{
    $sum = 10 + 5;
    echo $sum; // 15
}
getSum();
```

Однако подход, который демонстрируется в листинге 11.2, обычно 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 для обработки другими функциями.

В PHP функция может вызываться до ее объявления (листинг 11.3).

Листинг 11.3. Файл function_call.php

```
<?php
echo getSum();
function getSum()
{
    $sum = 10 + 5;
    return $sum;
}
```

Это правило изменяется, если объявление функции осуществляется внутри фигурных скобок. Функции могут быть объявлены в блоке, обрамленном фигурными

скобками. Такой способ объявления функций часто используется, если объявление должно быть условным (листинг 11.4).

Листинг 11.4. Условное объявление функции. Файл function_cond.php

```
<?php
// Объявляем логическую переменную
$flag = true;

// Если переменная $flag равна true, объявляем функцию
if (true) {
    function getSum()
    {
        $sum = 10 + 5;
        return $sum;
    }
}

// Вызываем функцию, если переменная $flag равна true
if ($flag) {
    echo getSum(); // 15
}
```

Результатом выполнения скрипта из листинга 11.4 будет вывод числа 15 в окно браузера. Однако объявить функцию позднее ее вывода в этом случае уже не получится (листинг 11.5).

Листинг 11.5. Файл function_declare_fail.php

```
<?php
// Объявляем логическую переменную
$flag = true;

// Вызываем функцию, если переменная $flag равна true
if ($flag) {
    echo getSum(); // Fatal error
}

// Если переменная $flag равна true, объявляем функцию
if ($flag) {
    function getSum()
    {
        $sum = 10 + 5;
        return $sum;
    }
}
```

Попытка вызова функции, объявленной условно, раньше объявления приводит к генерации ошибки "Fatal error: Uncaught Error: Call to undefined function getSum()."

11.2. Параметры и аргументы функции

Можно значительно увеличить гибкость функции из листинга 11.2, если складываемые числа будут передаваться в качестве параметров (листинг 11.6).

Листинг 11.6. Параметры функции. Файл params.php

```
<?php
function getSum($left, $right)
{
    $sum = $left + $right;
    return $sum;
}
echo getSum(10, 5); // 15
```

Переменные `$left` и `$right`, которые задаются в круглых скобках при объявлении, называются *параметрами* функции. При вызове функции вместо них могут быть подставлены переменные с другими названиями или просто значения, например, в листинге 11.6 это числа 10 и 5. Значения и переменные, передаваемые функции при вызове, называются *аргументами* функции.

В качестве аргументов могут выступать выражения и даже другие функции. В PHP выражения в этом случае вычисляются слева направо (листинг 11.7).

Листинг 11.7. Файл params_dynamic.php

```
<?php
function funct($left, $middle, $right)
{
    echo "$left<br />";
    echo "$middle<br />";
    echo "$right<br>";
}
$i = 10;
funct(++$i, $i = $i * 2, --$i);
```

Результатом выполнения скрипта из листинга 11.7 будет такая последовательность чисел:

```
11
22
21
```

11.3. Типы параметров и возвращаемого значения

При объявлении функции допускается указывать типы параметров. В случае необходимости задания типа возвращаемого значения он указывается через двоеточие непосредственно перед телом функции (листинг 11.8).

Листинг 11.8. Типы параметров и возвращаемого значения. Файл types.php

```
<?php
function getSum(int $fst, int $snd) : int
{
    return $fst + $snd;
}
echo getSum(2, 2); // 4
echo getSum(2.5, 2.5); // 4
```

11.4. Передача параметров по значению и ссылке

В рассмотренных 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 11.9).

Листинг 11.9. 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
```

Для того 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 get_sum(&$var)
```

В случае, если аргумент передается *по ссылке*, при любом изменении значения параметра происходит изменение переменной-аргумента (листинг 11.10).

Замечание

Объекты и массивы передаются в функцию по ссылке, поэтому применительно к таким параметрам можно не применять символ амперсанда `&`.

Листинг 11.10. 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
```

11.5. Необязательные параметры

Параметры можно объявить как необязательные, присвоив им значение по умолчанию (листинг 11.11).

Листинг 11.11. Необязательные параметры. Файл params_default.php

```
<?php
function getSum($left = 10, $right = 5)
{
    $sum = $left + $right;
    return $sum;
}
echo getSum();      // выводит 15
echo getSum(5);     // выводит 10
echo getSum(5, 0); // выводит 5
```

Как видно из листинга 11.11, даже если функции `getSum()` не передаются аргументы, она успешно производит вычисления с участием параметров по умолчанию.

Если функция содержит множество обязательных и необязательных параметров, то все обязательные параметры следует располагать до необязательных. В листинге 11.12 приводится некорректное объявление функции `getSum()`, в котором необязательный параметр предшествует обязательному.

Листинг 11.12. Файл params_fail.php

```
<?php
function getSum($left = 10, $right)
```

```
{
    $sum = $left + $right;
    return $sum;
}
echo getSum(5);
```

Впрочем, функция отрабатывает "успешно" — PHP-интерпретатор ограничивается выводом предупреждения и замечания. Единственный параметр 5 подставляется вместо `$left`, параметр `$right` получает неопределенное значение, которое при сложении приводится к нулю.

11.6. Переменное количество параметров

Для того чтобы создать функцию, которая принимает переменное количество аргументов, перед последним параметром следует указать многоточие. Внутри функции такой параметр рассматривается как массив, содержащий все дополнительные параметры (листинг 11.13).

> ***Замечание***
>
> До версии 5.6 PHP не поддерживал описываемый в разделе механизм обработки переменного количества параметров. Вместо этого использовались функции `func_get_args()`, `func_num_args()` и `func_get_arg()`, подробнее с которыми можно ознакомиться в документации.

Листинг 11.13. Переменное число параметров. Файл varargs.php

```
<?php
function echoList(...$items)
{
    foreach ($items as $v) {
        echo "$v<br />\n"; // выводим элемент
    }
}
// Отображаем строки одну под другой
echoList('PHP', 'Python', 'Ruby', 'JavaScript');
```

Оператор ... может использоваться не только перед аргументами функций, но и при вызове с массивом. Это позволяет осуществить "развертывание" массива. Пусть имеется функция `tooManyArgs()` с большим количеством параметров. Можно поместить значения параметров в массив `$args` и передать его функции, предварив оператором ..., который развернет элементы массива в соответствующие параметры (листинг 11.14).

Листинг 11.14. Использование оператора ... Файл toomanyargs.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 />";
}
// Отображаем строки одну под другой
$items = ['PHP', 'Python', 'Ruby', 'JavaScript'];
tooManyArgs(...$items);
```

11.7. Глобальные переменные

Переменные в функциях имеют локальную область видимости. Это означает, что даже если локальная (внутри функции) и внешняя (в не функции) переменные имеют одинаковые имена, то изменение локальной переменной никак не повлияет на внешнюю переменную (листинг 11.15).

Листинг 11.15. Файл vars_scope.php

```
<?php
function getSum()
{
    $var = 5;       // локальная переменная
    return $var;
}
$var = 10;          // внешняя переменная
echo getSum();      // 5 (локальная переменная)
echo "<br />$var"; // 10 (внешняя переменная)
```

Локальную переменную можно сделать внешней, если перед ее именем указать ключевое слово `global`. В этом случае изменения как внутри функции, так и вне ее будут влиять на переменную, а сама переменная будет называться *глобальной*. Если локальная переменная объявлена как `global`, то к ней возможен доступ из любой части программы (листинг 11.16).

> ***Замечание***
>
> Использовать глобальные переменные следует только в случае острой необходимости. Злоупотребление ими ведет к созданию запутанного, сложно сопровождаемого кода.

Листинг 11.16. Использование глобальной переменной. Файл global.php

```
<?php
function getSum()
{
    global $var;
```

```
    $var = 5; // изменяем глобальную переменную
    return $var;
}
$var = 10;
echo "$var<br />";       // выводит 10
echo getSum().'<br />'; // выводит 5(глобальная переменная изменена)
echo "$var<br />";       // выводит 5
```

11.8. Статические переменные

Временем жизни переменной называется интервал выполнения программы, в течение которого она существует. 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 11.17, всегда будет возвращать значение 1.

Листинг 11.17. 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 11.18).

Листинг 11.18. Файл counter_static.php

```
<?php
function counter()
{
    static $counter = 0;
    return ++$counter;
}
```

В скрипте из листинга 11.18 `$counter` устанавливается в ноль при первом вызове функции, и 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.

11.9. Возврат массива функцией

Функция может возвращать массив в качестве значения, для этого достаточно передать его конструкции `return`. Более того, такой массив может создаваться динамически при помощи конструкции `array()` или `[]`. К этому приему прибегают всякий раз, когда функция должна вернуть несколько значений, а передача значений по ссылке не допускается.

В листинге 11.19 демонстрируется функция `formatSize()`, принимающая в качестве значения размер файла в байтах и возвращающая массив, первый элемент которого содержит размер в байтах, второй — в килобайтах, третий — в мегабайтах, а четвертый — в гигабайтах.

Листинг 11.19. Функция возвращает массив. Файл return_array.php

```
<?php
function formatSize($bytes)
{
    $kbytes = $bytes / 1024;
    $mbytes = $kbytes / 1024;
    $gbytes = $mbytes / 1024;

    return [$bytes, $kbytes, $mbytes, $gbytes];
}
```

Оперировать массивом в скрипте не всегда удобно, особенно если он имеет постоянное небольшое количество элементов. Поэтому часто при вызове функции элементы массива сразу же сопоставляются переменным при помощи конструкции `list()` (листинг 11.20).

Листинг 11.20. Преобразование массива в переменные. Файл list.php

```
<?php
require_once('return_array.php');
list($bytes, $kbytes, $mbytes, $gbytes) = formatSize(18642678);
```

11.10. Рекурсивные функции

Рекурсия — это вызов функции самой себя. Пример рекурсивной функции приведен в листинге 11.21.

Листинг 11.21. Рекурсивная функция. Файл recursion.php

```
<?php
function callself($counter)
{
    // Если параметр $counter больше, продолжаем рекурсивный спуск
    if ($counter > 0) {
        // Уменьшаем значение параметра $counter и выводим его значение
        echo ($counter--) . '<br />';
        // Осуществляем рекурсивный вызов функции callself()
        callself($counter);
    }
    // Если значение параметра меньше или равно 0, прекращаем
    // рекурсивный спуск
    else return;
}
// Вызываем функцию callself()
callself(4);
```

Результатом работы функции будет последовательность цифр:

```
4
3
2
1
```

Функция `callself()` вызывает саму себя до тех пор, пока ее параметр `$counter` положителен и не равен нулю. Рекурсивных функций по возможности стараются избегать, т. к. они относятся к трудным по восприятию конструкциям языка, и отладка их достаточно сложна, особенно когда приходится иметь дело не с простейшей рекурсивной функцией, представленной в листинге 11.21, а со сложной функцией, осуществляющей рекурсивный вызов в нескольких местах.

> ***Замечание***
>
> Опасность использования неотлаженных рекурсивных функций заключается в возможности перехода их в режим бесконечной рекурсии, когда условие, прекращающее спуск вниз по рекурсии, из-за ошибки не наступает, в результате чего, как и в случае бесконечных циклов, наступает зависание программы.

Практически в любом случае можно избежать рекурсивных функций. Исключение составляют задачи, так или иначе связанные с обходом деревьев. К таким задачам относится, например, удаление каталогов, когда число файлов и подкаталогов заранее не известно, и необходимо вызывать функцию удаления до тех пор, пока не будут удалены файлы на самом глубоком уровне вложенности. К таким задачам относится и подсчет вложенных сумм, когда число вложений заранее не известно: $\Sigma\Sigma\Sigma...\Sigma\Sigma\, x_i$.

11.11. 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 осуществлен вызов основной функции, который произведет объявление вложенной (листинг 11.22).

Листинг 11.22. Объявление вложенной функции. Файл nested.php

```
<?php
// Объявление внешней и вложенной функций
function outter()
{
    function inner()
    {
        return "Hello, world!";
    }
}

// Вызываем функцию outter(), чтобы объявить функцию inner()
outter();

// Функция inner() не может быть вызвана до тех пор,
// пока не будет вызвана функция outter();
echo inner();
```

11.12. Динамическое имя функции

По аналогии с переменными, имя функции может быть динамическим и храниться в строковой переменной — передача такой переменной оператором круглых скобок (с параметрами, если они требуются) приводит к вызову функции. В листинге 11.23 демонстрируется вызов функций `hello()` и `bye()` по случайному закону.

Листинг 11.23. Использование динамических имен функций. Файл dynamic.php

```
<?php
// Объявление функций
function hello()
{
    return 'Hello!';
}
function bye()
{
    return 'Bye!';
}
```

```
// Случайный выбор функции
$var = rand(0, 1) ? 'hello' : 'bye';

// Вызов функции
$var();
```

11.13. Анонимные функции

Анонимные функции — это функции без имени. В листинге 11.24 приводится пример объявления такой функции.

Листинг 11.24. Анонимная функция. Файл anonim.php

```
<?php
$echoList = function (...$str)
{
    foreach ($str as $v) {
        echo "$v<br />\n";
    }
};
// Вызов функции
$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 11.25).

Листинг 11.25. Анонимная функция. Файл sort.php

```
<?php
$arr = ['PHP', 'Python', 'Ruby', 'JavaScript'];
sort($arr);
echo '<pre>';
print_r($arr); // JavaScript, PHP, Python, Ruby
```

Однако если мы имеем дело с массивом объектов, корректно отсортировать его функцией `sort()` довольно сложно, т. к. 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 двумерного пространства (листинг 11.26).

Листинг 11.26. 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 значения массива таких объектов, потребуется сравнить их расстояния. В этом случае удобнее воспользоваться функцией `usort()`, которая в качестве второго параметра принимает функцию

`bool usort(array &$array, callable $value_compare_func)`

В качестве такой функции может выступать анонимная функция, определенная непосредственно в списке аргументов (листинг 11.27). 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 11.27. 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);
```

Как видно из листинга, внутри анонимной функции для каждого из объектов вычисляется расстояние `$distance_a` и `$distance_b`, которые затем сравниваются друг

с другом при помощи оператора <=>. Результат выполнения скрипта выглядит следующим образом:

```
Array
(
    [0] => Point Object
        (
            [x] => 1
            [y] => 1
        )
    [1] => Point Object
        (
            [x] => 4
            [y] => 10
        )
    [2] => Point Object
        (
            [x] => 12
            [y] => 5
        )
)
```

11.14. Замыкания

Замыкание — это функция, которая запоминает состояние окружения в момент своего создания. Даже если состояние затем изменяется, замыкание содержит первоначальное состояние. Замыкание в PHP применимо только к анонимным функциям. В отличие от других языков программирования, вроде JavaScript, замыкания не действуют автоматически. Для активизации необходимо использовать ключевое слово use, а за ним в скобках можно указать переменные, которые должны войти в замыкание (листинг 11.28).

Листинг 11.28. Замыкания. 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([]);
// ...
$erorrs[] = 'Заполните имя пользователя';
$check($erorrs);
// ...
$message = 'Список требований'; // Уже не изменить
$erorrs = ['PHP', 'PostgreSQL', 'Redis'];
$check($erorrs);
```

В листинге 11.28 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

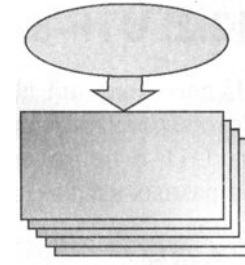
```
Работа не может быть продолжена из-за ошибок:
Заполните имя пользователя
Работа не может быть продолжена из-за ошибок:
PHP
PostgreSQL
Redis
```

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. Самое главное — никакие изменения глобальной переменной в других частях программы не смогут повлиять на значение, переданное через замыкание.

Задания

1. В документации к языку PHP найдите и изучите приемы использования функции `function_exists()`, осуществляющей проверку существования функции.
2. Создайте функцию `odd()`, которая принимает в качестве аргумента целое число и возвращает `true` в случае, если число нечетное, и `false` — в противном случае.
3. Создайте функцию `sum()`, которая принимает любое количество числовых аргументов и возвращает их сумму.

ГЛАВА 12



Строковые функции

Листинги данной главы
можно найти в подкаталоге strings.

Строки и функции их обработки являются одним из главных инструментов в скриптовых языках программирования, особенно если прикладной деятельностью является Web-разработка. Поэтому работе со строковыми переменными в PHP уделяется пристальное внимание.

12.1. Строки как массивы

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 12.1 приводится пример, в котором строка `"PHP"` выводится вертикально.

Листинг 12.1. 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
```

12.2. UTF-8. Расширение mbstring

На сегодняшний день все новые приложения ориентируются на кодировку UTF-8, которая является стандартом де-факто для кодирования всех языков мира. Переход на UTF-8 не только дает возможность использовать в одном документе тексты на разных языках, но и позволяет расширить кодировку при появлении новых символов.

В PHP 7 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 12.2).

Листинг 12.2. Вывод UTF-8 русской буквы А. Файл utf8.php

```
<?php
echo "\u{0410}";
```

В UTF-8 английские символы занимают один байт, а русские — два. Однако движок PHP будет создаваться задолго до повсеместного перехода на UTF-8. В результате, если обратиться к UTF-8 строке, как к массиву символов, используя квадратные скобки, в случае английского языка будет получен символ, а в случае русского — только половина символа (листинг 12.3).

Листинг 12.3. Кодировка ASCII. Файл utf8crash.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Проблемы с обработкой UTF-8 в PHP</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
$str = "Hello, world";
echo "{$str[2]}<br />";
$str = "Привет, мир!";
echo "{$str[2]}<br />";
?>
</body>
</html>
```

Результатом работы сценария будут следующие строки:

```
l
◆
```

Проблеме с поддержкой UTF-8 на уровне языка в PHP уже больше 10 лет. Команда разработчиков предприняла титанические усилия по переработке подсистемы строк

в PHP 7. Однако на момент написания книги проблема все еще не была решена до конца.

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.

В UNIX-подобных операционных системах, будь то Linux или Mac OS X, PHP, как правило, скомпилирован с расширением mbstring. В этом легко убедиться, запросив список расширений при помощи команды `php -m`. Windows-дистрибутив так же поставляется с mbstring-расширением, но для его установки потребуется раскомментировать (убрать точку с запятой) следующие две строки в конфигурационном файле php.ini:

```
extension_dir = "ext"
extension=php_mbstring.dll
```

После этого следует перезагрузить сервер, чтобы он перечитал конфигурационный файл php.ini.

Теперь можно попробовать использовать библиотеку. Подсчитаем количество символов в строке, для этого воспользуемся стандартной строковой функцией `strlen()` и ее многобайтным аналогом `mb_strlen()` (листинг 12.4).

Листинг 12.4. 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 видно, стандартная функция `strlen()` для строки `'Привет, мир!'` вернула 21 байт. При этом 3 байта отводится под запятую, пробел и восклицательный знак, а оставшиеся 18 байт 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 12.4 вернет правильное значение в 12 символов.

Как поступать на практике: переключать директиву или использовать функции расширения mbstring — решать вам.

12.3. Функции для работы с символами

Упоминавшаяся выше функция `strlen()` принимает в качестве единственного аргумента строку *`$string`* и возвращает ее длину.

```
int strlen (string $string)
```

Функция `strlen()` часто используется совместно с другими функциями, а также при работе с посимвольным представлением строки. В листинге 12.5 в цикле `for` в столбик выводится строка `$str`, содержащая слово `"Hello"`. Скрипт основан на том факте, что элемент `$str[0]` содержит символ `H`, `$str[1]` — символ `e`, `$str[2]` — символ `l`, и т. д. до конца строки.

Листинг 12.5. Обход строки в цикле. Файл for.php

```
<?php
$str = 'Hello';
for($i = 0; $i < strlen($str); $i++) {
    echo $str[$i] . '<br />';
}
```

Результатом работы скрипта из листинга 12.5 будет вертикальный вывод слова `"Hello"`:

```
H
e
l
l
o
```

Функция `chr()` принимает в качестве аргумента ASCII-код *`$ascii`* символа и возвращает соответствующий этому коду фактический символ:

```
string chr(int $ascii)
```

В листинге 12.6 приводится пример использования функции для получения символа с кодом 36.

Листинг 12.6. Использование функции `chr()`. Файл chr.php

```
<?php
echo chr(36); // $
```

Функция `ord()` выполняет действие, обратное функции `chr()` — возвращает ASCII-код символа, переданного ей в качестве аргумента (листинг 12.7).

```
int ord(string $str)
```

Если параметр *`$str`* функции `ord()` содержит более одного символа, ASCII-код будет возвращен только для первого.

Листинг 12.7. Использование функции ord(). Файл ord.php

```
<?php
echo ord('$'); // 36
```

12.4. Поиск в строке

Функция `substr()` возвращает часть строки (подстроку) и имеет следующий синтаксис:

```
string substr(string $str, int $start [, int $length ])
```

В качестве первого аргумента *$str* функции передается исходная строка, из которой вырезается текст; второй аргумент *$start* 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.

Листинг 12.8. Извлечение даты, месяца и года из строки. Файл substr.php

```
<?php
$str = '04.05.2017';
echo 'день - ' . substr($str, 0, 2) . '<br />';  // день - 04
echo 'месяц - ' . substr($str, 3, 2) . '<br />'; // месяц - 05
echo 'год - ' . substr($str, 6) . '<br />';      // год - 2017
```

В листинге 12.8 приводится пример использования функции `substr()` — из строки `"04.05.2017"` извлекается дата, месяц и год.

Функция `strpos()` возвращает позицию вхождения подстроки в строку и имеет следующий синтаксис:

```
mixed strpos(string $str, mixed $search [, int $offset = 0 ])
```

Функция возвращает позицию в строке *$str* первого вхождения подстроки *$search*. Необязательный параметр *$offset* позволяет задать позицию, начиная с которой будет осуществляться поиск. Если строка не найдена, возвращается `false`. В листинге 12.9 приводится простейший пример использования функции `strpos()`.

> ***Замечание***
>
> Функция `strpos()` при поиске учитывает регистр. Для того чтобы поиск осуществлялся без его учета, следует воспользоваться функцией `strpos()`.

Листинг 12.9. Использование функции strpos(). Файл strpos.php

```
<?php
echo strpos('Hello, world!', 'world'); // 7
```

Функции `strpos()` редко используется сама по себе, чаще в комбинации с другими строковыми функциями (листинг 12.10).

Листинг 12.10. Файл strpos_substr.php

```
<?php
$str = 'PHP - интерпретируемый язык';
echo substr($str, strpos($str, 'интер')); // интерпретируемый язык
```

Скрипт в листинге 12.10 ищет позицию, с которой начинается подстрока "интер", и выводит в окно браузера строку, начиная с этой позиции до конца строки.

12.5. Замена в тексте

Функции замены осуществляют преобразования строк, связанные с заменой одних подстрок другими, а также удалением подстрок.

Наиболее часто используемой функцией является `str_replace()`, которая позволяет заменить подстроку в тексте другой подстрокой и имеет следующий синтаксис:

```
mixed str_replace(
  mixed $search,
  mixed $replace,
  mixed $str
  [, int &$count ])
```

Функция заменяет в строке *$str* все вхождения подстроки *$search* на *$replace* и возвращает результат замены. Одной из распространенных задач является замена тегов форматирования в стиле bbCode их HTML-эквивалентами (листинг 12.11).

Листинг 12.11. 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 12.11 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 `<b>` и `</b>` (листинг 12.12).

Листинг 12.12. 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()` вместо параметров *`$search`* на *`$replace`* может принимать массивы с равным количеством элементов. В строке элемент массива *`$search`* заменяется соответствующим элементом массива *`$replace`*. Скрипт в листинге 12.13 аналогичен по результату скрипту из листинга 12.12.

Листинг 12.13. 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 было осуществлено в строке, 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 12.14).

Листинг 12.14. 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 12.14 будут следующие строки:

```
Это очень жирный текст.
Осуществлено замен: 4
```

Функция `trim()` удаляет символы из начала и конца строки и имеет следующий синтаксис:

```
string trim(string $str [, string $charlist = " \t\n\r\0\x0B" ])
```

Функция принимает в качестве аргумента *`$str`* строку и удаляет из нее ведущие и конечные пробельные символы, к которым относят:

- `" "` (ASCII 32 (0x20)), символ пробела;
- `"\t"` (ASCII 9 (0x09)), символ табуляции;
- `"\n"` (ASCII 10 (0x0A)), символ перевода строки;
- `"\r"` (ASCII 13 (0x0D)), символ возврата каретки;
- `"\0"` (ASCII 0 (0x00)), NUL-байт;
- `"\x0B"` (ASCII 11 (0x0B)), вертикальная табуляция.

Зачастую данных символов бывает недостаточно, в этом случае во втором необязательном параметре *$charlist* приводится список символов, которые необходимо считать пробельными.

Для демонстрации работы функции создадим скрипт, в котором будет контролироваться длина строки до и после удаления из нее пробельных символов (листинг 12.15).

Листинг 12.15. 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
```

12.6. Работа с HTML-кодом

Так как PHP задумывался как язык для Web-проектирования, среди строковых функций имеется несколько специально предназначенных для обработки языка разметки HTML.

Браузеры интерпретируют переводы строк `\n` как обычный пробельный символ. Поэтому для вывода данных с новой строки применяется специальный HTML-тег `<br />`, который неоднократно использовался в предыдущих главах.

Для решения задачи замены обычных переводов строк в HTML-эквивалент PHP предоставляет специальную функцию `nl2br()`, которая имеет следующий синтаксис:

```
string nl2br(string $str [, bool $is_xhtml = true])
```

Для перевода строк в HTML используется тег `<br>` или `<br />` в зависимости от того, принимает второй параметр функции *$is_xhtml* значение `false` или `true`.

Листинг 12.16. Использование функции `nl2br()`. Файл nl2br.php

```
<?php
$str = <<<text
hello
php
text;
```

```
echo $str;
echo '<br /><br />';
echo nl2br($str);
```

Результатом работы скрипта из листинга 12.16 будут следующие строки:

```
hello php

hello
php
```

Особенность функции `nl2br()` в том, что тег `<br />` вставляется рядом с символом перевода строки, а не заменяет его собой. Такое поведение функции `nl2br()` может быть не удобным, если текст предназначен для подстановки в JavaScript-скрипт. 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.

> ***Замечание***
>
> Всегда применяйте эту функцию при обработке текстовых сообщений из формы, так как в этом случае вы будете надежно защищены от злоумышленников.

Функция `htmlspecialchars()` имеет следующий синтаксис:

```
string htmlspecialchars (
  string $str [,
  int $quote_style = ENT_COMPAT | ENT_HTML401 [,
  string $charset = ini_get("default_charset") [,
  bool $double_encode = true ]]])
```

Первый аргумент *`$str`* — строка, в которой требуется выполнить преобразование. Второй необязательный аргумент *`$quote_style`* определяет режим обработки двойных и одиночных кавычек и может принимать одну из констант:

- `ENT_COMPAT` — режим по умолчанию, в этом режиме двойные кавычки заменяются символом `&quot;`, а одиночные кавычки остаются без изменений;
- `ENT_QUOTES` — в этом режиме преобразуются и двойные, и одиночные кавычки, последние заменяются символом `&#039;`;
- `ENT_NOQUOTES` — в данном режиме двойные и одиночные кавычки остаются без изменений.

Третий необязательный аргумент *`$charset`* принимает строку с названием кодировки, которую можно задать при помощи директивы `default_charset` конфигурационного файла php.ini.

Последний аргумент *$double_encode* позволяет управлять режимом повторного кодирования HTML-тегов. По умолчанию повторное кодирование включено, поэтому если в строке *$str* встречается последовательность `&`, являющаяся HTML-представлением амперсанда, она превратиться в `&amp;`, еще одно преобразование `htmlspecialchars()` превратит эту последовательность в `&amp;amp;`. Для того чтобы предотвратить такие преобразования при повторных вызовах, следует установить значение *$double_encode* в `false`.

В листинге 12.17 приведен код HTML-формы для ввода сообщений (index.html).

Листинг 12.17. Файл index.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Проблемы с обработкой HTML-форм</title>
  <meta charset='utf-8'>
</head>
<body>
  <form action='handler.php' method='post'>
  Сообщение:<br />
  <textarea cols='50' rows='5' name='msg'></textarea><br />
  <input type='submit' value='Добавить' />
  </form>
</body>
</html>
```

HTML-форма, представленная в листинге 12.17, имеет текстовую область `msg`, кнопку **Добавить**, после нажатия на которую данные отправляются в обработчик handler.php (листинг 12.18).

Листинг 12.18. Код обработчика HTML-формы. Файл handler.php

```
<?php
echo htmlspecialchars($_POST['msg']);
```

Суперглобальный массив `$_POST` содержит данные, отправленные скрипту из формы HTTP-методом POST, и более подробно рассматривается в *главе 13*.

Теперь добавим в форму для ввода сообщения вместо безобидного текста JavaScript-код (рис. 12.1):

```
<script language="JavaScript">
alert("Приветик!");
</script>
```

С JavaScript-кодом форма должна выглядеть так, как это представлено на рис. 12.1.

Рис. 12.1. HTML-форма с введенным JavaScript-кодом

Так как при обработке сообщения использовалась функция `htmlspecialchars()`, ничего страшного не произойдет, обработчик просто выведет текст скрипта, который был набран (рис. 12.2).

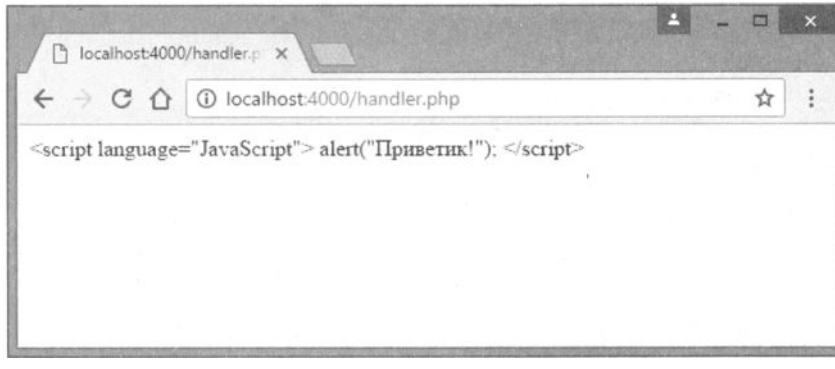


Рис. 12.2. Результат обработки введенного текста при помощи функции `htmlspecialchars()`

Если же введенный пользователем текст не обрабатывать функцией `htmlspecialchars()`, то вместо текста скрипта будет выведен результат его выполнения (рис. 12.3).

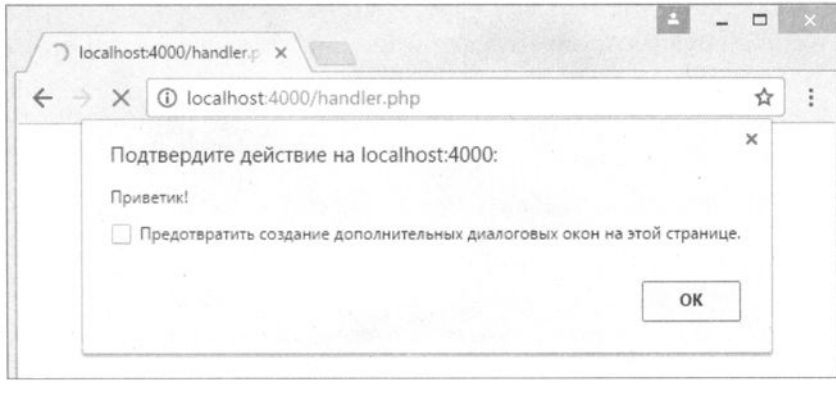


Рис. 12.3. Вот что получается без использования функции `htmlspecialchars()`

Злоумышленник может поместить редирект на страницу гостевой книги или форума, а в случае удачи похитить пароли.

В ряде случаев удобнее вообще удалить HTML-теги из текста. Для этого предназначена специальная функция `strip_tags()`, которая имеет следующий синтаксис:

```
string strip_tags(string $str [, string $allowable_tags])
```

Функция удаляет из строки *`$str`* HTML-теги кроме тех, которые указываются в параметре *`$allowable_tags`*.

> ***Замечание***
>
> Особенность функции заключается в том, что теги не удаляются, а заменяются пробельными символами.

В листинге 12.19 приводится пример использования функции `strip_tags()`. Для того чтобы был виден результат, перед выводом в окно браузера он пропускается через функцию `htmlspecialchars()`.

Листинг 12.19. Использование функции `strip_tags()`. Файл strip_tags.php

```
<?php
$str = '<p>Параграф.</p>
        <!-- Comment -->
        Еще немного текста';
echo htmlspecialchars(strip_tags($str));
echo '<br />';
echo htmlspecialchars(strip_tags($str, '<p>'));
```

Результатом работы скрипта из листинга 12.19 будут следующие строки:

```
Параграф. Еще немного текста
<p>Параграф.</p> Еще немного текста
```

12.7. Форматный вывод

Функции данной группы осуществляют вывод информации в окно браузера и ее форматирование.

Для форматного вывода предназначены функции семейства `printf()`. Функция `printf()` имеет следующий синтаксис:

```
int printf(string $format [, mixed $args [, mixed $... ]])
```

> ***Замечание***
>
> Функция `printf()` отличается от `sprintf()` тем, что первая функция выводит результат непосредственно в окно браузера, а вторая возвращает строку, которую можно сохранить в переменной. Функция `fprintf()` результат помещает в открытый файл, дескриптор которого передается ей в качестве первого параметра.

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 12.20, определитель преобразования `%d` подставляет в строку число, которое передается в качестве второго аргумента функции.

Листинг 12.20. 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.), поэтому называется *определителем типа*. В табл. 12.1 представлены определители типа, которые допускаются в строке формата функции `printf()`.

***Таблица 12.1.** 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 данной нотации записывается как `1.2e+03`, а 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

В листинге 12.21 демонстрируется форматный вывод числа 1024 с использованием разнообразных определителей типа.

Листинг 12.21. 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 12.22).

Листинг 12.22. 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 12.23 под вывод числа 45 отводится пять символов, поскольку само число занимает лишь два символа, три ведущих символа будут содержать символ заполнения.

Листинг 12.23. Форматирование числового вывода. Файл printf_int.php

```
<?php
echo '<pre>';
printf('% 4d\n', 45); // '   45'
printf('%04d\n', 45); // '00045'
echo '</pre>';
```

Применение определителя типа `f` позволяет вывести число в десятичном формате. 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 12.24).

Листинг 12.24. 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.

12.8. Объединение и разбиение строк

Функция `explode()` предназначена для разбивки строки по определенному разделителю и имеет следующий синтаксис:

```
array explode(
  string $delimiter,
  string $str
  [, int $limit = PHP_INT_MAX ])
```

Функция возвращает массив из строк, каждая из которых соответствует фрагменту исходной строки *$str*, находящемуся между разделителями, определяемыми параметром *$delimiter*.

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 12.25.

Листинг 12.25. Использование функции explode(). Файл explode.php

```
<?php
$str = 'Имя, Фамилия, e-mail';
echo '<pre>';
print_r(explode(', ', $str)); // ['Имя', 'Фамилия', 'e-mail']
```

Функция `implode()` является обратной `explode()`-функцией и осуществляет объединение элементов массива в строку. Функция имеет следующий синтаксис:

```
string implode(string $delimiter, array $arr)
```

Функция возвращает строку, которая содержит элементы массива, заданного в параметре *$arr*, между которыми вставляется значение, указанное в параметре *$delimiter*. Пример использования функции приводится в листинге 12.26.

Листинг 12.26. 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, т. к. слишком длинное слово или предложение может нарушить дизайн страницы. Для этого предназначена функция `wordwrap()`, которая осуществляет перенос на заданное количество символов с использованием символа разрыва строки. Функция имеет следующий синтаксис:

```
string wordwrap(
  string $str
  [, int $width = 75
  [, string $break = "\n"
  [, bool $cut = false ]]])
```

Функция разбивает блок текста *$str* на несколько строк, которые завершаются символами *$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 12.27).

> ***Замечание***
>
> Проблему авторазбиения слов можно решить на стороне браузера, воспользовавшись CSS-свойством `word-wrap`.

Листинг 12.27. 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* 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, если при уменьшении размера окна слово перестает помещаться целиком в строке.

12.9. Сериализация объектов и массивов

Пара функций `serialize()` и `unserialize()` позволяет осуществлять упаковку и распаковку массивов и объектов.

Синтаксис функции `serialize()` таков:

```
string serialize(mixed $value)
```

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.

```
mixed unserialize(string $str [, array $options])
```

В листинге 12.28 демонстрируется использование обеих функций.

Листинг 12.28. 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);
```

Результатом работы скрипта из листинга 12.28 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
)
```

12.10. JSON-формат

Процедура сериализации, описанная в предыдущем разделе, имеет несколько недостатков. При попытке сериализовать уже сериализованные данные, восстановить строку при помощи функции `unserialize()` может не получиться. Кроме того, для

восстановления сериализованных данных в других языках программирования может потребоваться воссоздание собственной функции `unserialize()`. Последнее может быть весьма утомительно, т. к. 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 приобрел большую популярность среди Web-разработчиков. Формат представляет собой объект JavaScript. Одним из достоинств данного формата является его легкое восприятие человеком. Формат позволяет задавать строковые, числовые значения, 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.

```
string json_encode(mixed $value, int $options = 0, int $depth = 512)
```

Функция преобразует переменную `$value` в JSON-последовательность. Параметр `$value` может принимать любой тип за исключением `resource`. Следующий параметр `$options` представляет собой битовую маску из флагов, приведенных в табл. 12.2. Последний параметр `$depth` задает максимальную глубину генерируемого JSON-объекта.

Таблица 12.2. *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

Таблица 12.2 (окончание)

Константа	Действие
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 как \u*XXXX*)

В листинге 12.29 приводится пример использования функции `json_encode()`.

Листинг 12.29. Файл json_encode.php

```
<?php
$arr = [
    'employee' => 'Иван Иванов',
    'phones' => [
        '916 153 2854',
        '916 643 8420'
    ]
];
echo json_encode($arr);
```

Результат работы скрипта из приведенного выше листинга 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 \u*XXXX*.

```
{"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 *$options*, который может принимать значения из табл. 9.2.

Нам потребуется константа `JSON_UNESCAPED_UNICODE` для того, чтобы предотвратить кодирование UTF-8, который и так прекрасно обрабатывается всеми остальными языками, включая JavaScript (листинг 12.30).

Листинг 12.30. 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 7*.

```
echo json_encode($arr, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
```

Для преобразования JSON-строки в массив PHP предназначена функция `json_decode()`:

```
mixed json_decode (
 string $json,
 bool $assoc = false,
 int $depth = 512,
 int $options = 0)
```

Функция принимает параметр *$json* с JSON-строкой и возвращает ассоциативный массив PHP, если значение параметра *$assoc* 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 12.31 приводится пример использования функции `json_decode()`, которая переводит JSON-строку, полученную в предыдущем примере, в массив.

Листинг 12.31. 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>';
```

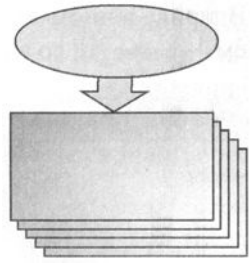
Результат выполнения скрипта из листинга 12.31 выглядит следующим образом:

```
Array
(
    [employee] => Иван Иванов
    [phones] => Array
        (
            [0] => 916 153 2854
            [1] => 916 643 8420
        )
)
```

Задания

1. В документации с сайта **http://php.net** изучите синтаксис функций `hightlight_string()` и `hightlight_file()`, преобразующих PHP-код в HTML-код с подсветкой синтаксиса. Создайте скрипт, который выводит собственный код с подсветкой синтаксиса PHP.
2. Создайте класс двумерной декартовой точки `Point`, объявите объект с координатами (1, 1), сохраните сериализованный объект в файле с именем point.txt. При помощи другого скрипта извлеките и восстановите объект из файла point.txt.
3. Создайте скрипт, преобразующий число в арабской нотации (от 1 до 2000) в римское. Арабские числа соотносятся с римскими следующим образом: 1 — I, 5 — V, 10 — X, 50 — L, 100 — C, 500 — D, 1000 — M. Например, 116 — это CXVI, 199 — это CXCIX, 14 — это XIV.

ГЛАВА 13



Взаимодействие PHP с HTML

Листинги данной главы
можно найти в подкаталоге html.

PHP выполняется на сервере, а HTML интерпретируется браузером на стороне клиента. При работе с Web-приложением посетитель переходит по ссылкам, заполняет HTML-формы и отправляет результаты на сервер, что вызывает выполнение PHP-скриптов.

Текущая глава посвящена тому, как PHP взаимодействует с HTML. В ней будут рассмотрены два метода протокола HTTP: GET — передача параметров в строке запроса и POST — передача параметров в теле HTTP-документа. Остальные способы взаимодействия будут освещены в *главе 14*, посвященной суперглобальным массивам PHP.

13.1. Передача параметров методом GET

GET-параметры передаются в строке запроса после символа вопроса (`?`).

http://localhost:4000/get.php?forum_id=1

В приведенном выше 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 13.1 выводится значение GET-параметра `forum_id`.

Листинг 13.1. 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 приведенном выше URL передаются три GET-параметра: `forum_id` со значением 1, `theme_id` со значением 2 и `post_id` со значением 10 (листинг 13.2).

Листинг 13.2. Файл get_array.php

```
<?php
echo '<pre>';
print_r($_GET);
echo '</pre>';
```

В результате выполнения скрипта из листинга 13.2 может быть выведен следующий дамп массива `$_GET`:

```
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 13.2 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. Для запроса

http://localhost:4000/get_array.php?id[a]=1&id[b]=2&id[c]=10

скрипт из листинга 13.2 выведет следующий дамп:

```
Array
(
    [id] => Array
        (
            [a] => 1
            [b] => 2
            [c] => 10
        )
)
```

GET-параметры и их значения могут содержать недопустимые с точки зрения URL символы (пробелы, русские символы), которые при формировании URL 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:

```
string urlencode(string $str)
```

В листинге 13.3 формируется ссылка на файл test.php, через GET-параметр `phrase` которому передается фраза "Привет, мир!".

Листинг 13.3. Использование функции `urlencode()`. Файл urlencode.php

```
<?php
echo "<a href='test.php?phrase=".
              urlencode("Привет, мир!")."'>ссылка</a>";
```

Исходный код результирующей HTML-страницы будет содержать следующую строку:

```
<a href='test.php?phrase=%CF%F0%E8%E2%E5%F2%2C+%EC%E8%F0%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:

```
mixed parse_url(string $url [, int $component = -1 ])
```

Функция принимает в качестве первого параметра строку запроса и возвращает отдельные его 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 13.4 приводится пример разбора URL при помощи функции `parse_url()`.

Листинг 13.4. 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 скрипта из листинга 13.4 будет следующий дамп массива `$arr`:

```
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 13.5 из строки запроса при помощи функции `parse_url()` и константы `PHP_URL_HOST` извлекается лишь доменное имя.

Листинг 13.5. 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
```

13.2. HTML-форма и ее обработчик

HTML-формы создаются при помощи парных тегов `<form>` и `</form>`, между которыми располагаются теги элементов управления. В листинге 13.6 представлен HTML-код формы, содержащей два элемента управления с однострочной текстовой областью `text` и кнопку подтверждения `submit`.

Листинг 13.6. 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 HTML-кода из листинга 13.6 будет простейшая HTML-форма, представленная на рис. 13.1.

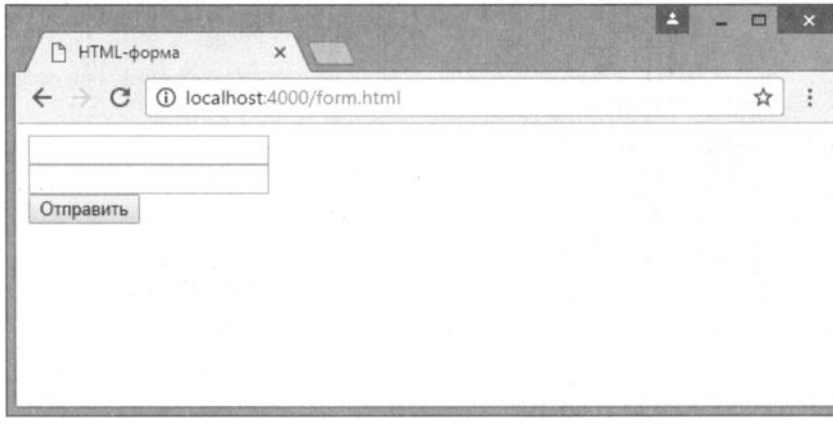


Рис. 13.1. HTML-форма в окне браузера

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. 13.1 представлены основные атрибуты, позволяющие управлять поведением HTML-формы.

Как видно из табл. 13.1, адрес обработчика указывается в атрибуте `action`. Различают два подхода к созданию обработчика HTML-формы:

- обработчик расположен в отдельном файле, после выполнения манипуляций над полученными данными клиент направляется на главную страницу;
- обработчик располагается в том же самом файле, где находится HTML-форма.

Таблица 13.1. 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
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, 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; данный 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

Рассмотрим каждый из случаев более подробно. В листинге 13.7 приводится пример HTML-формы, расположенной в файле form_first.php и содержащей единственное текстовое поле first и кнопку.

Листинг 13.7. 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 отправляются обработчику, расположенному в файле form_first_handler.php, который осуществляет вывод в окно браузера введенной в поле `first` строки (листинг 13.8).

Листинг 13.8. 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']);
}
```

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.

> ***Замечание***
>
> 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 возвращаться обратно и заполнять HTML-форму заново, что может оказаться очень утомительным, если HTML-форма содержит множество обязательных полей.

Выходом из данной ситуации является расположение обработчика непосредственно в файле HTML-формы (листинг 13.9).

Листинг 13.9. 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 массив `$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`.

13.3. 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 тегами `<form>` и `</form>` тег такого вида:

```
<input type='text' />
```

Следует отметить, 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 интерпретирует элемент управления как текстовое поле.

Помимо атрибута `type` тег `<input>` может содержать дополнительные атрибуты, представленные в табл. 13.2.

Таблица 13.2. 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, т. к. 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

13.4. Поле для приема пароля

Для ввода пароля обычно используют не текстовое поле, а специальное поле типа `password`, которое совпадает по атрибутам и внешнему виду с текстовым полем `text`, однако вводимый текст остается скрытым за символами звездочек или точек (листинг 13.10).

Листинг 13.10. 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 13.10 представлен на рис. 13.2.

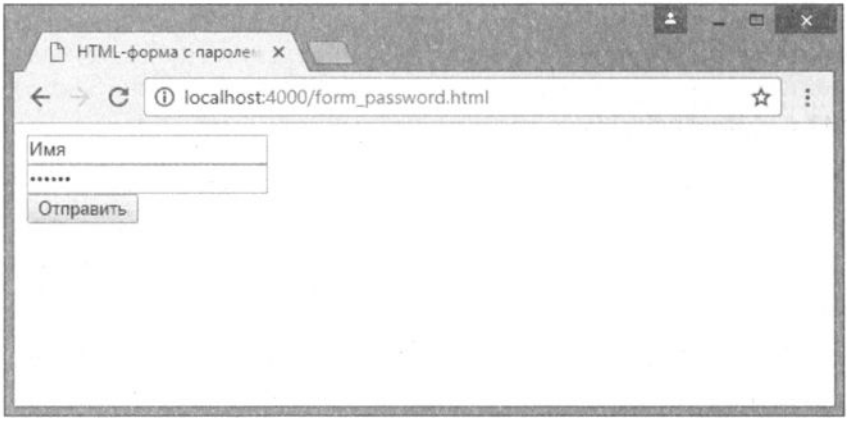


Рис. 13.2. Использование поля для приема пароля

13.5. Текстовая область

Текстовая область предназначена для ввода нескольких строк текста и создается при помощи парного тега `<textarea>`. В отличие от текстового поля `<input>` в текстовой области допускается создание переводов строк (абзацев). Тег `<textarea>` имеет следующий синтаксис:

```
<textarea>...</textarea>
```

Как можно заметить, текстовая область определяется не с помощью атрибута, а посредством собственного тега. Допустимые атрибуты тега `<textarea>` приводятся в табл. 13.3.

***Таблица 13.3.** 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
`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 при нажатии клавиши <Enter>, в противном случае должна появляться горизонтальная полоса прокрутки

В листинге 13.11 приводится пример использования текстовой области для приема большого объема текста.

Листинг 13.11. 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 13.11 представлен на рис. 13.3.

Рис. 13.3. Использование текстовой области

13.6. Скрытое поле

Скрытое поле служит для передачи незаметно от пользователя служебной информации. Скрытые поля не отображаются на странице.

Замечание

Дело в том, что 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 (session), однако такой подход не всегда удобен, т. к. 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 13.11 так, чтобы она включала скрытое поле id, заполняемое из GET-параметра (листинг 13.12).

Листинг 13.12. 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']); ?>">
  </form>
</body>
</html>
```

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 13.13).

Листинг 13.13. Файл form_hidden_handler.php

```
<?php
echo '<pre>';
print_r($_POST);
echo '</pre>';
```

Результат выполнения обработчика form_hidden_handler.php из листинга 13.13 может выглядеть следующим образом:

```
Array
(
    [name] => 123
    [id] => 1
)
```

13.7. Флажок

Флажок — это элемент управления, позволяющий представлять логическое значение в HTML-форме, находясь в установленном или снятом состоянии. Синтаксис 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 пять флажков, два из которых являются предустановленными (отмеченными) (листинг 13.14).

Листинг 13.14. 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='mysql' checked>Вы знакомы с MySQL?<br />
    <input type='checkbox' name='perl'>Вы знакомы с Perl?<br />
    <input type='checkbox' name='phyton'>Вы знакомы с Python?<br />
    <input type='checkbox' name='ruby'>Вы знакомы с Ruby?<br />
    <input type='submit' value='Отправить'>
  </form>
</body>
</html>
```

Внешний вид HTML-формы из листинга 13.14 представлен на рис. 13.4.

Интересно отметить, что 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 13.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [php] => on
    [mysql] => on
)
```

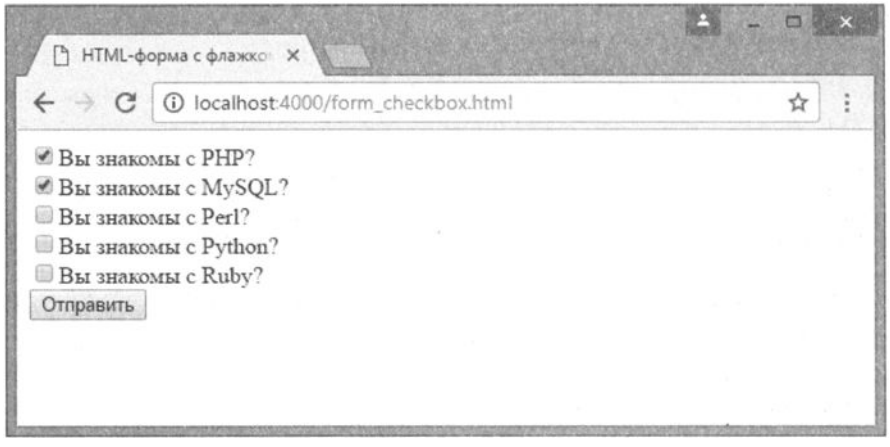


Рис. 13.4. Использование флажков

По умолчанию в качестве значения для флажка выступает строка `"on"`, однако это значение можно изменить, если снабдить флажок атрибутом `value` (листинг 13.15).

Листинг 13.15. 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='mysql' value='2' checked='checked'>
    Вы знакомы с MySQL?<br />
    <input type='checkbox' name='perl' value='3'>
    Вы знакомы с Perl?<br />
    <input type='checkbox' name='phyton' value='4'>
    Вы знакомы с Python?<br />
    <input type='checkbox' name='ruby' value='5'>
    Вы знакомы с Ruby?<br />
    <input type='submit' value='Отправить'>
  </form>
</body>
</html>
```

Результат отправки данных из HTML-формы из листинга 13.15 может выглядеть следующим образом:

```
Array
(
    [php] => 1
    [mysql] => 2
)
```

13.8. 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 выше примере список имеет три пункта.

Помимо традиционного атрибута `name`, тег `<select>` может иметь атрибуты `multiple` и `size`. Параметр `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 теге `<option>`. Помимо этого, тег `<option>` может иметь атрибут `selected` для обозначения выделения текущего пункта.

В листинге 13.16 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.

> ***Замечание***
>
> Если используется множественный список, для корректной передачи всех выбранных значений в качестве названия элемента `<select>` должен выступать массив.

Листинг 13.16. 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 13.16 представлен на рис. 13.5.

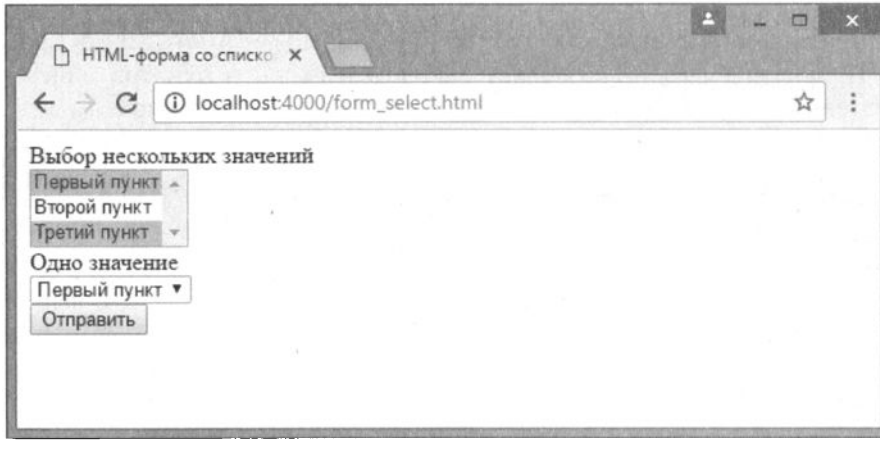


Рис. 13.5. Множественный и одиночный выпадающие списки

Если в качестве обработчика используется скрипт из листинга 13.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [fst] => Array
        (
            [0] => 1
            [1] => 3
        )

    [snd] => 1
)
```

13.9. Переключатель

Переключатель, или, иначе, 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` для того, чтобы отметить один из переключателей по умолчанию.

В листинге 13.17 приводится пример использования переключателей для оценки по пятибалльной системе.

Листинг 13.17. 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 13.17 представлен на рис. 13.6.

Если в качестве обработчика используется скрипт из листинга 13.13, то результатом отправки данных обработчику будет вывод следующего дампа массива:

```
Array
(
    [mark] => 3
)
```

Рис. 13.6. Переключатели

13.10. Загрузка файла на сервер

Для загрузки пользовательских файлов на сервер применяется специальный элемент управления, позволяющий указать путь к загружаемому файлу (при помощи кнопки **Обзор**). Элемент управления имеет следующий синтаксис:

```
<input type='file' />
```

Помимо атрибута `type`, элемент управления допускает указания атрибутов `name` и `size`. Простая форма для отправки файла на сервер демонстрируется в листинге 13.18.

Листинг 13.18. Файл form_file.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Загрузка файлов на сервер</title>
  <meta charset='utf-8'>
</head>
<body>
  <h2><b> Форма для загрузки файлов </b></h2>
  <form action='upload.php' method='post' enctype='multipart/form-data'>
    <input type='file' name='filename'><br />
    <input type='submit' value='Загрузить'>
  </form>
</body>
</html>
```

Атрибут `entype` формы определяет вид кодировки, которую браузер применяет к параметрам формы. Для того чтобы отправка файлов на сервер действовала,

атрибуту `entype` необходимо присвоить значение `multipart/form-data`. По умолчанию этот атрибут имеет значение `application/x-www-form-urlencoded`.

Если все сделано правильно, форма для отправки файлов на сервер должна выглядеть так, как это показано на рис. 13.7.

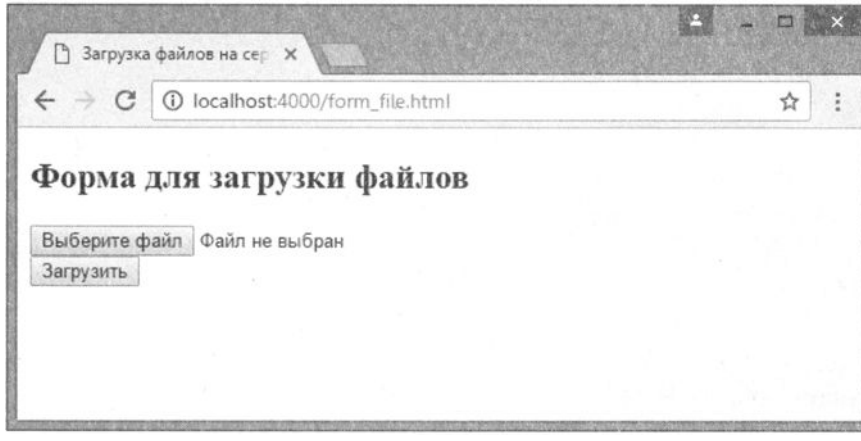


Рис. 13.7. Загрузка файла на сервер

После того как получен HTTP-запрос, содержимое загруженного файла записывается во временный файл, который создается в каталоге сервера, заданном по умолчанию для временных файлов, если другой каталог не задан в файле php.ini (директива `upload_tmp_dir`).

Характеристики загруженного файла доступны через двумерный суперглобальный массив `$_FILES`. При этом переменная со значениями этого массива может иметь следующий вид:

- `$_FILES['filename']['name']` (содержит исходное имя файла на клиентской машине);
- `$_FILES['filename']['size']` (содержит размер загруженного файла в байтах);
- `$_FILES['filename']['type']` (содержит MIME-тип файла);
- `$_FILES['filename']['tmp_name']` (содержит имя временного файла, в который сохраняется загруженный файл).

В листинге 13.19 приведен скрипт upload.php, который загружает файл на сервер и копирует его из временного каталога в каталог temp.

Листинг 13.19. Файл upload.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Результат загрузки файла</title>
  <meta charset='utf-8'>
</head>
```

```
<body>
<?php
if (move_uploaded_file($_FILES['filename']['tmp_name'],
                       'temp/'.$_FILES['filename']['name'])) {
    echo 'Файл успешно загружен';
} else {
   echo 'Ошибка загрузки файла';
}
?>
</body>
</html>
```

Проверить успешность загрузки файла на сервер можно при помощи специальной функции `is_uploaded_file()`, которая принимает в качестве единственного параметра имя файла (`$_FILES['filename']['name']`) и возвращает `true` в случае успешной загрузки и `false` в случае неудачи.

Переместить файл можно при помощи функции `move_uploaded_file()`, которая имеет следующий синтаксис:

```
bool move_uploaded_file(string $filename, string $destination)
```

В качестве первого параметра *`$filename`* функция принимает путь к загруженному файлу, в качестве второго параметра *`$destination`* — путь, куда файл должен быть перемещен.

После выполнения этого скрипта выбранный для загрузки файл будет помещен в подкаталог temp каталога, в котором расположен скрипт, а браузер выдаст фразу "Файл успешно загружен".

Скрипт из листинга 13.20 позволяет вывести характеристики загруженного файла.

Листинг 13.20. Файл upload_params.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Результат загрузки файла</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
if (move_uploaded_file($_FILES['filename']['tmp_name'],
                       'temp/'.$_FILES['filename']['name']))
{
    echo 'Файл успешно загружен <br />';
    // далее выводится информация о файле
    echo 'Характеристики файла: <br />';
    echo 'Имя файла: ';
    echo $_FILES['filename']['name'];
```

```
    echo '<br />Размер файла: ';
    echo $_FILES['filename']['size'];
    echo '<br />Каталог для загрузки: ';
    echo $_FILES['filename']['tmp_name'];
    echo '<br />Тип файла: ';
    echo $_FILES['filename']['type'];
} else {
    echo 'Ошибка загрузки файла';
}
?>
</body>
</html>
```

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 13.21.

Листинг 13.21. Ограничение размера файла. Файл upload_limit.php

```
<?php
if ($_FILES['filename']['size'] > 3*1024*1024) {
    exit 'Размер файла превышает три мегабайта';
}
if (move_uploaded_file($_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`, значение которой по умолчанию равно 2 Мбайт:

```
if ($_FILES['filename']['size'] > upload_max_filesize)
```

> ***Замечание***
>
> Значение директивы `upload_max_filesize` можно изменить в конфигурационном файле php.ini.

13.11. Переадресация

Часто после обработки формы пользователя необходимо перенаправить на другую страницу сайта. Так как страницы открывает браузер, запущенный на компьютере пользователя, а Web-приложение расположено на сервере, надо сообщить браузеру

о необходимости открыть другую страницу. Такой сигнал посылается средствами протокола HTTP, по которому общаются браузер и Web-сервер. Поэтому придется немного в него погрузиться.

Общение по протоколу HTTP сводится к обмену HTTP-документов 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. 13.8).

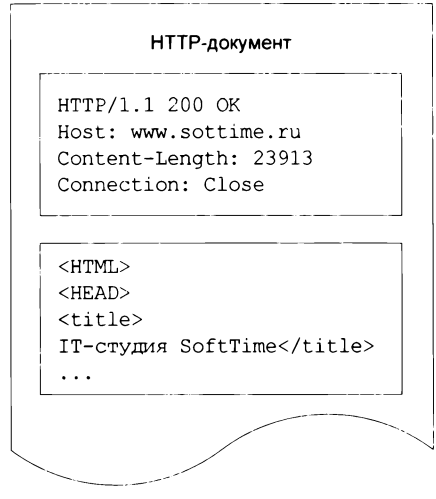


Рис. 13.8. HTTP-документ состоит из HTTP-заголовков и тела документа

HTTP-заголовки, как правило, формируются сервером автоматически и в большинстве случаев Web-разработчику нет надобности отправлять их вручную.

Браузер, получая HTTP-заголовки, автоматически выполняет предписания, даже не показывая посетителю их содержимое. Таким образом, 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, кэширование, 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:

```
void header(
  string $header
  [, bool $replace = true
  [, int $http_response_code ]]
)
```

Первый параметр *`$header`* содержит строку с HTTP-заголовком. Второй параметр *`$replace`* определяет поведение интерпретатора PHP, если тот встречает два одинаковых заголовка: если параметр принимает значение `true`, отправляется последний

заголовок, в противном случае отправляется первый заголовок. Третий параметр *$http_response_code* позволяет задать код HTTP-статуса.

Для того чтобы осуществить переадресацию клиента с одной страницы на другую, браузеру необходимо отправить HTTP-заголовок `location`. В листинге 13.22 представлена переадресация на главную страницу сайта **http://php.net**.

> ***Замечание***
>
> Вместо полного сетевого адреса (начинающегося с префикса http://) можно использовать относительные адреса: в этом случае браузер сам подставит адрес текущего сайта.

Листинг 13.22. Файл location.php

```
<?php
header('location: http://php.net');
```

Пусть имеется HTML-форма (листинг 13.23), которая в текстовом поле принимает имя пользователя, а после отправки его на сервер осуществляет редирект на страницу с приветствием.

Листинг 13.23. 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 13.23, при помощи конструкции `require()` в начало скрипта вставляется обработчик формы form_greets_handler.php, содержимое которого представлено в листинге 13.24.

Листинг 13.24. 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. В противном случае в тело HTTP-документа попадает форма из form_greets.php. В небольшой промежуток времени между получением ответа от сервера и редиректом на другую страницу форма 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 13.25), которая выводит содержимое GET-параметра, предварительно обработав его функцией `htmlspecialchars()`.

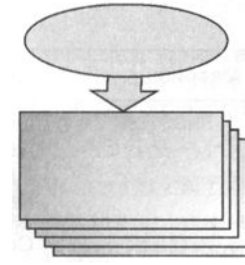
Листинг 13.25. Файл greets.php

```
<?php
echo 'Привет, ' . htmlspecialchars($_GET['name']) . '!';
```

Задания

1. Создайте скрипт, который через HTML-форму принимал бы координаты двух точек в декартовой системе координат, а после нажатия на кнопку типа `submit` выводил бы расстояние между двумя точками.
2. Создайте форму, содержащую текстовую область `textarea` и кнопку `submit`. При нажатии на кнопку содержимое тестовой области должно сохраняться в файл content.txt. При повторной загрузке формы в другом окне содержимое файла content.txt должно подставляться в тестовую форму.
3. Создайте скрипт, который читал бы содержимое текстового файла list.txt и выводил бы его содержимое в HTML-форме со списком флажков перед каждой из строк. После выбора флажков и нажатия на кнопку `submit` содержимое файла list.txt необходимо переписать таким образом, чтобы выбранные строки были исключены.

ГЛАВА 14



Суперглобальные массивы

Листинги данной главы
можно найти в подкаталоге superglobals.

Как уже упоминалось в конце предыдущей главы, 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 Web-сервером и PHP.

14.1. Типы суперглобальных массивов

В предыдущей главе 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, их полный список представлен в табл. 14.1.

***Таблица 14.1.** Суперглобальные массивы PHP*

Массив	Описание
`$_GET`	Содержит GET-параметры, т. е. данные, переданные через параметры строки запроса
`$_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 *(см. разд. 14.2)*

Таблица 14.1 (окончание)

Массив	Описание
$_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 *(см. разд. 14.3)*
$_REQUEST	Содержит все параметры, переданные скрипту методами POST, GET, а также через Cookie
$_ENV	Содержит все переменные окружения, переданные скрипту Web-сервером или командной оболочкой *(см. разд. 14.4)*
$_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

В оставшихся разделах главы не рассмотренные ранее суперглобальные массивы будут освещены более подробно.

14.2. Cookie

HTTP-протокол, лежащий в основе Интернета, не сохраняет информации о состоянии сеанса. Это означает, что любое обращение клиента сервер воспринимает как обращение нового клиента, даже если клиент формирует запрос для загрузки картинок с текущей страницы.

Данная 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, форме, 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 данный 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. 14.1).

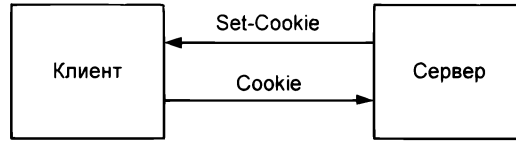


Рис. 14.1. Обмен HTTP-заголовками с Cookie между клиентом и сервером

> **Замечание**
>
> Дословно cookie переводится как "кекс" или сладкий бонус, выдаваемый клиентам ресторана, чтобы они запомнили его и посетили вторично. Из-за достаточно сумбурного английского названия для cookie так и не было подобранного адекватного русского перевода.

PHP по возможности скрывает от разработчика процесс отправки и анализа HTTP-заголовков. Поэтому для установки cookie достаточно воспользоваться функцией `setcookie()`, которая имеет следующий синтаксис:

```
bool setcookie(
  string $name
  [, string $value = ""
  [, int $expire = 0
  [, string $path = ""
  [, string $domain = ""
  [, bool $secure = false
  [, bool $httponly = false ]]]]]])
```

Функция принимает следующие аргументы:

- `$name` — имя cookie;
- `$value` — значение, хранящееся в cookie с именем `$name`;
- `$expire` — время в секундах, прошедших 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, следовательно, такой cookie невозможно похитить через XSS-атаку.

Функция `setcookie()` возвращает `true` при успешной установке cookie и `false` — в противном случае. После того как cookie установлен, его значение можно получить на всех страницах Web-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 функциями `echo()`, `print()` и т. п., а также до включения в файл HTML-тегов. Работа с cookie без установки времени жизни продемонстрирована в листинге 14.1.

> **Замечание**
>
> 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 14.1. 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 14.1, 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 `$_COOKIE['counter']` не установлено, функция возвращает значение 1, при последующих обращениях, когда посетитель присылает значение cookie с каждым запросом, `$_COOKIE['counter']` будет возвращать присланное значение, которое будет увеличено на единицу.

14.3. Сессии

Сессия во многом походит на cookie, данные в сессии так же хранятся в виде пар "ключ/значение", однако уже не на стороне клиента, а на сервере. Во многих случаях сессии являются более предпочтительным вариантом, чем cookies. Для каждого из посетителей сохраняется собственный набор данных. Изменение данных одного посетителя никак не отражается на данных другого посетителя.

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 on-line и т. п.). Поэтому использовать тот или иной механизм следует в зависимости от поставленных целей.

> ***Замечание***
>
> Помимо традиционного хранения сессий на жестком диске в папке, сессии допускают альтернативные механизмы хранения. В *главе 30* 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 удобным для отладки Web-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.

> ***Замечание***
>
> 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 14.2).

Листинг 14.2. 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 14.3).

> ***Замечание***
>
> Массив `$_SESSION` перед сохранением в файл подвергается сериализации, поэтому хранить сериализованные данные в сессии крайне не рекомендуется, т. к. массивы, два раза подряд подвергающиеся действию функции `serialize()`, зачастую восстановлению не подлежат.

Листинг 14.3. Файл session_get.php

```
<?php
session_start();

echo '<pre>';
print_r($_SESSION);
echo '</pre>';
```

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:

```
bool session_destroy(void)
```

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'])
```

14.4. 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 можно при помощи команды `echo`, добавив знак доллара перед именем переменной:

```
$ echo $HELLO
world
```

Для того чтобы получить значение переменной окружения внутри PHP-скрипта, следует обратиться к суперглобальному массиву `$_ENV`. Однако для того чтобы зна-

чение 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 примере выше 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 2)* значение директивы variables_order можно установить при старте сервера при помощи параметра -d:

```
$ HELLO=world php -d variables_order=EGPCS -S localhost:4000
```

После того как переменная окружения передана серверу, к ней можно обратиться из PHP-скрипта (листинг 14.4).

Листинг 14.4. Файл env.php

```
<?php
echo $_ENV['HELLO']; // world
```

В современной Web-разработке переменные окружения интенсивно используются. Web-приложение может поддерживать несколько окружений: режим разработки, тестовое окружение для автоматических тестов, демосцены 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 держать в голове, 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, не безопасно, с другой — не удобно, т. к. потребует развертывания приложения в случае их смены или приведет к конфликтам при использовании индивидуальных паролей разными разработчиками. В этом случае так же часто прибегают к использованию переменных окружения.

14.5. 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.

14.5.1. Элемент $_SERVER['DOCUMENT_ROOT']

Элемент $_SERVER['DOCUMENT_ROOT'] содержит путь к корневому каталогу сервера. Если скрипт выполняется в виртуальном хосте, в данном 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 компактное (листинг 14.5).

Листинг 14.5. Файл document_root.php

```
<?php
echo $_SERVER['DOCUMENT_ROOT'];
echo '<br />';
echo __DIR__;
```

14.5.2. 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 данного 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 * используется для группирования типов в медиаряд. К примеру, символами */* задается использование всех типов, а обозначение type/* определяет использование всех подтипов выбранного типа type.

> ***Замечание***
>
> 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.

> ***Замечание***
>
> 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 данном заголовке первым идет тип `audio/*`, включающий в себя все музыкальные документы и характеризующийся коэффициентом предпочтения `0.2`. Через запятую указан тип `audio/basic`, для которого коэффициент предпочтения не указан и принимает значение по умолчанию, равное единице. Данный 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 данный запрос, предпочтет `text/x-dvi`, а если и его нет, то он может принять тип `text/plain`.

14.5.3. Элемент *$_SERVER['HTTP_ACCEPT_LANGUAGE']*

В элементе `$_SERVER['HTTP_ACCEPT_LANGUAGE']` описываются предпочтения клиента относительно языка. Данная информация извлекается из HTTP-заголовка `Accept-Language`, который присылает клиент серверу. Можно привести следующий пример:

```
Accept-Language: ru, en; q=0.7
```

Его можно интерпретировать следующим образом: клиент предпочитает русский язык, но в случае его отсутствия согласен принимать документы на английском. Элемент `$_SERVER['HTTP_ACCEPT_LANGUAGE']` будет содержать точно такую же информацию, но без заголовка `Accept-Language`:

```
ru, en; q=0.7
```

Содержимое элемента `$_SERVER['HTTP_ACCEPT_LANGUAGE']` можно использовать для определения национальной принадлежности посетителей. Однако результаты будут приблизительными, т. к. многие пользователи используют английские варианты браузеров, которые будут извещать сервер о том, что посетитель предпочитает лишь один язык — английский.

14.5.4. Элемент $_SERVER['HTTP_HOST']

В элементе `$_SERVER['HTTP_HOST']` содержится имя сервера, которое, как правило, совпадает с доменным именем сайта, расположенного на сервере. Как правило, имя, указанное в данном параметре, совпадает с именем `$_SERVER['SERVER_NAME']`.

14.5.5. Элемент $_SERVER['HTTP_REFERER']

В элемент `$_SERVER['HTTP_REFERER']` помещается адрес, с которого посетитель пришел на данную страницу. Переход должен осуществляться по ссылке. Создадим две страницы: page.php (листинг 14.6) и referer.php (листинг 14.7).

Листинг 14.6. Страница page.php

```
<?php
echo "<a href='referer.php'>Ссылка на страницу PHP</a><br />";
if(isset($_SERVER['HTTP_REFERER'])) {
    echo $_SERVER['HTTP_REFERER'];
}
```

Листинг 14.7. Страница 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.

14.5.6. 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 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36"`. Наличие подстроки `"Chrome/57.0.2987.133"` говорит о том, что посетитель просматривает страницу при помощи браузера Chrome версии 57.0. Строка `"Windows NT 6.3"` сообщает, что в качестве операционной системы используется Windows 8.

14.5.7. 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 `$_SERBER['HTTP_X_FORWARDED_FOR']`.

> ***Замечание***
>
> 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 различают так называемые *анонимные прокси-серверы*, которые позволяют скрывать истинный IP-адрес клиента. Такие серверы не возвращают переменную окружения `HTTP_X_FORWARDED_FOR`.

14.5.8. 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"`.

14.5.9. Элемент $_SERVER['SERVER_NAME']

В элемент `$_SERVER['SERVER_NAME']` помещается имя сервера, как правило, совпадающее с доменным именем сайта, расположенного на нем. Например,

```
www.softtime.ru
```

Содержимое элемента `$_SERVER['SERVER_NAME']` часто совпадает с содержимым элемента `$_SERVER['HTTP_HOST']`. Помимо имени сервера суперглобальный массив `$_SERVER` позволяет выяснить еще ряд параметров сервера, например, прослушиваемый порт, тип Web-сервера и версию HTTP-протокола. Эта информация помещается в элементы `$_SERVER['SERVER_PORT']`, `$_SERVER['SERVER_SOFTWARE']` и `$_SERVER['SERVER_PROTOCOL']`, соответственно. В листинге 14.8 приводится пример с использованием данных элементов.

Листинг 14.8. Информация о сервере. Файл server.php

```
<?php
echo "Имя сервера - {$_SERVER['SERVER_NAME']}<br />";
echo "Порт сервера - {$_SERVER['SERVER_PORT']}<br />";
echo "Web-сервер - {$_SERVER['SERVER_SOFTWARE']}<br />";
echo "Версия HTTP-протокола - {$_SERVER['SERVER_PROTOCOL']}<br />";
```

Результат работы скрипта из листинга 14.8 может выглядеть следующим образом:

```
Имя сервера - localhost
Порт сервера - 4000
Web-сервер - PHP 7.0.0 Development Server
Версия HTTP-протокола - HTTP/1.1
```

14.5.10. Элемент $_SERVER['REQUEST_METHOD']

В элемент `$_SERVER['REQUEST_METHOD']` помещается метод запроса, который применяется для вызова скрипта: GET или POST (листинг 14.9).

Листинг 14.9. Определение метода запроса. Файл request_method.php

```
<?php
echo $_SERVER['REQUEST_METHOD']; // GET
```

14.5.11. 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 14.10, помещая в строке запроса произвольный текст после знака `?`, получим страницу с введенным текстом.

Листинг 14.10. Файл query_string.php

```
<?php
if(isset($_SERVER['QUERY_STRING'])) {
    echo urldecode($_SERVER['QUERY_STRING']);
}
```

Результат работы скрипта из листинга 14.10 представлен на рис. 14.2.

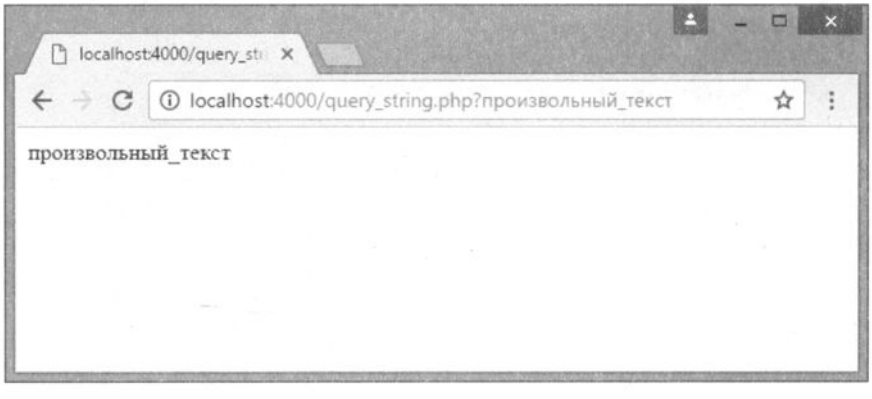


Рис. 14.2. Результат работы скрипта из листинга 14.9

14.5.12. Элемент $_SERVER['PHP_SELF']

В элемент `$_SERVER['PHP_SELF']` помещается имя скрипта, начиная от корневого каталога виртуального хоста, т. е. 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']`.

14.5.13. Элемент $_SERVER['REQUEST_URI']

Элемент `$_SERVER['REQUEST_URI']` содержит имя скрипта, начиная от корневого каталога виртуального хоста, и параметры, т. е. если строка запроса представляет собой адрес:

`http://www.mysite.ru/test/index.php?id=1&test=wet&theme_id=512`

то элемент `$_SERVER['REQUEST_URI']` будет содержать фрагмент `"/test/index.php?id=1&test=wet&theme_id=512"`. Для того чтобы восстановить в скрипте полный адрес, который помещен в строке запроса, достаточно использовать комбинацию элементов массива `$_SERVER`, представленную в листинге 14.11.

Листинг 14.11. Полный адрес к скрипту. Файл fulladdress.php

```
<?php
echo "http://".$_SERVER['SERVER_NAME'].$_SERVER['REQUEST_URI'];
```

Задания

1. Создайте скрипт, который бы запрашивал у пользователя его имя и фамилию. Добейтесь, чтобы при каждом обращении к страницам сайта выводилось приветствие пользователя в зависимости от текущего времени суток:

   С 5 до 11 часов: Доброе утро, ...!

   С 11 до 16 часов: Добрый день, ...!

   С 16 до 0 часов: Добрый вечер, ...!

   С 0 до 5 часов: Доброй ночи, ...!

   Вместо многоточия скрипт должен подставлять имя и фамилию пользователя.

2. Создайте скрипт, который бы отслеживал состояние переменной окружения `ENVIRONMENT` и создавал бы одноименную константу, если переменная окружения установлена.

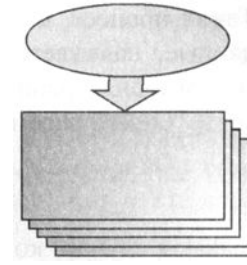
   В случае если переменная окружения пустая, результатом работы скрипта должна быть строка "Режим разработки", если значение переменной имеет в своем составе подстроку "test" — "Режим тестирования", если значение `ENVIRONMENT` принимает значение "production" — "Режим эксплуатации".

3. Создайте скрипт, который бы определял и выводил следующие сведения о текущем посетителе: название браузера, его версию, название операционной системы.

4. Создайте скрипт, который бы запрещал посещение сайта пользователям браузера Internet Explorer.

5. Создайте скрипт, который бы складировал в файл ips.txt уникальные IP-адреса посетителей. Для каждого из IP-адресов следует сохранять количество посещений. Таким образом, при первом посещении в файле ips.txt появляется новая запись, а при последующих посещениях увеличивается счетчик этой записи.

ГЛАВА 15



Фильтрация и проверка данных

Листинги данной главы
можно найти в подкаталоге filters.

Язык PHP специализируется на создании Web-приложений, поэтому к нему предъявляются повышенные требования в области безопасности.

Одна из главных угроз для Web-приложения исходит от пользовательского ввода (ошибочного или злонамеренного). В предыдущих изданиях довольно много внимания уделялось специальному мини-языку — *регулярным выражениям*, при помощи которых контролировался ввод пользователя. Этот отдельный и довольно сложный язык хотя и полезен в разработке приложений, все-таки не обязателен к изучению. Современный PHP предоставляет функции фильтрации и проверки, при помощи которых можно проконтролировать пользовательский ввод меньшими усилиями.

15.1. 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.

В качестве примера фильтрации можно привести типичный способ обработки GET-параметра, например, целочисленного идентификатора пользователя `id`.

http://example.com?id=52561

Любой пользователь, тем не менее, может попробовать передать вместо числа строку:

http://example.com?id=hello

Для того чтобы гарантировать, что `$_GET['id']` содержит число, а не строку, мы можем явно преобразовать ее 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 12*.

```
echo htmlspecialchars('Тег <p> служит для создания параграфа');
echo strip_tag('<p>Hello, world!</p>');
```

Бывают случаи, когда важно сохранить любой пользовательский ввод, например, чтобы можно было воспроизвести последовательность действий конкретного пользователя. 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.

```
mixed filter_var(
  mixed $variable [,
  int $filter = FILTER_DEFAULT [,
  mixed $options]])
```

Функция принимает значение `$variable` и фильтрует его в соответствии с правилом `$filter`. Необязательный массив `$options` позволяет задать дополнительные параметры, которые будут рассмотрены далее. Если переменная `$variable` успешно проходит проверку, функция возвращает ее в качестве результата, в противном случае возвращается `false`.

В листинге 15.1 приводится пример проверки двух электронных адресов: корректного `$correct` и некорректного `$wrong`.

> ***Замечание***
>
> На момент написания книги функция `filter_var()` некорректно обрабатывала электронные адреса с русскими доменными именами.

Листинг 15.1. 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 видно, 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_VALIDATE_EMAIL` (листинг 15.2).

Листинг 15.2. Фильтрация электронного адреса. Файл email_sanitize.php

```
<?php
$correct = 'igorsimdyanov@gmail.com';
$wrong   = 'igorsimdyanov@//gmail.com';
echo filter_var($email_correct, FILTER_SANITIZE_EMAIL) . '<br />';
echo filter_var($email_wrong, FILTER_SANITIZE_EMAIL) . '<br />';
```

Результатом работы скрипта будут следующие строки:

```
igorsimdyanov@gmail.com
igorsimdyanov@gmail.com
```

Как видно, некорректные символы были удалены из электронного адреса.

15.2. Фильтры проверки

Электронный адрес — далеко не единственный тип данных, которые может проверять и фильтровать функция `filter_var()`. В табл. 15.1 представлен полный список фильтров, которые могут быть использованы для проверки данных.

***Таблица 15.1.** Фильтры проверки данных*

Фильтр	Описание
`FILTER_VALIDATE_BOOLEAN`	Возвращает `true` для значений "1", "true", "on" и "yes", иначе возвращается `false`
`FILTER_VALIDATE_EMAIL`	Возвращает `true`, если значение является корректным электронным адресом, иначе возвращает `false`
`FILTER_VALIDATE_FLOAT`	Возвращает `true`, если значение является корректным числом с плавающей точкой, иначе возвращает `false`
`FILTER_VALIDATE_INT`	Возвращает `true`, если значение является корректным целым числом и при необходимости входит в диапазон от `min_range` до `max_range`, значения которых задаются третьим параметром функции `filter_var()`

Таблица 15.1 (окончание)

Фильтр	Описание
FILTER_VALIDATE_IP	Возвращает true, если значение является корректным IP-адресом, иначе возвращает false
FILTER_VALIDATE_REGEXP	Возвращает true, если значение соответствует регулярному выражению, которое задается параметром regexp в дополнительных параметрах функции filter_var()
FILTER_VALIDATE_URL	Возвращает true, если значение соответствует корректному URL, иначе возвращает false

Как видно из табл. 15.1, функция `filter_var()` позволяет проверять довольно разнородные данные (листинг 15.3).

Листинг 15.3. Проверка данных. Файл filter_var.php

```
<?php
echo filter_var('yes', FILTER_VALIDATE_BOOLEAN).'<br />';
echo filter_var('3.14', FILTER_VALIDATE_FLOAT).'<br />';
echo filter_var('https://github.com', FILTER_VALIDATE_URL);
```

Результатом работы скрипта из листинга 15.3 будут следующие строки:

```
1
3.14
https://github.com
```

Третий параметр функции `filter_var()` позволяет передать флаги, изменяющие режим работы функции. Например, при использовании `FILTER_VALIDATE_BOOLEAN` можно в качестве третьего параметра передать значение флага `FILTER_NULL_ON_FAILURE`, в результате функция будет возвращать `true` для значений "1", "true", "on" и "yes", `false` для значений "0", "false", "off", "no" и "". Для всех остальных значений будет возвращаться `null` (листинг 15.4).

Листинг 15.4. Использование FILTER_NULL_ON_FAILURE. Файл boolean_validate.php

```
<?php
$arr = [
    filter_var('yes', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE),
    filter_var('no', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE),
    filter_var('Hello', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)
];
echo '<pre>';
var_dump($arr);
echo '</pre>';
```

Результатом работы скрипта из листинга 15.4 будут следующие строки:

```
array(3) {
  [0]=> bool(true)
  [1]=> bool(false)
  [2]=> NULL
}
```

Еще одним фильтром, допускающим задание дополнительных параметров, является `FILTER_VALIDATE_INT`, который 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 задаются в виде ассоциативного массива в третьем параметре функции `filter_var()` (листинг 15.5).

Листинг 15.5. 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
```

Каждый из фильтров, представленных в табл. 15.1, 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 15.6).

Листинг 15.6. 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
```

15.3. Фильтры очистки

Помимо фильтров проверки (validate), PHP предоставляет большой набор фильтров очистки данных (sanitize), которые представлены в табл. 15.2. Напомним, что очистка осуществляется, как правило, уже при выводе данных на страницу.

***Таблица 15.2.** Фильтры очистки данных*

Фильтр	Описание
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_MAGIC_QUOTES	К строке применяется функция 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 15.7).

Листинг 15.7. Фильтрация URL-адреса. Файл encoded_sanitize.php

```
<?php
echo filter_var('params=Привет мир!', 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
```

Фильтр FILTER_SANITIZE_MAGIC_QUOTES экранирует одиночные ' и двойные кавычки ", обратный слеш \ и нулевой байт \0 (листинг 15.8).

Листинг 15.8. Экранирование. Файл magic_quotes_sanitize.php

```
<?php
$arr = [
    'Deb\'s files',
    'Symbol \\',
    'print "Hello, world!"'
];
echo '<pre>';
print_r($arr);
$result = filter_var_array($arr, FILTER_SANITIZE_MAGIC_QUOTES);
print_r($result);
```

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 15.9).

Листинг 15.9. 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 15.10).

Листинг 15.10. Функция `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 тысячей;
- `FILTER_FLAG_ALLOW_SCIENTIFIC` — разрешает экспоненциальную запись числа *(см. главу 4)*.

Фильтр `FILTER_SANITIZE_FULL_SPECIAL_CHARS` эквивалентен обработке значения функцией `htmlspecialchars()` *(см. главу 12)*. В листинге 15.11 приводится пример использования фильтра.

Листинг 15.11. Обработка текста. Файл special_chars_sanitize.php

```
<?php
$str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
echo '<pre>';
echo filter_var($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>
<p>Первый параграф, посвященный "проверке"</p>
```

15.4. Пользовательская фильтрация данных

До текущего момента мы рассматривали фильтры, которые предоставляются расширением. Однако функция `filter_var()` допускает создание собственных механизмов проверки. Для активации этого режима в качестве фильтра следует использовать `FILTER_CALLBACK`. В параметрах `'options'` следует передать имя функции, которая будет выполнять фильтрацию.

В листинге 15.12 приводится пример, который очищает строку от HTML-тегов при помощи функции `strip_tags()` *(см. главу 12)*.

Листинг 15.12. Пользовательская фильтрация данных. Файл callback_sanitize.php

```
<?php
function filterTags($value) {
    return strip_tags($value);
}
$str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
echo '<pre>';
echo filter_var($str, FILTER_CALLBACK, ['options' => 'filterTags']);
```

Результатом работы программы будут строки, в которых удалены все теги:

```
Заголовок
Первый параграф, посвященный "проверке"
```

Необязательно определять отдельную функцию обратного вызова, особенно если она больше нигде не будет использоваться. В примере выше вместо функции обратного вызова `filterTags()` можно воспользоваться анонимной функцией (листинг 15.13).

Листинг 15.13. Использование анонимной функции. Файл anonim_sanitize.php

```
<?php
$str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
echo '<pre>';
echo filter_var(
    $str,
    FILTER_CALLBACK,
    [
        'options' => function ($value) {
                        return strip_tags($value);
                    }
    ]
);
```

15.5. Фильтрация внешних данных

Напомним, что 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 Web-сервером;
- `$_ENV` — переменные окружения, установленные процессу Web-сервером.

Помимо этого списка, приходится иметь дело с дополнительными суперглобальными массивами:

- `$_FILES` — загруженные на сервер файлы;
- `$_SESSION` — данные сессии;
- `$_REQUEST` — объединение всех перечисленных выше 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:

```
mixed filter_input(
  int $type,
  string $variable_name [,
  int $filter = FILTER_DEFAULT [,
  mixed $options]])
```

Функция принимает в качестве первого параметра *`$type`* одну из констант `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, `INPUT_ENV`, соответствующих суперглобальным массивам из первого списка. В качестве второго значения *`$variable_name`* указывается ключ параметра в суперглобальном массиве, который нужно либо проверить на соответствие условиям, либо отфильтровать. Третьим параметром *`$filter`* указывается одна из констант, представленных в табл. 15.1 или 15.2. В качестве четвертого параметра *`$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 15.14).

Листинг 15.14. 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`, при помощи кода из листинга 15.15.

Перезагрузка страницы будет приводить к тому, что поле `search` будет снова заполняться отправленными данными через атрибут `value`. Чуть пониже формы будет выводиться отфильтрованный текст, содержащий не менее трех символов.

Листинг 15.15. 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-х символов
            $value = (strlen($value) >= 3) ? $value : '';
            // Удаляем теги
            return strip_tags($value);
        }
    ]
);
```

Как видно из листинга 15.15, для решения задачи даже не потребовалось использовать суперглобальный массив `$_POST` и проверять на существование элементы этого массива.

Задания

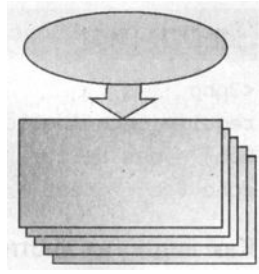
1. Создайте форму регистрации, которая бы сохраняла имя, фамилию и электронный адрес пользователя в файл. Для каждого пользователя в файле должна отводиться одна строка. Не допускается появление дублирующих записей.

   Все поля формы обязательны для заполнения и должны быть корректны. Имя и фамилия должны быть набраны на русском языке и содержать не менее двух и не более 20 символов. В случае некорректного заполнения элементов управления формы или если такой пользователь уже зарегистрирован, необходимо выводить предупреждающие сообщения.

2. Создайте HTML-форму, состоящую из двух текстовых полей, в первом из которых вводится количество товарных позиций, а во втором их цена в формате `###.##`. Обработчик формы должен проверить, является ли введенная в первом поле информация целым числом, а во втором — удовлетворяющим денежному формату. Если все верно, необходимо вывести произведение этих двух чисел.

3. Познакомьтесь с регулярными выражениями и набором preg-функций PHP. Используя регулярные выражения, извлеките из HTML-содержимого страницы **http://php.net** текст, расположенный между тегами `<title>` и `</title>`.

4. Создайте регулярное выражение, которое заменит в тексте все символы точки (.) многоточием (. . .), только в том случае, если точка не стоит в конце сокращений "г.", "рис." и "табл.".



ГЛАВА 16

Методы

Листинги данной главы
можно найти в подкаталоге methods.

В предыдущих главах были рассмотрены встроенные функции языка PHP, а также создание собственных функций. В *главе 5* были введены классы и объекты. Одним из самых полезных свойств классов является возможность объявления внутри них функций, которые могут выполнять операции над данными объекта, вызывать другие функции объекта. Такие функции называются *методами*. Именно им будет посвящена текущая глава.

16.1. Определение метода

Классы и их экземпляры — объекты, представляют собой не просто контейнеры для хранения переменных (в качестве таких контейнеров в PHP могут выступать ассоциативные массивы). Помимо переменных в классы можно включать методы, которые представляют собой обычные функции PHP.

В листинге 16.1 приводится пример класса `Hello`, в состав которого входит метод `greet()`, возвращающий сообщение "Hello, world!".

Листинг 16.1. Объявление метода класса. Файл hello.php

```
<?php
class Hello
{
    public function greet()
    {
        return 'Hello, world!';
    }
}
```

В листинге 16.2 приводится пример использования метода `greet()`.

Листинг 16.2. Использование метода. Файл hello_use.php

```
<?php
require_once 'hello.php'
$obj = new Hello;
echo $obj->greet(); // Hello, world!
```

Как видно из листинга 16.2, обратиться к методу можно, как и в случае переменных, при помощи оператора `->`.

16.2. Обращение к переменным объекта

Для того чтобы обратиться к переменным или другим методам внутри метода, используется специальная переменная `$this`, после которой следуют оператор `->` и название переменной или класса.

Создадим класс `Point`, который моделирует точку двухмерной декартовой системы координат. Для этого внутри класса объявим две закрытые *(см. главу 5)* 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 `$obj->x` к ним уже не получится. Потребуется способ для того, чтобы назначить им значения или прочитать их. Для этого можно ввести набор методов-аксессоров:

- `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`.

> ***Замечание***
>
> В *главе 17* будет показан более элегантный способ создания методов-аксессоров.

Кроме методов, представленных в списке выше, введем дополнительный метод `distance()`, который вычисляет расстояние от начала координат до точки. Возможная реализация класса `Point` представлена в листинге 16.3.

Листинг 16.3. Класс `Point`. Файл point.php

```
<?php
class Point
{
    private $x;
    private $y;

    public function setX($x)
    {
        $this->x = $x;
    }
    public function setY($y)
    {
        $this->y = $y;
    }
    public function getX()
    {
        return $this->x;
    }
    public function getY()
    {
        return $this->y;
    }
    public function distance()
    {
        return sqrt($this->getX() ** 2 + $this->getY() ** 2);
    }
}
```

В листинге 16.4 приводится пример использования класса `Point`.

Листинг 16.4. Файл point_use.php

```
<?php
require_once 'point.php';

$point = new Point;
$point->setX(5);
$point->setY(7);

echo $point->distance(); // 8.6023252670426;
```

16.3. Статические методы

До текущего момента использовались методы объекта, которые можно было вызывать только после того, как объект класса создавался при помощи 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 16.5).

Листинг 16.5. Объявление статического метода. Файл greet.php

```
<?php
class Greet
{
    public static function hello()
    {
        return 'Hello, world!';
    }
}
```

PHP допускает любой порядок следования спецификаторов доступа и ключевого слова `static` перед именем функции. Однако согласно требованиям PSR ключевое слово `static` всегда располагается после спецификаторов доступа.

Для вызова статического метода необходимо предварить его именем класса и оператором разрешения области видимости `::` (листинг 16.6).

Листинг 16.6. Файл greet_use.php

```
<?php
require_once('greet.php');
echo Greet::hello(); // Hello, world!
```

16.4. Ключевое слово *self*

Внутри метода класса, для того чтобы сослаться на статическую переменную этого же класса, не обязательно использовать имя класса перед оператором разрешения области видимости. Вместо него может использоваться ключевое слово `self`.

В листинге 16.7 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, т. к. позволяет переименовывать класс без многочисленных исправлений и, как правило, более короток в написании.

Листинг 16.7. 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 16.8 приводится пример вызова метода `render()` класса `Page`.

Листинг 16.8. Файл greet_use.php

```
<?php
require_once 'page.php';

Page::render();
```

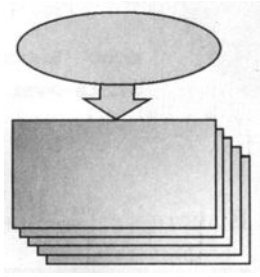
Результатом работы скрипта будут следующие строки:

```
header
about
footer
```

Задания

1. Создайте класс с методом `find()`, принимающий единственный аргумент — строку с названием функции PHP. В качестве результата метод должен возвращать текст с сайта **http://php.net** с описанием функции.
2. Создайте класс-обертку для сессии, метод `set()` которого позволяет устанавливать значение сессии, `get()` считывать ранее установленное значение, `listKeys()` выводит список всех установленных ключей, а `existsKey()` проверяет наличие ключа в сессии.

ГЛАВА 17



Специальные методы

Листинги данной главы
можно найти в подкаталоге magic.

При создании классов и объектов в PHP разработчик не действует с чистого листа. Класс уже содержит ряд специальных методов, вызываемых неявно при создании, удалении объекта, при обращении к его переменным и методам.

Разработчик PHP имеет возможность переопределить эти методы и тем самым вмешаться в жизненный цикл объекта. Такие специальные методы в сообществе называют *магическими методами*, а чтобы отличить их от остальных методов, их название предваряется двумя символами подчеркивания, например, `__constructor()`.

17.1. Конструктор *__construct()*

Конструктор — это специальный метод класса, который автоматически выполняется в момент создания объекта до вызова всех остальных нестатических методов класса. Данный метод используется главным образом для инициализации объекта, обеспечивая согласованное состояние объекта перед началом работы.

Для объявления конструктора в классе необходимо создать метод с именем `__construct()`. В листинге 17.1 приводится пример объявления класса `Constructor`, содержащего конструктор, который выводит сообщение "Вызов конструктора" и инициализирует закрытый член `$var`.

Листинг 17.1. Переопределение конструктора. Файл constructor.php

```
<?php
class Constructor
{
    private $var;

    public function __construct()
```

```
    {
        echo 'Вызов конструктора<br />';
        $this->var = 100;
    }
}
```

В листинге 17.2 приводится пример использования класса `Constructor`.

Листинг 17.2. Файл constructor_use.php

```
<?php
require_once 'constructor.php';

$obj = new Constructor();

echo '<pre>';
print_r($obj);
echo '</pre>';
```

Результатом работы скрипта из листинга 17.2 являются следующие строки:

```
Вызов конструктора
cls Object
(
    [var:private] => 100
)
```

Вызов конструктора производится автоматически во время выполнения оператора `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, поскольку это противоречит принципу инкапсуляции, однако в PHP конструктор можно вызвать не только в самом классе, но и из внешнего кода. Модифицируем класс `Constructor`, добавив в него дополнительный метод `inner()`, вызывающий конструктор (листинг 17.3).

Листинг 17.3. Файл constructor_call.php

```
<?php
class Constructor
{
    private $var;

    public function __construct()
    {
        echo 'Вызов конструктора<br />';
        $this->var = 100;
    }
```

```
    public function inner()
    {
        $this->__construct();
    }
}
```

В листинге 17.4 приводится пример явного и неявного вызова конструктора.

Листинг 17.4. Файл constructor_call_use.php

```
<?php
require_once 'constructor_call.php';

$obj = new Constructor(); // Вызов конструктора
$obj->__construct();      // Вызов конструктора
$obj->inner();            // Вызов конструктора
```

В первый раз конструктор вызывается неявно при создании объекта `$obj`, 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.

17.2. Параметры конструктора

Конструктор, как и любой другой метод, может принимать параметры, которые передаются ему при объявлении объекта в круглых скобках, следующих после имени класса. В листинге 17.5 приводится класс `Point`, предназначенный для моделирования точки в двумерной декартовой системе координат. Две его закрытые переменные `$x` и `$y` определяют координаты точки по оси абсцисс и ординат соответственно.

Листинг 17.5. Передача параметров конструктору. Файл point.php

```
<?php
class Point
{
    private $x;
    private $y;

    public function __construct($x, $y)
    {
        $this->x = $x;
        $this->y = $y;
    }
```

```
    public function getX()
    {
        return $this->x;
    }
    public function getY()
    {
        return $this->y;
    }
}
```

В листинге 17.6 демонстрируется использование конструктора, инициализирующего закрытые члены класса.

Листинг 17.6. Передача аргументов конструктору. Файл point_use.php

```
<?php
require_once 'point.php';

// $obj = new Point(); // Fatal error: Uncaught ArgumentCountError
$obj = new Point(10, 20);
echo $obj->getX().' '.$obj->getY(); // 10 20
```

Важно отметить, что если не указать аргументы при объявлении объекта, то произойдет ошибка "Fatal error: Uncaught ArgumentCountError: Too few arguments to function Point::__construct(), 0 passed".

PHP не поддерживает создание нескольких разных конструкторов (или других методов) с разным набором параметров. Однако можно обойти это ограничение необязательными параметрами. В листинге 17.7 представлен переработанный класс `Point`, в котором закрытые переменные `$x` и `$y` получают значение 0, если их значение не устанавливается через параметры конструктора.

Листинг 17.7. Использование параметров по умолчанию. Файл default.php

```
<?php
class Point
{
    private $x;
    private $y;

    public function __construct($x = 0, $y = 0)
    {
        $this->x = $x;
        $this->y = $y;
    }
    public function getX()
    {
        return $this->x;
    }
```

```
    public function getY()
    {
        return $this->y;
    }
}
```

Если один из параметров конструктору не передается, он получает значение по умолчанию.

17.3. Деструктор __destruct()

Деструктор — это специальный метод класса, который автоматически выполняется в момент уничтожения объекта. Данный метод 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 17.8 приводится пример объявления класса `Destructor`, конструктор которого выводит сообщение "Вызов конструктора", а деструктор — "Вызов деструктора".

Листинг 17.8. Файл destructor.php

```
<?php
class Destructor
{
    public function __construct()
    {
        echo 'Вызов конструктора<br />';
    }
    public function __destruct()
    {
        echo 'Вызов деструктора<br />';
    }
}
```

В листинге 17.9 приводится пример создания объекта класса `Destructor`.

Листинг 17.9. Файл destructor_use.php

```
<?php
require_once('destructor.php');

$obj = new Destructor();
```

Скрипт, представленный в листинге 17.9, 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 довольно редко, даже в тех ситуациях, где он мог быть полезным. Дело в том, что деструктор вызывается лишь в том случае, когда объект штатно собирается сборщиком мусора. Это событие может произойти далеко не сразу или вообще не наступить, если возникает ошибка, исключительная ситуация или скрипт завершается извне (листинг 17.10).

> ***Замечание***
>
> В листинге 17.10 используются ключевое слово `throw` и предопределенный класс `Exception` для обработки исключительных ситуаций, более подробно рассмотренных в *главе 21*.

Листинг 17.10. Файл destructor_throw.php

```
<?php
require_once('destructor.php');

$obj = new Destructor(); // Вызов конструктора
throw new Exception;     // Fatal error: Uncaught exception 'Exception'
print_r($obj);           // Эта точка никогда не достигается
```

Скрипт из листинга 17.10 никогда не вызовет деструктора.

17.4. Методы-аксессоры __set() и __get()

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 заключается в том, что приходится создавать довольно много однообразных в реализации методов для чтения и присваивания. Причем имена таких методов зачастую не совпадают с именами переменных `getX()` *(см. разд. 16.2)*.

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, принимает единственный параметр, который служит ключом. Метод `__set()` позволяет присвоить свойству новое значение и принимает два параметра, первый из которых является ключом, а второй — значением свойства.

В листинге 17.11 при помощи метода `__set()` объекту присваиваются новые свойства, которые помещаются в массив `$this->arr`, а перегруженный метод `__get()` позволяет извлечь их из массива.

Листинг 17.11. Файл accessor.php

```
<?php
class Accessor
{
    private $arr = [];

    public function __get($key)
    {
        if(array_key_exists($key, $this->arr)) {
            return $this->arr[$key];
        } else {
            return null;
        }
    }

    public function __set($key, $value)
    {
        $this->arr[$key] = $value;
    }
}
```

Как видно из листинга 17.11, класс `Accessor` перехватывает все обращения к свойствам объекта и создает соответствующий элемент в закрытом массиве `$arr`. В листинге 17.12 демонстрируется, как обращение к свойству `$name` приводит к созданию соответствующего элемента массива.

Листинг 17.12. Файл accessor_use.php

```
<?php
require_once 'accessor.php';

$obj = new Accessor();

$obj->name = 'Hello, world!<br />';
echo $obj->name;

echo "<pre>";
print_r($obj);
echo "</pre>";
```

Результатом работы скрипта из листинга 17.12 являются следующие строки:

```
Hello, world!
cls Object
(
    [arr:private] => Array
        (
            [name] => Hello, world!
        )
)
```

Любая попытка присвоить свойству значения приводит к созданию нового элемента закрытого массива `$arr` или к обновлению значения уже существующего элемента этого массива.

17.5. Динамические методы

Специальный метод `__call()` предназначен для создания динамических методов: если в классе не перегружен 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 17.13 представлен класс `MinMax`, который предоставляет пользователю два метода: `min()` и `max()`, принимающие произвольное количество числовых параметров и определяющие минимальное и максимальное значения соответственно.

Листинг 17.13. Файл minmax.php

```
<?php
class MinMax
{
    public function __call($method, $arg)
    {
        if (!is_array($arg)) {
            return false;
        }
        $value = $arg[0];
        if ($method == 'min') {
            for($i = 0; $i < count($arg); $i++)
            {
                if ($arg[$i] < $value) {
                    $value = $arg[$i];
                }
            }
        }
```

```
        if ($method == 'max') {
            for($i = 0; $i < count($arg); $i++) {
                if ($arg[$i] > $value) {
                    $value = $arg[$i];
                }
            }
        }
        return $value;
    }
}
```

В примере из листинга 17.13 в зависимости от вызываемого метода — `min()` или `max()` — используются два разных алгоритма для поиска результата. Кроме того, для класса `MinMax` допустим вызов метода с произвольным именем, и, если оно отлично от `min` или `max`, будет возвращаться первый аргумент. Независимо от имени метода, если не передано ни одного аргумента, возвращается значение `false`.

В листинге 17.14 приводится пример использования класса `MinMax` для получения максимального и минимального значений последовательности.

Листинг 17.14. 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
```

По аналогии со специальным методом `__call()` существует статический вариант `__callStatic()`, позволяющий динамически создавать статические методы. В листинге 17.15 приводится пример реализации класса `MinMax`, в котором методы `min()` и `max()` определяются как статические.

Листинг 17.15. Файл minmax_static.php

```
<?php
class MinMax
{
    public static function __callStatic($method, $arg)
    {
        if (!is_array($arg)) {
            return false;
        }
        $value = $arg[0];
```

```
        if ($method == 'min') {
            for($i = 0; $i < count($arg); $i++)
            {
                if ($arg[$i] < $value) {
                    $value = $arg[$i];
                }
            }
        }
        if ($method == 'max') {
            for($i = 0; $i < count($arg); $i++) {
                if ($arg[$i] > $value) {
                    $value = $arg[$i];
                }
            }
        }
        return $value;
    }
}
```

В листинге 17.16 приводится пример использования класса `MinMax` со статическими вариантами методов поиска максимального и минимального значений.

Листинг 17.16. 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
```

17.6. Интерполяция объекта

Специальный метод `__toString()` позволяет интерполировать (подставлять) объект в строку. Для подстановки значений переменных необходимо заключить строку в двойные кавычки (листинг 17.17).

Листинг 17.17. 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 `Point` (см. листинг 17.7) таким образом, чтобы его подстановка в строку приводила к выводу координат по оси абсцисс и ординат в круглых скобках (листинг 17.18).

> ***Замечание***
>
> **Следует обратить внимание, что метод `__toString()` выводит результат при помощи конструкции `return`, а не `echo`.**

Листинг 17.18. Файл point_interpolation.php

```
<?php
class Point
{
    private $point;

    public function __construct($x = 0, $y = 0)
    {
        $this->point = [];
        $this->point['x'] = $x;
        $this->point['y'] = $y;
    }
    public function __get($key)
    {
        if(array_key_exists($key, $this->point)) {
            return $this->point[$key];
        } else {
            return null;
        }
    }

    public function __set($key, $value)
    {
        if(array_key_exists($key, $this->point)) {
            $this->point[$key] = $value;
        }
    }

    public function __toString() {
        return "({$this->x}, {$this->y})";
    }
}
```

Как видно из листинга 17.18, координаты точки хранятся в ассоциативном массиве `$point`, который при необходимости может быть расширен до трехмерного или многомерного случая. Для доступа к свойствам `x` и `y` перезагружаются методы `__get()` и `__set()`. В самом конце класса перезагружается специальный метод `__toString()`, ответственный за формирование строки при интерполяции объекта.

В листинге 17.19 приводится пример использования класса `Point`.

Листинг 17.19. Файл point_interpolation_use.php

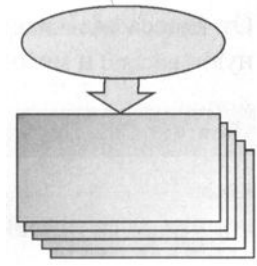
```
<?php
require_once 'point_interpolation.php';

$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 "Catchable fatal error: Object of class Point could not be converted to string".

Задания

1. По документации с сайта **http://php.net** изучите возможности специального метода `__unset()`, предназначенного для управления процессом уничтожения свойства объекта при помощи конструкции `unset()`. Модифицируйте класс `Accessor` из листинга 17.11 таким образом, чтобы он поддерживал удаление свойств класса.
2. По документации с сайта **http://php.net** изучите возможности специальных методов `__sleep()` и `__wakeup()`, предназначенных для управления процессом сериализации объекта. Модифицируйте класс `Accessor` из листинга 17.11 таким образом, чтобы его объекты можно было сериализовать при помощи функции `serialize()` и восстановить при помощи функции `unserialize()`. По возможности организуйте сериализацию таким образом, чтобы объект сохранялся в JSON-формате.

ГЛАВА 18



Наследование

Листинги данной главы
можно найти в подкаталоге inheritance.

Одной из главных целей объектно-ориентированного подхода является повторное использование кода. Достигается это несколькими механизмами, главным из которых является наследование, рассматриваемое в текущей главе.

18.1. Наследование

Объектно-ориентированная методология предоставляет три возможных способа повторного использования кода:

- включение объектов в класс;
- подмешивание классов в виде трейтов *(см. главы 19–20)*;
- использование наследования.

Наследование позволяет создать новый класс на основе уже существующего, автоматически включив в новый класс переменные и методы старого. В рамках наследования "старый" класс называется *базовым*, а вновь создаваемый класс — *производным*. При объявлении производного класса необходимо указать имя базового класса с помощью ключевого слова `extends`. В листинге 18.1 создается базовый класс `Base`, содержащий единственную переменную `$first` и метод `printFirst()`.

Листинг 18.1. Базовый класс Base. Файл base.php

```
<?php
class Base
{
    public $first;
    public function printFirst()
    {
        echo $this->first;
    }
}
```

От класса `Base` наследуется класс `Derived`, содержащий, в свою очередь, переменную `$second` и метод `printSecond()` (листинг 18.2).

Листинг 18.2. Производный класс `Derived`. Файл derived.php

```
<?php
require_once 'base.php';

class Derived extends Base
{
    public $second;
    public function printSecond()
    {
        echo $this->second;
    }
}
```

В листинге 18.3 приводится пример, в котором объявляется объект `$obj` производного класса `Derived`.

Листинг 18.3. Файл derived_use.php

```
<?php
require_once 'derived.php';

$obj = new Derived;

$obj->first = 3;
$obj->second = 5;

$obj->printFirst(); // 3
echo '<br />';
$obj->printSecond(); // 5
```

Объект производного класса `Derived` содержит не только собственную переменную `$second` и метод `printSecond()`, но и переменную `$first` и метод `printFirst()`, объявленные в базовом классе `Base`. Производный класс, в свою очередь, может выступать в качестве базового для других классов. В результате можно получить разветвленную иерархию классов, расширяя функциональность без повторного создания переменных и методов, используя уже имеющиеся из существующих классов.

18.2. Спецификаторы доступа и наследование

Переменные и методы, объявленные со спецификаторами доступа `public` и `private`, при наследовании ведут себя по отношению к производному классу точно так же, как и внешний код в отношении объекта. Это означает, что из производного класса

доступны все переменные и методы, объявленные со спецификатором доступа `public`, и не доступны компоненты, объявленные как `private`.

В листинге 18.4 приводится пример базового класса `Base` с закрытой переменной `$var`, которая инициализируется конструктором и доступна через метод `getVar()`.

Листинг 18.4. Файл base_private.php

```
<?php
class Base
{
    private $var;

    public function __construct($var)
    {
        $this->var = $var;
    }
    public function getVar()
    {
        return $this->var;
    }
}
```

Теперь если унаследовать от класса `Base` новый класс `Derived` и попробовать обратиться к закрытой переменной `$var`, выяснится, что в объекте класса `Derived` будет создана новая открытая переменная `$var`, не имеющая отношения к закрытой переменной класса `Base` (листинг 18.5).

Листинг 18.5. Файл derived_private.php

```
<?php
require_once 'base_private.php';

class Derived extends Base
{
    public function __construct($var)
    {
        $this->var = $var;
    }
}
```

В листинге 18.6 приводится пример использования класса `Derived`.

Листинг 18.6. Файл derived_private_use.php

```
<?php
require_once 'derived_private.php';
```

```
$obj = new Derived(20);
echo $obj->getVar(); // null
echo '<pre>';
print_r($obj);
echo '</pre>';
```

Метод `getVar()` возвращает `null`, несмотря на то что в конструкторе присваивается значение переменной `$var`. Дело в том, что производный класс даже не видит попытки обращения к закрытому члену, он просто создает новый открытый член в объекте производного класса:

```
Derived Object
(
    [var:Base:private] =>
    [var] => 20
)
```

Именно поэтому обращение к методу `getVar()` не приводит к выводу переменной `$var` — данная переменная остается не инициализированной в рамках класса `Base`.

Иногда удобно, чтобы переменная или метод базового класса, оставаясь закрытыми для внешнего кода, были открыты для производного класса. В этом случае прибегают к спецификатору доступа `protected`. Снабженные им компоненты классы называют *защищенными*.

В листинге 18.7 приводится пример базового класса `Base`, в котором переменная `$var` объявлена защищенной.

Листинг 18.7. Файл base_protected.php

```
<?php
class Base
{
    protected $var;
    public function __construct($var)
    {
        $this->var = $var;
    }
}
```

Производный класс `Derived` не претерпевает изменений (листинг 18.8).

Листинг 18.8. Файл derived_protected.php

```
<?php
require_once 'base_protected.php';

class Derived extends Base
{
    public function __construct($var)
```

```
    {
        $this->var = $var;
    }
}
```

В листинге 18.9 приводится скрипт, объявляющий объект класса `Derived`. Попытки обращения к защищенной переменной `$var` завершаются ошибкой.

Листинг 18.9. Файл derived_protected_use.php

```
<?php
require_once 'derived_protected.php';

$obj = new derived(20);
// echo $obj->var; // Ошибка
echo '<pre>';
print_r($obj);
echo '</pre>'
```

Результатом работы скрипта из листинга 18.9 будут следующие строки:

```
Derived Object
(
    [var:protected] => 20
)
```

Как видно из результатов работы скрипта, конструктор производного класса имеет возможность инициализировать переменную `$var` в обход конструктора базового класса, в то же время переменная `$var` остается закрытой по отношению к внешнему коду.

Спецификатор `protected` используется как с переменными класса, так и с методами. Например, если конструктор базового класса объявлен со спецификатором `protected`, то получить его объект невозможно; доступны будут только объекты производных классов.

18.3. Перегрузка методов

В производном классе можно создать метод с таким же названием, что и в базовом классе, который заменит метод базового класса при вызове. Такая процедура называется *перегрузкой методов*.

В листингах 18.10–18.12 приводится пример перегрузки метода `overridden()` в производном классе `Derived`, который наследуется от базового класса `Base`.

Листинг 18.10. Файл base_overridden.php

```
<?php
class Base
```

```
{
    public function overridden()
    {
        echo 'Вызов метода Base::overridden()<br />';
    }
}
```

Листинг 18.11. Файл derived_overridden.php

```
<?php
require_once 'base_overridden.php';

class Derived extends Base
{
    public function overridden()
    {
        echo 'Вызов метода Derived::overridden()<br />';
    }
}
```

Листинг 18.12. Файл derived_overridden_use.php

```
<?php
require_once 'derived_overridden.php';

$obj = new Derived();
$obj->overridden();
```

Результатом работы скрипта из листинга 18.12 будет следующая строка:

```
Вызов метода Derived::overridden()
```

Таким образом, метод `Base::overridden()` не вызывается при обращении к объекту производного класса `Derived`.

Однако в рамках производного класса остается возможность вызвать метод базового класса, обратившись к нему при помощи префикса `parent::`. В листингах 18.13 и 18.14 представлена перегрузка метода `overridden()`, при которой метод производного класса сначала вызывает метод базового класса.

Листинг 18.13. Файл derived_overridden_alt.php

```
<?php
require_once 'base_overridden.php';

class Derived extends Base
{
    public function overridden()
```

```
    {
        parent::overridden();
        echo 'Вызов метода Derived::overridden()<br />';
    }
}
```

Листинг 18.14. Файл derived_overridden_alt_use.php

```
<?php
require_once 'derived_overridden_alt.php';

$obj = new Derived();
$obj->overridden();
```

Результатом работы скрипта из листинга 18.14 будут уже две строки:

```
Вызов метода Base::overridden()
Вызов метода Derived::overridden()
```

Таким образом, при помощи перегрузки метода можно как расширить метод базового класса, так и полностью заменить его уже новым.

18.4. Позднее статическое связывание

При наследовании классов надо уделять особое внимание статическим методам. В *разд. 16.4* было рассмотрено ключевое слово `self`, которое позволяет ссылаться на статический метод. Ключевое слово `self` всегда ссылается на текущий класс, при помощи его нельзя обратиться к статическим методам базового класса. В листингах 18.15–18.17 в производном классе `Derived` осуществляется попытка переопределить статический метод `title()` из базового класса `Base`.

Листинг 18.15. Файл base_self.php

```
<?php
class Base
{
    public static function title()
    {
        return __CLASS__;
    }

    public static function test() {
        return self::title();
    }
}
```

Листинг 18.16. Файл derived_self.php

```
<?php
require_once 'base_self.php';

class Derived extends Base
{
    public static function title()
    {
        return __CLASS__;
    }
}
```

Листинг 18.17. Файл derived_self_use.php

```
<?php
require_once 'derived_self.php';

echo Derived::test(); // Base
```

Как видно из примера, при попытке воспользоваться методом `test()` в классе-наследнике, `self`, вместо того чтобы вернуть метод из класса `Derived`, вызвал метод из базового класса `Base`. Для решения этой проблемы PHP предоставляет специальное ключевое слово `static`, которое можно задействовать вместо `self` (листинги 18.18–18.20).

Листинг 18.18. Файл base_static.php

```
<?php
class Base
{
    public static function title()
    {
        return __CLASS__;
    }

    public static function test() {
        return static::title();
    }
}
```

Листинг 18.19. Файл derived_static.php

```
<?php
require_once 'base_static.php';
```

```
class Derived extends Base
{
    public static function title()
    {
        return __CLASS__;
    }
}
```

Листинг 18.20. Файл derived_static_use.php

```
<?php
require_once 'derived_static.php';

echo Derived::test(); // Derived
```

18.5. Полиморфизм

При использовании разветвленных наследственных иерархий все объекты производных классов автоматически снабжаются методами базового класса. Производные классы могут использовать методы базового класса без изменений, адаптировать их или заменять собственной реализацией. Как бы ни были реализованы эти методы, можно утверждать, что все объекты наследственной иерархии классов будут обладать некоторым количеством методов с одними и теми же названиями. Это явление называется *полиморфизмом*.

При использовании полиморфизма появляется возможность программировать работу объектов, абстрагируясь от их типа. Так, в транспортной сети, независимо от того, является ли текущий объект автомобильным, железнодорожным, воздушным или водным транспортным средством, он характеризуется способностью к движению, и для всех объектов "транспорт" можно ввести единый метод движения `move()`. Каждый объект из транспортной иерархии будет обладать этим методом. Несмотря на то, что движение по асфальтовой или железной дороге, по воздуху или по реке отличается, методы будут называться одинаково, и их можно будет вызывать для каждого из объектов, чей класс является наследником класса "транспорт" (рис. 18.1).

Скорее всего, метод `move()` придется переопределить для классов "автомобиль", "авиатранспорт", "водный транспорт" и "железнодорожный транспорт", т. к. свобода движения у этих классов транспортных средств различна и даже зависит от сезона. При дальнейшем развитии иерархии для наследников класса "водный транспорт" переопределять методы, вероятно, уже не придется, поскольку реализация метода `move()` базового класса "водный транспорт" подойдет для каждого из них.

Однако такие объемные системы, как транспортные сети, редко моделируют на PHP; чаще задачи сводятся к разработке Web-компонентов. В качестве примера рассмотрим постраничную навигацию. Объемный список неудобно отображать на странице целиком, т. к. это требует значительных ресурсов и затрудняет поиск.

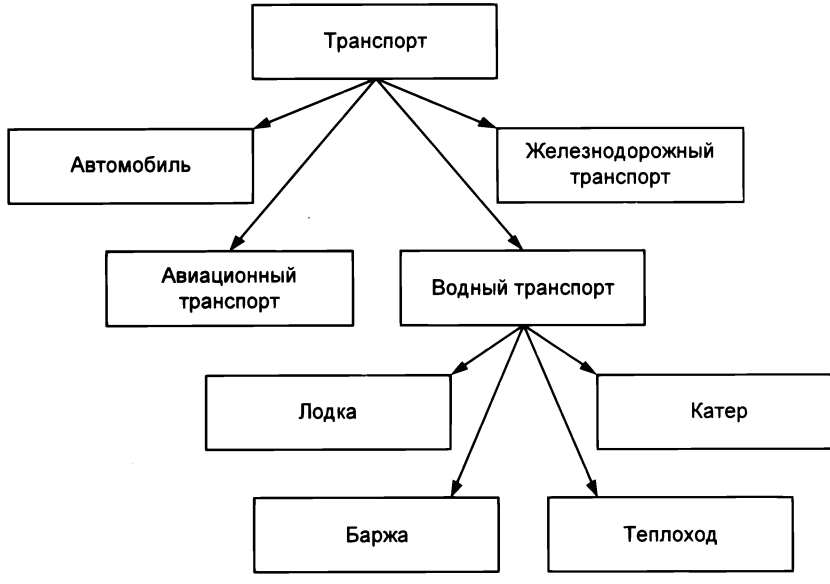


Рис. 18.1. Схема иерархии объектов "транспорт"

Гораздо нагляднее выводить список, например, по 10 элементов, предоставляя ссылки на оставшиеся страницы. В большинстве случаев такая задача решается без привлечения объектно-ориентированного подхода и тем более без наследуемой иерархии и полиморфизма. Однако в Web-приложении источником списка элементов, к которому следует применить постраничную навигацию, могут выступать база данных, файл со строками, каталог с файлами изображений и т. п. Каждый раз создавать отдельную функцию для реализации постраничной навигации не всегда удобно, т. к. придется дублировать значительную часть кода в нескольких функциях. В данном случае удобнее реализовать постраничную навигацию в методе базового класса `Pager`, а методы, работающие с конкретными источниками, переопределить в производных классах:

- `PagerDb` — постраничная навигация для СУБД;
- `PagerFile` — постраничная навигация для текстового файла;
- `PagerDir` — постраничная навигация для файлов в каталоге.

Иерархия классов постраничной навигации представлена на рис. 18.2.

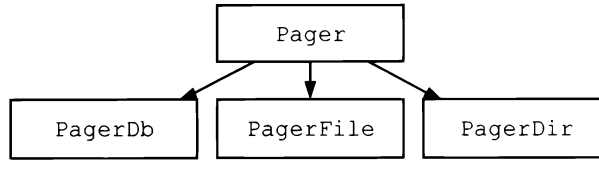


Рис. 18.2. Иерархия классов постраничной навигации

Класс `Pager` "не знает", каким образом производные классы будут узнавать общее количество элементов в списке, сколько элементов будет отображаться на одной странице, сколько ссылок находится слева и справа от текущей страницы. Поэтому помимо метода `__ToString()` класс будет содержать четыре защищенных (`protected`) метода, которые возвращают:

- `total()` — общее количество элементов в списке;
- `pnumber()` — количество элементов на странице;
- `pageLink()` — количество элементов слева и справа от текущей страницы;
- `parameters()` — строку, которую необходимо передать по ссылкам на другую страницу (например, при постраничном выводе результатов поиска по ссылкам придется передавать результаты поиска).

Эти методы не имеют реализации в классе `Pager` (пустые) и перегружаются производными классами, которые формируют параметры в зависимости от источника.

18.6. Абстрактные классы

Возвращаясь к иерархии классов постраничной навигации, рассмотренной в *разд. 18.5*, можно утверждать, что объект класса `Pager` не имеет смысла и не может быть использован по назначению — работают только его наследники. Для того чтобы предотвратить возможность создания объекта такого класса, PHP предоставляет специальный инструмент — класс можно объявить абстрактным. Для этого объявление класса предваряют ключевым словом `abstract` (листинг 18.21).

Листинг 18.21. Объявление абстрактного класса. Файл page_abstract.php

```
<?php
abstract class Pager
{
    //...
}
```

Теперь попытка создать экземпляр класса `Pager` (листинг 18.22) будет заканчиваться сообщением об ошибке "Fatal error: Uncaught Error: Cannot instantiate abstract class Pager".

Листинг 18.22. Файл page_abstract_use.php

```
<?php
require_once 'page_abstract.php';

$obj = new Pager(); // Fatal error
```

Абстрактными следует объявлять классы, которые не существуют в реальности и объекты которых заведомо не потребуются. Например, возвращаясь к транспорт-

ной сети, объекты "транспорт", "автомобиль", "авиатранспорт", "водный транспорт", "железнодорожный транспорт" не понадобятся, и их можно сделать абстрактными. В то же время конкретные классы ("лодка", "баржа", "теплоход", "катер") могут быть полезными при оценке грузо- и пассажиропотока, расчета налогообложения и т. п.

18.7. Абстрактные методы

Если во время работы над сайтом появляется новый источник данных, к которому нужно будет применить постраничную навигацию *(см. разд. 18.6)*, достаточно унаследовать от базового класса `Pager` новый производный класс.

Однако продемонстрированный подход обладает некоторым изъяном: программист может забыть реализовать один из методов, который используется базовым классом. Напомним, что в базовом классе `Pager` находятся только заглушки методов `total()`, `pnumber()`, `pageLink()` и `parameters()`.

Для решения этой задачи в объектно-ориентированной модели PHP предусмотрены абстрактные методы, объявление которых предваряется ключевым словом `abstract`.

Для абстрактных методов задают их описание и список параметров без реализации (без тела метода). Каждый класс, который наследуется от базового класса, содержащего абстрактные методы, обязан их реализовать. Если хотя бы один метод не реализован — работа PHP-скрипта будет остановлена, а интерпретатор сообщит о необходимости реализации абстрактного метода в производном классе. При помощи абстрактных методов исключается возможность неумышленного нарушения полиморфизма для производных классов.

В листинге 18.23 методы `total()`, `pnumber()`, `pageLink()` и `parameters()` класса `Pager` объявляются абстрактными.

> ***Замечание***
>
> Согласно требованиям стандарта PSR, ключевое слово `abstract` располагается перед спецификатором области видимости.

Листинг 18.23. Объявление абстрактных методов. Файл abstract.php

```
<?php
abstract class Pager
{
    abstract public function total();
    abstract public function pnumber();
    abstract public function pageLink();
    abstract public function parameters();
}
```

Наличие в классе абстрактного метода требует, чтобы класс также был объявлен абстрактным.

18.8. *Final*-методы класса

Если абстрактные методы и интерфейсы предназначены для того, чтобы обеспечить обязательную перегрузку методов в производных классах, то ключевое слово `final` решает обратную задачу. Методы, объявленные в базовом классе с ключевым словом `final`, не могут быть перегружены в производном классе.

В листингах 18.24 и 18.25 приводится пример базового класса `Base`, снабженного `final`-методом `finalMethod()`, и производного класса `Derived`.

> ***Замечание***
>
> Согласно стандарту кодирования PSR, ключевое слово `final` располагается перед спецификатором доступа.

Листинг 18.24. Файл base_final.php

```
<?php
class Base
{
    final public function finalMethod()
    {
        return 'Base::finalMethod()';
    }
}
```

Листинг 18.25. Файл derived_final.php

```
<?php
require_once 'base_final.php';

class Derived extends Base
{
    // public function finalMethod()
    // {
    //     echo 'Derived::finalMethod()';
    // }
}
```

Попытка перегрузить метод `finalMethod()` в производном классе `Derived` заканчивается ошибкой "Fatal error: Cannot override final method Base::finalMethod()".

Помимо обычных методов в качестве `final`-метода можно объявлять и специальные методы. Например, в качестве `final`-методов могут объявляться конструкторы и деструкторы.

18.9. *Final*-классы

Совместно с ключевым словом `final` можно объявлять не только отдельные методы, но и целые классы. Класс, объявленный при помощи ключевого слова `final`, не может иметь наследников (листинг 18.26).

> ***Замечание***
>
> Класс не может одновременно являться и абстрактным (`abstract`), и `final`-классом.

Листинг 18.26. Объявление `final`-класса. Файл final.php

```
<?php
final class Base
{
    public function __construct()
    {
    }
}
```

Попытка объявить производный класс, наследующий от `final`-класса, заканчивается сообщением об ошибке: "Fatal error: Class derived may not inherit from final class (base)".

18.10. Анонимные классы

Анонимные классы появились, начиная с PHP 7. Для их демонстрации создадим класс `Dumper`, который имеет единственный статический метод, выводящий дамп своего аргумента (листинг 18.27).

Листинг 18.27. Файл dumper.php

```
<?php
class Dumper
{
    public static function print($obj)
    {
        print_r($obj);
    }
}
```

В листинге 18.28 приводится пример использования класса.

Листинг 18.28. Использование анонимного класса. Файл anonym.php

```
<?php
require_once 'dumper.php';

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 скрипта будет следующая строка:

```
class@anonymous Object ( [title] => Hello, world! )
```

При определении анонимного класса внутри другого класса, для получения доступа к защищенным членам, допускается наследование анонимных классов (листинг 18.29).

Листинг 18.29. Вложенные анонимные классы. Файл container.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})";
            }
        };
    }
}
```

В листинге 18.30 приводится пример использования класса `Container`.

Листинг 18.30. Файл container_use.php

```
<?php
require_once 'container.php';

(new Container)->anonym()->print();
```

Обратите внимание, что в примере анонимный класс возвращается оператором `return`, в конце которого обязательно требуется точка с запятой. Анонимный класс, будучи унаследованным от класса `Container`, получает доступ к защищенному члену `$id`, в то время как закрытый член `$title` мы вынуждены были передать через его конструктор. Результатом выполнения скрипта будет следующая строка:

```
Класс Container (1)
```

18.11. Оператор *instanceof*

Для того чтобы определить, является ли текущий объект экземпляром класса, используется оператор `instanceof`. Оператор возвращает `true` в случаях, когда объект является экземпляром класса или его класс наследуется одним из классов. Вернем в начало главы к иерархии классов `Base` (см. листинг 18.1) и `Derived` (см. листинг 18.2). Создадим объект класса `Derived` (листинг 18.31) и проверим, в каких случаях оператор `instanceof` возвращает `true`, а в каких `false`.

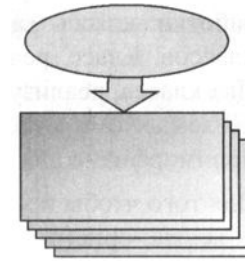
Листинг 18.31. Файл instanceof.php

```
<?php
require_once 'derived.php';

$obj = new Derived;

echo $obj instanceof Derived; // true
echo $obj instanceof Base;    // true
echo $obj instanceof Dir;     // false
```

Задания

1. По документации с сайта **http://php.net** познакомьтесь с предопределенными классами `Directory`, `DateTime`, `DateTimeZone`, `DateInterval`, `DatePeriod`. Допускается ли наследование от этих предопределенных классов?
2. Реализуйте класс `Pager` из *разд. 18.5*.

ГЛАВА 19



Интерфейсы

Листинги данной главы
можно найти в подкаталоге interfaces.

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.

19.1. 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 производные классы; базовые классы иерархии необходимы лишь для сокрытия реализации и формирования общего набора методов производных классов.

В результате базовые классы зачастую становятся абстрактными, вернее, классами, которые определяют 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. 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 раз-

работки: классы задают поведение объектов, а интерфейсы — поведение группы классов. Класс, реализующий интерфейс, обязан реализовать методы интерфейса. Два класса, реализующих 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 большая объектно-ориентированная система. Пусть это будет Web-сайт, содержащий статьи, новости, возможность регистрации и входа пользователей и систему администрирования.

Сосредоточимся на небольшой части данного сайта — на его пользователях (рис. 19.1). Общим для всех пользователей классом является абстрактный класс `User`, от которого наследуется два класса:

- `FrontUser` — зарегистрированный посетитель сайта;
- `BackendUser` — сотрудник компании, имеющий доступ к системе администрирования.

От класса `BackendUser` наследуется 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 фронт-части сайта, так и в специальном разделе системы администрирования;
- `Administrator` — администратор, помимо возможностей редактора имеющий доступ к системе логирования действий, панели управления пользователями как сайта, так и системы администрирования.

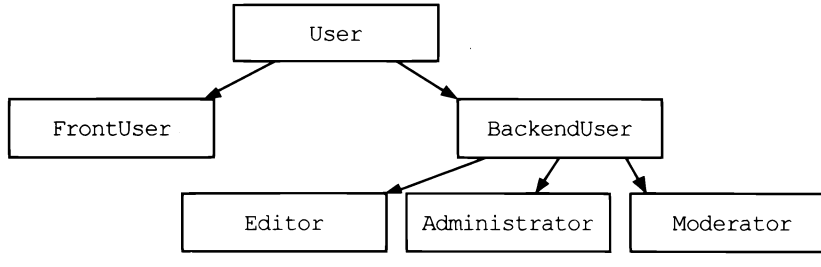


Рис. 19.1. Иерархия наследования пользователей

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 19.1).

Листинг 19.1. Файл user.php

```
<?php
class User
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }

    public function fullName() {
        $arr_name = array_filter([$this->first_name, $this->last_name]);
        $full_name = implode(' ', $arr_name);
        return $full_name ? $full_name : 'Анонимный пользователь';
    }
}
```

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, фильтрует его при помощи функции `array_filter()`, в результате чего в новом массиве `$arr_name` остаются только непустые значения. При помощи 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 19.2).

Листинг 19.2. 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 19.3–19.7.

Листинг 19.3. Файл frontuser.php

```
<?php
require_once 'user.php';
class FrontUser extends User {}
```

Листинг 19.4. Файл backenduser.php

```
<?php
require_once 'user.php';
class BackendUser extends User {}
```

Листинг 19.5. Файл editor.php

```
<?php
require_once 'backenduser.php';
class Editor extends BackendUser {}
```

Листинг 19.6. Файл administrator.php

```
<?php
require_once 'backenduser.php';
class Administrator extends BackendUser {}
```

Листинг 19.7. 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 потребуется два метода: установки `setImage()` и получения `getImage()`. Как видно из рис. 19.1, единственная возможность добавить их и в класс `FrontUser`, и в класс `Moderator` — на уровне класса `User`. Если мы добавим готовые методы 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 данную проблему, удобно воспользоваться интерфейсами.

19.2. Создание интерфейса

Для создания интерфейса используется ключевое слово `interface`. В листинге 19.8 приводится пример интерфейса `Avatar`, который требует от классов, реализующих этот интерфейс, объявления двух методов: установки `setImage()` и получения `getImage()` аватарки.

Листинг 19.8. Файл avatar.php

```
<?php
interface Avatar
{
    public function setImage($path);
    public function getImage();
}
```

Для того чтобы объявить класс, реализующий интерфейс, необходимо воспользоваться ключевым словом `implements`, который располагается в конце конструкции `class`. Например, задачу снабжения аватарками зарегистрированных пользователей и модераторов из предыдущего раздела можно решить следующим образом (листинги 19.9 и 19.10).

Листинг 19.9. Файл frontuser_avatar.php

```
<?php
require_once 'user.php';
require_once 'avatar.php';

class FrontUser extends User implements Avatar
{
    private $path;

    public function getImage()
```

```
    {
        return $this->path;
    }
    public function setImage($path)
    {
        $this->path = $path;
    }
}
```

Листинг 19.10. Файл moderator_avatar.php

```
<?php
require_once 'backenduser.php';
require_once 'avatar.php';

class Moderator extends BackendUser implements Avatar
{
    private $path;

    public function getImage()
    {
        return $this->path;
    }
    public function setImage($path)
    {
        $this->path = $path;
    }
}
```

В листинге 19.11 приводится пример использования модифицированного класса `FrontUser`.

Листинг 19.11. Файл moderator_avatar.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 выполнения

скрипта завершается сообщением об ошибке: "Fatal error: Access type for interface method must be omitted".

19.3. 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 так же 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. 19.2).

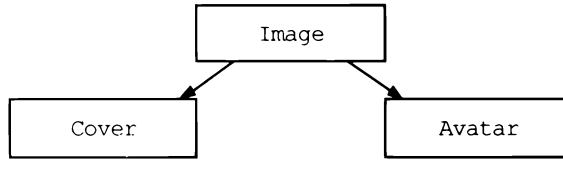


Рис. 19.2. Иерархия наследования интерфейсов

Реализация интерфейсов представлена в листингах 19.12–19.14. Точно так же, как в случае классов, наследование интерфейсов осуществляется при помощи ключевого слова `extends`.

Листинг 19.12. Файл image.php

```
<?php
interface Image
{
    public function setImage($path);
    public function getImage();
}
```

Листинг 19.13. Файл image_avatar.php

```
<?php
require_once 'image.php';
interface Avatar extends Image {}
```

Листинг 19.14. 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. 19.3).

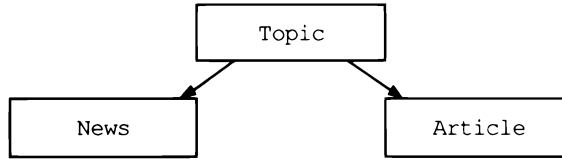


Рис. 19.3. Иерархия наследования классов

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 19.15).

Листинг 19.15. Файл topic.php

```
<?php
require_once 'image_cover.php';

class Topic implements Cover
{
    public $title;
    public $content;
    public $published_at;
    private $path;

    public function __construct(
         $title,
         $content,
         $published_at = null)
    {
        $this->title = $title;
        $this->content  = $content;
```

```
        if(empty($published_at)) {
            $this->published_at = time();
        } else {
            $this->published_at = $published_at;
        }
    }
    public function getImage()
    {
        return $this->path;
    }
    public function setImage($path)
    {
        $this->path = $path;
    }
}
```

Теперь можно реализовать классы новостей `News` и статей `Article`, унаследовав их от базового класса `Topic` (листинги 19.16 и 19.17).

Листинг 19.16. Файл news.php

```
<?php
require_once 'topic.php';
class News extends Topic {}
```

Листинг 19.17. Файл article.php

```
<?php
require_once 'topic.php';
class Article extends Topic {
    public $authors;

    public function __construct(
         $title,
         $content,
         $authors,
         $published_at = null)
    {
        parent::__construct($title, $content, $published_at);
        $this->authors = $authors;
    }
}
```

Обычно новости поступают из информационных агентств и их авторство не указываются, в то время как статьи являются авторским материалом и для них необходимо указать 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 19.18 приводится пример использования класса `Article`.

Листинг 19.18. Файл article_use.php

```
<?php
require_once 'article.php';

$obj = new Article(
    'Заголовок',
    'Содержимое',
    ['Максим Кузнецов', 'Игорь Симдянов']);

echo $obj->title; // Заголовок
```

19.4. Реализация нескольких интерфейсов

В предыдущих разделах в класс статей `Article` была добавлена дополнительная переменная `$article`. Авторство может быть у видеороликов, фотогалерей, поэтому имеет смысл выделить возможность добавления авторства в виде еще одного интерфейса `Author` (листинг 19.19).

Листинг 19.19. Файл author.php

```
<?php
interface Author
{
    public function setAuthor($authors);
    public function getAuthor();
}
```

Пусть одновременно каждый из опубликованных материалов, обладающих детальной страницей, должен содержать SEO-информацию, которая встраивается в head-части HTML-страницы. Пусть интерфейс, требующий реализации поддержки SEO-информации, `Seo` содержит следующие четыре метода:

- `title()` — заголовок для тега `<title>`;
- `description()` — заголовок для META-тега `description`;
- `keywords()` — заголовок для META-тега `keywords`;
- `seo($title, $description, $keywords)` — установка всех трех значений.

Возможная реализация интерфейса `Seo` представлена в листинге 19.20.

Листинг 19.20. Файл seo.php

```
<?php
interface Seo
{
    public function seo($title, $description, $keywords);
    public function title();
    public function description();
    public function keywords();
}
```

Для того чтобы реализовать два интерфейса одновременно, достаточно их перечислить через запятую после ключевого слова `implements` (листинг 19.21).

Листинг 19.21. 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 $authors;
    private $seo_title;
    private $seo_description;
    private $seo_keywords;

    public function getAuthor()
    {
        return $this->authors;
    }
    public function setAuthor($authors)
    {
        $this->authors = $authors;
    }
    public function seo(
        $title = null,
        $description = null,
        $keywords = null)
    {
        $this->seo_title = $title;
        $this->seo_description = $description;
        $this->seo_keywords = $keywords;
    }
    public function title()
    {
        if(!empty($this->seo_title)) {
            return $this->seo_title;
```

```
        } else {
            return $this->title;
        }
    }
    public function description()
    {
        return $this->seo_description;
    }
    public function keywords()
    {
        return $this->seo_keywords;
    }
}
```

В листинге 19.22 приводится пример использования получившегося класса `Article`.

Листинг 19.22. Файл article_author_seo_use.php

```
<?php
require_once 'article_author_seo.php';

$obj = new Article(
    'Заголовок',
    'Содержиомое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

19.5. 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 создадим два интерфейса: `first` и `second`, которые будут требовать от классов реализации одноименных методов (листинг 19.23).

Листинг 19.23. 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 />';
}
```

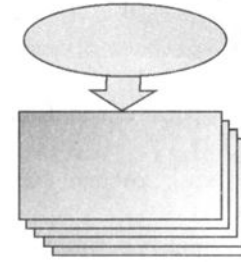
Результатом выполнения скрипта из листинга 19.23 будут следующие строки:

```
Объект реализует интерфейс Seo
Объект реализует интерфейс Author
```

Задание

Организуйте клонирование объектов класса `User` таким образом, чтобы пароль пользователя `password` в клонированном объекте отличался от оригинала.

ГЛАВА 20



Трейты

Листинги данной главы
можно найти в подкаталоге traits.

Трейты похожи на интерфейсы тем, что в одном классе может использоваться несколько разных трейтов. Однако, в отличие от интерфейсов, трейты содержат готовые методы. Их использование позволяет исключить повторные участки кода в тех случаях, когда реализация той или иной функциональности не отличается от класса к классу.

20.1. Создание трейта

В предыдущей главе разрабатывалась система пользователей сайта. Для добавления поддержки аватарок отдельными классами объектно-ориентированной схемы использовались интерфейсы. В результате каждый класс, которому необходима была поддержка аватарок, 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 20.1 приводится возможная реализация трейта `Image`, который содержит переменную `$path` и реализацию двух методов: `getImage()` и `setImage()`.

Листинг 20.1. Файл image.php

```
<?php
trait Image
```

```
{
    private $path;

    public function getImage()
    {
        return $this->path;
    }
    public function setImage($path)
    {
        $this->path = $path;
    }
}
```

Для включения трейта в класс используется ключевое слово `use`, после которого указывается имя трейта (листинг 20.2).

Листинг 20.2. Файл moderator.php

```
<?php
require_once 'backenduser.php';
require_once 'image.php';

class Moderator extends BackendUser
{
    use Image;
}
```

В листинге 20.3 приводится пример использования класса `Moderator`.

Листинг 20.3. Файл moderator.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 20.4 представлен трейт `Author`, в листинге 20.5 — трейт `Seo`.

Листинг 20.4. Файл author.php

```
<?php
trait Author
{
    public $authors;

    public function getAuthor()
    {
        return $this->authors;
    }
    public function setAuthor($authors)
    {
        $this->authors = $authors;
    }
}
```

Листинг 20.5. Файл seo.php

```
<?php
trait Seo
{
    private $seo_title;
    private $seo_description;
    private $seo_keywords;

    public function seo(
        $title = null,
        $description = null,
        $keywords = null)
    {
        $this->seo_title = $title;
        $this->seo_description = $description;
        $this->seo_keywords = $keywords;
    }
    public function title()
    {
        if(!empty($this->seo_title)) {
            return $this->seo_title;
        } else {
            return $this->title;
        }
    }
    public function description()
    {
        return $this->seo_description;
    }
```

```
    public function keywords()
    {
        return $this->seo_keywords;
    }
}
```

Теперь можно включить в класс `Article` трейты `Author` и `Seo` для поддержки META-данных и авторов (листинг 20.6).

Листинг 20.6. 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 20.7 приводится пример использования класса `Article`.

Листинг 20.7. Файл article_use.php

```
<?php
require_once 'article.php';

$obj = new Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

20.2. Трейты и наследование

Трейты встраиваются в цепочку наследования таким образом, что переопределяют методы базового класса, однако методы класса, куда трейт включен, перезаписывает методы трейта (рис. 20.1).

Для демонстрации представленной выше схемы вернемся к базовому классу `User`, от которого наследуется класс `BackendUser`, от которого в конце концов наследуется `Moderator` *(см. главу 19)*. В классе `User` был реализован метод `fullName()` (листинг 20.8).

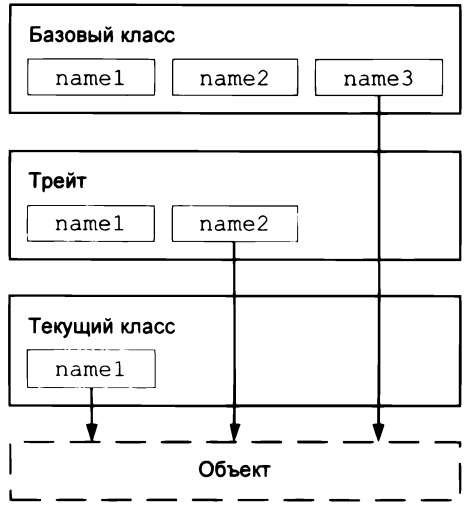


Рис. 20.1. Перегрузка методов в цепочке наследования при использовании трейтов

Листинг 20.8. Файл user.php

```
<?php
class User
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }

    public function fullName() {
        $arr_name = array_filter([$this->first_name, $this->last_name]);
        $full_name = implode(' ', $arr_name);
        return $full_name ? $full_name : 'Анонимный пользователь';
    }
}
```

Создадим демонстрационный трейт `Name`, который будет предоставлять собственную реализацию метода `fullName()` (листинг 20.9). Новый метод `fullName()` использует одноименный родительский метод и добавляет к сформированной строке метку `' (модератор)'`.

Листинг 20.9. Файл name.php

```
<?php
trait Name
{
    public function fullName()
    {
        return parent::fullName() . ' (модератор)';
    }
}
```

Теперь если включить в класс `Moderator` трейт `Name` (листинг 20.10), то модераторы будут помечены (листинг 20.11).

Листинг 20.10. 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 20.11. 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 20.12), он будет экранировать метод в трейте (листинг 20.13). В новом методе вместо метки к имени модератора добавляется звездочка.

Листинг 20.12. 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()
    {
        return parent::fullName() . '*';
    }
}
```

Листинг 20.13. 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*
```

20.3. Разрешение конфликтов

Если в двух трейтах определен метод с одним и тем же именем, возникнет конфликт. Впрочем, его можно разрешить, явно указав, какой из методов следует использовать, в фигурных скобках после оператора `use`. Внутри фигурных скобок можно добавить ключевое слово `insteadof` для указания, какой из методов следует использовать. Кроме этого, допускается применение ключевого слова `as` для указания нового псевдонима для конфликтующего метода.

Организуем конфликт намеренно, для этого создадим два трейта: `Tag` и `Theme`, которые будут включать два одинаковых метода: `tags()` и `themes()` (листинги 20.14 и 20.15).

Листинг 20.14. Файл tag.php

```
<?php
trait Tag
{
    public function tags()
    {
        echo 'Tag::tags<br />';
    }
    public function themes()
    {
        echo 'Tag::themes<br />';
    }
}
```

Листинг 20.15. Файл theme.php

```
<?php
trait Theme
{
    public function tags()
    {
        echo 'Theme::tags<br />';
    }
    public function themes()
    {
        echo 'Theme::themes<br />';
    }
}
```

В листинге 20.16 представлен вариант разрешения конфликта при одновременном включении трейтов `Tag` и `Theme` в класс `Page`.

Листинг 20.16. 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
```

В скрипте, представленном выше, 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 изменять спецификатор доступа метода. В листинге 20.17 приводится альтернативная реализация класса `Page`, в котором конфликтующим методам не только назначаются новые названия, но и изменяется спецификатор доступа на `private`.

Листинг 20.17. 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
```

20.4. Вложенные трейты

Трейты могут включать другие трейты. В листинге 20.5 был представлен трейт `Seo`, который добавляет в класс поддержку метаинформации. Помимо META-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 20.18.

Листинг 20.18. Файл opengraph.php

```
<?php
trait OpenGraph
{
    public $og_title;
    public $og_type;
    public $og_sitename;
    public $og_description;
    public $og_url;
    public $og_image;
}
```

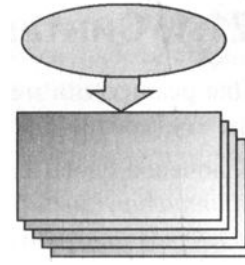
Для того чтобы не включать по отдельности трейты `Seo` и `OpenGraph`, их можно объединить в рамках нового трейта `Meta` (листинг 20.19).

Листинг 20.19. Файл meta.php

```
<?php
require_once 'seo.php';
require_once 'opengraph.php';

trait Meta
{
    use Seo, OpenGraph;
}
```

Задание

Создайте трейт `Auth`, который при наличии переменных класса `$email` и `$password` добавлял бы в класс метод `auth()`, принимающий в качестве аргумента электронный адрес и пароль. В случае успешного сопоставления метод должен помещать в сессию информацию о том, что пользователь аутентифицирован. Другой метод `is_auth()` при этом должен возвращать `true` или `false` в зависимости от того, пройдена аутентификация или нет. Если переменных `$email` и `$password` в классе нет, методы не должны появляться, даже если трейт `Auth` подмешан в класс.



ГЛАВА 21

Исключения

Листинги данной главы
можно найти в подкаталоге exceptions.

Исключения не являются непременным атрибутом объектно-ориентированного подхода, однако сопровождают каждый объектно-ориентированный язык де-факто, т. к. он серьезно изменяет структуру кода. Разработчик отчасти сам становится автором нового языка программирования, создавая классы в рамках предметной области. Структурные изменения кода требуют новых способов обработки ошибок (нештатных ситуаций).

Разработка класса происходит на абстрактном уровне: при этом создается не конкретная область памяти, а лишь определяется поведение объектов. Класс выступает инструментом, который, как и язык программирования, может применяться в совершенно разных областях и приложениях. Обработка нештатных ситуаций, ошибок как кода, так и ввода данных может быть различной для разных приложений: где-то достаточно вывести сообщение при помощи конструкции `echo`; где-то сообщение следует оформить в виде HTML-страницы с дизайном, согласованным с остальными страницами приложения; где-то сообщение об ошибке должно быть помещено в журнал (в файл или базу данных). Предусмотреть заранее формат ошибки невозможно, и любой формат будет сужать область применения класса и возможность его повторного использования.

Выходом из ситуации является разделение кода класса и кода обработки ошибок, что достигается при помощи специального механизма — исключений. Разработчик класса может сгенерировать исключение, а пользователь — обработать его по своему усмотрению.

Разработчики могут проектировать собственные исключения, которые являются классами, при этом генерация исключений сводится к передаче объекта исключения из точки возникновения нештатной ситуации в обработчик исключений.

21.1. Синтаксис исключений

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 следует один или несколько *блоков обработки исключений*, реализуемых с помощью ключевого слова `catch`.

```
<?php
try
{
    // Операторы
    ...
    // Генерация исключений
    throw Выражение_генерации_исключения;
    ...
    // Операторы
}
catch(Exception $exp)
{
    // Блок обработки исключительной ситуации
}
```

Обработчик (или обработчики) всегда располагаются после контролируемого оператором `try` блока кода. Среди операторов контролируемого блока могут быть любые операторы и объявления PHP. Если в теле контролируемого блока исключение генерируется при помощи ключевого слова `throw`, то интерпретатор PHP переходит в `catch`-обработчик. В листинге 21.1 представлен скрипт, в котором случайным образом либо генерируется, либо не генерируется исключение.

Листинг 21.1. Файл throw_rand.php

```
<?php
try
{
    // Генерируем исключение по случайному закону
    if(mt_rand(0, 1)) {
        // Генерация исключения
        throw new Exception();
    }
}
catch(Exception $exp)
{
    // Фраза выводится, если было сгенерировано исключение
    exit('Произошла исключительная ситуация');
}
// Фраза выводится, если исключение не генерировалось
echo 'Штатная работа скрипта';
```

В зависимости от того, возвращает функция mt_rand() 0 или 1, выводится либо фраза "Произошла исключительная ситуация", либо фраза "Штатная работа скрипта". Следует обратить внимание, что если исключение не генерируется, то код в catch-блоке не выполняется. В качестве исключения выступает объект класса Exception, который создается при помощи ключевого new непосредственно при вызове оператора throw. При генерации исключения ключевое слово throw принимает объект класса Exception или производного класса.

21.2. Интерфейс класса *Exception*

Для того чтобы эффективно использовать класс Exception, следует познакомиться с интерфейсом Throwable. В листинге 21.2 представлено определение встроенного класса Exception. Кода, представленного в листинге, не существует, и работать он не будет, нам он требуется лишь для того, чтобы познакомиться с возможностями объектов класса Exception. Жирным шрифтом выделены методы, реализации которых требует интерфейс Throwable.

Листинг 21.2. Файл throwable.php

```
<?php
class Exception implements Throwable
{
    protected $message;   // Сообщение
    private   $string;    // Свойство для __toString
    protected $code;      // Код исключения
    protected $file;      // Файл, в котором произошло исключение
    protected $line;      // Строка, в которой произошло исключение
    private   $trace;     // Трассировка вызовов методов и функций
    private   $previous;  // Предыдущее исключение (вложенные try-блоки)

    public function __construct(
        $message = null,
        $code = 0,
        Exception $previous = null);

    // Запрещает клонировать исключения
    final private function __clone();

    final public  function getMessage();
    final public  function getCode();
    final public  function getFile();
    final public  function getLine();
    final public  function getTrace();
    final public  function getPrevious();
    final public  function getTraceAsString();
    public function __toString();
}
```

Замечание

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 производных классах.

Конструктор класса принимает два необязательных аргумента, которые он записывает в соответствующие свойства объекта. Он также заполняет свойства `$file`, `$line` и `$trace`, соответственно, именем файла, номером строки и стеком вызова.

Как видно из листинга 21.2, конструктор класса `Exception` позволяет инициализировать защищенные переменные `$message` и `$code`, содержимое которых можно получить в обработчике соответственно при помощи методов `getCode()` и `getMessage()` (листинг 21.3).

Листинг 21.3. Файл exception_interface.php

```
<?php
try
{
    $code = mt_rand(0, 1);
    if (!$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 `mt_rand()` число 0 или 1, скрипт из листинга 21.3 может выводить в окно браузера последовательность строк либо для первого оператора `throw`:

```
Исключение 0 : Первая точка входа
в файле D:\main\oop\08\index.php
в строке 8
```

либо для второго:

```
Исключение 1 : Вторая точка входа
в файле D:\main\oop\08\index.php
в строке 13
```

Таким образом можно однозначно определить, в каком контексте было сгенерировано исключение, даже если контролируемый блок `try` содержит несколько вызовов оператора `throw`.

21.3. 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 выводить сообщение в окно браузера невозможно: 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 модифицированный класс `User` *(см. разд. 19.1)*, который реализует специальные методы `__get()` и `__set()` для обращений к закрытым членам. Напомним его упрощенную реализацию (листинг 21.4).

Листинг 21.4. Класс `User`. Файл user.php

```
<?php
class User
{
    private $first_name;
    private $last_name;
    private $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }

    private function __get($index)
    {
        return $this->$index;
    }
```

```
    private function __set($index, $value)
    {
        if (isset($this->$index)) {
            $this->$index = $value;
        }
    }
}
```

Класс содержит четыре закрытых члена: `$last_name` (фамилия), `$first_name` (имя), `$email` (электронный адрес) и `$password` (пароль). Значения закрытых членов устанавливаются при помощи конструктора класса и 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, т. к. обращение к несуществующей переменной класса автоматически приводит к ее созданию.

Обращение к несуществующей переменной в рамках метода `__get()` не так критично, поскольку метод не изменяет состояние объекта.

Однако обращение к несуществующей переменной в методе `__set()` практически наверняка вызвано ошибкой, поэтому крайне полезно сгенерировать исключение, которое сигнализировало бы внешнему коду об ошибке (листинг 21.5).

Листинг 21.5. Файл user_exception.php

```
<?php
class User
{
    private $first_name;
    private $last_name;
    private $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }
```

```
    public function __get($index)
    {
        if (isset($this->$index)) {
            return $this->$index;
        } else {
            throw new Exception("Атрибут $index не существует");
        }
    }
    public function __set($index, $value)
    {
      if (isset($this->$index)) {
          $this->$index = $value;
      }
      else {
          throw new Exception("Атрибут $index не существует");
      }
    }
}
```

Теперь достаточно поместить код, обращающийся к объектам класса `User`, в контролируемый блок, чтобы предотвратить попытку обращения к несуществующим переменным (листинг 21.6).

Листинг 21.6. 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 листинге 21.6 производится попытка осуществить доступ к несуществующей переменной `$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 D:\code\exceptions\user_exception.php
в строке 35
#0 D:\code\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.

21.4. Создание собственных исключений

Класс `Exception` может выступать в качестве базового класса для пользовательских классов исключений. Создадим два новых производных класса исключений:

- `AttributeException` — генерируется при попытке обращения к несуществующему атрибуту класса;
- `PasswordException` — генерируется при попытке прямого обращения к паролю.

На рис. 21.1 представлена схема классов, а в листингах 21.7 и 2.18 приведены их возможные реализации.

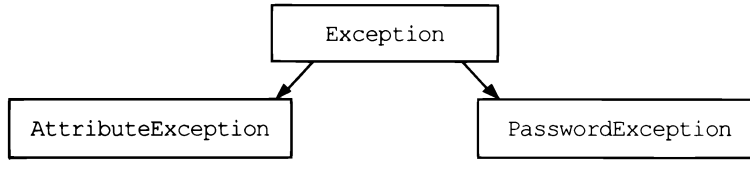


Рис. 21.1. Иерархия наследования исключений

Листинг 21.7. 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 21.8. Файл password_exception.php

```
<?php
class PasswordException extends Exception {
    public function __construct(
        $message = 'Не допускается прямого обращения к атрибуту password'
    )
    {
        parent::__construct($message, 1002);
    }
}
```

Теперь, используя эти два исключения, можно переписать класс `User` (листинг 21.9).

Листинг 21.9. Файл user_own_exceptions.php

```
<?php
require_once 'attribute_exception.php';
require_once 'password_exception.php';

class User
{
    private $first_name;
    private $last_name;
    private $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }

    public function __get($index)
    {
        if($index == 'password') {
```

```
            throw new PasswordException;
        }
        if (isset($this->$index)) {
            return $this->$index;
        } else {
            throw new AttributeException($index);
        }
    }
    public function __set($index, $value)
    {
      if (isset($this->$index)) {
          $this->$index = $value;
      }
      else {
          throw new AttributeException($index);
      }
    }
    public function isPasswordCorrect($password)
    {
        return $this->password == $password;
    }
}
```

Так как обращение к атрибуту `$password` теперь недоступно, класс `User` реализует метод `isPasswordCorrect()`, возвращающий `true`, если ему передан правильный пароль, и в `false`, если пароль ошибочен. В листинге 21.10 приводится пример использования класса `User` с новыми пользовательскими классами.

Листинг 21.10. 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 $exp)
{
    // Блок обработки исключительной ситуации
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
```

```
    echo "в строке {$exp->getLine()}<br />";
    echo "<pre>";
    echo $exp->getTraceAsString();
    echo "</pre>";
}
```

21.5. Перехват исключений производных классов

Как видно из предыдущего раздела, исключения производных типов можно перехватывать при помощи исключений базового типа. Допускается перехват исключений как по их собственному классу, так и по базовому (листинг 21.11).

Листинг 21.11. 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 $exp)
{
    echo "Исключение: {$exp->getMessage()}<br />";
    echo "в файле {$exp->getFile()}<br />";
    echo "в строке {$exp->getLine()}<br />";
}
catch(Exception $exp)
{
    echo "Исключение: {$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 данного типа перехватились бы `Exception`-обработчиком.

21.6. 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 21.12 класс `User` способен генерировать исключения `AttribueException` и `PasswordException`. Обработчик базового типа `Exception` предшествует обработчикам специализированных классов. После вывода имени класса посредством функции `get_class()` происходит повторная инициализация исключения, которое еще раз перехватывается одним из специализированных обработчиков.

Листинг 21.12. 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 $exp)
    {
        echo 'Exception-исключение ' . get_class($exp) . '<br />';
        // Передача исключения далее по каскаду
        throw $exp;
    }
}
catch(AttribueException $exp)
{
    echo 'AttribueException-исключение';
}
catch(PasswordException $exp)
{
    echo 'PasswordException-исключение';
}
```

Результатом работы скрипта из листинга 21.12 будут следующие строки:

```
Exception-исключение PasswordException
PasswordException-исключение
```

Таким образом, какое бы из исключений производного класса не 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 21.13 повторно сгенерированное исключение не будет обработано ни одним из последующих обработчиков.

Листинг 21.13. 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 $exp)
{
    echo 'ExceptionSQL-исключение ' . get_class($exp) . '<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 21.13 может выглядеть следующим образом:

```
ExceptionSQL-исключение PasswordException
Fatal error: Uncaught exception 'PasswordException' with message
'Не допускается прямого обращения к атрибуту password'
in D:\code\exceptions\user_own_exceptions.php:27 Stack trace:
#0 D:\code\exceptions\rethrow_fail.php(11): User->__get('password')
#1 {main} thrown in D:\code\exceptions\user_own_exceptions.php on line 27
```

21.7. Блок *finally*

Инструкция `throw` заставляет программу немедленно покинуть охватывающий `try`-блок, даже если при этом будет необходимо выйти из нескольких промежуточных функций. Если такой выход нежелателен, 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 21.14).

Листинг 21.14. 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 скрипта из листинга 21.14 будет строка

```
Эта строка выводится не всегда
```

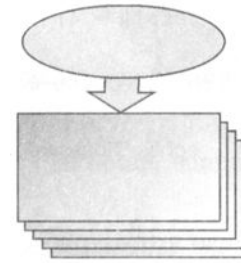
Если же в листинге выше снять комментарий напротив строки с обращением к атрибуту `password`, вывод изменится следующим образом:

```
PasswordException-исключение
Эта строка выводится не всегда
```

Задание

Модифицируйте обработчик исключений в данной главе таким образом, чтобы информация об исключении, времени и месте его возникновения сохранялась в журнальный файл exceptions.log.

ГЛАВА 22



Ошибки

Листинги данной главы
можно найти в подкаталоге errors.

На протяжении предыдущих глав мы уже сталкивались с ошибками, которые генерируются интерпретатором PHP. В текущей главе мы более подробно познакомимся с механизмом сообщений об ошибках, а также способами влияния на него.

22.1. Ошибки и исключения

Избежать ошибок довольно сложно, поэтому PHP предоставляет развитый механизм информирования о возникающих проблемах. В листинге 22.1 приводится пример, в котором осуществляется попытка применить квадратные скобки к строке.

Листинг 22.1. Файл error.php

```
<?php
$str = 'Hello, world!';
$str[] = 4;
```

PHP не может выполнить такую операцию, поэтому останавливает программу и выводит сообщение об ошибке:

```
Fatal error: Uncaught Error: [] operator not supported for strings in
```

Начиная с PHP 7, при возникновении ошибки инициируется исключение специального класса `Error`. Этот класс не наследуется от `Exception`, поэтому обработать ошибку при помощи `catch`-блока для класса `Exception` не получится (листинг 22.2).

Листинг 22.2. Файл error_fail_catch.php

```
<?php
try {
    $str = 'Hello, world!';
```

```
    $str[] = 4;
}
catch(Exception $exp)
{
    echo 'Попытка обработать ошибку при помощи catch-блока';
}
```

Дело в том, что встроенные классы `Exception` и `Error` являются двумя независимыми классами, реализующими один и тот же интерфейс `Throwable` (рис. 22.1)

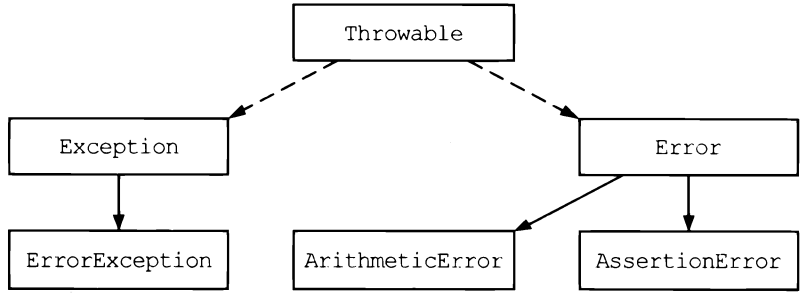


Рис. 22.1. Ошибки и исключения реализуют один и тот же интерфейс `Throwable`

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 22.3).

Листинг 22.3. Файл error_catch.php

```
<?php
try {
    $str = 'Hello, world!';
    $str[] = 4;
}
catch(Error $err)
{
    echo 'Попытка обработать ошибку при помощи catch-блока';
}
```

Результатом выполнения файла из листинга 22.3 будет следующая строка:

```
Попытка обработать ошибку при помощи catch-блока
```

В PHP 7 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;

- `AssertionError` — исключение для функции `assert()`;
- `DivisionByZeroError` — деление на ноль, исключение реализовано таким образом, что его невозможно отловить при помощи конструкции `catch`;
- `ParseError` — исключение, возникающее при ошибке разбора PHP-кода, например, выполняющегося функцией `eval()`;
- `TypeError` — исключение, возникающее при ошибках использования типа. Ситуация, описанная в листинге 22.3, как раз подходит под данный тип исключения.

22.2. Типы уведомлений

PHP имеет несколько уровней обработки ошибок. Это связано с тем, что ошибка в ядре, при разборе скрипта, его интерпретации, пользовательские ошибки относятся к разным логическим уровням, и их смешение не желательно. Такое разделение ошибок на уровни позволяет более гибко настраивать интерпретатор. В табл. 22.1 приводятся список уровней и соответствующие им предопределенные константы, используемые для настройки чувствительности интерпретатора PHP к ошибкам, как в конфигурационном файле php.ini, так и из скриптов.

Таблица 22.1. *Константы, управляющие контролем ошибок*

Бит	Константа PHP	Назначение
1	`E_ERROR`	Критические ошибки во время выполнения PHP-программы. Вызывает 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`	Критические ошибки, которые генерируются ядром PHP
32	`E_CORE_WARNING`	Предупреждения, которые генерируются ядром PHP
64	`E_COMPILE_ERROR`	Критические ошибки на этапе компиляции. Генерируются движком Zend
128	`E_COMPILE_WARNING`	Предупреждения на этапе компиляции. Генерируются движком Zend
256	`E_USER_ERROR`	Критические ошибки, которые генерируются PHP-скриптом при помощи функции `trigger_error()`, рассматриваемой ниже
512	`E_USER_WARNING`	Предупреждение, которое генерируется PHP-скриптом при помощи функции `trigger_error()`, рассматриваемой ниже

Таблица 22.1 (окончание)

Бит	Константа PHP	Назначение
1024	E_USER_NOTICE	Уведомление, которое генерируется пользователем при помощи функции trigger_error(), рассматриваемой ниже
2048	E_STRICT	Различные "рекомендации" PHP по улучшению кода (например, замечания насчет вызова устаревших функций)
4096	E_RECOVERABLE_ERROR	Критические 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 перечисленные флаги, за исключением E_STRICT

При помощи директивы error_reporting в конфигурационном файле php.ini можно указать, сообщения о каких видах ошибок должны выводиться в окно браузера или в журнал сообщений об ошибках.

Кроме того, в конфигурационном файле php.ini допускается использование директив display_errors и error_log. Директива display_errors принимает значение On, если требуется вывод сообщений об ошибках в стандартный поток (в окно браузера), и Off, если сообщения об ошибках не должны появляться на сайте (в этом случае результат работы скрипта выглядит как белая страница). Директива error_log позволяет задать путь к журналу сообщений об ошибках. Выключив display_errors и указав путь к журналу в error_log, можно добиться, чтобы пользователи никогда не видели сообщения об ошибках, но тем не менее иметь возможность получать к ним доступ через журнал.

В листинге 22.4 приводится фрагмент конфигурационного файла php.ini, в котором PHP предписывается выводить все сообщения (E_ALL), кроме замечаний (E_NOTICE).

Замечания

Вместо констант из табл. 22.1 можно использовать их числовые значения, однако это не рекомендуется делать, т. к. они могут изменяться от версии к версии.

Листинг 22.4. Фрагмент конфигурационного файла php.ini. Файл php.ini

```
...
error_reporting  =  E_ALL & ~E_NOTICE
display_errors = On
...
```

Для комбинации нескольких констант можно использовать поразрядные операторы |, ~, !, ^ и &, правила работы с которыми подробно освещаются в *главе 7*.

Директива `error_reporting` устанавливает чувствительность PHP на уровне сервера. Установить собственный уровень для скрипта или Web-приложения можно при помощи одноименной функции, которая имеет следующий синтаксис:

```
int error_reporting([int $level])
```

Если указан необязательный параметр *$level*, функция устанавливает уровень чувствительности интерпретатора PHP. В качестве результата возвращает старый уровень (в виде числового значения).

В листинге 22.5 приводится пример использования функции `error_reporting()` для установки локального уровня чувствительности интерпретатора PHP.

Листинг 22.5. Файл error_reporting.php

```
<?php
error_reporting(E_ALL & ~E_NOTICE);
```

22.3. Пользовательские ошибки

Помимо сообщений, генерируемых интерпретатором PHP, разработчики могут создавать собственные сообщения об ошибках при помощи функции `trigger_error()`, которая имеет следующий синтаксис:

```
bool trigger_error (
  string $error_msg
  [, int $error_type = E_USER_NOTICE])
```

Первый параметр *$error_msg* содержит текст сообщения об ошибке, второй необязательный параметр *$error_type* может задавать уровень ошибки. Функция `trigger_error()` позволяет инициировать только ошибки пользовательского уровня, т. е. `E_USER_ERROR`, `E_USER_WARNING` и `E_USER_NOTICE`. Если задан один из этих уровней, функция возвращает `false`, в противном случае возвращается `true`.

В листинге 22.6 приводится пример использования функции `trigger_error()`. В функции `printAge()` пользовательская ошибка `E_USER_ERROR` инициируется, если аргумент `$age` является отрицательным.

Листинг 22.6. Файл trigger_error.php

```
<?php
function printAge($age)
{
    $age = intval($age);
    if ($age < 0) {
        trigger_error("Функция printAge(): ".
                      "возраст не может быть".
                      " отрицательным", E_USER_ERROR);
    }
```

```
    echo "Возраст составляет: $age";
}

printAge(-10);
// Fatal error: Функция printAge(): возраст не может быть отрицательным
```

22.4. Подавление ошибок

PHP предоставляет специальный оператор @, расположив который перед выражением, можно подавить вывод ошибок и предупреждений в окно браузера.

Если передать скрипту из листинга 22.7 GET-параметр `name`, например `errorcontrol.php?name=hello`, то он отработает штатно; если скрипт вызывать без параметров, будет выдано уведомление "Notice: Undefined index: name".

Листинг 22.7. Файл errorcontrol.php

```
<?php
echo $_GET['name'];
```

Избавиться от этого уведомления можно двумя способами: либо явно проверив наличие переданного GET-параметра (листинг 22.8), либо подавив вывод сообщения об ошибке при помощи оператора @ (листинг 22.9).

Листинг 22.8. Файл errorcontrol_if.php

```
<?php
if(isset($_GET['name'])) {
    echo $_GET['name'];
}
```

Листинг 22.9. Файл errorcontrol.php

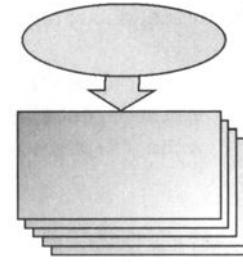
```
<?php
echo @$_GET['name'];
```

Использование оператора @ является признаком плохого тона в программировании, т. к. может скрывать опасные ошибки, которые без уведомлений довольно трудно локализовать. При штатной разработке лучше воспользоваться дополнительной проверкой (см. листинг 22.8). Для подавления вывода сообщений неучтенных ошибок лучше применить директивы `display_errors` и `error_log` *(см. разд. 22.2)*.

Задания

1. По документации с сайта **http://php.net** изучите возможности функции `error_reporting()`, устанавливающей уровень тревожности интерпретатора.
2. По документации изучите возможности функции `debug_backtrace()`, выводящей стек вызова. Реализуйте `catch`-обработчик исключительной ситуации таким образом, чтобы он выводил стек вызова до точки возникновения исключения.
3. По документации изучите возможности функции `set_error_handler()`, предназначенной для перегрузки механизма обработки ошибок. Реализуйте обработчик, который бы сохранял сообщения об ошибках в файл errors.txt.

ГЛАВА 23



Пространство имен

Листинги данной главы
можно найти в подкаталоге namespace.

Помимо классов, PHP предоставляет еще один способ организации проекта — пространство имен. Оно позволяет организовать код проекта в 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 в разных пространствах имен. Это позволяет избегать конфликтов со сторонними библиотеками, а также облегчает поиск и загрузку файлов. Разрабатывая собственные библиотеки и выбирая уникальное пространство имен для своих классов, можно быть уверенным, что названия классов, функций и констант не будут конфликтовать с какими-то другими именами.

23.1. Создание пространства имен

Пространство имен — это имеющий имя фрагмент программы, содержащий функции, переменные, константы и другие именованные сущности. Для объявления пространства имен используется ключевое слово `namespace`, после которого следует имя пространства (листинг 23.1).

> ***Примечание***
>
> Названия пространств имен `PHP` и `php` являются зарезервированными и не могут использоваться в пользовательском коде.

Листинг 23.1. Объявление пространства имен. Файл namespace.php

```
<?php
namespace SelfPhp;

const VERSION = '1.0';
```

```
function debug($obj)
{
    echo "<pre>";
    print_r($obj);
    echo "</pre>";
}

class User
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }
}
```

В пространстве имен может находиться любой PHP-код, однако действие пространства имен распространяется только на классы (включая абстрактные и трейты), интерфейсы, функции и константы.

Для того чтобы обратиться к функции и классу из пространства имен `SelfPhp`, потребуется включить файл namespace.php и обращаться к элементам пространства имен, используя квалифицированные имена `SelfPhp\VERSION`, `SelfPhp\debug` и `SelfPhp\User` (листинг 23.2).

Листинг 23.2. Использование пространства имен. Файл namespace_use.php

```
<?php
require_once 'namespace.php';

$user = new SelfPhp\User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo 'Версия ' . SelfPhp\VERSION . '<br />';
SelfPhp\debug($user);
```

Результатом выполнения скрипта из листинга 23.2 будут следующие строки:

```
Версия 1.0

SelfPhp\User Object
(
    [first_name] => Игорь
    [last_name] => Симдянов
    [email] => igorsimdyanov@gmail.com
    [password:SelfPhp\User:private] => password
)
```

Конструкция `namespace` должна располагаться в файле первой, до любых объявлений, в том числе до тега `<?php` (листинг 23.3).

Листинг 23.3. Файл namespace_fail.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>HTML-форма с флажком</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
namespace SelfPhp;

const VERSION = '2.0';

function debug($obj)
{
    echo "<pre>";
    print_r($obj);
    echo "</pre>";
}
?>
</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 примере выше можно поместить константы в пространство имен `SelfPhp\constants`, функции — в `SelfPhp\functions`, а классы — в `SelfPhp\classes` (листинг 23.4).

Листинг 23.4. Файл namespaces.php

```
<?php
namespace SelfPhp\constants;

const VERSION = '3.0';

namespace SelfPhp\functions;

function debug($obj)
{
    echo "<pre>";
    print_r($obj);
    echo "</pre>";
}

namespace SelfPhp\classes;

class User
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
        $email,
        $password,
        $first_name = null,
        $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }
}
```

Если необходимо использовать несколько пространств имен в одном файле, рекомендуется применять альтернативный синтаксис, в котором принадлежащие пространству имен классы и функции заключаются в фигурные скобки (листинг 23.5).

Листинг 23.5. Файл namespaces_alt.php

```
<?php
namespace SelfPhp\constants
{
    const VERSION = '3.0';
}
```

```
namespace SelfPhp\functions
{
    function debug($obj)
    {
        echo "<pre>";
        print_r($obj);
        echo "</pre>";
    }
}

namespace SelfPhp\classes
{
    class User
    {
        public $first_name;
        public $last_name;
        public $email;
        private $password;

        public function __construct(
             $email,
             $password,
             $first_name = null,
             $last_name = null)
        {
            $this->first_name = $first_name;
            $this->last_name  = $last_name;
            $this->email      = $email;
            $this->password   = $password;
        }
    }
}
```

В листинге 23.6 приводится пример использования новых пространств имен.

Листинг 23.6. Файл namespaces_alt_use.php

```
<?php
require_once 'namespaces_alt.php';

$user = new SelfPhp\classes\User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo 'Версия ' . SelfPhp\constants\VERSION . '<br />';
SelfPhp\functions\debug($user);
```

Результатом выполнения скрипта из листинга 23.6 будут следующие строки:

```
Версия 3.0

SelfPhp\classes\User Object
(
    [first_name] => Игорь
    [last_name] => Симдянов
    [email] => igorsimdyanov@gmail.com
    [password:SelfPhp\classes\User:private] => password
)
```

23.2. Иерархия пространств имен

Из предыдущего раздела видно, что пространство имен очень напоминает файловую систему. Используя обратный слеш, можно 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`. Однако, с учетом пространства имен, полное квалифицированное имя все равно остается слишком длинным.

В случае файловой системы имеется возможность использовать длинные полные абсолютные и короткие относительные пути. Аналогичные правила предусмотрены и в отношении пространства имен.

В файле, где объявлено пространство имен, можно ссылаться на его элементы через относительные ссылки. В листинге 23.7 объявляется пространство `SelfPhp`, в связи с чем вместо полного имени `SelfPhp\classes\User` можно использовать относительное имя `classes\User`.

Листинг 23.7. Файл relative.php

```
<?php
namespace SelfPhp;

require_once 'namespaces_alt.php';

$user = new classes\User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo 'Версия ' . constants\VERSION . '<br />';
functions\debug($user);
```

Так же, как и в файловой системе, можно указать абсолютное имя, которое начинается с ведущего слеша: `\SelfPhp\classes\User`.

23.3. Глобальное пространство имен

В скриптах, где объявлено пространство имен, для обращения к стандартным функциям, например к строковой функции `strlen()`, может потребоваться воспользоваться абсолютным именем `\strlen()`, чтобы сообщить PHP, что `strlen()` является функцией глобального пространства имен, а не `\Self` (листинг 23.8). Безымянное пространство имен, обозначаемое слешем, называется *глобальным пространством имен*.

Листинг 23.8. Файл absolute.php

```
<?php
namespace SelfPhp;

function strlen($str)
{
    return count(str_split($str));
}

// Это SelfPhp\strlen
echo strlen('Hello world!') . '<br />';

// Стандартная функция strlen()
echo \strlen('Hello world!') . '<br />';
```

23.4. Текущее пространство имен

В файле, где объявлено пространство имен, мы можем использовать ключевое слово `namespace` для ссылки на *текущее пространство имен*. В листинге 23.9 все три вызова функции `strlen()` являются эквивалентными.

Листинг 23.9. Файл namespace_key.php

```
<?php
namespace SelfPhp;

function strlen($str)
{
  return count(str_split($str));
}

// Это SelfPhp\strlen
echo \SelfPhp\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 23.10).

Листинг 23.10. Файл namespace_const.php

```
<?php
namespace SelfPhp;

echo __NAMESPACE__; // SelfPhp
```

В случае глобального пространства имен константа возвращает пустую строку.

23.5. Импортирование

Объявление пространства имен в начале файла при помощи ключевого слова namespace помогает сократить имена только текущего пространства имен. В реальности приходится работать с несколькими различными пространствами. Здесь на помощь приходит механизм *импортирования*, который при помощи ключевого слова use позволяет создавать псевдонимы (листинг 23.11).

Листинг 23.11. Импортирование. Файл use.php

```
<?php
require_once 'namespaces_alt.php';

use SelfPhp\constants as constants;
use SelfPhp\functions as functions;
use SelfPhp\classes\User as User;

$user = new User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo 'Версия ' . constants\VERSION . '<br />';
functions\debug($user);
```

Как видно из листинга 23.11, допускается создание псевдонимов как для отдельных элементов пространства имен (класс User), так и для подпространств (functions). Если имя псевдонима, которое мы указываем после ключевого слова as, совпадает с последним элементом, то as можно не указывать (листинг 23.12).

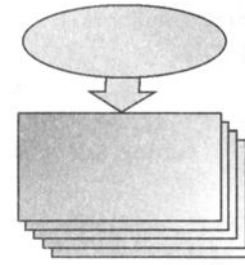
Листинг 23.12. Импортирование. Файл use.php

```
<?php
require_once 'namespaces_alt.php';

use SelfPhp\constants;
use SelfPhp\functions;
use SelfPhp\classes\User;

$user = new User(
    'igorsimdyanov@gmail.com',
    'password',
    'Игорь',
    'Симдянов');

echo 'Версия ' . constants\VERSION . '<br />';
functions\debug($user);
```

Задания

1. Допустим, имеется иерархия классов материалов (`News`, `Post`), авторов, создавших эти материалы (`Author`), изображения (`Image`), постраничная навигация (`Pager`). Какие пространства имен вы использовали бы для этих классов?
2. По документации на сайте **http://php.net** познакомьтесь с функциями управления буфером вывода (ob-функциями). Создайте объектно-ориентированную оболочку для этого набора функций. Расположите классы в подходящих пространствах имен.
3. По документации на сайте **http://php.net** познакомьтесь с расширением `curl`, которое предоставляет возможности для низкоуровневого сетевого взаимодействия. Создайте объектно-ориентированную оболочку для функций этого расширения, включающую как минимум класс для установки соединения, класс для доступа к результатам запроса, исключения для обработки ошибок. Расположите классы в подходящих пространствах имен.

ГЛАВА 24



Автозагрузка

Листинги данной главы
можно найти в подкаталоге autoload.

Обычно класс оформляется в виде отдельного файла, который вставляется в месте использования при помощи конструкции `require_once()`. Если используется большое количество классов, то в начале скрипта выстраивается целая вереница объявлений, 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.

24.1. Функция *__autoload()*

PHP предоставляет разработчику специальную функцию `__autoload()`, которая позволяет задать путь к каталогу с классами и автоматически подключать классы при обращении к ним в теле программы.

Для демонстрации работы функции создадим папку SelfPhp, в которой разместим два уже знакомых нам по *главе 20* трейта: `Author` (листинг 24.1) и `Seo` (листинг 24.2).

Трейты и классы размещаются в пространстве имен `SelfPhp`. 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, в котором объявлен класс или трейт.

Листинг 24.1. Трейт SelfPhp\Author. Файл SelfPhp/Author.php

```
<?php
namespace SelfPhp;
```

```
trait Author
{
    public $authors;

    public function getAuthor()
    {
        return $this->authors;
    }
    public function setAuthor($authors)
    {
        $this->authors = $authors;
    }
}
```

Листинг 24.2. Трейт SelfPhp\Seo. Файл SelfPhp/Seo.php

```
<?php
namespace SelfPhp;

trait Seo
{
    private $seo_title;
    private $seo_description;
    private $seo_keywords;

    public function seo(
        $title = null,
        $description = null,
        $keywords = null)
    {
        $this->seo_title = $title;
        $this->seo_description = $description;
        $this->seo_keywords = $keywords;
    }
    public function title()
    {
        if(!empty($this->seo_title)) {
            return $this->seo_title;
        } else {
            return $this->title;
        }
    }
    public function description()
    {
        return $this->seo_description;
    }
```

```
    public function keywords()
    {
        return $this->seo_keywords;
    }
}
```

В папку SelfPhp поместим класс `Article`, который использует указанные выше трейты (листинг 24.3).

Листинг 24.3. Класс SelfPhp\Article. Файл Self/Article.php

```
<?php
namespace SelfPhp;

class Article {
    use Author, Seo;
}
```

Теперь, чтобы объявить объект класса `Article`, нам потребуется подключить все вышеопределенные файлы (листинг 24.4).

Листинг 24.4. Файл wrong.php

```
<?php
require_once(__DIR__ . "/SelfPhp/Author.php");
require_once(__DIR__ . "/SelfPhp/Seo.php");
require_once(__DIR__ . "/SelfPhp/Article.php");

$obj = new \SelfPhp\Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

В больших промышленных системах количество классов может достигать сотен и тысяч, подключать их вручную в начале каждого сценария или даже в отдельном выделенном скрипте довольно утомительно. Для того чтобы автоматизировать этот процесс, используется функция `__autoload()`(листинг 24.5).

> ***Замечание***
>
> Функция `__autoload()` не является методом класса. Это независимая функция, которую PHP-разработчик может перегружать.

Листинг 24.5. Файл autoload.php

```
<?php
function __autoload($classname)
{
    $classname = str_replace('\\', '/', $classname);
    require_once(__DIR__ . "/$classname.php");
}

$obj = new \SelfPhp\Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

Функция `__autoload()` принимает единственный аргумент — строку с именем класса. Внутри функции должна быть реализация загрузки класса при помощи одной из директив: `require_once`, `include_once`, `require` или `include`. За счет того, что выбранное пространство имен `SelfPhp` и названия классов совпадают с физическими каталогами и файлами, достаточно одной директивы `require_once`, чтобы загрузить любой файл из пространства имен `SelfPhp`.

Вызывать функцию нет необходимости, она автоматически вызывается в момент, когда интерпретатор встречает имя еще незагруженного класса. Таким образом будут загружены только те классы, которые реально используются.

24.2. Функция *spl_autoload_register()*

В современной практике функция `__autoload()` практически не используется. Причина заключается в том, что она предоставляет лишь один вариант загрузки, который может быть довольно сложным по реализации, но вряд ли учтет все особенности сторонних библиотек. Вместо этого применяется функция `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.

Функция `spl_autoload_register()` позволяет зарегистрировать очередь функций-автозагрузчиков. Таким образом, каждая библиотека может реализовать собственный механизм автозагрузки и загрузить его при помощи функции `spl_autoload_register()`. Функция полностью изменяет механизм автозагрузки, поэтому если вы реализуете функцию `__autoload()`, ее потребуется явно зарегистрировать в `spl_autoload_register()`.

```
bool spl_autoload_register (
  [ callable $autoload_function
```

```
  [, bool $throw = true
  [, bool $prepend = false ]]] )
```

В качестве первого параметра `$autoload_function` функция принимает имя функции обратного вызова или реализацию в виде анонимной функции. Параметр `$throw` определяет, генерируется ли исключение, 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, в общем случае функция может не принимать ни одного аргумента. Пример из листинга 24.5 можно переписать с использованием функции `spl_autoload_register()` (листинг 24.6).

Листинг 24.6. Файл spl_autoload_register.php

```
<?php
spl_autoload_register();

$obj = new \SelfPhp\Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

В качестве аргумента функции часто удобно использовать анонимные функции (листинг 24.7).

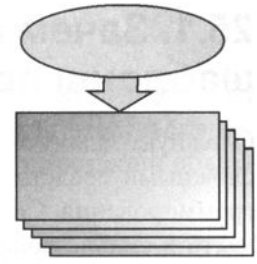
Листинг 24.7. Файл anonim.php

```
<?php
spl_autoload_register(function($classname){
    $classname = str_replace('\\', '/', $classname);
    require_once(__DIR__ . "/$classname.php");
});

$obj = new \SelfPhp\Article(
    'Заголовок',
    'Содержимое');

$obj->setAuthor(['Максим Кузнецов', 'Игорь Симдянов']);
$obj->seo('SEO-заголовок');
echo $obj->title(); // SEO-заголовок
```

Задание

Найдите стандарт PSR-4 и ознакомьтесь с его содержимым.

ГЛАВА 25



Шаблоны проектирования

Листинги данной главы
можно найти в подкаталоге patterns.

Сфера разработки программного обеспечения является, с одной стороны, довольно молодой инженерной специальностью, а с другой, находящейся в условиях многолетнего взрывного роста. Причиной такого роста является прогресс в микроэлектронике, который свыше пятидесяти лет определялся законом Мура. Согласно закону, количество транзисторов, размещаемых на интегральной схеме, удваивается каждые два года.

Экспоненциальный рост возможностей компьютеров перевернул жизнь всей цивилизации на планете. Однако в сфере разработки программного обеспечения образовался огромный разрыв между возможностями аппаратного обеспечения и средствами разработки. Дело в том, что совершенствование языков программирования, инструментария для разработки, да и просто человеческого сознания не поспевало 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 IT связаны с тем, что отрасль предоставляет слишком много возможностей для дальнейшего совершенствования языков программирования и средств разработки.

На примере других инженерных дисциплин можно интерполировать развитие отрасли ПО. 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 наблюдать и участвовать в создании таких типовых решений, которые в сообществе принято называть *шаблонами* или *паттернами проектирования*.

> ***Замечание***
>
> Данную главу ни в коем случае нельзя рассматривать как исчерпывающее руководство по шаблонам проектирования, скорее, как введение в область. Детальному разбору, классификации шаблонов посвящены десятки книг. Для более детального изучения следует обратиться к ним.

25.1. Зачем нужны шаблоны проектирования?

Сложную задачу можно решить либо сложно простыми средствами, либо просто сложными средствами. Сложным средством в разработке ПО 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, т. к. он уже содержит интегралы и дифференциалы, как самостоятельные объекты языка. Разработка такого решения на PHP или любом другом универсальном языке потребует гораздо больше усилий, 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.

> ***Замечание***
>
> 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 10 из 24 предыдущих глав были посвящены именно им.

Однако знать объектно-ориентированные возможности языка недостаточно, необходимо умение их применять на практике. Вариантов использования очень много, 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, мы не сможем рассмотреть даже их малую часть. В последующих разделах будут рассмотрены наиболее известные паттерны и возможные варианты их использования.

25.2. Одиночка (Singleton)

Можно создать любое количество объектов класса. Однако в ряде случаев объекты должны существовать в единственном экземпляре, например, если это объект настроек 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 — довольно нетривиальная задача. В первую очередь необходимо исключить возможность создания нескольких объектов при помощи конструкции `new`. Добиться этого можно, объявив конструктор закрытым. 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, чтобы предотвратить создание копии этим способом, потребуется перегрузить специальный метод `__clone()`, объявив его закрытым. В листинге 25.1 представлена возможная реализация паттерна в виде класса `\Singleton\Settings`.

Листинг 25.1. Паттерн Одиночка. Файл Singleton/Settings.php

```
<?php
namespace Singleton;

final class Settings
{
    private static $_object = null;
    private $_settings;

    private function __construct()
    {
        $_settings = [];
    }

    private function __clone() {}

    public static function get()
    {
        if (is_null(self::$_object)) {
            self::$_object = new self();
        }

        return self::$_object;
    }
```

```
    public function __get($key)
    {
        if(array_key_exists($key, $this->_settings)) {
            return $this->_settings[$key];
        } else {
            return null;
        }
    }

    public function __set($key, $value)
    {
        $this->_settings[$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 25.2 приводится пример использования полученного класса.

Листинг 25.2. Файл settings_use.php

```
<?php
spl_autoload_register();

use Singleton\Settings;

Settings::get()->items_per_page = 20;
echo Settings::get()->items_per_page; // 20
```

25.3. Фабричный метод (Factory Method)

Рассмотренный в предыдущем разделе паттерн Одиночка относится к классу *порождающих паттернов*. Основная задача паттернов данного 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. Страница со списком пользователей будет 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.

Пусть для доступа к информации о пользователях и страницах предоставляется следующая система URL:

- /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 списка выше и создавал бы объект соответствующего класса.

Класс `Router` является базовым классом для всех представляемых классов (рис. 25.1). Классы коллекций `Users` и `Pages` наследуются от `Router` не напрямую, а через промежуточный класс `Collection`, который содержит общий для коллекций обслуживающий код.

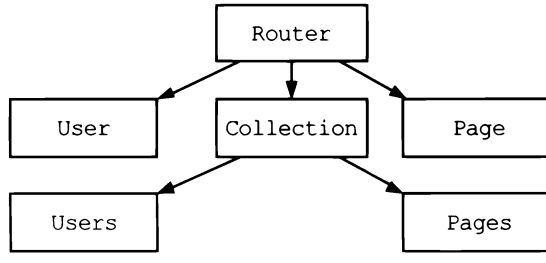


Рис. 25.1. Иерархия классов

Разработку иерархии классов разумно начать с классов `User` и `Page` (листинги 25.3 и 25.4).

Листинг 25.3. Файл Factory/Models/User.php

```
<?php
namespace Factory\Models;

class User extends \Factory\Router
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
        $email,
        $password,
        $first_name = null,
        $last_name = null)
```

```
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }

    public function render()
    {
        $name = implode(' ', [$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 25.4. Файл Factory/Models/Page.php

```
<?php
namespace Factory\Models;

class Page extends \Factory\Router
{
    public $title;
    public $content;

    public function __construct(
         $title,
         $content)
    {
        $this->title   = $title;
        $this->content = $content;
    }

    public function render()
    {
        return '<h1>' . htmlspecialchars($this->title) . '</h1>' .
               '<p>' . htmlspecialchars($this->content) . '</p>';
    }
}
```

Коллекции `Users` и `Pages` наследуются от базового класса `Collection`, который хранит коллекцию в переменной класса `$collection`, а методом `render()` 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 25.5).

> ***Замечание***
>
> PHP предоставляет стандартный интерфейс `ArrayAccess`, реализация которого позволит получить доступ к коллекции, как к обычному массиву PHP, при помощи квадратных скобок и обходом через цикл `foreach`.

Листинг 25.5. Файл Factory/Models/Collection.php

```
<?php
namespace Factory\Models;

class Collection extends \Factory\Router
{
    public $collection;

    public function __construct($collection = null)
    {
        $this->collection = $collection;
    }

    public function render()
    {
        return array_map(
                    function($item){
                        return $item->render();
                    },
                    $this->collection);
    }
}
```

Теперь можно унаследовать классы-коллекции `Users` и `Pages` от `Collection` (листинги 25.6 и 25.7).

Листинг 25.6. Файл Factory/Models/Users.php

```
<?php
namespace Factory\Models;

class Users extends Collection
{
    public $users;

    public function __construct($users = null)
    {
        if(is_null($users))
```

```
        {
            $users = [
                new User(
                    'makkuz@yandex.ru',
                    'password',
                    'Максим',
                    'Кузнецов'),
                new User(
                    'igorsimdyanov@gmail.com',
                    'password',
                    'Игорь',
                    'Симдянов')
            ];
        }
        parent::__construct($users);
    }

    public function render()
    {
        return implode('<br />', parent::render());
    }
}
```

Листинг 25.7. Файл Factory/Models/Pages.php

```
<?php
namespace Factory\Models;

class Pages extends Collection
{
    public $pages;

    public function __construct($pages = null)
    {
        if(is_null($pages))
        {
            $pages = [
                new Page(
                    'Первая статья',
                    'Содержимое первой статьи'),
                new Page(
                    'Вторая статья',
                    'Содержимое второй статьи')
            ];
        }
        parent::__construct($pages);
    }
```

```
    public function render()
    {
        return implode('', parent::render());
    }
}
```

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.

Теперь, когда базовые классы готовы, можно реализовать класс `Router`, реализующий фабричный метод `parse()` (листинг 25.8).

Листинг 25.8. Файл Factory/Router.php

```
<?php
namespace Factory;

abstract class Router
{
    public static function parse($url)
    {
        $arr = explode('/', $url);
        $class = 'Factory\\Models\\' . ucfirst($arr[0]);
        $id = count($arr) > 1 ? $arr[1] : null;
        $obj = new $class();
        if(is_null($id)) {
            return $obj;
        } else {
            return $obj->collection[$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`. Из строки `"users/1"` в конечном итоге получаются класс `\Factory\Models\Users` и идентификатор 1. В листинге 25.9 приводится пример использования класса `Router`.

Листинг 25.9. Файл factory_use.php

```
<?php
spl_autoload_register();
```

```
use Factory\Router;

$obj = Router::parse('users');
echo $obj->render();
```

Результатом выполнения кода из листинга 25.9 будут следующие строки:

```
Максим Кузнецов (makkuz@yandex.ru)
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)
```

25.4. Модель-Представление-Контроллер

Представленная в предыдущем разделе система вывода пользователей и статей имеет существенные ограничения. Представление статей и пользователей определяется 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 представления существует множество паттернов, наиболее популярным из которых является Модель-Представление-Контроллер (Model-View-Controller, MVC).

Основная задача паттерна — разделить содержимое документов (модель), их обработку (контроллер) и их отображение (представление). Как видно из названия, паттерн состоит из трех компонентов (рис. 25.2).

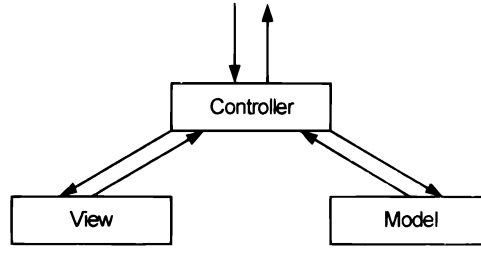


Рис. 25.2. Высокоуровневая схема паттерна MVC

Класс модели предоставляет данные, это может быть класс, обслуживающий базу данных, JSON-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 представление на основании полученных

данных. После этого отправляет результат клиенту. Обращение к базе данных напрямую из представления или хранение представления в базе данных не допускается.

> ***Замечание***
>
> Представленная на рис. 25.2 схема реализации MVC, получившая широкое распространение в Web-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 Web-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 25.10) и класс-коллекция `Users` (листинг 25.11).

Листинг 25.10. Файл MVC/Models/User.php

```
<?php
namespace MVC\Models;

class User
{
    public $first_name;
    public $last_name;
    public $email;
    private $password;

    public function __construct(
         $email,
         $password,
         $first_name = null,
         $last_name = null)
    {
        $this->first_name = $first_name;
        $this->last_name  = $last_name;
        $this->email      = $email;
        $this->password   = $password;
    }
}
```

Листинг 25.11. Файл MVC/Models/Users.php

```
<?php
namespace MVC\Models;

class Users
{
    public $collection;

    public function __construct($users = null)
    {
        if(is_null($users))
        {
            $users = [
                new User(
                    'makkuz@yandex.ru',
                    'password',
                    'Максим',
                    'Кузнецов'),
                new User(
                    'igorsimdyanov@gmail.com',
                    'password',
                    'Игорь',
                    'Симдянов')
            ];
        }
        $this->collection = $users;
    }
}
```

В отличие от предыдущего раздела, в классах отсутствует метод `render()`, т. к. за отображение будет отвечать подсистема представлений. Разделение функций представления и отображения данных позволяет избавиться от фабричного метода при получении моделей.

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 25.12) два разных класса: `HtmlView` (листинг 25.13) и `RssView` (листинг 25.14), которые учитывали бы компоновку каждой отдельной страницы (рис. 25.3).

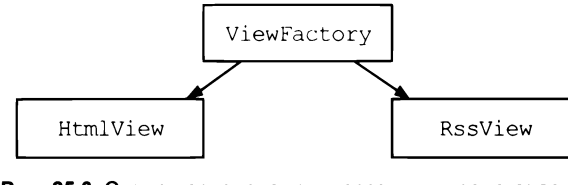


Рис. 25.3. Схема наследования классов для представления

Листинг 25.12. Файл MVC/Views/ViewFactory.php

```
<?php
namespace MVC\Views;

abstract class ViewFactory
{
    abstract public function render();

    public static function create($type, $class, $decorator)
    {
        $class = 'MVC\\Views\\' . ucfirst($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 25.13. 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($decorator)
    {
        $this->replacements = [
            '{{{title}}}' => $decorator->title(),
            '{{{body}}}' => $decorator->body()
        ];
    }
```

```
    public function render()
    {
        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 25.14. Файл MVC/Views/RssView.php

```
<?php
namespace MVC\Views;

class RssView extends ViewFactory
{
    const LAYOUT = <<<HTML
      <?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>
HTML;

    protected $replacements;

    public function __construct($decorator)
    {
        $this->replacements = [
            '{{{title}}}' => $decorator->title(),
            '{{{items}}}' => $decorator->items()
        ];
    }

    public function render()
    {
        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 выше двумя классами-представлениями. Однако для каждой страницы можно унаследовать свой класс `UserHtmlView`, `UserRssView`, `UsersHtmlView` и `UsersRssView` (рис. 25.4) и расширить их функционал.

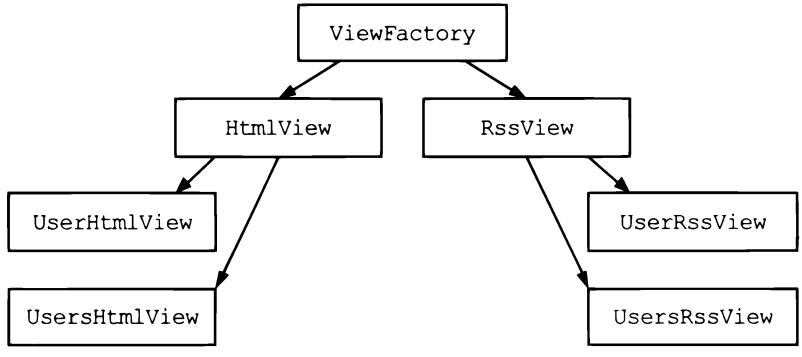


Рис. 25.4. Альтернативная схема наследования классов для представления

Так как URL, на которые отзывается система, уже довольно сложные, можно выделить их обработку в отдельный класс `Router` (листинг 25.15).

Листинг 25.15. Файл MVC/Controllers/Router.php

```
<?php
namespace MVC\Controllers;

class Router
{
    public $model;
    public $ext;
    public $id;

    public static function parse($path)
    {
        list($path, $ext) = explode('.', $path);
        $arr = explode('/', $path);
```

```
        return new self($arr[0], $ext, count($arr) > 1 ? $arr[1] : null);
    }

    private function __construct($model, $ext = null, $id = null)
    {
        $this->model = $model;
        $this->ext = $ext;
        $this->id = $id;
    }
}
```

Класс `Router` реализован по принципу фабричного метода, за исключением того, что мы не предполагаем, что от него будет кто-то наследоваться. Конструктор класса закрыт, а за создание объекта ответственен статический метод `parse()`, который принимает строку `$path`, разбирает и заполняет переменные 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 25.16).

Листинг 25.16. Файл MVC/Decorators/DecoratorFactory.php

```
<?php
namespace MVC\Decorators;

abstract class DecoratorFactory
{
    public static function create($type, $class, $model)
    {
        $class = 'MVC\\Decorators\\' . ucfirst($class) . 'Decorator';
        return new $class($model);
    }

    abstract public function title();
    abstract public function body();
    abstract public function items();
}
```

В листинге 25.17 представлена возможная реализация декоратора модели `User`.

Листинг 25.17. Файл MVC/Decorators/UserDecorator.php

```
<?php
namespace MVC\Decorators;

class UserDecorator extends DecoratorFactory
{
    public $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function title()
    {
       return implode(' ', [$this->user->first_name, $this->user->last_name]);
    }

    public function body()
    {
        return '<strong>'.htmlspecialchars($this->title()).'</strong> '.
               '(' . htmlspecialchars($this->user->email) . ')';

    }

    public function items()
    {
        return '<item>'.
               '<title>'.htmlspecialchars($this->title()).'</title>' .
               '<email>'.htmlspecialchars($this->user->email).'</email>'.
               '</item>';
    }
}
```

В декораторе коллекции `UsersDecorator` перед тем, как использовать коллекцию, каждый элемент коллекции необходимо преобразовать в объект класса `UserDecorator` (листинг 25.18). Это можно сделать либо один раз в конструкторе, либо, как это демонстрируется ниже, преобразовывать коллекцию всякий раз, когда она требуется, при помощи специального метода `collection_render()`, в основе которого лежит обход массива при помощи стандартной функции `array_map()`.

Листинг 25.18. Файл MVC/Decorators/UserDecorator.php

```
<?php
namespace MVC\Decorators;
```

```
class UsersDecorator extends DecoratorFactory
{
    public $users;

    public function __construct($users)
    {
        $this->users = $users;
    }

    public function title()
    {
       return 'Пользователи';
    }

    public function collection_render($call, $separator = '<br />')
    {
        return implode(
                    $separator,
                    array_map($call, $this->users->collection)
                );
    }

    public function body()
    {
        return $this->collection_render(
            function($item){
                $decorated_item = new UserDecorator($item);
                return $decorated_item->body();
            });
    }

    public function items()
    {
        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, каждый ресурс вроде пользователей, статей, новостей и т. п. должен обслуживаться собственным контроллером. Однако у нас только один ресурс — пользователи, поэтому контроллер у нас также присутствует в единственном экземпляре (листинг 25.19). Именно поэтому у нас контроллер может позволить себе вызов подсистемы роутинга, обычно класс `Router` вызывает тот или иной контроллер системы для обработки запроса.

Листинг 25.19. Файл MVC/Controllers/Controller.php

```
<?php
namespace MVC\Controllers;

class Controller
{
    public $path;
    public $router;
    public $model;

    public function __construct($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()
    {
        $class = get_class($this->model);
        $class = substr($class, strrpos($class, '\\') + 1);
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $this->router->ext,
            $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 25.20 приводится пример использования полученного мини-приложения.

Листинг 25.20. 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 25.20 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>Максим Кузнецов</title>
      <email>makkuz@yandex.ru</email>
    </item>
    <item>
      <title>Игорь Симдянов</title>
      <email>igorsimdyanov@gmail.com</email>
    </item>
  </channel>
</rss>
```

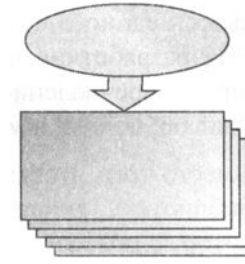
Если заменить `'users.rss'` на `'users/1.html'`, будет получено HTML-представление для единичного пользователя:

```
<!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>
```

Задания

1. Переработайте модель, представленную в *разд. 25.4*, с учетом замечаний в главе таким образом, чтобы она поддерживала несколько типов моделей: пользователи (`User`, `Users`) и статьи (`Page`, `Pages`).
2. Найдите в Сети описание паттерна Репозиторий, попробуйте его реализовать средствами PHP.
3. Найдите в Сети описание паттерна Стратегия, попробуйте его реализовать средствами PHP.

ГЛАВА 26



Компоненты

Листинги данной главы
можно найти в подкаталоге composer.

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, Web-приложение собирается из нескольких совместимых друг с другом *компонентов*.

Управление компонентами осуществляется при помощи утилиты Composer, которая подробно описывается в данной главе.

26.1. Composer: управление компонентами

Долгое время PHP-разработчики сосредотачивались вокруг одной известной системы управления контента (CMS) вроде Drupal или вокруг известного фреймворка вроде Yii или Symfony. Любая система управления контентом или фреймворк — это шаг в определенном направлении. Если вам по пути: вы экономите время и усилия. Если философия и приемы разработки фреймворка идут в разрез 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 окажитесь в той же самой точке в полном одиночестве в окружении неподдерживаемого кода.

Во многих языках имеются флагманские фреймворки для Web-разработки: Ruby on Rails в Ruby, Django в Python, Spring в Java. В отличие от них, в PHP не сформиро-

валось 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: Yii, Symfony, Zend, Phalcon, Laravel и многие другие.

Вместо того чтобы конкурировать друг с другом и разрабатывать несовместимые библиотеки, разработчики объединились и выработали стандарты кодирования, которые позволяют разрабатывать совместимые компоненты. Их можно использовать в любом современном фреймворке. Более того, можно вообще не использовать фреймворки и собирать свои приложения из подходящих компонентов.

Компоненты, или библиотеки — это коллекция связанных классов, интерфейсов и трейтов, которые решают определенную задачу. Например, 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. Аналогичные утилиты можно обнаружить в любом современном языке программирования, связанном с Web: 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 Mac OS X.

26.2. Установка Composer

Существует множество способов установки Composer, мы рассмотрим установку этой утилиты для трех наиболее популярных операционных систем: Windows, Mac OS X и Linux (дистрибутив Ubuntu).

26.2.1. Установка в Windows

Установить Composer в операционной системе Windows можно двумя способами: при помощи автоматического установщика и вручную. В первом случае его следует загрузить по ссылке **https://getcomposer.org/Composer-Setup.exe** и запустить на выполнение (рис. 26.1).

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, кроме того, пропишет путь к утилите в переменной окружения `PATH`, сделав доступной команду `composer` в любой точке файловой системы.

Рис. 26.1. Мастер установки Composer

Замечание

Для успешной установки Composer необходимо установить расширение OpenSSL, для этого в конфигурационном файле php.ini следует снять комментарий с директивы `extension=php_openssl.dll`.

После успешной установки в командной строке можно обратиться к утилите `composer`, например, запросить ее версию:

```
> composer --version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

В случае если автоматическая установка невозможна, можно 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
```

Замечание

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.

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 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Для того чтобы создать команду `composer`, достаточно создать файл composer.bat следующего содержания:

```
@php "%~dp0composer.phar" %*
```

После чего файлы composer.phar и composer.bat нужно поместить в каталог, прописанный в переменной окружения PATH *(см. главу 2)*.

26.2.2. Установка в Mac OS X

Установить Composer в Mac OS X проще всего, воспользовавшись менеджером пакетов Homebrew. Для этого достаточно выполнить команду:

```
$ sudo brew install composer
Composer version 1.0.0-apha10
```

26.2.3. Установка в Ubuntu

Для установки в Linux можно воспользоваться командой, описанной в *разд. 26.2.1*:

```
$ 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 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Для того чтобы установить Composer глобально, PHAR-архив следует переименовать в удобную форму, например просто в composer. При этом файлу должны быть выставлены права доступа на выполнение, например 0755. Для того 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:

```
$ php composer -version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

26.3. 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. 26.2).

Выбирая компонент для решения задачи, обычно ориентируются на количество загрузок, звездочек (положительных оценок). Если вы только начинаете знакомиться с компонентами, хорошим путеводителем может стать список, представленный на странице **https://github.com/ziadoz/awesome-php**.

Рис. 26.2. Поиск компонентов на сайте Packagist

26.4. 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`.

Для того чтобы воспользоваться пакетом, потребуется создать файл composer.json, в котором описываются зависимости приложения. Файл composer.json является конфигурационным, и его следует располагать в корне проекта.

В листинге 26.1 представлен вариант файла composer.json, который при помощи ключевого слова `require` сообщает о необходимости поставить компонент `monolog/monolog` версии не ниже 1.17.0.

Листинг 26.1. Файл monolog/composer.json

```
{
    "require": {
        "monolog/monolog": "1.17.*"
    }
}
```

Значение `1.17.*` указывает на необходимость установки версии пакета в диапазоне от 1.17.0 (включительно) до 1.18. Следующая запись аналогична представленному в листинге 26.1 варианту со звездочкой:

```
"monolog/monolog": ">=1.17.0 <1.18.0"
```

Помимо операторов >, >=, <= и <, можно воспользоваться диапазонами, например:

```
"monolog/monolog": "1.17.0 - 2.0.0"
```

Вдобавок предусмотрен специальный оператор ~, который позволяет задать интервалы для текущей версии. Запись `~1.17` эквивалентна `>=1.17 <2.0.0`, а запись `~1.17.2` эквивалентна `>=1.17.2 <1.18.0`.

Можно указать конкретную версию, например `1.17.2`. 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 не пугает, можно пользоваться диапазонами.

> ***Замечание***
>
> Дальнейшее повествование будет вестись в предположении, что в командной строке доступна команда `composer`. Если Composer не установлен глобально, эту команду следует заменять на `php composer.phar`.

Для того чтобы установить пакет, в папке с конфигурационным файлом composer.json следует выполнить команду `composer install`:

```
$ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing psr/log (1.0.0)
    Downloading: 100%

  - Installing monolog/monolog (1.17.2)
    Downloading: 100%

monolog/monolog suggests installing graylog2/gelf-php (Allow sending log
messages to a GrayLog2 server)
monolog/monolog suggests installing raven/raven (Allow sending log messages
to a Sentry server)
monolog/monolog suggests installing doctrine/couchdb (Allow sending log
messages to a CouchDB server)
monolog/monolog suggests installing ruflin/elastica (Allow sending log messages
to an Elastic Search server)
monolog/monolog suggests installing videlalvaro/php-amqplib (Allow sending log
messages to an AMQP server using php-amqplib)
monolog/monolog suggests installing ext-amqp (Allow sending log messages
to an AMQP server (1.0+ required))
monolog/monolog suggests installing ext-mongo (Allow sending log messages
to a MongoDB server)
monolog/monolog suggests installing aws/aws-sdk-php (Allow sending log
messagesto AWS services like DynamoDB)
monolog/monolog suggests installing rollbar/rollbar (Allow sending log
messagesto Rollbar)
monolog/monolog suggests installing php-console/php-console (Allow sending log
messages to Google Chrome)
Writing lock file
Generating autoload files
```

Как видно из отчета, были установлены два компонента: `monolog/monolog` версии `1.17.2` и `psr/log` версии `1.0.0`, от которого зависит компонент Monolog.

После выполнения команды `composer install` в папке 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 из мониторинга. Это связано с тем, что, 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.

Если вышла новая версия библиотеки и ею необходимо воспользоваться, 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.

26.5. Использование компонента

В *главе 24* рассматривался 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 директивы `require` или `require_once` предоставляет доступ ко всем компонентам, загруженным посредством Composer (листинг 26.2).

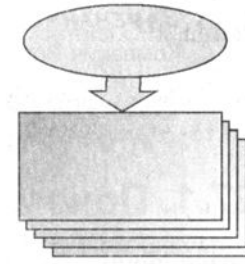
Листинг 26.2. Файл monolog/index.php

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$log = new Logger('name');
$handler = new StreamHandler('app.log', Logger::WARNING);
$log->pushHandler($handler);
$log->addWarning('Предупреждение');
```

Задания

1. Подключите компонент `phpmailer/phpmailer` и отправьте при помощи его почтовое сообщение.
2. Познакомьтесь с шаблонами HAML, подключите компонент `tiutalk/haml`. Переработайте HTML-представление MVC-системы из *главы 25* с использованием шаблонов HAML.
3. Познакомьтесь с фреймворком Yii 2. Создайте проект, выводящий "Hello, world!" с его помощью. Для инициализации проекта используйте Composer.
4. По документации с сайта **http://php.net** познакомьтесь с PHAR-архивами, допускающими создание исполняемых PHP-архивов. Познакомьтесь с классом `Phar`. Создайте PHAR-архив, выводящий фразу "Hello, world!".
5. По документации с сайта **http://php.net** познакомьтесь с возможностями расширения GDLib. Подключите компонент `imagine/imagine` и создайте при помощи его изображение с надписью "Hello world!".

ГЛАВА 27



База данных PostgreSQL

Листинги данной главы
можно найти в подкаталоге postgresql.

До текущей главы в разрабатываемых приложениях не хватало одной детали — возможности сохранения результатов работы Web-приложения. Во время работы пользователь может регистрироваться, добавлять комментарии и записи на сайте, возвращаться к ним для последующего редактирования. Введенную им информацию нужно хранить длительное время. Переменные, массивы и объекты для этого не подходят, т. к. они расположены в оперативной памяти и после того, как скрипт прекращает свою работу, уничтожаются и более не доступны для использования.

Чтобы воспользоваться результатами в следующий раз, потребуется сохранить их на жесткий диск сервера: в файл или 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 двумя или более потоками. Чтобы избежать конфликтов при записи в один файл, одновременные обращения приходится обрабатывать через очередь, создавать систему поиска, обновления, удаления записей, т. е. строить довольно сложную прослойку между обычным файлом и приложением.

В настоящее время из Web-разработчиков никто не использует обычные файлы и не строит системы управления ими. В качестве хранилища данных используются готовые решения — *системы управления базами данных* (СУБД). Подавляющее большинство современных Web-приложений, при помощи какой бы технологии они не были разработаны, используют ту или иную СУБД, которые для краткости мы будем называть просто *базами данных*.

В этой главе мы рассмотрим свободную и бесплатную базу данных PostgreSQL. Данную главу ни в коем случае нельзя рассматривать как полноценное описание всех возможностей PostgreSQL. Только официальная документация занимает объем в пять раз больше, чем книга, которую вы держите в руках.

Замечание

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/postgrespro/9.6/**.

27.1. Почему PostgreSQL?

Традиционно сообщество PHP-разработчиков ориентировалось на СУБД MySQL. В предыдущих изданиях книги рассматривалась именно MySQL, т. к. эта СУБД долго удерживала лидерство, особенно в PHP-сообществе.

В настоящий момент MySQL развивается медленнее, чем раньше, и имеет ряд сложно разрешимых проблем с производительностью, полноценной поддержкой кодировки UTF-8, да и просто является старой базой данных, во многом ориентированной на работу в условиях дефицита оперативной памяти, т. е. ориентированной 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. До некоторой степени именно Oracle является главным игроком на рынке баз данных.

После серии перепродаж MySQL сначала принадлежала корпорации Sun, а затем после ее банкротства — корпорации Oracle. За несколько лет до того, как главная СУБД с открытым кодом MySQL стала собственностью Oracle, последней была приобретена компания InnoDB. Эта компания являлась разработчиком наиболее продвинутого движка баз данных для MySQL (СУБД имеет ядро, к которому могут подключаться движки баз данных, разработанных сторонними компаниями). InnoDB в свою очередь разрабатывалась как бесплатная альтернатива СУБД Oracle.

В результате главному коммерческому игроку на рынке баз данных достались ключевые компоненты MySQL, которые кроме ощутимой конкуренции являлись альтернативой одной из старых реализаций Oracle.

Хотя закрытие проекта MySQL не состоялось, как многие опасались, и как Oracle делал с ранними приобретениями, СУБД оказалась в двусмысленном положении. С одной стороны, компания Oracle совершенно не заинтересована в разработке бесплатного конкурента, к тому же реализующего возможности не первой свежести основного продукта. С другой стороны, сообщество разработчиков MySQL не доверяет корпорации, и многие покинули проект. Результатом стало образование нескольких альтернативных центров разработки (MariaDB, Percona).

Развитие MySQL сильно затормозилось. Сообщество переключилось на альтернативный свободный проект — PostgreSQL, долго оставшейся второй по известности открытой базы данных.

PostgreSQL разработана в RedHat — легендарной компании, одной из первых начавшей выпускать дистрибутивы Linux. PostgreSQL задумывалась как СУБД, не-

сколько нарушающая 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 предтечей современных NoSQL баз данных.

Очень долго PostgreSQL оставалась непопулярной в силу большой доли MySQL, отсутствия реализации для Windows. Однако, благодаря драматической ситуации с MySQL, переносу на Windows, динамическому развитию, введению поддержки современных стандартов SQL, СУБД вышла на первые позиции. В настоящий момент при создании нового проекта в качестве свободной СУБД чаще выбирается PostgreSQL, нежели MySQL.

> ***Замечание***
>
> Роль и вес MySQL по-прежнему довольно велики в PHP-разработке. Существует множество уже разработанных приложений с использованием MySQL. Мы не рассматриваем здесь MySQL, однако, если она необходима вам по работе, вы можете познакомиться с приемами работы с ней из PHP по книге "PHP 7" Котерова Дмитрия и Симдянова Игоря[1].

27.2. Установка PostgreSQL

PostgreSQL — это в первую очередь сервер, к которому могут обращаться клиенты, например консольный клиент psql, или скрипт, написанный на PHP *(см. главу 28)*.

Рассмотрим установку PostgreSQL для трех наиболее популярных операционных систем: Windows, Mac OS X и Linux (дистрибутив Ubuntu).

27.2.1. Установка в Windows

Установку PostgreSQL в операционной системе Windows легко выполнить при помощи автоматического установщика, загрузить который можно с сайта **https://www.bigsql.org/postgresql/installers.jsp**. Выбрав дистрибутив для Windows, следует его загрузить и запустить на выполнение (рис. 27.1).

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.

После установки в командной строке будут доступны все стандартные утилиты PostgreSQL. Так как 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
```

[1] Котеров В. Д., Симдянов И. В. PHP 7. — СПб.: БХВ-Петербург, 2016. — 1088 с.: ил. — (В подлиннике).

Рис. 27.1. Мастер установки PostgreSQL

Как видно из примера выше, `psql` сообщает о том, что пользователь с именем Igor не существует. Лучше всего его создать при помощи утилиты `createuser`, указав в параметре `-U`, что создание осуществляется от пользователя postgres:

```
C:\> createuser -U postgres Igor
```

Кроме этого для текущего пользователя Igor лучше сразу задать базу данных по умолчанию при помощи утилиты `createdb`:

```
C:\> createdb -U postgres Igor
```

> ***Замечание***
>
> 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 при помощи команды `chcp 1251`.

Теперь в диалоговый режим утилиты `psql` можно войти без указания дополнительных параметров:

```
C:\> psql
psql (9.6.2)
Введите "help", чтобы получить справку.

Igor=> SELECT CURRENT_USER;
 current_user
--------------
 Igor
(1 строка)

Igor=> \q
```

В примере выше, после того как был осуществлен вход в диалоговый режим `psql`, был запрошен текущий пользователь при помощи запроса `SELECT CURRENT_USER`, а затем осуществлен выход из утилиты посредством `psql`-команды `\q`.

27.2.2. Установка в Mac OS X

Установить PostgreSQL в Mac OS X 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 (9.6.2)
Type "help" for help.

igor=# SELECT CURRENT_USER;
 current_user
--------------
 igor
(1 row)

igor=# \q
```

Для того 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
```

27.2.3. Установка в 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 при помощи команды `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
```

27.3. 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 ими. СУБД — это программный комплекс, который выполняет все низкоуровневые операции по работе с файлами базы данных.

Язык программирования, с помощью которого клиент (оператор или программный код) общается с СУБД, называется SQL (Structured Query Language, язык структурированных запросов).

Таким образом, для получения информации из базы данных необходимо направить ей запрос, созданный с использованием SQL, результатом выполнения которого будет результирующая таблица.

Несмотря на то, что SQL называется "языком запросов", в настоящее время он представляет собой нечто большее. С помощью SQL осуществляется реализация всех возможностей, которые предоставляются пользователям разработчиками 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/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, Python, Ruby 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 SQL строго стандартизирован, существует множество его диалектов: каждая база данных реализует собственный вариант со своими особенностями и конструкциями, 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.

Например, СУБД PostgreSQL значительно расширяет возможности традиционных СУБД за счет наследования таблиц и коллекционных типов данных (массивы, json-объекты, пользовательские типы данных).

Тем не менее, в PostgreSQL доступны и традиционные реляционные отношения, на которых мы сосредоточимся в данной главе.

Главной особенностью реляционных СУБД является хранение данных в *таблицах*, состоящих из строк и столбцов. На пересечении каждого столбца и строки находится только одно значение. При этом строки и столбцы не равнозначны, у каждого столбца есть свое имя и тип (например, целочисленный столбец `id` или текстовый столбец `name`). При этом столбцы располагаются в определенном порядке, который задается при создании таблицы.

На рис. 27.2 приведен пример таблицы `users`. Каждая строка этой таблицы представляет собой одного пользователя, для описания которого используются следующие поля:

- `id` — уникальный номер пользователя;
- `first_name` — имя пользователя;
- `last_name` — его фамилия.

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.

Таблица `users`

id	first_name	last_name
1	Максим	Кузнецов
2	Игорь	Симдянов
3	Сергей	Голышев
4	Дмитрий	Котеров
5	Алексей	Костарев

Строка

Столбец

Рис. 27.2. Таблица реляционной базы данных

На рис. 27.3 приведена структура базы данных, состоящей из двух таблиц: `users` и `articles`. Таблица `users` определяет авторов статей, а таблица `articles` содержит сами статьи. Одному автору или статье соответствует одна строка таблицы.

Последнее поле таблицы `articles` с именем `user_id` содержит значение поля `id` таблицы `users`. По этому значению можно однозначно определить автора статьи. Таким образом, таблицы оказываются связанными друг с другом. Связь условна и может отсутствовать. Проявляется она только в результате специальных запросов, однако именно благодаря наличию связей такая форма организации информации получила название реляционной (связанной отношениями).

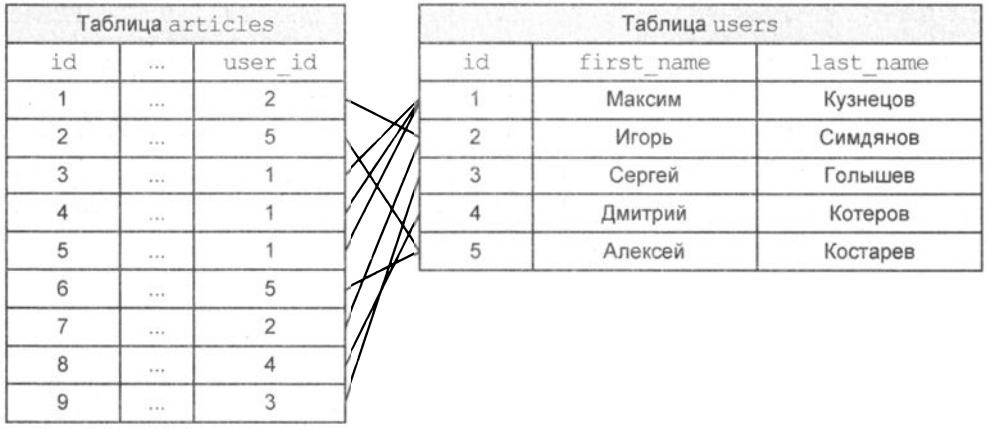


Таблица articles

id	...	user_id
1	...	2
2	...	5
3	...	1
4	...	1
5	...	1
6	...	5
7	...	2
8	...	4
9	...	3

Таблица users

id	first_name	last_name
1	Максим	Кузнецов
2	Игорь	Симдянов
3	Сергей	Голышев
4	Дмитрий	Котеров
5	Алексей	Костарев

Рис. 27.3. Связанные друг с другом таблицы

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, можно объединить в одну таблицу. И, наконец, это говорит о том, что результаты запроса сами могут стать предметом дальнейших запросов.

27.4. Первичные ключи

Строки в реляционной базе данных неупорядочены: в таблице нет "первой", "последней" и "сорок третьей" строк. Для того чтобы выбрать в таблице конкретную строку, 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 таб-

лицы обладает своим уникальным идентификатором. На рис. 27.4 в качестве первичного ключа таблиц `articles` и `users` выступают поля `id`.

По способу задания первичных ключей различают *логические* (естественные) и *суррогатные* (искусственные) ключи.

Таблица articles

id	...	user_id
1	...	2
2	...	5
3	...	1
4	...	1
5	...	1
6	...	5
7	...	2
8	...	4
9	...	3

Таблица users

id	first_name	last_name
1	Максим	Кузнецов
2	Игорь	Симдянов
3	Сергей	Голышев
4	Дмитрий	Котеров
5	Алексей	Костарев

Рис. 27.4. Связь между таблицами `articles` и `users`

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. А вот дата рождения уже не уникальна, поэтому соответствующее 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.

> ***Замечание***
>
> Даже если в базе данных содержится естественный первичный ключ, лучше использовать суррогатные ключи, поскольку их применение позволяет абстрагировать первичный ключ от реальных данных. Это облегчает работу с таблицами, т. к. суррогатные ключи не связаны ни с какими фактическими данными таблицы.

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 уже упомянутую выше базу данных (рис. 27.4). Каждая таблица 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 `user_id` таблицы `articles` принимает значения из столбца `articles.id`. Поле `user_id` таблицы `articles` называют *внешним* или *вторичным ключом*. Внешний ключ сокращенно обозначают как FK (foreign key).

27.5. Управление базами данных

Для того 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""
```

Замечание

По умолчанию утилита `psql` выдает результаты в виде таблицы. Чтобы развернуть каждую запись в отдельную подтаблицу, как это демонстрируется выше, достаточно выполнить команду `\x`.

Как видно из отчета выше, 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 *разд. 27.2* при создании системной базы данных `igor` при помощи утилиты `createdb` на самом деле создавалась копия шаблонной базы данных `template1`. Поэтому если в нее внести изменения, то все новые базы данных будут их содержать.

Помимо утилиты `createdb` создать базу данных можно при помощи оператора `CREATE DATABASE`. В листинге 27.1 создается новая база данных с именем `catalog`.

Листинг 27.1. Файл database_create.sql

```
igor=# CREATE DATABASE catalog TEMPLATE template0;
```

При помощи необязательного ключевого слова `TEMPLATE` можно уточнить, какая база данных выступает шаблоном для вновь создаваемой.

Для удаления базы данных можно либо воспользоваться консольной утилитой `dropdb`, либо выполнить в диалоговом режиме клиента `psql` оператор `DROP DATABASE` (листинг 27.2).

Листинг 27.2. Файл database_drop.sql

```
igor=# DROP DATABASE catalog;
```

При входе в консольного клиента `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 видно, 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)
```

27.6. Управление таблицами

Как правило, Web-приложению выделяется одна база данных, где создается несколько таблиц.

Для создания таблиц используется оператор `CREATE TABLE`. В листинге 27.3 приводится пример создания таблицы `users`, содержащей три столбца: `id` — первичный ключ, `first_name` — имя пользователя, `last_name` — фамилия пользователя.

Листинг 27.3. Файл users.sql

```
CREATE TABLE users (
  id SERIAL,
  first_name VARCHAR(40),
  last_name VARCHAR(40)
);
```

Определение столбца в операторе `CREATE TABLE` содержит имя, например `first_name`, и его тип, в данном случае `VARCHAR(40)`.

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, цифра в скобках указывает максимально возможную длину строки.

В листинге 27.4 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 27.4. 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` используются 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, в текущей главе рассматриваются лишь те, которые встречаются в примерах. За полным перечнем и детальным описанием типов следует обращаться к документации PostgreSQL.

Для того 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
```

Оператор `DROP TABLE` предназначен для удаления одной или нескольких таблиц. В листинге 27.5 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.

> ***Замечание***
>
> Для изменения существующей таблицы предназначен оператор `ALTER TABLE`, ознакомиться с синтаксисом которого можно в документации PostgreSQL.

Листинг 27.5. Файл tables_drop.sql

```
catalog=# DROP TABLE IF EXISTS articles, users;
```

27.7. Вставка записей в таблицу

Вставить запись в таблицу можно при помощи оператора `INSERT`. В листинге 27.6 приводятся различные варианты вставки новых записей в таблицу пользователей `users`.

Листинг 27.6. Файл users_insert.sql

```
INSERT INTO users VALUES (DEFAULT, 'Игорь', 'Симдянов');
INSERT INTO users (last_name, first_name) VALUES ('Кузнецов', 'Максим');
```

Как видно из листинга 27.6, после ключевого слова INSERT INTO указывается имя таблицы, в которую вставляется запись, после ключевого слова VALUES в круглых скобках следуют вставляемые значения. При помощи дополнительных круглых скобок перед VALUES можно указать количество и порядок следования столбцов. Для автоматического заполнения поля id типа SERIAL можно указать значение DEFAULT или просто исключить его из списка вставляемых значений. При каждой вставке нового значения поле id автоматически будет получать максимальное значение в таблице, увеличенное на единицу.

Убедиться в том, что значения успешно вставлены, легко при помощи оператора SELECT (листинг 27.7), возможности которого более детально будут рассмотрены далее в главе.

Листинг 27.7. Файл users_select.sql

```
catalog=# SELECT * FROM users;
 id | first_name | last_name
----+------------+-----------
  1 | Игорь      | Симдянов
  2 | Максим     | Кузнецов
```

Оператор INSERT имеет многострочную форму, позволяющую за один раз вставить сразу несколько значений (листинг 27.8).

Листинг 27.8. Файл users_multi_insert.sql

```
INSERT INTO
  users (first_name, last_name)
VALUES
  ('Максим', 'Кузнецов'),
  ('Игорь', 'Симдянов');
```

27.8. Удаление записей

Время от времени возникает задача удаления записей из базы данных, для которой предназначены следующие два оператора:

- DELETE — удаление всех или части записей из таблицы;
- TRUNCATE TABLE — удаление всех записей из таблицы.

В простейшей форме операторы DELETE и TRUNCATE TABLE полностью эквивалентны (листинг 27.9).

Листинг 27.9. Файл users_truncate.sql

```
TRUNCATE TABLE users;
DELETE FROM users;
```

Однако, в отличие от TRUNCATE, оператор DELETE допускает использование ключевых слов WHERE и LIMIT. При помощи WHERE можно задать условие, при котором запись удаляется. В листинге 27.10 из таблицы users удаляются лишь пользователи с именем 'Игорь'.

Листинг 27.10. Файл users_delete_where.sql

```
catalog=# DELETE FROM users WHERE first_name = 'Игорь';
DELETE 1
catalog=# SELECT * FROM users;
 id | first_name | last_name
----+------------+-----------
  9 | Максим     | Кузнецов
```

При помощи ключевого слова LIMIT можно указать количество удаляемых за один раз значений (листинг 27.11).

Листинг 27.11. Файл users_delete_limit.sql

```
DELETE FROM users LIMIT 1;
```

Допускается и совместное использование WHERE и LIMIT. При этом порядок расположения ключевых слов имеет значение, WHERE всегда предшествует LIMIT.

27.9. Обновление записей

Для обновления записей предназначен оператор UPDATE, сразу после которого указывается имя таблицы. После чего при помощи ключевого слова SET задаются столбцы, которые подвергаются обновлению. Так же как и в случае оператора DELETE, необязательное ключевое слово WHERE позволяет задать критерий отбора записей. В листинге 27.12 приводится пример замены русского имени пользователя 'Игорь' на английский эквивалент 'Igor'.

Листинг 27.12. Файл users_update.sql

```
UPDATE
  users
SET
  first_name = 'Igor'
WHERE
  first_name = 'Игорь';
```

27.10. Выборка записей

Для выборки записей из таблицы используется уже упоминавшийся ранее оператор SELECT. Сразу после ключевого слова SELECT указывается список столбцов через запятую, значения которых окажутся в результирующей таблице (листинг 27.13).

Листинг 27.13. Файл users_select_field.sql

```
catalog=# SELECT first_name, last_name FROM users;
 first_name | last_name
------------+-----------
 Максим     | Кузнецов
 Игорь      | Симдянов
```

Если требуется вывести все столбцы, перечислять их имена после ключевого слова SELECT не обязательно, вместо списка можно использовать символ *, обозначающий все столбцы (листинг 27.14).

Листинг 27.14. Файл users_select.sql

```
catalog=# SELECT * FROM users;
 id | first_name | last_name
----+------------+-----------
 13 | Максим     | Кузнецов
 14 | Игорь      | Симдянов
```

Как видно из примера выше, в результирующей таблице имя столбца совпадает с именем в исходной таблице. При помощи ключевого слова AS можно переименовывать столбцы. Это особенно полезно при извлечении результатов в PHP-скриптах. В листинге 27.15 столбец с именами пользователей переименовывается в name, а с фамилией в family.

Листинг 27.15. Файл users_as.sql

```
catalog=# SELECT
catalog-#   first_name AS name,
catalog-#   last_name AS family
catalog-# FROM
catalog-#   users;
  name  |  family
--------+----------
 Максим | Кузнецов
 Игорь  | Симдянов
```

Оператор SELECT также допускает использование ключевых слов WHERE и LIMIT. В дополнение к ним разрешено использование ключевого слова ORDER BY, позво-

ляющего задать сортировку записей по столбцу. В листинге 27.16 приводится пример сортировки записей по имени.

Листинг 27.16. Файл users_order.sql

```
catalog=# SELECT
catalog-#   first_name,
catalog-#   last_name
catalog-# FROM
catalog-#   users
catalog-# ORDER BY
catalog-#   first_name;
 first_name | last_name
------------+-----------
 Игорь      | Симдянов
 Максим     | Кузнецов
```

Для того чтобы отсортировать записи в обратном порядке, после названия столбца в конструкции `ORDER BY` следует добавить ключевое слово `DESC` (листинг 27.17).

Листинг 27.17. Файл users_order_desc.sql

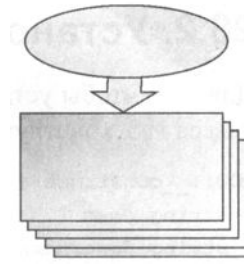
```
catalog=# SELECT
catalog-#   first_name,
catalog-#   last_name
catalog-# FROM
catalog-#   users
catalog-# ORDER BY
catalog-#   first_name DESC;
 first_name | last_name
------------+-----------
 Максим     | Кузнецов
 Игорь      | Симдянов
```

Задания

1. По документации к PostgreSQL ознакомьтесь с операторами `BETWEEN`, `IN` и `LIKE`.
2. Создайте таблицу `monthes`, предназначенную для хранения названий месяцев. Заполните ее. Составьте запрос, который извлекает текущий месяц. Составьте запрос, который извлекает случайную запись из таблицы `monthes`.
3. По документации познакомьтесь с ключевым словом `JOIN`, предназначенным для составления многотабличных запросов. Чем отличается `JOIN` от `LEFT JOIN`?
4. Заполните базу таблицы `users` и `articles` из текущей главы тестовыми данными. Извлеките только тех авторов, у которых есть хотя бы одна статья. Извлеките авторов без статей.

5. По документации ознакомьтесь с ключевыми словами `DISTINCT` и `GROUP BY`.
6. По документации познакомьтесь с функциями `COUNT()`, `MIN()` и `MAX()` для поиска количества записей, минимального и максимального значений. Составьте запрос, который бы извлекал список авторов `users` и количество их статей `articles`.
7. Создайте собственную хранимую функцию, которая выводит фразу "Hello, world!".
8. Установите СУБД MySQL и выполните примеры и задания главы в ней.

ГЛАВА 28



PHP-расширение PDO

Листинги данной главы
можно найти в подкаталоге pdo.

В предыдущей главе была рассмотрена СУБД PostgreSQL, которая в современных приложениях часто играет роль долговременного хранилища данных. Текущая глава будет посвящена взаимодействию PHP-скриптов и PostgreSQL. Для решения этой задачи в PHP предусмотрено универсальное расширение — PDO.

До введения расширения PDO для каждой базы данных использовалось собственное уникальное расширение. Функции, предоставляемые таким расширением, имели префикс, сигнализирующий об используемой базе данных, например `pg_query()`, `mysql_query()`, `mssql_query()`. Переход с одной базы данных на другую требовал множество изменений по всему коду. Новая объектно-ориентированная библиотека PDO предоставляет одинаковый интерфейс для всех типов баз данных, значительно облегчая переход от одной базы данных к другой.

28.1. Настройка расширения PDO

Расширение PDO предоставляет интерфейс для доступа к базам данных из PHP-скриптов. Помимо самого расширения, для его корректного взаимодействия с СУБД необходим драйвер конкретной базы данных, 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 случае Ubuntu, установить расширение можно при помощи команды:

```
$ sudo apt-get install php7.0-pgsql
```

Для Mac OS X можно воспользоваться менеджером пакетов Homebrew:

```
$ brew install php70-pdo-pgsql
```

28.2. Установка соединения с базой данных

Для того чтобы установить соединение с базой данных, необходимо создать объект класса PDO, конструктор которого имеет следующий синтаксис:

```
PDO::__construct (
  string $dsn [,
  string $username [,
  string $password [,
  array $options]]])
```

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', 'igor', '');
```

В примере осуществляется обращение к серверу, расположенному на локальной машине localhost, выбирается база данных catalog, в качестве имени PostgreSQL-пользователя передается igor с пустым паролем. Если соединение в силу причин не может быть установлено, генерируется исключение PDOException, которое может быть перехвачено блоком try/catch (листинг 28.1).

Листинг 28.1. Соединение с базой данных. Файл connect_db.php

```
<?php
try {
    $pdo = new PDO('pgsql:host=localhost;dbname=catalog', 'igor', '');
} catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

Файл connect_db.php будет использоваться во всех следующих примерах для установки соединения. Для этого он будет подключаться в начале скрипта при помощи конструкции require_once. В качестве иллюстрации можно извлечь текущую версию PostgreSQL-сервера, для чего можно воспользоваться встроенной PostgreSQL-функцией VERSION() (листинг 28.2).

Листинг 28.2. Соединение с базой данных. 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 отличаться, однако в полученной строке обязательно будет присутствовать версия сервера. Например, под управлением Mac OS X можно получить следующий результат:

```
PostgreSQL 9.6.2 on x86_64-apple-darwin15.6.0, compiled by Apple LLVM version
8.0.0 (clang-800.0.42.1), 64-bit
```

Для выполнения запроса используется объект соединения `$pdo` и его метод `query()`, в качестве результата метод возвращает объект класса `PDOStatement`, который сохраняется в переменной `$version`. Класс предоставляет интерфейс для доступа к результирующей таблице. В примере выше эта таблица имеет одну строку с единственным значением. В последующих разделах порядок работы с запросами и результирующими таблицами будет рассмотрен более подробно.

28.3. 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 int PDO::exec(string $statement)`

В листинге 28.3 приводится пример использования метода `exec()` для создания таблицы `users`.

Листинг 28.3. Использование метода `PDO::exec()`. Файл exec.php

```
<?php
require_once('connect_db.php');

$query = 'CREATE TABLE users (
            id SERIAL,
            first_name VARCHAR(40),
            last_name VARCHAR(40)
          )';
```

```
$count = $pdo->exec($query);
if ($count !== false)
    echo 'Таблица создана успешно';
else {
    echo 'Не удалось создать таблицу';
    echo '<pre>';
    print_r($pdo->errorInfo());
    echo '<pre>';
}
```

Примечательно, что скрипт лишь один раз выполнится успешно, все последующие разы он будет возвращать сообщение о неудачной попытке создания таблицы, т. к. она уже существует. Выяснить, почему последняя операция не завершилась успешно, можно при помощи метода `errorInfo()`, возвращающего массив с кодом ошибки и текстовым сообщением от базы данных.

```
Не удалось создать таблицу
Array
(
    [0] => 42P07
    [1] => 7
    [2] => ERROR:  relation "users" already exists
)
```

Следует обязательно проверить, не возвращает ли СУБД PostgreSQL ошибку выполнения последнего запроса, поскольку интерпретатор PHP никак не сигнализирует об ошибках синтаксиса PostgreSQL: PHP-скрипт и SQL-запросы выполняются в разных процессах, и PHP-интерпретатор не имеет сведений об успешности или неудаче выполнения SQL-запроса.

28.4. Обработка ошибок

Осуществлять проверку после каждого выполненного запроса, как было показано в предыдущем разделе, не очень удобно. Лучше воспользоваться механизмом исключений, как это демонстрировалось ранее при установке соединения (см. листинг 28.1). Создадим ошибочный запрос и попробуем обработать возникновение ошибки (листинг 28.4).

Листинг 28.4. 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: "Fatal error: Uncaught Error: Call to a member function fetch() on boolean".

Так как СУБД PostgreSQL не смогла выполнить запрос, вместо результирующей таблицы было возвращено сообщение об ошибке. Метод `query()` вернул `false`, поэтому обращение с переменной `$ver` как с объектом терпит неудачу — вместо ожидаемого объекта класса `PDOStatement` переменная содержит логический тип. В примере использование переменной `$ver` довольно близко к месту выполнения ошибочного запроса, и восстановить причину возникновения ошибки просто. Однако в объемном приложении это уже гораздо труднее.

Расширение PDO предоставляет несколько режимов обработки ошибок. По умолчанию используется "тихий" режим, работа которого только что была продемонстрирована. Извлечь сообщение об ошибке в нем можно, только явно обратившись к методу `errorInfo()`.

- `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 класса `PDO`. Для этого скрипт инициализации объекта `$pdo` из листинга 28.1 следует переписать таким образом, чтобы обработка запросов протекала в режиме исключения (листинг 28.5).

Листинг 28.5. Обработка ошибки соединения с базой данных. Файл connect.php

```
<?php
try {
    $pdo = new PDO(
      'pgsql:host=localhost;dbname=catalog',
      'igor',
      '',
      [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
} catch (PDOException $e) {
    echo 'Невозможно установить соединение с базой данных';
}
```

Замена файла connect_db.php на connect.php в листинге 28.4 приведет к генерации исключительной ситуации и ее перехвату в блоке `catch`. В результате будет выведено сообщение об ошибке:

```
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 сообщению можно легко определить, что PostgreSQL не может обнаружить функцию с именем `VERSION1()`.

28.5. Извлечение данных

Рассмотрим задачу извлечения данных более подробно, для этого воспользуемся таблицей `users` из листинга 27.3. В листинге 28.6 представлен скрипт, выводящий содержимое таблицы в окно браузера.

Листинг 28.6. Вывод содержимого таблицы users. Файл fetch.php

```
<?php
require_once('connect.php');

$query = 'SELECT * FROM users';
$usr = $pdo->query($query);

try {
    while($user = $usr->fetch()) {
        echo $user['first_name'] . ' ' . $user['last_name'] . '<br />';
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Результатом выполнения скрипта будут следующие строки:

```
Максим Кузнецов
Игорь Симдянов
```

Как видно из листинга 28.6, метод `query()` объекта `PDO` возвращает объект результирующей таблицы `$usr`. Извлечение данных осуществляется при помощи метода `fetch()`. За один вызов функция извлекает из результирующей таблицы одну запись, которая представляется в виде ассоциативного массива `$user`.

В качестве ключей массива выступают имена столбцов таблицы `users`, 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.

Если посмотреть на содержимое массива `$user`, который возвращается методом `fetch()`:

```
$user = $usr->fetch();
echo '<pre>';
print_r($user);
echo '</pre>';
```

то можно увидеть, что в качестве результата будут выведены следующие строки:

```
Array
(
    [id] => 1
    [0] => 1
    [first_name] => Максим
    [1] => Максим
    [last_name] => Кузнецов
    [2] => Кузнецов
)
```

Метод `fetch()` возвращает ассоциативный массив, где ключами выступают имена столбцов, а значениями — содержимое ячеек. Кроме того, содержимое ячеек дублируется в индексном виде. В качестве индекса выступает порядок извлекаемого столбца (начиная с 0). Изменить такое поведение можно при помощи констант класса `PDO`:

```
$user = $usr->fetch(PDO::FETCH_ASSOC);
echo '<pre>';
print_r($user);
echo '</pre>';
```

Использование константы `PDO::FETCH_ASSOC` приводит к тому, что в результирующем массиве остаются только ассоциативные элементы:

```
Array
(
    [id] => 1
    [first_name] => Максим
    [last_name] => Кузнецов
)
```

Помимо константы `FETCH_ASSOC`, класс `PDO` предоставляет еще несколько констант, при помощи 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 массивы (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 28.7).

Листинг 28.7. Использование метода `fetchAll()`. Файл fetch_all.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT * FROM users';
    $usr = $pdo->query($query);

    $users = $usr->fetchAll();
    foreach($users as $user) {
        echo $user['first_name'] . ' ' . $user['last_name'] . '<br />';
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Метод `fetchAll()` может принимать такие же управляющие константы, как и метод `fetch()`.

28.6. Параметризация SQL-запросов

До текущего момента извлекалось все содержимое таблицы. В большинстве случаев SQL-запросы будут содержать какие-то значения, которые должны быть подставлены в SQL-запрос. Для того чтобы выполнить такой запрос, придется воспользоваться параметризованным запросом. Такой запрос проходит несколько стадий: подготовки, связывания с переменными и выполнения.

Метод `PDO::query()` в предыдущем разделе выполнял все три стадии (за исключением связывания, т. к. параметры отсутствовали).

В листинге 28.8 приводится пример извлечения из таблицы `users` пользователя с именем 'Игорь'.

Листинг 28.8. Параметризованный запрос. Файл prepare.php

```
<?php
require_once('connect.php');

try {
    $query = 'SELECT
                first_name,
                last_name
              FROM
                users
              WHERE
                first_name = :name';
    $usr = $pdo->prepare($query);
    $usr->execute(['name' => 'Игорь']);
```

```
    $user = $usr->fetch(PDO::FETCH_NUM);
    if ($user) {
        echo implode(' ', $user); // Игорь Симдянов
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Запросы с параметрами передаются специальному методу `prepare()`. Сам запрос содержит плейсмент `:name`, который заполняется на этапе выполнения методом `execute()`. Для заполнения параметра методу `execute()` передается ассоциативный массив, ключи которого содержат названия плейсментов.

Параметры могут быть безымянными, в запросе они обозначаются символом вопроса `?`. Методу же `execute()` передается индексный массив, элементы которого заменят символы `?` в параметризованном запросе. Первый знак вопроса заменяется первым элементом, второй — вторым и т. д. (листинг 28.9).

Листинг 28.9. Файл prepare_alt.php

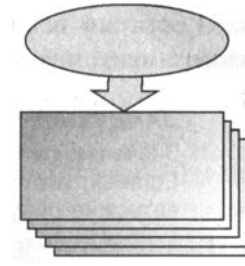
```
<?php
require_once('connect.php');

try {
    $query = 'SELECT
                first_name,
                last_name
              FROM
                users
              WHERE
                first_name = ?';
    $usr = $pdo->prepare($query);
    $usr->execute(['Игорь']);
    $user = $usr->fetch(PDO::FETCH_NUM);
    if ($user) {
        echo implode(' ', $user); // Игорь Симдянов
    }
} catch (PDOException $e) {
    echo 'Ошибка выполнения запроса: ' . $e->getMessage();
}
```

Задания

1. Создайте форму с текстовой областью, информация из которой сохраняется в таблицу `messages`, где помимо текста сохраняются дата и время сообщения. Выведите сохраненные в таблице сообщения перед формой в порядке убывания даты сообщения.

2. Пусть имеется таблица `messages` с текстовыми сообщениями. Организуйте постраничный вывод записей из этой таблицы.
3. По документации изучите метод `lastInsertId()` класса `PDO`, предназначенный для извлечения автосгенерированного первичного ключа только что вставленной записи. Создайте скрипт, который бы заполнял связанные таблицы `users` и `authors` из *главы 27*.
4. При помощи Composer установите фреймворк phinx для выполнения миграций. Ознакомьтесь по документации фреймворка с его возможностями. Создайте таблицы `users` и `authors` из *главы 27* при помощи миграций.
5. При помощи Composer установите фреймворк Doctrine, предоставляющий объектно-ориентированный интерфейс для выполнения SQL-запросов. По документации фреймворка ознакомьтесь с его возможностями.

ГЛАВА 29



NoSQL база данных Redis

За последние десятилетия серверы значительно нарастили объем оперативной памяти, которая больше не является дефицитным ресурсом. В связи с этим меняется и концепция хранения данных. Совместно с классическими реляционными СУБД (PostgreSQL, MySQL) используются альтернативные NoSQL СУБД. В подавляющем большинстве случаев, NoSQL базы данных используют в качестве кэша. Разместив сгенерированную HTML-страницу или ее части в оперативной памяти, можно значительно уменьшить время ее отдачи за счет отсутствия обращения к жесткому диску и необходимости формировать ее средствами PHP при каждом обращении клиента.

Помимо кэша, NoSQL базы данных применяются для счетчиков, очередей, учета кликов, 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.

За счет всех этих факторов скорость обработки данных и количество одновременно обрабатываемых запросов значительно, и часто на порядки превосходит традиционные СУБД.

Одной из самых первых NoSQL баз данных, которая послужила прототипом для многих современных NoSQL-решений, являлась 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, рассмотреть их все детально не представляется возможным. Поэтому мы сосредоточимся лишь на одной NoSQL базе данных Redis, которая благо-

даря богатым возможностям и концепции хранения данных "ключ-значение" все чаще заменяет memcached в PHP-проектах.

> ***Замечание***
>
> Значение memcached и частота использования в PHP-проектах по-прежнему велики. Если вы хотите познакомиться с приемами работы с ней из PHP, можно ориентироваться на книгу "PHP 7" Котерова Дмитрия и Симдянова Игоря.

29.1. Почему Redis?

Redis — исключительно быстрый однопоточный сервер для хранения данных в оперативной памяти. За счет того, что процессору не приходится переключаться между потоками, при большом количестве соединений достигается огромная производительность.

Причем Redis — это не просто хранилище пар "ключ-значение", значения могут быть как обычными строками, так и коллекциями (массивами, хэшами, множествами), над которыми можно осуществлять операции.

Допускается сохранение данных на жесткий диск, тем самым разрешается проблема холодного старта, когда после запуска группировки серверов должно пройти некоторое время, прежде чем заполнится кэш и страницы будут отдаваться клиенту быстро.

Допускается назначение ключам времени жизни, тем самым можно 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. Эта операция может занять длительное время, и лучше ее выполнить в фоновом режиме, не задерживая ответ пользователю. В этом случае задание можно поместить в очередь и завершить обслуживание запроса. В очереди, как правило, на отдельном сервере задание будет рано или поздно выполнено.

Redis допускает создание скриптов на языке программирования Lua. На Lua также разрабатываются расширения для Web-сервера nginx. В результате можно поме-

щать данные непосредственно в Redis и забирать их из nginx. 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 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.

29.2. Установка сервера

29.2.1. Установка в среде 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. Во время установки 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
```

29.2.2. Установка в среде Mac OS X

В Mac OS X для установки Redis проще все 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
```

29.2.3. Установка в Windows

На момент написания книги официальных дистрибутивов Redis под Windows не собиралось. Однако в GitHub-репозитории Microsoft Open Technologies без труда можно обнаружить порт Redis под Windows: **https://github.com/MSOpenTech/redis**. На странице релизов **https://github.com/MSOpenTech/redis/releases** можно найти собранные и готовые к использованию дистрибутивы.

После того как дистрибутив загружен и запущен на выполнение, мастер установки предложит несколько диалоговых окон (рис. 29.1).

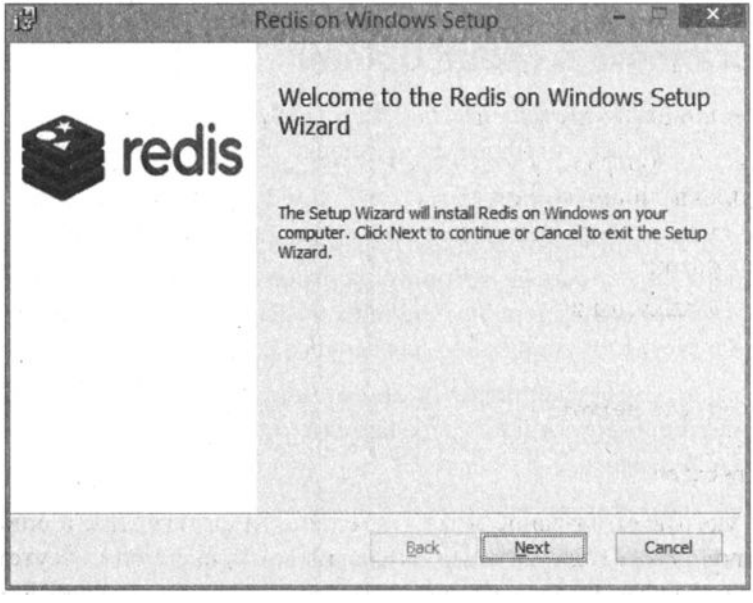


Рис. 29.1. Установка Redis под Windows

29.2.4. Проверка работоспособности

Убедиться в работоспособности сервера можно, обратившись к нему через 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:3.0.7
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:aa27a151289c9b98
redis_mode:standalone
...
```

29.3. Клиент *redis-cli*

Для доступа к серверу потребуется клиент, это может быть PHP-скрипт, GUI-клиент Redis Desktop Manager или стандартный консольный клиент `redis-cli`. Последний чрезвычайно важен при отладке на удаленных серверах, поэтому следует познакомиться хотя бы с базовыми приемами работы в этом клиенте.

Для запуска клиента следует выполнить команду `redis-cli`:

```
$ redis-cli
```

По умолчанию клиент пытается соединиться с локальным сервером по порту 6379. Команда, приведенная выше, 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 при помощи команды `HELP`, после которой следует указать название команды. Например, запросить справочную информацию по команде `PING` можно следующим образом:

```
localhost:6379> HELP PING

  PING -
  summary: Ping the server
  since: 1.0.0
  group: connection
```

Первое поле указывает название команды, в данном случае `PING`, и список возможных 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 `@`, разместив после которого название группы, можно получить список всех команд, входящих в группу.

```
localhost:6379> HELP @connection

  AUTH password
  summary: Authenticate to the server
  since: 1.0.0

  ECHO message
  summary: Echo the given string
  since: 1.0.0

  PING -
  summary: Ping the server
  since: 1.0.0
```

```
QUIT -
summary: Close the connection
since: 1.0.0

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>, можно перебирать доступные группы. Ниже приводится список доступных групп:

- `@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 хэшами;
- `@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 геокоординатами.

29.4. 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 перечисляются через пробел вслед за командой:

```
> MGET fst snd thd fth
1) "1"
2) "2"
3) "3"
4) "4"
```

29.5. 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)
```

29.6. 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 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 название 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)
```

29.7. Время жизни ключа

Одна из основных специализаций Redis — быстрый кэш, расположенный в оперативной памяти. В связи с этим особо важна актуальность кэша, 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 EXISTS 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. Для того 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
```

Для того 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
```

29.8. 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 (2^{32}) 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);
- хэш — 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 освещены в предыдущих разделах. В последующих разделах будут более детально рассмотрены команды для работы с коллекционными типами.

29.9. Хэш

Хэши хранят пары "ключ-значение", т. е. помимо ключа к самому хэшу, каждый элемент, который входит в его состав, также снабжается своим ключом. Для создания хэша можно воспользоваться командой HSET, которая принимает в качестве первого параметра ключ хэша, в качестве второго параметра — ключ пары, а в качестве третьего — значение. Ниже создается хэш admin с регистрационными данными пользователя:

```
> HSET admin login "root"
(integer) 1
> HSET admin pass "password"
(integer) 1
> HSET admin register_at "2017-09-01"
(integer) 1
```

Создать представленный выше хэш можно при помощи одной команды HMSET, которая позволяет задать сразу все пары "ключ-значение":

```
> HMSET admin login "root" pass "password" register_at "2017-09-01"
OK
```

Для чтения элементов хэша можно воспользоваться командой HGET:

```
> HGET admin login
"root"
```

Или извлечь все содержимое хэша при помощи команды HVALS:

```
> HVALS admin
1) "root"
2) "password"
3) "2017-09-01"
```

Проверить существование поля с заданным именем можно с помощью команды HEXISTS:

```
> HEXISTS admin login
(integer) 1
> HEXISTS admin none
(integer) 0
```

Кроме того, в любой момент можно запросить все ключи хэша при помощи команды HKEYS:

```
> HKEYS admin
1) "login"
2) "pass"
3) "register_at"
```

С помощью команды HGETALL можно извлечь все содержимое хэша, 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 хэше можно с помощью команды HLEN:

```
> HLEN admin
(integer) 3
```

29.10. 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 email support@softtime.info
(integer) 1
> SADD email igor@softtime.info
(integer) 1
> SADD email support@softtime.info
(integer) 0
```

Команда SADD позволяет вставлять в коллекцию сразу несколько значений:

```
> SADD email igor@softtime.ru softtime@softtime.ru
(integer) 2
```

Сколько бы повторяющих значений не было вставлено в коллекцию email, содержать она будет только уникальные значения, в чем можно убедиться, воспользовавшись командой SMEMBERS:

```
> SMEMBERS email
1) "igor@softtime.info"
2) "softtime@softtime.ru"
3) "igor@softtime.ru"
4) "support@softtime.info"
```

Выяснить количество элементов в множестве позволяет команда SCARD:

```
> SCARD email
(integer) 4
```

Для удаления элемента из коллекции предназначена команда SREM:

```
> SREM email igor@softtime.info
(integer) 1
```

Для извлечения (случайного) значения из множества можно воспользоваться командой SPOP:

```
> SPOP email
"igor@softtime.ru"
```

Сильной стороной множеств является возможность поиска, объединения и пересечения нескольких множеств. Для демонстрации этих возможностей создадим дополнительную коллекцию subscribers, также содержащую список электронных адресов:

```
> SADD subscribers igor@simdyanov.ru igor@softtime.info igor@softtime.ru
(integer) 3
> SMEMBERS subscribers
1) "igor@softtime.info"
2) "igor@simdyanov.ru"
3) "igor@softtime.ru"
```

Для поиска общих электронных адресов коллекций email и subscribers можно воспользоваться командой SINTER:

```
> SINTER email subscribers
1) "igor@softtime.info"
2) "igor@softtime.ru"
```

Для поиска во множестве email-адресов, не входящих во множество subscribers, можно воспользоваться командой SDIFF:

```
> SDIFF email subscribers
1) "softtime@softtime.ru"
2) "support@softtime.info"
```

При помощи команды SUNION можно объединить оба множества email и subscribers в одно (дубликаты автоматически отбрасываются):

```
> SUNION subscribers email
1) "igor@softtime.info"
2) "softtime@softtime.ru"
```

```
3) "igor@simdyanov.ru"
4) "igor@softtime.ru"
5) "support@softtime.info"
```

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 subscribers email
(integer) 5
> SMEMBERS result
1) "igor@softtime.info"
2) "softtime@softtime.ru"
3) "igor@simdyanov.ru"
4) "igor@softtime.ru"
5) "support@softtime.info"
```

Аналогичные команды существуют для пересечения SINTERSTORE и разности SDIFFSTORE.

Команда SMOVE перемещает элемент из одного множества в другое:

```
> SMOVE result new igor@softtime.ru
(integer) 1
> SMOVE result new softtime@softtime.ru
(integer) 1
> SMEMBERS result
1) "igor@simdyanov.ru"
2) "support@softtime.info"
3) "igor@softtime.info"
> SMEMBERS new
1) "softtime@softtime.ru"
2) "igor@softtime.ru"
```

29.11. Отсортированное множество

Отсортированные множества в некотором смысле являются объединением всех остальных типов коллекций — списков, хэшей и множеств. Точно так же как в хэшах, 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 данным 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 при помощи команды 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, 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.

29.12. Базы данных

Как и в СУБД, в Redis есть базы данных, однако они не имеют названия и являются нумерованными. По умолчанию утилита redis-cli открывает базу данных 0. Для того 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.

29.13. 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
...
```

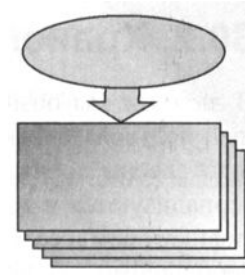
Как видно, производительность сервера Redis может достигать свыше 100 000 RPS (операций в секунду), даже на запись. Столь впечатляющие результаты достигаются за счет EventLoop-механизма, когда соединения обрабатывает один поток в неблокирующем режиме.

Показанные в тестах результаты подтверждаются и на практике, даже без учета кластеризации, единичные экземпляры Redis способны обрабатывать запросы с десятка Web-серверов с PHP-приложениями.

Задания

1. Исследуйте подсистему Pub/Sub в Redis.
2. Пробуйте запустить на разных портах несколько экземпляров Redis и связать их в кластер. Исследуйте возможности подсистемы Sentinel в Redis.
3. Установите и познакомьтесь с NoSQL базой данных memcached.

ГЛАВА 30



PHP-расширение Redis

Листинги данной главы
можно найти в подкаталоге redis.

PHP не поддерживает работу с базой данных Redis "из коробки". Для того чтобы скрипты смогли подключиться к Redis, потребуется установить расширение php-redis.

30.1. Установка расширения php-redis

Для установки расширения php-reids в операционной системе Mac OS X можно воспользоваться менеджером пакетов Homebrew. Для этого следует выполнить команду

```
$ brew install php71-redis
```

Для установки расширения в операционной системе Ubuntu можно воспользоваться менеджером пакетов `apt-get`:

```
$ sudo apt-get install php7.0-redis
```

Чтобы проверить работоспособность расширения, можно воспользоваться скриптом из листинга 30.1.

Листинг 30.1. Файл ping.php

```
<?php
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
echo $redis->ping(); // +PONG
```

Расширение добавляет новый предопределенный класс `Redis`, объект которого позволяет установить соединение с сервером и выполнять `redis`-команды за счет вызова одноименных методов.

30.2. Хранение сессий в Redis

В *главе 14* 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 Web-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 30.2. Файл session.php

```
<?php
session_start();

if(!isset($_SESSION['count'])) {
    $_SESSION['count'] = 0;
}

$_SESSION['count']++;
echo $_SESSION['count'];
```

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:uttfavjbcnitdke4s4lc5dcifp"
```

```
127.0.0.1:6379> TYPE "PHPREDIS_SESSION:uttfavjbcnitdke4s4lc5dcifp"
string
127.0.0.1:6379> GET "PHPREDIS_SESSION:uttfavjbcnitdke4s4lc5dcifp"
"count|i:4;"
```

В примере выше видно, что в сессию сохранен сериализованный (serialize) массив `$_SESSION`.

В настройках php.ini, приведенных выше, сервер Redis располагается на том же сервере, где работает Web-приложение. Это не всегда удобно в случае больших Web-приложений, обслуживаемых несколькими Web-серверами. Возможно, удобнее будет выделить под Redis отдельный сервер и прописать его адрес в конфигурационных файлах других серверов (рис. 30.1).

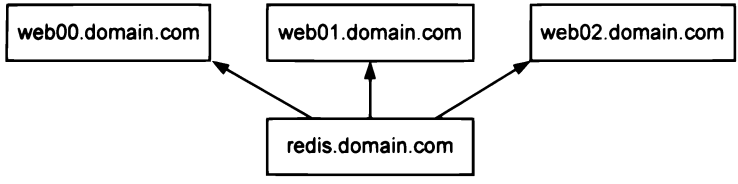


Рис. 30.1. Один сервер Redis может обслуживать сразу несколько Web-серверов

30.3. 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 30.3).

Листинг 30.3. 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 используется метод `set()`, а для извлечения — метод `get()`, для проверки существования ключа можно воспользоваться методом `exists()`, допускается и множественная вставка значений при помощи метода `mSet()` (листинг 30.4).

Листинг 30.4. 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'); // value
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 30.5).

Листинг 30.5. Файл keys.php

```
<?php
require_once 'config.php';

echo '<pre>';
print_r($redis->keys('*')); // value
echo '</pre>';
```

Результатом выполнения скрипта из листинга 30.5 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 30.6).

Листинг 30.6. 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
)
```

30.4. Кэширование данных

Так как класс `Redis` фактически воспроизводит весь функционал, доступный Redis-клиентам, мы не можем осветить все его возможности. Многочисленные методы класса придется освоить, опираясь на документацию расширения php-redis.

Как правило, Redis очень интенсивно используется для построения кэширующих 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.

Как правило, весь цикл от извлечения записей из базы данных до создания HTML проводят при первом обращении к странице, после чего результат помещают в кэш в Redis. При следующих обращениях идет проверка, имеется ли уже сгенерированный HTML-код в Redis, и при положительном ответе 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 кэша, которые могут быть затронуты изменениями. Этот процесс называется *инвалидацией* кэша.

Для того чтобы продемонстрировать пример кэширования, обратимся к системе отображения пользователей, разработанной в *разд. 25.4*. Из всего объема кода нам потребуется лишь контроллер, содержимое которого приводится в листинге 30.7.

Листинг 30.7. Файл MVC/Controllers/Controller.php

```
<?php
namespace MVC\Controllers;

class Controller
{
    public $path;
    public $router;
    public $model;
```

```
    public function __construct($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()
    {
        $class = get_class($this->model);
        $class = substr($class, strrpos($class, '\\') + 1);
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $this->router->ext,
            $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 выше разделение труда не очень удобно для кэширования, 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 30.8).

Листинг 30.8. Файл MVC/Controllers/ControllerLazy.php

```
<?php
namespace MVC\Controllers;

class ControllerLazy
{
    public $path;
    private $class;
```

```
    private $router;
    private $model;

    public function __construct($path)
    {
        $this->path = $path;
    }

    public function render()
    {
        $decorator = \MVC\Decorators\DecoratorFactory::create(
            $this->getRouter()->ext,
            $this->getClass(),
            $this->getModel());
        $view = \MVC\Views\ViewFactory::create(
            $this->getRouter()->ext,
            $this->getClass(),
            $decorator);
        return $view->render();
    }

    private function getClass() {
        if(empty($this->class)) {
            $class = get_class($this->getModel());
            $this->class = substr($class, strrpos($class, '\\') + 1);
        }
        return $this->class;
    }

    private function getRouter() {
        if(empty($this->router)) {
            $this->router = Router::parse($this->path);
        }
        return $this->router;
    }

    private function getModel() {
        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 примера выше, 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 кэширования данных в Redis. Для демонстрации создадим новый вариант контроллера `ControllerCache` (листинг 30.9).

Листинг 30.9. Файл MVC/Controllers/ControllerCache.php

```
<?php
namespace MVC\Controllers;

class ControllerCache
{
    public $path;
    private $class;
    private $router;
    private $model;
    private $redis;

    public function __construct($path, $redis)
    {
        $this->path = $path;
        $this->redis = $redis;
    }

    public function render()
    {
        $cache = $this->redis->get($this->path);
        if (!$cache) {
            $decorator = \MVC\Decorators\DecoratorFactory::create(
                $this->getRouter()->ext,
                $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() {
        if(empty($this->class)) {
            $class = get_class($this->getModel());
            $this->class = substr($class, strrpos($class, '\\') + 1);
        }
        return $this->class;
    }

    private function getRouter() {
        if(empty($this->router)) {
            $this->router = Router::parse($this->path);
        }
        return $this->router;
    }

    private function getModel() {
        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 и затем отправляется клиенту.

В листинге 30.10 приводится пример использования класса `ControllerCache`.

Листинг 30.10. Файл mvc_ups.php

```
<?php
require_once 'config.php';

spl_autoload_register();
```

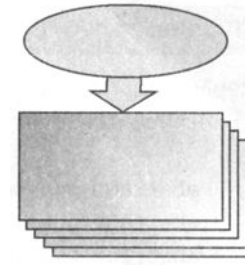
```
use MVC\Controllers\ControllerCache;

$obj = new ControllerCache('users/1.html', $redis);
echo $obj->render();
```

После обращения к скрипту из листинга 30.10 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"
```

Задания

1. Создайте скрипт, который бы сохранял в Redis-коллекции множество IP-адресов обращающихся к нему клиентов. Причем для уже существующих IP-адресов значения должны увеличиваться при каждом новом обращении.
2. Реализуйте матрешечный тип кэширования, когда инвалидация верхнеуровневого кэша определяется вложенными элементами. Например, страница состоит из шапки сайта, футера, сайдбара и области контента. В области контента может выводиться как детальное содержимое страницы, так и списочное представление для коллекции страниц. Каждый из фрагментов, включая отдельный элемент списка, закэширован в Redis. Необходимо построить такую систему, чтобы инвалидация любого из элементов приводила к обновлению кэша всех зависимых элементов.
3. При реализации регистрации пользователей на сайте в шапке или сайдбаре часто выводится меню, индивидуальное для каждого пользователя, с его данными (имя, статистическая информация). При кэшировании страницы информация, помещенная в Redis, общая для всех пользователей. Как избежать вывода индивидуальной информации пользователя, который первым обратился к странице и инициировал кэш?
4. Познакомьтесь с шаблоном проектирования Наблюдатель (Observer). Создайте реализацию шаблона на PHP с использованием механизма Pub/Sub NoSQL базы данных Redis.



ГЛАВА 31

Итераторы

Листинги данной главы
можно найти в подкаталоге iterators.

Итераторы — это объекты-коллекции, которые можно обходить в цикле `foreach`. Сами по себе массивы являются довольно 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 терять 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`.

31.1. Интерфейсы для создания итераторов

PHP предоставляет два интерфейса для создания классов-итераторов: `Iterator` и `IteratorAggregate`, которые наследуются от общего абстрактного интерфейса `Traversable` (рис. 31.1).

В листинге 31.1 представлена реализация интерфейса `Iterator`, который требует от реализующих его классов создания пяти обязательных методов.

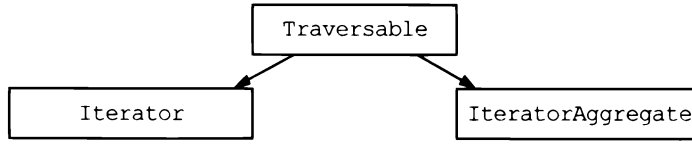


Рис. 31.1. Схема наследования интерфейсов для реализации итераторов

Листинг 31.1. Определение интерфейса `Iterator`. Файл iterator.php

```
<?php
interface Iterator extends Traversable
{
    abstract public mixed current();
    abstract public mixed key();
    abstract public next();
    abstract public rewind();
    abstract public boolean valid();
}
```

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`. Возвращаемый результат позволяет сделать вывод о достижении конца коллекции.

В листинге 31.2 представлена возможная реализация итератора `MyIterator`, осуществляющего навигацию по внутреннему индексному массиву `$collection`, который передается ему при инициализации объекта.

Листинг 31.2. Файл Iterators/MyIterator.php

```
<?php
namespace Iterators;

class MyIterator implements \Iterator
{
    private $index;
    private $collection;

    public function __construct($collection) {
        $this->collection = $collection;
        $this->rewind();
    }
```

```
    public function rewind() {
        $this->index = 0;
    }

    public function current() {
        return $this->collection[$this->key()];
    }

    public function key() {
        return $this->index;
    }

    public function next() {
        ++$this->index;
    }

    public function valid() {
        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 31.3 приведен пример использования класса `MyIterator` для обхода массива из пяти элементов.

Листинг 31.3. 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 приведенного выше 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` может показаться неочевидной, т. к. итерируемый массив `$collection` и так 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. Все это позволяет многократно усилить гибкость создаваемого кода.

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. В листинге 31.4 приводится пример класса `LimitMyIterator`, который использует ранее разработанный `MyIterator`, но ограничивает размер коллекции первыми `$limit`-элементами.

Листинг 31.4. Файл Iterators/LimitMyIterator.php

```
<?php
namespace Iterators;

class LimitMyIterator implements \IteratorAggregate {

    private $collection;
    private $limit;

    public function __construct($collection, $limit = 2) {
        $this->collection = $collection;
        $this->limit = $limit;
    }

    public function getIterator() {
        $limited = array_slice($this->collection, 0, $this->limit);
        return new MyIterator($limited);
    }
}
```

Класс включает закрытые переменные `$collection` и `$limit`, используемые 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 31.5 приводится пример использования класса `LimitMyIterator`.

Листинг 31.5. 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 />";
}
```

Результат выполнения скрипта из листинга 31.5 будет выглядеть следующим образом:

```
Элемент с индексом 0 и значением первый
Элемент с индексом 1 и значением второй
```

Помимо интерфейсов, которые необходимо реализовывать самостоятельно, PHP предоставляет несколько готовых итераторов, сосредоточенных в библиотеке SPL (Standard PHP Library, стандартная библиотека PHP). Рассмотреть полностью содержимое SPL не представляется возможным. В оставшейся части главы мы обсудим лишь часть наиболее часто используемых классов.

31.2. Интерфейс *ArrayAccess*

При использовании итератора `MyIterator` из предыдущего раздела имеется определенное ограничение: объект этого класса можно использовать в цикле `foreach`, однако обратиться к его элементам при помощи квадратных скобок не получится.

Это можно исправить, реализовав интерфейс `ArrayAccess`, определение которого представлено в листинге 31.6.

Листинг 31.6. Файл ArrayAccess.php

```
<?php
interface ArrayAccess {
    abstract public boolean offsetExists(mixed $index);
    abstract public mixed offsetGet(mixed $index);
    abstract public void offsetSet(mixed $index, mixed $value);
    abstract public void offsetUnset(mixed $index);
}
```

Как видно из листинга, интерфейс требует реализации четырех методов:

- `offsetExists()` — возвращает `true` или `false` в зависимости от того, существует элемент с 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 31.7 представлена расширенная версия итератора для обхода массива `MyArrayIterator`, к объекту которого помимо этого можно обращаться при помощи квадратных скобок, как к обычному массиву.

Листинг 31.7. Файл Iterators/MyArrayIterator.php

```
<?php
namespace Iterators;

class MyArrayIterator implements \Iterator, \ArrayAccess
{
    private $index;
    private $collection;

    public function __construct($collection) {
        $this->collection = $collection;
        $this->rewind();
    }

    public function offsetSet($index, $value) {
        if (is_null($index)) {
            $this->collection[] = $value;
        } else {
            $this->collection[$index] = $value;
        }
    }

    public function offsetExists($index) {
        return isset($this->collection[$index]);
    }

    public function offsetUnset($index) {
        unset($this->collection[$index]);
    }

    public function offsetGet($index) {
        if($this->offsetExists($index)) {
            return $this->collection[$index];
```

```
        } else {
            return null;
        }
    }

    public function rewind() {
        $this->index = 0;
    }

    public function current() {
        return $this->collection[$this->key()];
    }

    public function key() {
        return $this->index;
    }

    public function next() {
        ++$this->index;
    }

    public function valid() {
        return isset($this->collection[$this->index]);
    }
}
```

В листинге 31.8 приводится пример использования полученного класса.

Листинг 31.8. Файл use_myarrayiterator.php

```
<?php
spl_autoload_register();

$array = ['первый',
          'второй',
          'третий',
          'четвертый',
          'пятый'];

$collection = new Iterators\MyArrayIterator($array);

echo $collection[2]; // третий
echo '<br />';

foreach($collection as $key => $value) {
    echo "Элемент с индексом $key и значением $value<br />";
}
```

31.3. Класс *ArrayObject*

Необязательно реализовывать интерфейсы `Iterator`, `ArrayAccess` для того, чтобы добиться от объекта поведения массива. В библиотеке SPL имеется готовый класс `ArrayObject`, который позволяет обрабатывать полученный объект как массив (листинг 31.9).

Листинг 31.9. 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 />";
}
```

В отличие от разработанного ранее класса `MyArrayIterator`, класс `ArrayObject` может работать не только с массивом, но и с объектом. При этом итерация идет не по заранее подготовленной коллекции, а по открытым переменным объекта. Создадим объект на базе анонимного класса и передадим его `ArrayObject` (листинг 31.10).

Листинг 31.10. Файл use_array_object_object.php

```
<?php
$collection = new ArrayObject(new class {
    private $first = 'первый';
    public $second = 'второй';
    public $third = 'третий';

    public function __construct()
    {
        $this->fourth = 'четвертый';
    }
});

echo $collection['second']; // второй
echo '<br />';
```

```
foreach($collection as $key => $value) {
    echo "Элемент с индексом $key и значением $value<br />";
}
```

Результатом выполнения скрипта из примера выше будут следующие строки:

```
второй
Элемент с индексом second и значением второй
Элемент с индексом third и значением третий
Элемент с индексом fourth и значением четвертый
```

Как видно из примера, итератор получает доступ только к открытым переменным, закрытые переменные игнорируются итератором.

31.4. Класс *DirectoryIterator*

Еще одним готовым классом библиотеки SPL является `DirectoryIterator`, предоставляющий доступ к содержимому каталога. В листинге 31.11 приводится пример использования итератора.

Листинг 31.11. 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 класса `DirectoryIterator`, реализующего методы, часть из которых представлена в табл. 31.1. С полным списком методов можно ознакомиться в справочной документации.

Таблица 31.1. Методы класса `DirectoryIterator`

Метод	Описание
`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 имя каталога (без имени файла и подкаталога)
`getType()`	Возвращает 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, для того чтобы после имени файла вывести его размер, достаточно воспользоваться методом `getSize()` (листинг 31.12).

Листинг 31.12. Использование методов класса `DirectoryIterator`. Файл size.php

```
<?php
$dir = new DirectoryIterator('.');
foreach($dir as $file) {
    // Выводим только файлы
    if ($file->isFile()) {
        // Имя файла и его размер
        echo $file . ' ' . $file->getSize() . '<br />';
    }
}
```

31.5. Класс *FilterIterator*

Элементы коллекции могут быть отфильтрованы при помощи итератора, производного от класса `FilterIterator`. В листинге 31.13 приводится пример создания фильтра `ExtensionFilter` для класса `DirectoryIterator`, который отбирает все файлы с расширением PHP.

Листинг 31.13. Файл Iterator/ExtensionFilter.php

```
<?php
namespace Iterators;

class ExtensionFilter extends \FilterIterator
{
    // Фильтруемое расширение
    private $ext;
    // Итератор DirectoryIterator
    private $it;

    public function __construct(\DirectoryIterator $it, $ext)
    {
        parent::__construct($it);
        $this->it = $it;
        $this->ext = $ext;
    }

    // Метод, определяющий, удовлетворяет текущий элемент
    // фильтру или нет
    public function accept()
    {
        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 из результирующего списка файлов будут исключены все файлы с расширением PHP (листинг 31.14).

Листинг 31.14. 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 />';
}
```

31.6. 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 третьего — смещение от позиции. При этом итератор работает с участком коллекции, определяемым вторым и третьим параметрами. В листинге 31.15 приводится пример вывода первых пяти элементов текущего каталога с исключением PHP-файлов.

Листинг 31.15. Использование 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 />';
}
```

31.7. Рекурсивные итераторы

Рекурсивной называется функция, которая вызывает сама себя. 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.

Для решения этой проблемы можно воспользоваться итератором для рекурсивного обхода `RecursiveIteratorIterator` (листинг 31.16).

Листинг 31.16. Рекурсивный итератор. Файл recursion.php

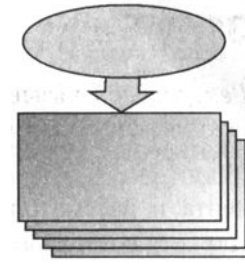
```
<?php
$dir = new RecursiveIteratorIterator(
         new RecursiveDirectoryIterator('.'),
       true);

foreach ($dir as $file) {
    echo str_repeat('-', $dir->getDepth()) . " $file<br />";
}
```

Метод `getDepth()` итератора `RecursiveIteratorIterator` возвращает глубину вложения элемента.

Задания

1. По документации на сайте **http://php.net** изучите интерфейс `Countable`. Расширьте класс `MyArrayIterator` из листинга 31.7 таким образом, чтобы применение функции `count()` к объекту возвращало количество элементов в коллекции.
2. По документации на сайте **http://php.net** изучите интерфейс `Serializable` для сериализации объектов. Реализуйте класс, объекты которого при применении к ним функции `serialize()` возвращали бы JSON-строку, а не стандартный сериализованный объект PHP.
3. Реализуйте задачу рекурсивного обхода каталогов из листинга 31.6 без использования итераторов, только опираясь на стандартные функции обработки каталогов.
4. По документации на сайте **http://php.net** познакомьтесь с классом SPL `SplDoublyLinkedList`, реализующим двусвязный список.

ГЛАВА 32



Генераторы и итераторы

Листинги данной главы
можно найти в подкаталоге generators.

Генераторы — относительно новая возможность PHP, доступная с версии 5.5, позволяющая создавать собственные итераторы, которые затем можно использовать в операторе `foreach`. В отличие от полноценных итераторов из *главы 31* генераторы не позволяют вернуться в начальное состояние при помощи `rewind()`. Однако они создаются более просто, предоставляют разработчикам более гибкие возможности по их использованию и допускают отложенные вычисления, т. е. значения вычисляются только тогда, когда они действительно необходимы.

32.1. Отложенные вычисления

Генератор — это обычная функция, но для возврата значения вместо ключевого слова `return` используется оператор `yield`. В листинге 32.1 приводится пример простейшего генератора, который по умолчанию формирует последовательность от 0 до 100 и возвращает значения при помощи оператора `yield`. В момент возврата при помощи оператора `echo` выводится текущее значение.

Листинг 32.1. 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 ниже, 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 оператора `break` после первых шести итераций (добиться этого можно было 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 оператора `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 и упростить картину, напишем генератор без цикла внутри (листинг 32.2).

> ***Замечание***
>
> Если после оператора `yield` не указать никакое значение, в качестве значения будет подразумеваться `null`.

Листинг 32.2. Простейший генератор. 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 32.2 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 первый оператор `echo`, после которого следует ключевое слово `yield`. Последнее выталкивает результат 1 из функции. В результате 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 операторы, но 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 оператора `foreach` (рис. 32.1).

Использование цикла внутри генератора лишь позволяет вызвать `yield` необходимое количество раз.

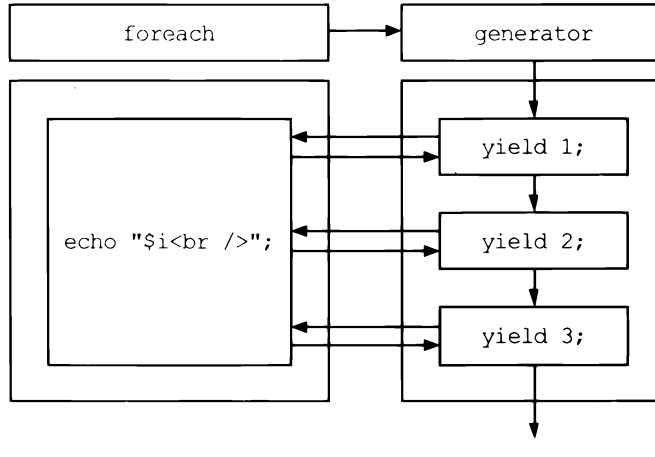


Рис. 32.1. Схема передачи управления от оператора `foreach` генератору и обратно

32.2. Манипуляция массивами

Пока все выглядит довольно запутанно и малополезно. Однако генераторы открывают двери в необъятный мир. Создадим несколько собственных генераторов, которые облегчат работу с массивами, а заодно помогут глубже понять конструкцию `yield`.

В листинге 32.3 приводится пример функции-генератора `collect()`, которая применяет к каждому элементу массива пользовательскую функцию.

Листинг 32.3. Обработка каждого элемента массива. Файл collect.php

```
<?php
function collect($arr, $callback)
{
    foreach($arr as $value) {
        yield $callback($value);
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$collect = collect($arr, function($e){ return $e * $e; });
foreach($collect as $val) {
    echo "$val ";
}
```

Результатом выполнения скрипта из листинга 32.3 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, к каждому из них применяется функция `$callback`, результат которой выталкивается из функции конструкцией `yield`. В качестве пользовательской функции выступает анонимная функция, возвращающая квадрат аргумента.

Похожие возможности предоставляет стандартная функция `array_walk()`. Однако, в отличие от генераторов, `array_walk()` не может фильтровать содержимое массива. Продемонстрируем эту возможность на примере функции-генератора `select()`. Данный генератор также принимает два параметра, первый из которых обрабатываемый массив, а второй является функцией обратного вызова, возвращающей `true`, если элемент следует обрабатывать в массиве, и `false`, если он должен игнорироваться (листинг 32.4).

Листинг 32.4. Извлекаем только четные элементы. Файл select.php

```
<?php
function select($arr, $callback)
{
    foreach($arr as $value) {
        if($callback($value)) yield $value;
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$select = select($arr, function($e){ return $e % 2 == 0 ? true : false; });
foreach($select as $val) {
    echo "$val ";
}
```

В качестве функции обратного вызова 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 элементы (листинг 32.5).

Листинг 32.5. Извлекаем только нечетные элементы. Файл reject.php

```
<?php
function reject($arr, $callback)
{
    foreach($arr as $value) {
        if(!$callback($value)) yield $value;
    }
}

$arr = [1, 2, 3, 4, 5, 6];
$reject = reject($arr, function($e){ return $e % 2 == 0 ? true : false; });
foreach($reject as $val) {
    echo "$val ";
}
```

В листинге 32.5 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 32.6 приводится пример вычисления последовательности квадратов четных элементов массива.

Листинг 32.6. 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, function($e){ return $e % 2 == 0 ? true : false; });
$collect = collect($select, function($e){ return $e * $e; });
foreach($collect as $val) {
    echo "$val ";
}
```

Результатом выполнения скрипта из листинга 32.6 будет последовательность:

```
4 16 36
```

32.3. Экономия ресурсов

Ранее в главе мы рассмотрели генераторы `collect()`, `select()` и `reject()`. Следует отметить, что при их работе не создается копий массива. Если исходный массив занимает несколько мегабайт оперативной памяти, это позволяет значительно сэкономить ресурсы сервера, т. к. на каждой итерации мы имеем дело только с объемом памяти, который занимает элемент массива.

Пусть стоит задача вывести на страницу числа от 0 до 1 024 000 через пробел. Такая страница займет объем где-то 7 Мбайт. Конечно, задача гипотетическая и ее можно решить без использования массива, однако предположим, что обойтись без массива в данной ситуации не получается (листинг 32.7).

Листинг 32.7. Неэкономное расходование памяти. Файл makerange_bad.php

```
<?php
function crange($size)
{
    $arr = [];
```

```
    for($i = 0; $i < $size; $i++) {
        $arr[] = $i;
    }
    return $arr;
}

$range = crange(1024000);
foreach($range as $i) echo "$i ";
// Определяем количество используемой скриптом памяти
echo memory_get_usage() . '<br />';
```

Для определения количества памяти, которое потребляет скрипт, используется функция `memory_get_usage()`, возвращающая количество байтов оперативной памяти, занятых сценарием. В 64-битной версии PHP 7 скрипт из листинга 32.7 потребляет порядка 32 Мбайт.

Если переписать сценарий с использованием генераторов, количество используемой памяти можно значительно сократить (листинг 32.8).

Листинг 32.8. 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 ";
// Определяем количество используемой скриптом памяти
echo memory_get_usage() . '<br />';
```

Запустив этот скрипт в тех же условиях, можно получить 347 Кбайт, т. е. экономия памяти составила почти два порядка. При этом сгенерированная страница занимает объем порядка 7 Мбайт.

32.4. Использование ключей

Оператор `foreach` может извлекать ключи ассоциативных массивов: для этого достаточно указать пару *ключ* `=>` *значение* после ключевого слова `as`:

```
foreach ($array as $key => $value) {
  ...
}
```

Генераторы допускают работу с ключами, для этого после ключевого слова `yield` указывается точно такая же пара (листинг 32.9).

Листинг 32.9. Использование ключей. Файл keys.php

```
<?php
function collect($arr, $callback)
{
    foreach($arr as $key => $value) {
        yield $key => $callback($value);
    }
}

$arr = [
  "first" => 1,
  "second" => 2,
  "third" => 3,
  "fourth" => 4,
  "fifth" => 5,
  "sixth" => 6];
$collect = collect($arr, function($e){ return $e * $e; });
foreach($collect as $key => $val) {
    echo "$val ($key) ";
}
```

Результатом выполнения скрипта из листинга 32.9 является следующая строка:

```
1 (first) 4 (second) 9 (third) 16 (fourth) 25 (fifth) 36 (sixth)
```

Так же как для обычной функции, для генераторов допускается возврат значения по ссылке. По сравнению с обычной функцией в этом гораздо больше смысла, т. к. мы можем влиять на значение внутри генератора (листинг 32.10).

Листинг 32.10. Использование ключей. 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 операторе `foreach`. Результатом выполнения скрипта из листинга 32.10 является следующая строка:

```
2 1 0
```

32.5. Связь генераторов с объектами

На самом деле генератор возвращает объект класса `Generator`, который в свою очередь реализует интерфейс `Iterator` *(см. главу 31)*.

Помимо методов интерфейса `Iterator`, класс `Generator` определяет метод `send()`, который позволяет отправить значение внутрь генератора и использовать `yield` для инициализации переменных внутри генератора (листинг 32.11).

Листинг 32.11. 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 />');
```

Причем нет ограничения на передачу 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 операторов после него, они никогда не выполняются, после `return` интерпретатор покидает функцию.

Однако, начиная с PHP 7, извлечь значение, которое возвращается при помощи оператора `return`, можно посредством еще одного метода — `getReturn()` (листинг 32.12).

Листинг 32.12. Использование `return` в генераторе. Файл return.php

```
<?php
function generator()
{
    yield 1;
    return yield from two_three();
    yield 5;
}
```

```
function two_three()
{
    yield 2;
    yield 3;
    return 4;
}

$generator = generator();

foreach($generator as $i) {
    echo "$i<br />";
}
echo 'return = ' . $generator->getReturn();
```

В результате выполнения скрипта будет возвращен следующий набор строк:

```
1
2
3
return = 4
```

Задания

1. Создайте генератор, который построчно считывал бы содержимое файла.
2. Создайте генератор, который принимал бы адрес RSS-канала, например **https://lenta.ru/rss**, и предоставлял бы доступ к коллекции объектов, представляющих item-позиции XML-файла.
3. Создайте генератор, который построчно считывал бы содержимое таблицы PostgreSQL.
4. Создайте генератор, который предоставлял бы возможность рекурсивного обхода каталога, выводя его содержимое, в том числе вложенных папок. При этом генератор должен принимать необязательные параметры, позволяющие фильтровать файлы по расширению и размеру.

Заключение

Книга завершена, но ваше обучение Web-технологиям на этом не окончено. За пределами книжных страниц осталось множество неохваченных областей, которые вам придется штурмовать при помощи других изданий или самостоятельно. Вот далеко не полный список того, что не вошло в книгу или освещено недостаточно подробно:

- шаблоны проектирования;
- стандарты кодирования PSR;
- современные PHP-фреймворки (Symfony, Laravel и Yii);
- система контроля версий Git;
- протокол, серверы и клиенты SSH;
- протокол HTTP;
- детальное знакомство с современными возможностями SQL;
- очереди (на базе Redis и RabbitMQ);
- Web-серверы (Apache, nginx);
- системы полнотекстового поиска (ElasticSearch, Sphinx);
- регулярные выражения;
- знакомство с основами UNIX;
- автоматическое развертывание серверов (Ansible, Puppet, Chef);
- современные JavaScript-фреймворки (React.js, Backbone.js, Angular);
- обработка изображений средствами GDLib и ImageMagic;
- технологии виртуализации (Docker, Vagrant, VirtualBox);
- тестирование PHP-кода (PHPUnit, phpspec).

Эта своеобразная дорожная карта, по маршрутам которой можно направить свои усилия. Технологии тут не выстроены в каком-то определенном порядке, но все они используются для создания современных Web-сайтов.

Кроме того, следует помнить, что основной залог успеха в программировании — это кодирование. Только разработка реальных проектов приносит глубокое понимание технологий, причин их возникновения, преимуществ и трудностей использования в той или иной ситуации.

Удачи!

Предметный указатель

J

L

M

N

O

P

R

S

U

V

W

Y

Z

А

Б

В

Г

Д

З

И

К

M

H

O

Р

С

Т

У

Ф

Ц

Ч

Ш

Э

Я

самоучитель

PHP 7

Современный PHP — от простого к сложному

Кузнецов Максим Валерьевич, дважды лауреат стипендии Президента РФ, лауреат премии UNESCO, лауреат диплома I степени МГУ им. М. В. Ломоносова. Автор двух десятков книг по Web-разработке и более 50 научных работ.

Симдянов Игорь Вячеславович, ведущий разработчик группы компаний Rambler&Co с 15-летним стажем разработки Web-проектов (Известия, Life.ru, Rambler.ru). Автор двух десятков книг по Web-разработке.

Книга опытных разработчиков описывает последнюю, седьмую версию популярного языка Web-программирования PHP. Рассматриваются не только все нововведения языка, но и изменения в разработке современных Web-сайтов. Объектно-ориентированный подход, необязательный в PHP еще 10 лет назад, стал основной методологией. На смену традиционным базам данных MySQL и memcached приходят объектно-ориентированная СУБД PostgreSQL и базы данных NoSQL (Redis и подобные). Библиотеки в PHP теперь распространяются через Интернет при помощи менеджера пакетов Composer. Возможности языка PHP и сопутствующих технологий настолько возросли, что описать их в рамках одной книги становится затруднительно. По этой причине авторы ставили перед собой двойную цель: во-первых, систематически изложить язык PHP настолько полно, насколько это возможно, а во-вторых, снабдить каждую из глав заданиями, выполняя которые можно закрепить материал и познакомиться с неохваченными разделами языка и инструментами современного Web-разработчика. Книга будет интересна не только читателям, впервые знакомящимся с языком, но и профессионалам, заинтересованным в освоении современного PHP.

Все исходные коды можно скачать по ссылкам ftp://ftp.bhv.ru/9785977538176.zip и https://github.com/igorsimdyanov/phpworkshop, а также со страницы книги на сайте www.bhv.ru

ISBN 978-5-9775-3817-6

БХВ-ПЕТЕРБУРГ
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