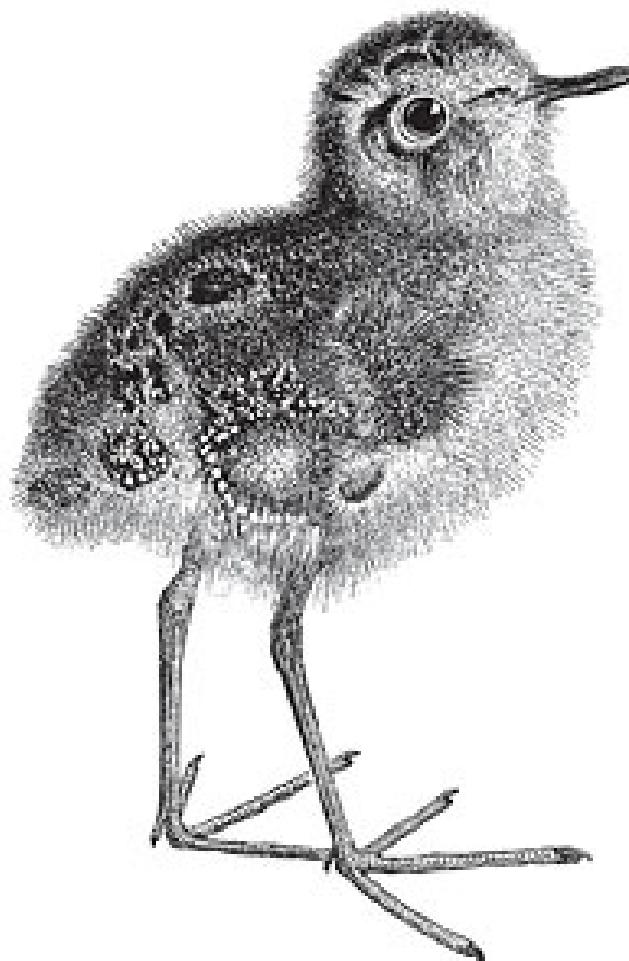


O'REILLY®

6th Edition

# Learning PHP, MySQL & JavaScript

A Step-by-Step Guide to Creating  
Dynamic Websites



Early  
Release  
RAW &  
UNEDITED

Robin Nixon

# **Learning PHP, MySQL & JavaScript**

SIXTH EDITION

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

With PHP 8, MySQL 8, PDO, CSS, HTML5, jQuery & React

**Robin Nixon**

# **Learning PHP, MySQL & JavaScript**

by Robin Nixon

Copyright © 2021 Robin Nixon. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,  
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Editor: Rachel Roumeliotis

Production Editor: Caitlin Ghegan

Copyeditor: TK

Proofreader: TK

Indexer: TK

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Catherine Dullea

July 2009: First Edition

August 2012: Second Edition

June 2014: Third Edition

December 2014: Fourth Edition

May 2018: Fifth Edition

November 2021

## **Revision History for the Early Release**

- 2021-02-04: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492093817> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.

*Learning PHP, MySQL & JavaScript*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-09381-7

[LSI]

# Preface

---

The combination of PHP and MySQL is the most convenient approach to dynamic, database-driven web design, holding its own in the face of challenges from integrated frameworks—such as Ruby on Rails—that are harder to learn. Due to its open source roots (unlike the competing Microsoft .NET Framework), it is free to implement and is therefore an extremely popular option for web development.

Any would-be developer on a Unix/Linux or even a Windows/Apache platform will need to master these technologies. And, combined with the partner technologies of JavaScript, React, CSS, and HTML5, you will be able to create websites of the caliber of industry standards like Facebook, Twitter, and Gmail.

## Audience

This book is for people who wish to learn how to create effective and dynamic websites. This may include webmasters or graphic designers who are already creating static websites but wish to take their skills to the next level, as well as high school and college students, recent graduates, and self-taught individuals.

In fact, anyone ready to learn the fundamentals behind responsive web design will obtain a thorough grounding in the core technologies of PHP, MySQL, JavaScript, CSS, and HTML5, and you'll learn the basics of the React library and React Native Framework, too.

## Assumptions This Book Makes

This book assumes that you have a basic understanding of HTML and can at least put together a simple, static website, but does not assume that you have any prior knowledge of PHP, MySQL, JavaScript, CSS, or HTML5—although if you do, your progress through the book will be even quicker.

## Organization of This Book

The chapters in this book are written in a specific order, first introducing all of the core technologies it covers and then walking you through their installation on a web development server so that you will be ready to work through the examples.

In the first section, you will gain a grounding in the PHP programming language, covering the basics of syntax, arrays, functions, and object-oriented programming.

Then, with PHP under your belt, you will move on to an introduction to the MySQL database system, where you will learn everything from how MySQL databases are structured to how to generate complex queries.

After that, you will learn how you can combine PHP and MySQL to start creating your own dynamic web pages by integrating forms and other HTML features. You will then get down to the nitty-gritty practical aspects of PHP and MySQL development by learning a variety of useful functions and how to manage cookies and sessions, as well as how to maintain a high level of security.

In the next few chapters, you will gain a thorough grounding in JavaScript, from simple functions and event handling to accessing the Document Object Model, in-browser validation, and error handling. You'll also get a comprehensive primer on using the popular React library for JavaScript.

With an understanding of all three of these core technologies, you will then learn how to make behind-the-scenes Ajax calls and turn your websites into highly dynamic environments.

Next, you'll spend two chapters learning all about using CSS to style and lay out your web pages, before discovering how the React libraries can make your development job a great deal easier. You'll then move on to the final section on the interactive features built into HTML5, including geolocation, audio, video, and the canvas. After this, you'll put together everything you've learned in a complete set of programs that together constitute a fully functional social networking website.

Along the way, you'll find plenty of advice on good programming practices and tips that can help you find and solve hard-to-detect programming errors. There are also plenty of links to websites containing further details on the topics covered.

## Supporting Books

Once you have learned to develop using PHP, MySQL, JavaScript, CSS, and HTML5, you will be ready to take your skills to the next level using the following O'Reilly reference books:

- *Dynamic HTML: The Definitive Reference* by Danny Goodman
- *PHP in a Nutshell* by Paul Hudson
- *MySQL in a Nutshell* by Russell Dyer
- *JavaScript: The Definitive Guide* by David Flanagan
- *CSS: The Definitive Guide* by Eric A. Meyer and Estelle Weyl
- *HTML5: The Missing Manual* by Matthew MacDonald

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Plain text*

Indicates menu titles, options, and buttons.

### *Italic*

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities. Also used for database, table, and column names.

### *Constant width*

Indicates commands and command-line options, variables and other code elements, HTML tags, and the contents of files.

### ***Constant width bold***

Shows program output and is used to highlight sections of code that are discussed in the text.

### *Constant width italic*

Shows text that should be replaced with user-supplied values.

#### **NOTE**

This element signifies a tip, suggestion, or general note.

#### **WARNING**

This element indicates a warning or caution.

## **Using Code Examples**

Supplemental material (code examples, exercises, etc.) is available for download at [github.com/RobinNixon/lpmj6](https://github.com/RobinNixon/lpmj6).

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a set of examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Learning PHP, MySQL & JavaScript 6th Edition* by Robin Nixon (O’Reilly). Copyright 2021 Robin Nixon, [[[ISBN NUMBER GOES HERE]]].”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## O’Reilly Online Learning

### NOTE

For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (in the United States or Canada)

(707) 829-0515 (international or local)

(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *[[ERRATA URL GOES HERE]]*.

To comment or ask technical questions about this book, send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

For news and more information about our books and courses, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

## Acknowledgments

I would like to thank Senior Content Acquisitions Editor, Amanda Quinn, Content Development Editor, Melissa Potter, and everyone who worked so hard on this book, including ???, ??? & ??? for their comprehensive

technical reviews, ??? for overseeing production, ??? for copy editing, ??? for proofreading, ??? for creating the index, Karen Montgomery for the original sugar glider front cover design, ??? for the latest book cover, my original editor, Andy Oram, for overseeing the first five editions, and everyone else too numerous to name who submitted errata and offered suggestions for this new edition.

# Chapter 1. Introduction to Dynamic Web Content

---

The World Wide Web is a constantly evolving network that has already traveled far beyond its conception in the early 1990s, when it was created to solve a specific problem. State-of-the-art experiments at CERN (the European Laboratory for Particle Physics, now best known as the operator of the Large Hadron Collider) were producing incredible amounts of data—so much that the data was proving unwieldy to distribute to the participating scientists, who were spread out across the world.

At this time, the internet was already in place, connecting several hundred thousand computers, so Tim Berners-Lee (a CERN fellow) devised a method of navigating between them using a hyperlinking framework, which came to be known as Hypertext Transfer Protocol, or HTTP. He also created a markup language called Hypertext Markup Language, or HTML. To bring these together, he wrote the first web browser and web server.

Today we take these tools for granted, but back then, the concept was revolutionary. The most connectivity so far experienced by at-home modem users was dialing up and connecting to a bulletin board that was hosted by a single computer, where you could communicate and swap data only with other users of that service. Consequently, you needed to be a member of many bulletin board systems in order to effectively communicate electronically with your colleagues and friends.

But Berners-Lee changed all that in one fell swoop, and by the mid-1990s, there were three major graphical web browsers competing for the attention of 5 million users. It soon became obvious, though, that something was missing. Yes, pages of text and graphics with hyperlinks to take you to other pages was a brilliant concept, but the results didn't reflect the instantaneous potential of computers and the internet to meet the particular needs of each user with dynamically changing content. Using the web was a very dry and

plain experience, even if we did now have scrolling text and animated GIFs!

Shopping carts, search engines, and social networks have clearly altered how we use the web. In this chapter, we'll take a brief look at the various components that make up the web, and the software that helps make using it a rich and dynamic experience.

### NOTE

It is necessary to start using some acronyms more or less right away. I have tried to clearly explain them before proceeding, but don't worry too much about what they stand for or what these names mean, because the details will become clear as you read on.

## HTTP and HTML: Berners-Lee's Basics

HTTP is a communication standard governing the requests and responses that are sent between the browser running on the end user's computer and the web server. The server's job is to accept a request from the client and attempt to reply to it in a meaningful way, usually by serving up a requested web page—that's why the term *server* is used. The natural counterpart to a server is a *client*, so that term is applied both to the web browser and the computer on which it's running.

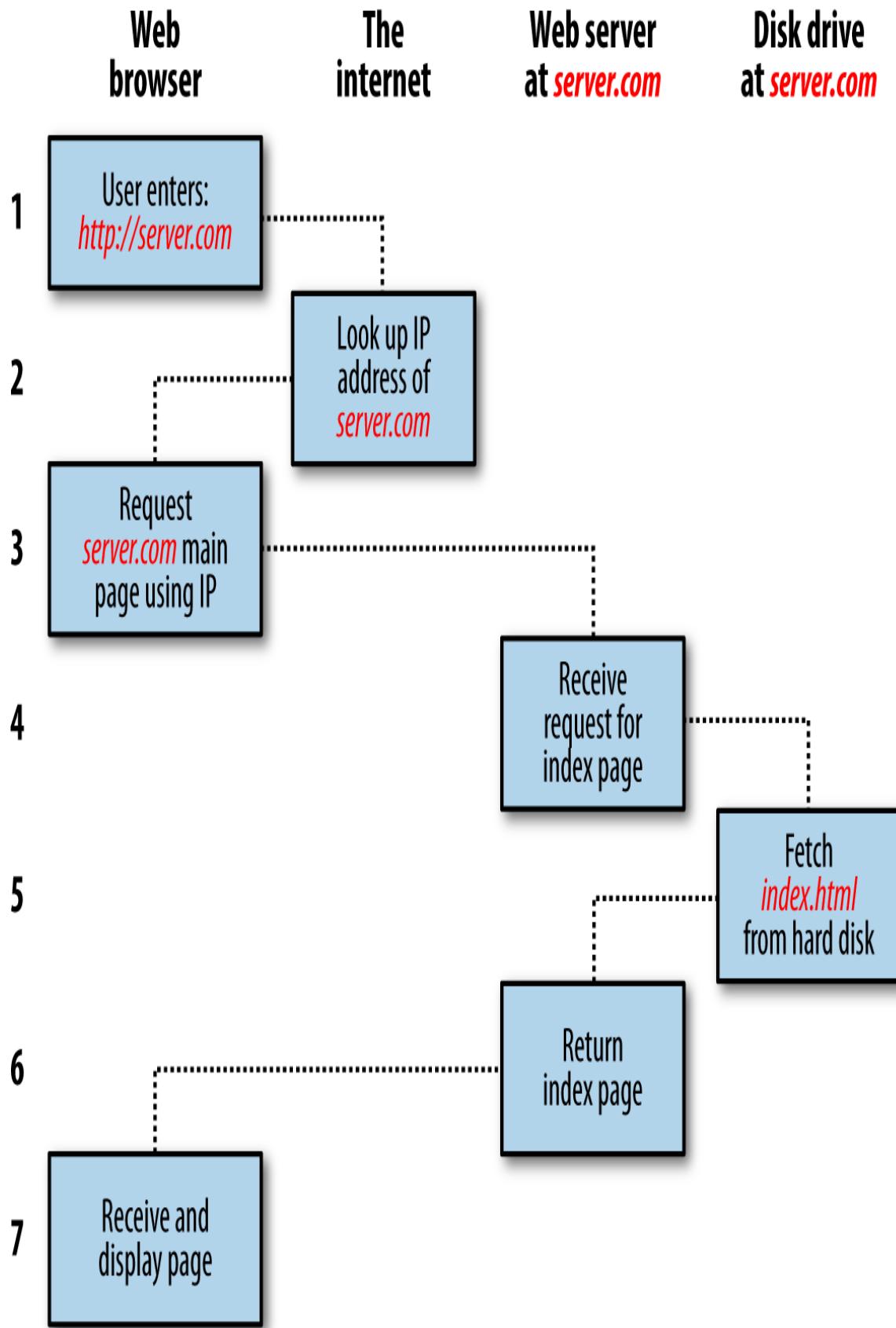
Between the client and the server there can be several other devices, such as routers, proxies, gateways, and so on. They serve different roles in ensuring that the requests and responses are correctly transferred between the client and server. Typically, they use the internet to send this information. Some of these in-between devices can also help speed up the internet by storing pages or information locally in what is called a *cache*, and then serving this content up to clients directly from the cache rather than fetching it all the way from the source server.

A web server can usually handle multiple simultaneous connections, and when not communicating with a client, it spends its time listening for an

incoming connection. When one arrives, the server sends back a response to confirm its receipt.

## The Request/Response Procedure

At its most basic level, the request/response process consists of a web browser asking the web server to send it a web page and the server sending back the page. The browser then takes care of displaying the page (see [Figure 1-1](#)).



*Figure 1-1. The basic client/server request/response sequence*

The steps in the request and response sequence are as follows:

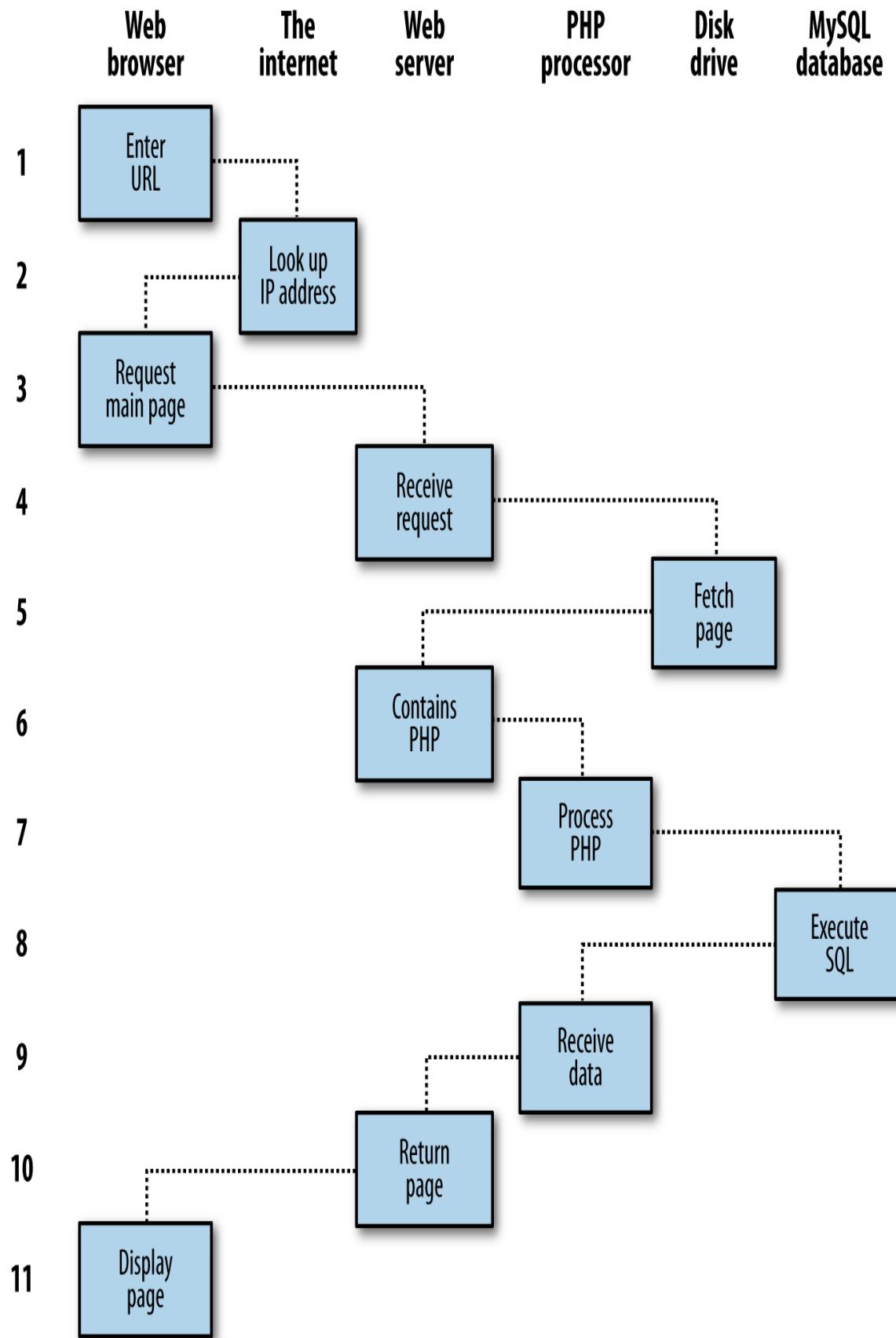
1. You enter `http://server.com` into your browser's address bar.
2. Your browser looks up the Internet Protocol (IP) address for `server.com`.
3. Your browser issues a request for the home page at `server.com`.
4. The request crosses the internet and arrives at the `server.com` web server.
5. The web server, having received the request, looks for the web page on its disk.
6. The web server retrieves the page and returns it to the browser.
7. Your browser displays the web page.

For an average web page, this process also takes place once for each object within the page: a graphic, an embedded video or Flash file, and even a CSS template.

In step 2, notice that the browser looks up the IP address of `server.com`. Every machine attached to the internet has an IP address—your computer included—but we generally access web servers by name, such as `google.com`. As you probably know, the browser consults an additional internet service called the Domain Name Service (DNS) to find the server's associated IP address and then uses it to communicate with the computer.

For dynamic web pages, the procedure is a little more involved, because it may bring both PHP and MySQL into the mix. For instance, you may click on a picture of a raincoat. Then PHP will put together a request using the standard database language, SQL—many of whose commands you will learn in this book—and send the request to the MySQL server. The MySQL server will return information about the raincoat you selected, and the PHP

code will wrap it all up in some HTML, which the server will send to your browser (see [Figure 1-2](#)).



*Figure 1-2. A dynamic client/server request/response sequence*

The steps are as follows:

1. You enter `http://server.com` into your browser's address bar.
2. Your browser looks up the IP address for `server.com`.
3. Your browser issues a request to that address for the web server's home page.
4. The request crosses the internet and arrives at the `server.com` web server.
5. The web server, having received the request, fetches the home page from its hard disk.
6. With the home page now in memory, the web server notices that it is a file incorporating PHP scripting and passes the page to the PHP interpreter.
7. The PHP interpreter executes the PHP code.
8. Some of the PHP contains SQL statements, which the PHP interpreter now passes to the MySQL database engine.
9. The MySQL database returns the results of the statements to the PHP interpreter.
10. The PHP interpreter returns the results of the executed PHP code, along with the results from the MySQL database, to the web server.
11. The web server returns the page to the requesting client, which displays it.

Although it's helpful to be aware of this process so that you know how the three elements work together, in practice you don't really need to concern yourself with these details, because they all happen automatically.

The HTML pages returned to the browser in each example may well contain JavaScript, which will be interpreted locally by the client, and

which could initiate another request—the same way embedded objects such as images would.

## The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5

At the start of this chapter, I introduced the world of Web 1.0, but it wasn't long before the rush was on to create Web 1.1, with the development of such browser enhancements as Java, JavaScript, JScript (Microsoft's slight variant of JavaScript), and ActiveX. On the server side, progress was being made on the Common Gateway Interface (CGI) using scripting languages such as Perl (an alternative to the PHP language) and *server-side scripting*—inserting the contents of one file (or the output of running a local program) into another one dynamically.

Once the dust had settled, three main technologies stood head and shoulders above the others. Although Perl was still a popular scripting language with a strong following, PHP's simplicity and built-in links to the MySQL database program had earned it more than double the number of users. And JavaScript, which had become an essential part of the equation for dynamically manipulating Cascading Style Sheets (CSS) and HTML, now took on the even more muscular task of handling the client side of the asynchronous communication (exchanging data between a client and server after a web page has loaded). Using asynchronous communication, web pages perform data handling and send requests to web servers in the background—without the web user being aware that this is going on.

No doubt the symbiotic nature of PHP and MySQL helped propel them both forward, but what attracted developers to them in the first place? The simple answer has to be the ease with which you can use them to quickly create dynamic elements on websites. MySQL is a fast and powerful yet easy-to-use database system that offers just about anything a website would need in order to find and serve up data to browsers. When PHP allies with MySQL to store and retrieve this data, you have the fundamental parts

required for the development of social networking sites and the beginnings of Web 2.0.

And when you bring JavaScript and CSS into the mix too, you have a recipe for building highly dynamic and interactive websites—especially as there is now a wide range of sophisticated frameworks of JavaScript functions you can call on to really speed up web development, such as the well-known jQuery, which until very recently was one of the most common way programmers access asynchronous communication features, and the more recent React JavaScript library which has been growing quickly in popularity, and is now one of the most widely downloaded and implemented frameworks, so much so that since 2020 the Indeed job site lists more than twice as many positions for React developers than for jQuery.

## **MariaDB: The MySQL Clone**

After Oracle purchased Sun Microsystems (the owners of MySQL), the community became wary that MySQL might not remain fully open source, so MariaDB was forked from it to keep it free under the GNU GPL.

Development of MariaDB is led by some of the original developers of MySQL and it retains exceedingly close compatibility with MySQL. Therefore, you may well encounter MariaDB on some servers in place of MySQL—but not to worry, everything in this book works equally well on both MySQL and MariaDB, which is based on the same code base as MySQL Server 5.5. To all intents and purposes you can swap one with the other and notice no difference.

Anyway, as it turns out, many of the initial fears appear to have been allayed as MySQL remains open source, with Oracle simply charging for support and for editions that provide additional features such as geo-replication and automatic scaling. However, unlike MariaDB, MySQL is no longer community driven, so knowing that MariaDB will always be there if ever needed will keep many developers sleeping at night, and probably ensures that MySQL itself will remain open source.

## Using PHP

With PHP, it's a simple matter to embed dynamic activity in web pages. When you give pages the *.php* extension, they have instant access to the scripting language. From a developer's point of view, all you have to do is write code such as the following:

```
<?php  
    echo " Today is " . date("l") . ". ";  
?>
```

Here's the latest news.

The opening `<?php` tells the web server to allow the PHP program to interpret all the following code up to the `?>` tag. Outside of this construct, everything is sent to the client as direct HTML. So, the text `Here's the latest news.` is simply output to the browser; within the PHP tags, the built-in `date` function displays the current day of the week according to the server's system time.

The final output of the two parts looks like this:

**Today is Wednesday. Here's the latest news.**

PHP is a flexible language, and some people prefer to place the PHP construct directly next to PHP code, like this:

Today is <?php echo date("l"); ?>. Here's the latest news.

There are even more ways of formatting and outputting information, which I'll explain in the chapters on PHP. The point is that with PHP, web developers have a scripting language that, although not as fast as compiling your code in C or a similar language, is incredibly speedy and also integrates seamlessly with HTML markup.

## NOTE

If you intend to enter the PHP examples in this book into a program editor to work along with me, you must remember to add `<?php` in front and `?>` after them to ensure that the PHP interpreter processes them. To facilitate this, you may wish to prepare a file called `example.php` with those tags in place.

Using PHP, you have unlimited control over your web server. Whether you need to modify HTML on the fly, process a credit card, add user details to a database, or fetch information from a third-party website, you can do it all from within the same PHP files in which the HTML itself resides.

## Using MySQL

Of course, there's not a lot of point to being able to change HTML output dynamically unless you also have a means to track the information users provide to your website as they use it. In the early days of the web, many sites used "flat" text files to store data such as usernames and passwords. But this approach could cause problems if the file wasn't correctly locked against corruption from multiple simultaneous accesses. Also, a flat file can get only so big before it becomes unwieldy to manage—not to mention the difficulty of trying to merge files and perform complex searches in any kind of reasonable time.

That's where relational databases with structured querying become essential. And MySQL, being free to use and installed on vast numbers of internet web servers, rises superbly to the occasion. It is a robust and exceptionally fast database management system that uses English-like commands.

The highest level of MySQL structure is a database, within which you can have one or more tables that contain your data. For example, let's suppose you are working on a table called `users`, within which you have created columns for `surname`, `firstname`, and `email`, and you now wish to add another user. One command that you might use to do this is as follows:

```
INSERT INTO users VALUES('Smith', 'John', 'jsmith@mysite.com');
```

You will previously have issued other commands to create the database and table and to set up all the correct fields, but the SQL `INSERT` command here shows how simple it can be to add new data to a database. SQL is a language designed in the early 1970s that is reminiscent of one of the oldest programming languages, COBOL. It is well suited, however, to database queries, which is why it is still in use after all this time.

It's equally easy to look up data. Let's assume that you have an email address for a user and need to look up that person's name. To do this, you could issue a MySQL query such as the following:

```
SELECT surname,firstname FROM users WHERE email='jsmith@mysite.com';
```

MySQL will then return `Smith`, `John` and any other pairs of names that may be associated with that email address in the database.

As you'd expect, there's quite a bit more that you can do with MySQL than just simple `INSERT` and `SELECT` commands. For example, you can combine related data sets to bring related pieces of information together, ask for results in a variety of orders, make partial matches when you know only part of the string that you are searching for, return only the *n*th result, and a lot more.

Using PHP, you can make all these calls directly to MySQL without having to directly access the MySQL command-line interface yourself. This means you can save the results in arrays for processing and perform multiple lookups, each dependent on the results returned from earlier ones, to drill down to the item of data you need.

For even more power, as you'll see later, there are additional functions built right into MySQL that you can call up to efficiently run common operations within MySQL, rather than creating them out of multiple PHP calls to MySQL.

## Using JavaScript

The oldest of the three core technologies discussed in this book, JavaScript, was created to enable scripting access to all the elements of an HTML document. In other words, it provides a means for dynamic user interaction such as checking email address validity in input forms and displaying prompts such as “Did you really mean that?” (although it cannot be relied upon for security, which should always be performed on the web server).

Combined with CSS (see the following section), JavaScript is the power behind dynamic web pages that change in front of your eyes rather than when a new page is returned by the server.

However, JavaScript can also be tricky to use, due to some major differences in the ways different browser designers have chosen to implement it. This mainly came about when some manufacturers tried to put additional functionality into their browsers at the expense of compatibility with their rivals.

Thankfully, the developers have mostly now come to their senses and have realized the need for full compatibility with one another, so it is less necessary these days to have to optimize your code for different browsers. However, there remain millions of users using legacy browsers, and this will likely be the case for a good many years to come. Luckily, there are solutions for the incompatibility problems, and later in this book we’ll look at libraries and techniques that enable you to safely ignore these differences.

For now, let’s take a look at how to use basic JavaScript, accepted by all browsers:

```
<script type="text/javascript">
  document.write("Today is " + Date() );
</script>
```

This code snippet tells the web browser to interpret everything within the `<script>` tags as JavaScript, which the browser does by writing the text

**Today** is to the current document, along with the date, using the JavaScript function **Date**. The result will look something like this:

**Today is Wed Jan 01 2025 01:23:45**

### NOTE

Unless you need to specify an exact version of JavaScript, you can normally omit the `type="text/javascript"` and just use `<script>` to start the interpretation of the JavaScript.

As previously mentioned, JavaScript was originally developed to offer dynamic control over the various elements within an HTML document, and that is still its main use. But more and more, JavaScript is being used for *asynchronous communication*, the process of accessing the web server in the background.

Asynchronous communication is what allows web pages to begin to resemble standalone programs, because they don't have to be reloaded in their entirety to display new content. Instead, an asynchronous call can pull in and update a single element on a web page, such as changing your photograph on a social networking site or replacing a button that you click with the answer to a question. This subject is fully covered in Chapter 18.

Then, in Chapter 24, we take a good look at the jQuery framework, which you can use to save reinventing the wheel when you need fast, cross-browser code to manipulate your web pages. Of course, there are other frameworks available too, so we also take a look at React, one of the most popular choices of today, in Chapter 24 . Both are extremely reliable, and are major tools in the utility kits of many seasoned web developers.

## Using CSS

CSS is the crucial companion to HTML, ensuring that the HTML text and embedded images are laid out consistently and in a manner appropriate for

the user's screen. With the emergence of the CSS3 standard in recent years, CSS now offers a level of dynamic interactivity previously supported only by JavaScript. For example, not only can you style any HTML element to change its dimensions, colors, borders, spacing, and so on, but now you can also add animated transitions and transformations to your web pages, using only a few lines of CSS.

Using CSS can be as simple as inserting a few rules between `<style>` and `</style>` tags in the head of a web page, like this:

```
<style>
  p {
    text-align:justify;
    font-family:Helvetica;
  }
</style>
```

These rules change the default text alignment of the `<p>` tag so that paragraphs contained in it are fully justified and use the Helvetica font.

As you'll learn in Chapter 19 , there are many different ways you can lay out CSS rules, and you can also include them directly within tags or save a set of rules to an external file to be loaded in separately. This flexibility not only lets you style your HTML precisely, but can also (for example) provide built-in hover functionality to animate objects as the mouse passes over them. You will also learn how to access all of an element's CSS properties from JavaScript as well as HTML.

## And Then There's HTML5

As useful as all these additions to the web standards became, they were not enough for ever more ambitious developers. For example, there was still no simple way to manipulate graphics in a web browser without resorting to plug-ins such as Flash. And the same went for inserting audio and video into web pages. Plus, several annoying inconsistencies had crept into HTML during its evolution.

So, to clear all this up and take the internet beyond Web 2.0 and into its next iteration, a new standard for HTML was created to address all these shortcomings: *HTML5*. Its development began as long ago as 2004, when the first draft was drawn up by the Mozilla Foundation and Opera Software (developers of two popular web browsers), but it wasn't until the start of 2013 that the final draft was submitted to the World Wide Web Consortium (W3C), the international governing body for web standards.

It has taken a few years for HTML5 to develop, but now we are at a very solid and stable version 5.1 (since 2016). It's a never-ending cycle of development, though, and more functionality is sure to be built into it over time, with version 5.2 (planned to make the plugin system obsolete) released as a W3C recommendation in 2017, and HTML 5.3 (with proposed features such as auto-capitalisation) still in planning as of 2020, and so on. Some of the best features in HTML5 for handling and displaying media include the `<audio>`, `<video>`, and `<canvas>` elements, which add sound, video, and advanced graphics. Everything you need to know about these and all other aspects of HTML5 is covered in detail starting in Chapter 25 .

#### NOTE

One of the little things I like about the HTML5 specification is that XHTML syntax is no longer required for self-closing elements. In the past, you could display a line break using the `<br>` element. Then, to ensure future compatibility with XHTML (the planned replacement for HTML that never happened), this was changed to `<br />`, in which a closing / character was added (since all elements were expected to include a closing tag featuring this character). But now things have gone full circle, and you can use either version of these types of elements. So, for the sake of brevity and fewer keystrokes, in this book I have reverted to the former style of `<br>`, `<hr>`, and so on.

## The Apache Web Server

In addition to PHP, MySQL, JavaScript, CSS, and HTML5, there's a sixth hero in the dynamic web: the web server. In the case of this book, that means the Apache web server. We've discussed a little of what a web server

does during the HTTP server/client exchange, but it does much more behind the scenes.

For example, Apache doesn't serve up just HTML files—it handles a wide range of files, from images and Flash files to MP3 audio files, RSS (Really Simple Syndication) feeds, and so on. And these objects don't have to be static files such as GIF images. They can all be generated by programs such as PHP scripts. That's right: PHP can even create images and other files for you, either on the fly or in advance to serve up later.

To do this, you normally have modules either precompiled into Apache or PHP or called up at runtime. One such module is the GD (Graphics Draw) library, which PHP uses to create and handle graphics.

Apache also supports a huge range of modules of its own. In addition to the PHP module, the most important for your purposes as a web programmer are the modules that handle security. Other examples are the Rewrite module, which enables the web server to handle a range of URL types and rewrite them to its own internal requirements, and the Proxy module, which you can use to serve up often-requested pages from a cache to ease the load on the server.

Later in the book, you'll see how to use some of these modules to enhance the features provided by the three core technologies.

## Handling Mobile Devices

We are now firmly in a world of interconnected mobile computing devices, and the concept of developing websites solely for desktop computers has become rather dated. Instead, developers now aim to develop responsive websites and web apps that tailor themselves to the environment in which they find themselves running.

So, new in this edition, I show how you can easily create these types of products using just the technologies detailed in this book, along with the powerful jQuery Mobile library of responsive JavaScript functions. With it, you'll be able to focus on the content and usability of your websites and

web apps, knowing that how they display will be automatically optimized for a range of different computing devices—one less thing for you to worry about.

To demonstrate how to make full use of its power, the final chapter of this book creates a simple social networking example website, using jQuery to make it fully responsive and ensure it displays well on anything from a small mobile phone screen to a tablet or a desktop computer. We could equally have used React (or other JavaScript libraries or frameworks), but perhaps that's an exercise you'll like to set yourself once you've completed this book.

## About Open Source

The technologies in this book are open source: anyone is allowed to read and change the code. Whether or not this status is the reason these technologies are so popular has often been debated, but PHP, MySQL, and Apache *are* the three most commonly used tools in their categories. What can be said definitively, though, is that their being open source means that they have been developed in the community by teams of programmers writing the features they themselves want and need, with the original code available for all to see and change. Bugs can be found quickly and security breaches can be prevented before they happen.

There's another benefit: all these programs are free to use. There's no worrying about having to purchase additional licenses if you have to scale up your website and add more servers, and you don't need to check the budget before deciding whether to upgrade to the latest versions of these products.

## Bringing It All Together

The real beauty of PHP, MySQL, JavaScript (sometimes aided by React or other frameworks), CSS, and HTML5 is the wonderful way in which they all work together to produce dynamic web content: PHP handles all the

main work on the web server, MySQL manages all the data, and the combination of CSS and JavaScript looks after web page presentation. JavaScript can also talk with your PHP code on the web server whenever it needs to update something (either on the server or on the web page). And with the powerful new features in HTML5, such as the canvas, audio and video, and geolocation, you can make your web pages highly dynamic, interactive, and multimedia-packed.

Without using program code, let's summarize the contents of this chapter by looking at the process of combining some of these technologies into an everyday asynchronous communication feature that many websites use: checking whether a desired username already exists on the site when a user is signing up for a new account. A good example of this can be seen with Gmail (see [Figure 1-3](#)).

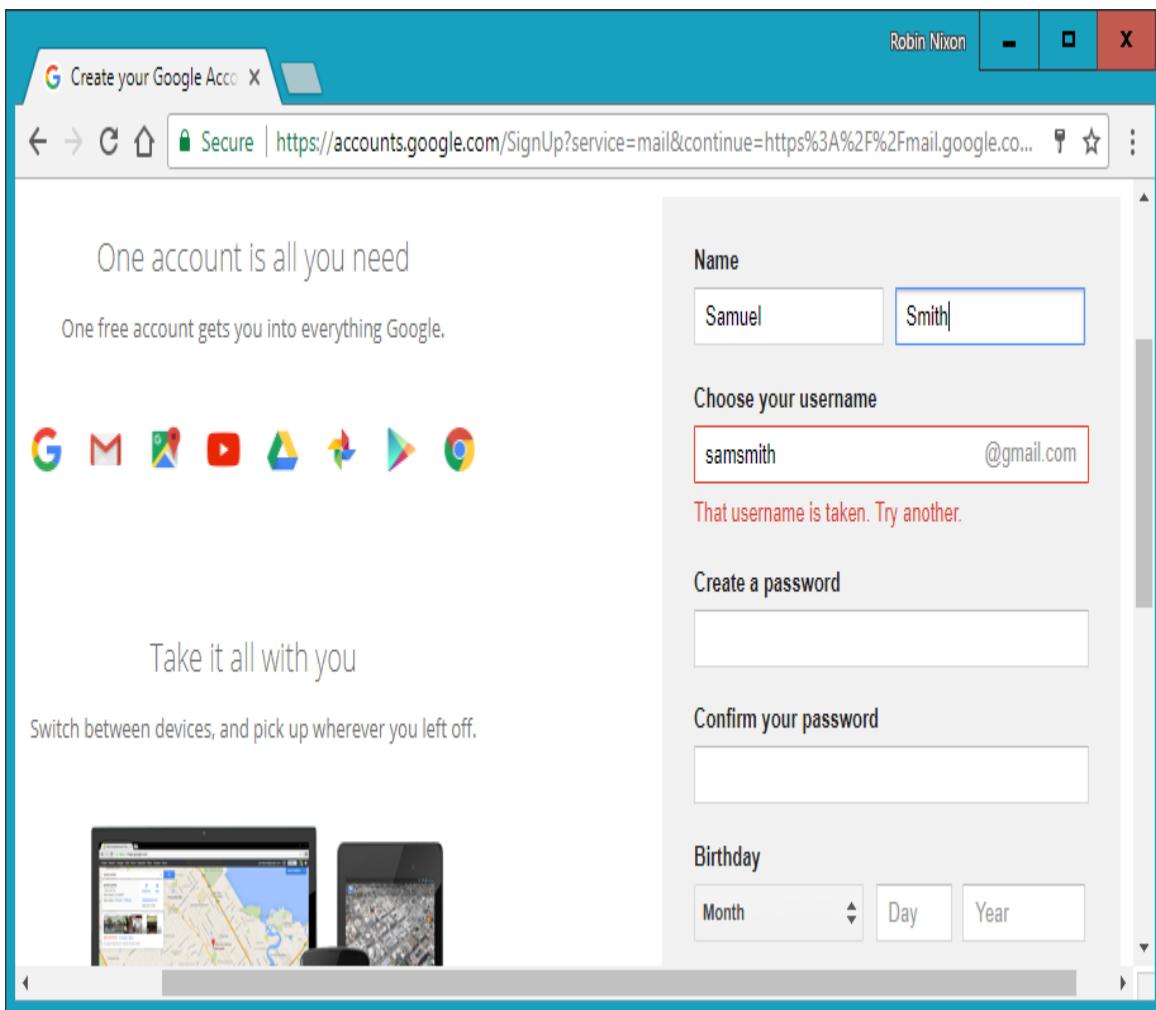


Figure 1-3. Gmail uses asynchronous communication to check the availability of usernames

The steps involved in this asynchronous process will be similar to the following:

1. The server outputs the HTML to create the web form, which asks for the necessary details, such as username, first name, last name, and email address.
2. At the same time, the server attaches some JavaScript to the HTML to monitor the username input box and check for two things: whether some text has been typed into it, and whether the input has been deselected because the user has clicked on another input box.

3. Once the text has been entered and the field deselected, in the background the JavaScript code passes the username that was entered back to a PHP script on the web server and awaits a response.
4. The web server looks up the username and replies back to the JavaScript regarding whether that name has already been taken.
5. The JavaScript then places an indication next to the username input box to show whether the name is available to the user—perhaps a green checkmark or a red cross graphic, along with some text.
6. If the username is not available and the user still submits the form, the JavaScript interrupts the submission and reemphasizes (perhaps with a larger graphic and/or an alert box) that the user needs to choose another username.
7. Optionally, an improved version of this process could even look at the username requested by the user and suggest an alternative that is currently available.

All of this takes place quietly in the background and makes for a comfortable and seamless user experience. Without asynchronous communication, the entire form would have to be submitted to the server, which would then send back HTML, highlighting any mistakes. It would be a workable solution, but nowhere near as tidy or pleasurable as on-the-fly form field processing.

Asynchronous communication can be used for a lot more than simple input verification and processing, though; we'll explore many additional things that you can do with it later in this book.

In this chapter, you have read a good introduction to the core technologies of PHP, MySQL, JavaScript, CSS, and HTML5 (as well as Apache), and have learned how they work together. In [Chapter 2](#), we'll look at how you can install your own web development server on which to practice everything that you will be learning.

# Questions

1. What four components (at the minimum) are needed to create a fully dynamic web page?
2. What does *HTML* stand for?
3. Why does the name *MySQL* contain the letters *SQL*?
4. PHP and JavaScript are both programming languages that generate dynamic results for web pages. What is their main difference, and why would you use both of them?
5. What does *CSS* stand for?
6. List three major new elements introduced in HTML5.
7. If you encounter a bug (which is rare) in one of the open source tools, how do you think you could get it fixed?
8. Why is a framework such as jQuery or React so important for developing modern websites and web apps?

See “[Chapter 1 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 2. Setting Up a Development Server

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

If you wish to develop internet applications but don’t have your own development server, you will have to upload every modification you make to a server somewhere else on the web before you can test it.

Even on a fast broadband connection, this can still represent a significant slowdown in development time. On a local computer, however, testing can be as easy as saving an update (usually just a matter of clicking once on an icon) and then hitting the Refresh button in your browser.

Another advantage of a development server is that you don’t have to worry about embarrassing errors or security problems while you’re writing and testing, whereas you need to be aware of what people may see or do with your application when it’s on a public website. It’s best to iron everything out while you’re still on a home or small office system, presumably protected by firewalls and other safeguards.

Once you have your own development server, you'll wonder how you ever managed without one, and it's easy to set one up. Just follow the steps in the following sections, using the appropriate instructions for a PC, a Mac, or a Linux system.

In this chapter, we cover just the server side of the web experience, as described in [Chapter 1](#). But to test the results of your work—particularly when we start using JavaScript, CSS, and HTML5 later in this book—you should ideally have an instance of every major web browser running on some system convenient to you. Whenever possible, the list of browsers should include at least Microsoft Edge, Mozilla Firefox, Opera, Safari, and Google Chrome. If you plan to ensure that your sites look good on mobile devices too, you should try to arrange access to a wide range of iOS and Android devices.

## What Is a WAMP, MAMP, or LAMP?

WAMP, MAMP, and LAMP are abbreviations for “Windows, Apache, MySQL, and PHP,” “Mac, Apache, MySQL, and PHP,” and “Linux, Apache, MySQL, and PHP.” These abbreviations each describe a fully functioning setup used for developing dynamic internet web pages.

WAMPs, MAMPs, and LAMPs come in the form of packages that bind the bundled programs together so that you don't have to install and set them up separately. This means you can simply download and install a single program and follow a few easy prompts to get your web development server up and running fast, with minimal hassle.

During installation, several default settings are created for you. The security configurations of such an installation will not be as tight as on a production web server, because it is optimized for local use. For these reasons, you should never install such a setup as a production server.

However, for developing and testing websites and applications, one of these installations should be entirely sufficient.

## WARNING

If you choose not to go the WAMP/MAMP/LAMP route for building your own development system, you should know that downloading and integrating the various parts yourself can be very time-consuming and may require a lot of research in order to configure everything fully. But if you already have all the components installed and integrated with one another, they should work with the examples in this book.

## Installing AMPPS on Windows

There are several available WAMP servers, each offering slightly different configurations. Of the various open source and free options, one of the best is AMPPS. You can download it by clicking the button on the website's [home page](#), shown in #ampps-website.

I recommend that you always download the latest stable release (as I write this, it's 3.9, which is about 114 MB in size). The various Windows, macOS, and Linux installers are listed on the download page.

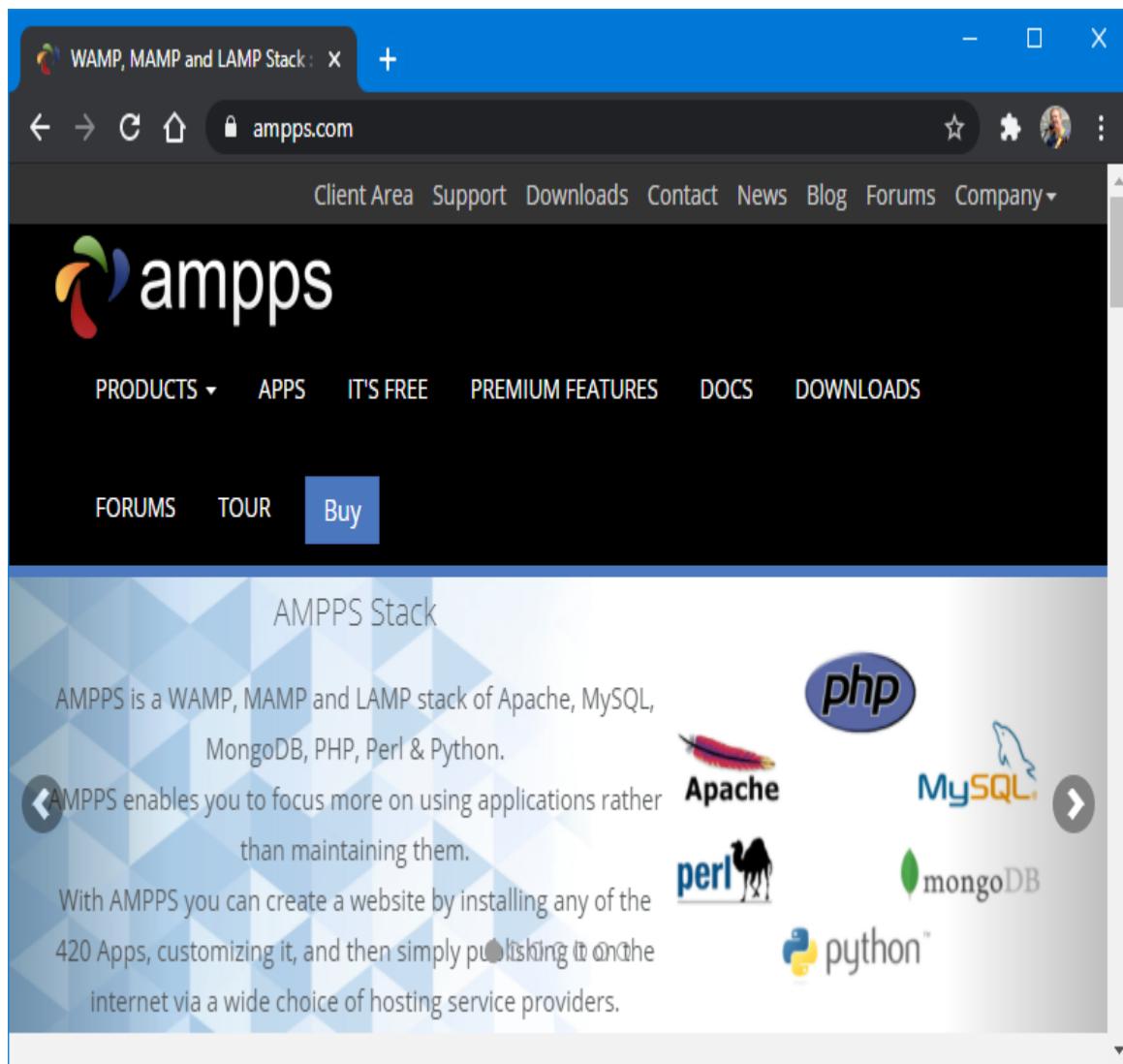


Figure 2-1. The AMPPS website

### NOTE

During the lifetime of this edition, some of the screens and options shown in the following walk-through may change. If so, just use your common sense to proceed in as similar a manner as possible to the sequence of actions described.

Once you've downloaded the installer, run it to bring up the window shown in the Installation Window. Before arriving at that window, though, if you use an antivirus program or have User Account Control activated on

Windows, you may first be shown one or more advisory notices, and will have to click Yes and/or OK to continue with the installation.

Click Next, after which you must accept the agreement. Click Next once again, and then once more to move past the information screen. You will now need to confirm the installation location. This will probably be suggested as something like the following, depending on the letter of your main hard drive, but you can change this if you wish:

C:\Program Files (x86)\Ampps

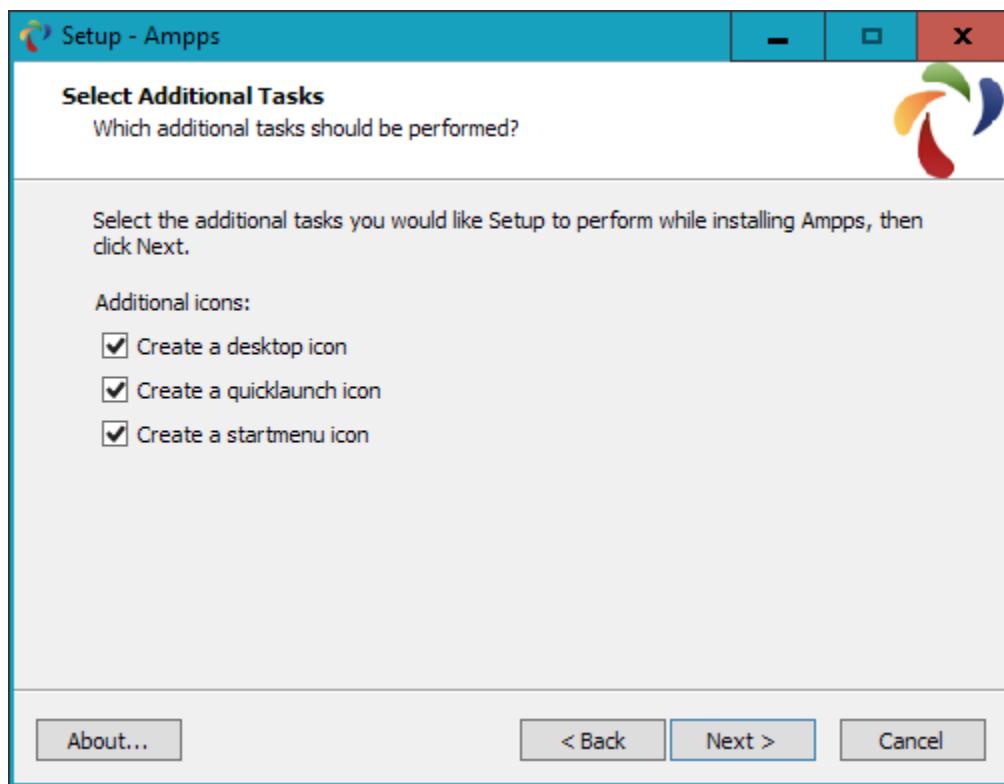


Figure 2-2. The opening window of the installer

Next you must accept the agreements in the following screen and click Next, then after reading the information summary click Next once more and you will be asked which folder you wish to install AMPPS into.

Once you have decided where to install AMPPS, click Next, decide where shortcuts should be saved (the default shown is usually just fine), click Next

again to choose which icons you wish to install, as shown in [Figure 2-3](#). On the screen that follows, click the Install button to start the process.



*Figure 2-3. Choose which icons to install*

Installation will take a few minutes, after which you should see the completion screen in [Figure 2-4](#), and you can click Finish.



Figure 2-4. AMPPS is now installed

The final thing you must do is install Microsoft Visual C++ Redistributable, if you haven't already. Visual Studio is an environment in which you'll be doing development work. A window will pop up to prompt you, as shown in Figure "amps-popup". Click Yes to start the installation or No if you are certain you already have it. Or, you can always proceed anyway and you will be told whether you don't need to reinstall it.

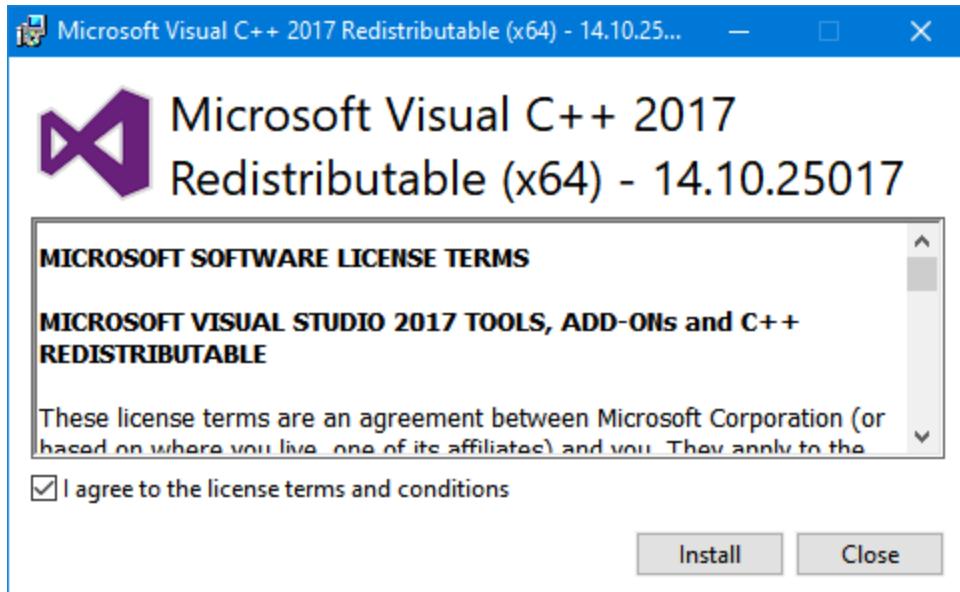


Figure 2-5. Install Visual C++ Redistributable if you don't already have it

If you choose to go ahead and install, you will have to agree to the terms and conditions in the pop-up window that appears, and then click Install. Installation of this should be fairly fast. Click Close to finish.

Once AMPPS is installed, the control window shown in “AMPPS Control” should appear at the bottom right of your desktop. You can also call up this window using the AMPPS application shortcut in the Start menu or on the desktop, if you allowed these icons to be created.



Figure 2-6. The AMPPS control window

Before proceeding, I recommend you acquaint yourself with the [AMPPS documentation](#). Once you have digested this, should you still have an issue, there's a Support link at the bottom of the control window that will take you to the AMPPS website, where you can open up a trouble ticket.

## NOTE

You may notice that the default version of PHP in AMPPS is 7.3. If you wish to try version 5.6 for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window, and then select Change PHP Version, whereupon a new menu will appear from which you can choose a version between 5.6 and 7.3.

## Testing the Installation

The first thing to do at this point is verify that everything is working correctly. To do this, enter either of the following two URLs into the address bar of your browser:

localhost  
127.0.0.1

This will call up an introductory screen, where you will have the opportunity to secure AMPPS by giving it a password (see “the Initial Security”). I recommend you don’t check the box and just click the Submit button to proceed without setting a password.

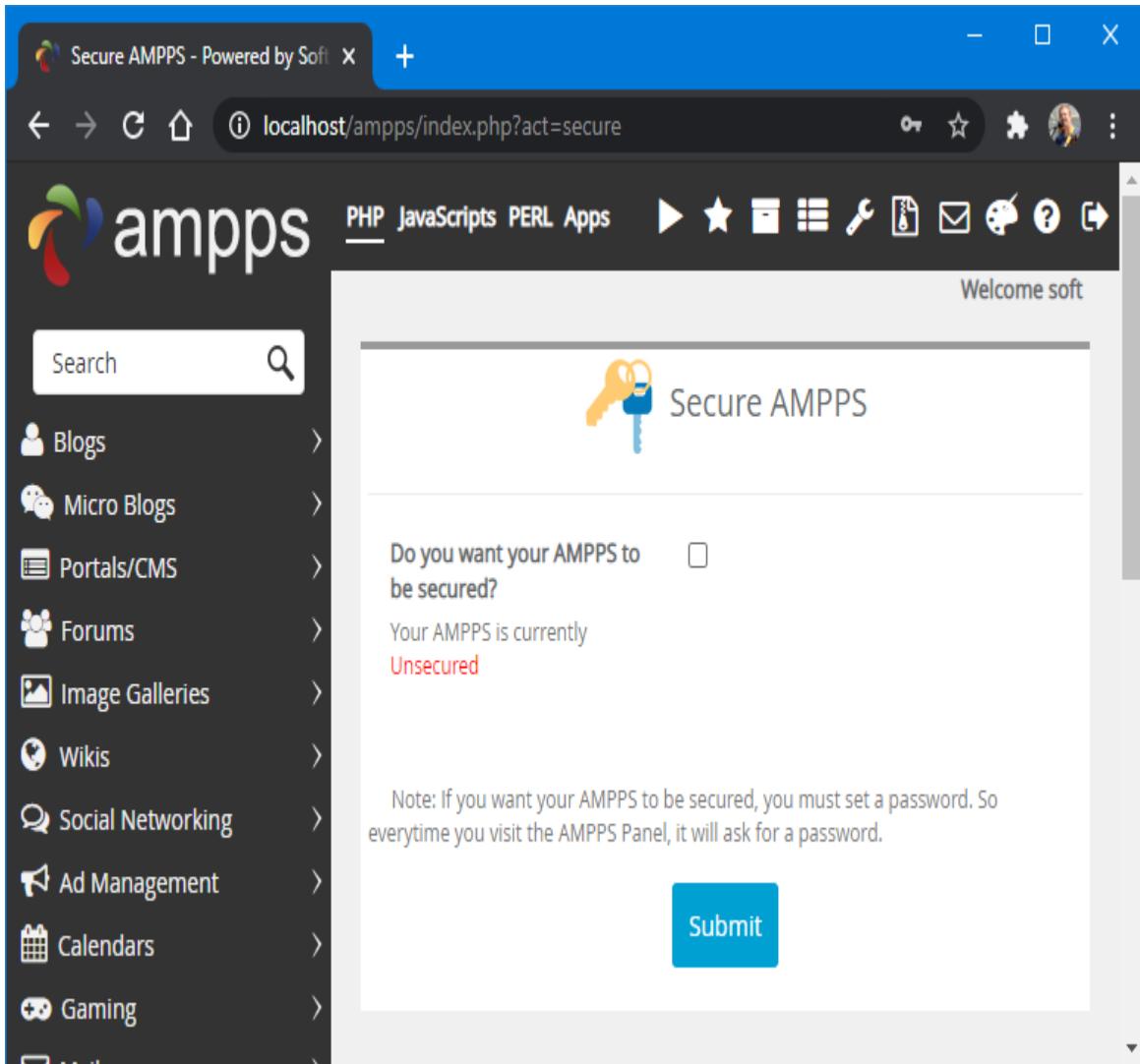


Figure 2-7. The initial security setup screen

Once this has been done you will be taken to the main control page at `localhost/ampps/` (from now on I will assume you are accessing AMPPS through `localhost` rather than `127.0.0.1`). From here you can configure and control all aspects of the AMPPS stack, so make a note of this for future reference, or perhaps set a bookmark in your browser.

Next, type the following to view the document root (described in the following section) of your new Apache web server:

```
localhost
```

This time, rather than seeing the initial screen about setting up security, you should see something similar to “Viewing the Document Root”.

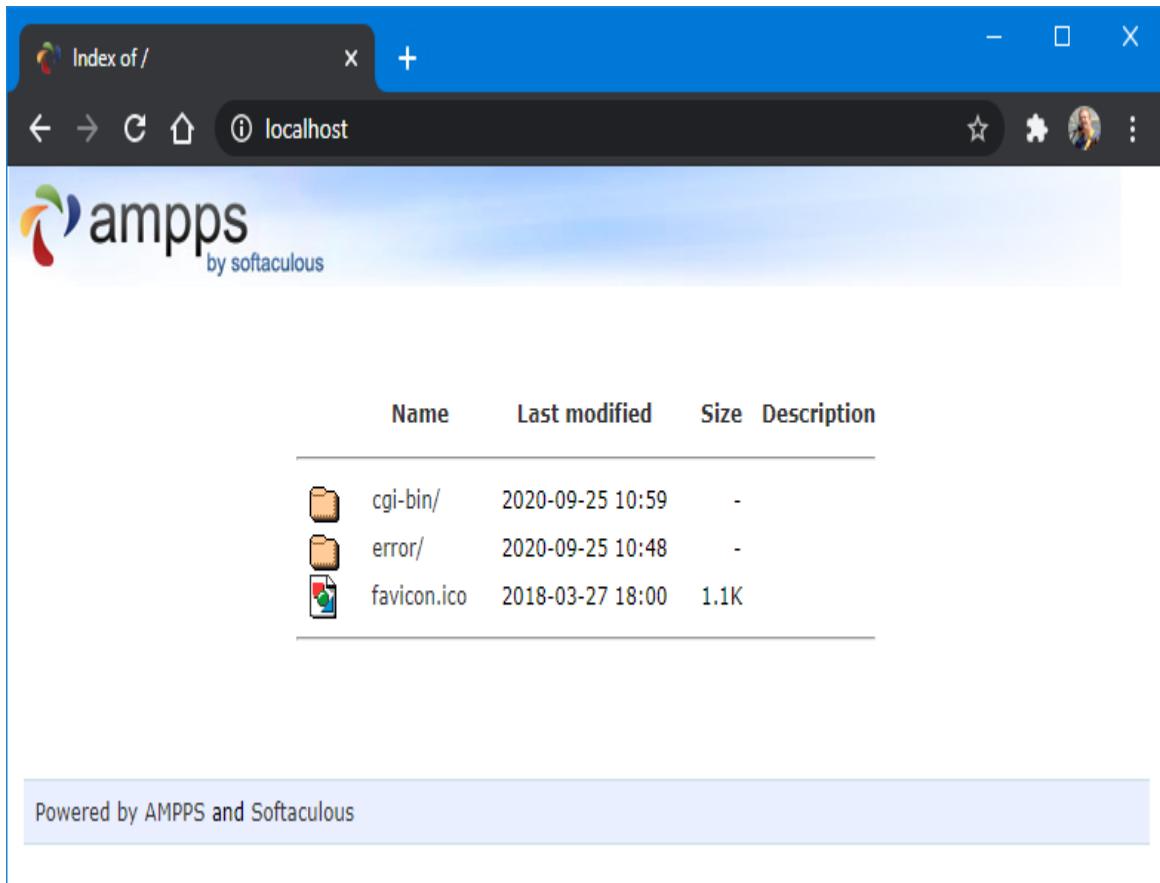


Figure 2-8. Viewing the document root

## Accessing the Document Root (Windows)

The *document root* is the directory that contains the main web documents for a domain. This directory is the one that the server uses when a basic URL without a path is typed into a browser, such as *http://yahoo.com* or, for your local server, *http://localhost*.

By default AMPPS will use the following location as the document root:

C:\Program Files\Ampps\www

To ensure that you have everything correctly configured, you should now create the obligatory “Hello World” file. So, create a small HTML file along the following lines using Windows Notepad or any other program or text editor, but not a rich word processor such as Microsoft Word (unless you save as plain text):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>A quick test</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

Once you have typed this, save the file into the document root directory, using the filename *test.html*. If you are using Notepad, make sure that the value in the “Save as type” box is changed from “Text Documents (\*.txt)” to “All Files (\*.\*)”.

You can now call this page up in your browser by entering the following URL in its address bar (see **Figure 2-9**):

localhost/test.html

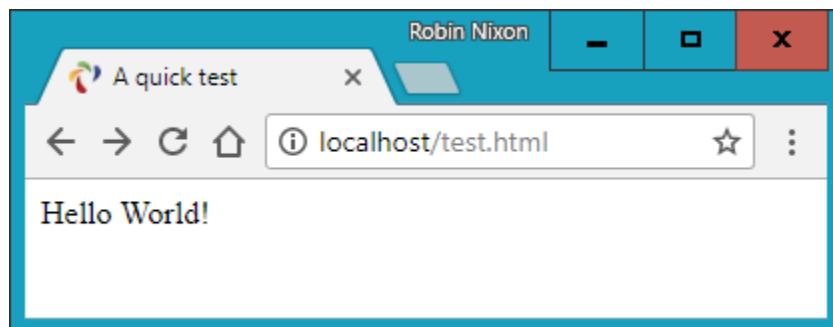


Figure 2-9. Your first web page

## NOTE

Remember that serving a web page from the document root (or a subfolder) is different from loading one into a web browser from your computer's filesystem. The former will ensure access to PHP, MySQL, and all the features of a web server, while the latter will simply load the file into the browser, which will do its best to display it but will be unable to process any PHP or other server instructions. So, you should generally run examples using the *localhost* preface from your browser's address bar, unless you are certain that the file doesn't rely on web server functionality.

## Alternative WAMPs

When software is updated, it sometimes works differently from how you expect, and bugs can even be introduced. So, if you encounter difficulties that you cannot resolve in AMPPS, you may prefer to choose one of the other solutions available on the web.

You will still be able to make use of all the examples in this book, but you'll have to follow the instructions supplied with each WAMP, which may not be as easy to follow as the preceding guide.

Here's a selection of some of the best, in my opinion:

- [EasyPHP](#)
- [XAMPP](#)
- [WAMPServer](#)
- [Glossword WAMP](#)

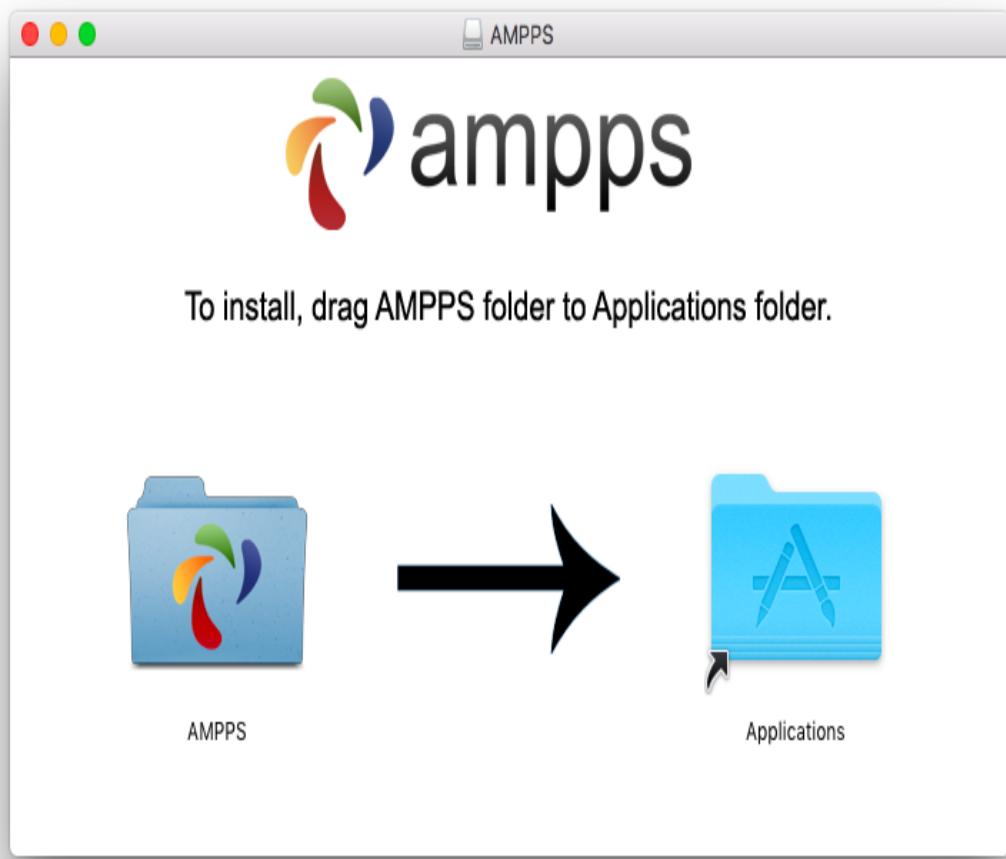
## UPDATES TO AMPPS

Over the life of this edition of the book, it is very likely that the developers of AMPPS will make improvements to the software, and therefore the installation screens and method of use may evolve over time, as may versions of Apache, PHP, or MySQL. So, please don't assume something is wrong if the screens and operation look different. The AMPPS developers take every care to ensure it is easy to use, so just follow any prompts given, and refer to the documentation on the [website](#).

## Installing AMPPS on macOS

AMPPS is also available on macOS, and you can download it from the [website](#), as shown previously in #ampps-website (as I write, the current version is 3.8 and its size is around 270 MB).

If your browser doesn't open it automatically once it has downloaded, double-click the *.dmg* file, and then drag the *AMPPS* folder over to your *Applications* folder (see [Figure 2-10](#)).



*Figure 2-10. Drag the AMPPS folder to Applications*

Now open your *Applications* folder in the usual manner, and double-click the AMPPS program. If your security settings prevent the file being opened,

hold down the Control key and click the icon once. A new window will pop up asking if you are sure you wish to open it. Click Open to do so. When the app starts you may have to enter your macOS password to proceed.

Once AMPPS is up and running, a control window similar to the one shown in “AMPPS Control” will appear at the bottom left of your desktop.

#### NOTE

You may notice that the default version of PHP in AMPPS is 7.3. If you wish to try out version 5.6 for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window, and then select Change PHP Version, whereupon a new menu will appear in which you can choose a version between 5.6 and 7.3.

## Accessing the Document Root (macOS)

By default, AMPPS will use the following location as the document root:

/Applications/Ampps/www

To ensure that you have everything correctly configured, you should now create the obligatory “Hello World” file. So, create a small HTML file along the following lines using theTextEdit program or any other program or text editor, but not a rich word processor such as Microsoft Word (unless you save as plain text):

```
<html>
  <head>
    <title>A quick test</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

Once you have typed this, save the file into the document root directory using the filename *test.html*.

You can now call this page up in your browser by entering the following URL in its address bar (to see a similar result to [Figure 2-9](#)):

`localhost/test.html`

#### NOTE

Remember that serving a web page from the document root (or a subfolder) is different from loading one into a web browser from your computer's filesystem. The former will ensure access to PHP, MySQL, and all the features of a web server, while the latter will simply load the file into the browser, which will do its best to display it but will be unable to process any PHP or other server instructions. So, you should generally run examples using the *localhost* preface from your browser's address bar, unless you are certain that the file doesn't rely on web server functionality.

## Installing a LAMP on Linux

This book is aimed mostly at PC and Mac users, but its contents will work equally well on a Linux computer. However, there are dozens of popular flavors of Linux, and each of them may require installing a LAMP in a slightly different way, so I can't cover them all in this book.

That said, many Linux versions come preinstalled with a web server and MySQL, and the chances are that you may already be all set to go. To find out, try entering the following into a browser and see whether you get a default document root web page:

`localhost`

If this works, you probably have the Apache server installed and may well have MySQL up and running too; check with your system administrator to be sure.

If you don't yet have a web server installed, however, there's a version of AMPPS available that you can download from the [website](#).

Installation is similar to the sequence shown in the preceding section. If you need further assistance on using the software, please refer to the [documentation](#).

## Working Remotely

If you have access to a web server already configured with PHP and MySQL, you can always use that for your web development. But unless you have a high-speed connection, it is not always your best option. Developing locally allows you to test modifications with little or no upload delay.

Accessing MySQL remotely may not be easy either. You should use the secure SSH protocol to log into your server to manually create databases and set permissions from the command line. Your web hosting company will advise you on how best to do this and provide you with any password it has set for your MySQL access (as well as, of course, for getting into the server in the first place). Unless you have no choice, I recommend you do not use the insecure Telnet protocol to remotely log into any server.

## Logging In

I recommend that, at minimum, Windows users should install a program such as [PuTTY](#), for Telnet and SSH access (remember that SSH is much more secure than Telnet).

On a Mac, you already have SSH available. Just select the *Applications* folder, followed by *Utilities*, and then launch Terminal. In the Terminal window, log in to a server using SSH as follows:

```
ssh mylogin@server.com
```

where *server.com* is the name of the server you wish to log into and *mylogin* is the username you will log in under. You will then be prompted for the correct password for that username and, if you enter it correctly, you will be logged in.

## Using FTP

To transfer files to and from your web server, you will need an FTP program. If you go searching the web for a good one, you'll find so many that it could take you quite a while to come across one with all the right features for you.

My preferred FTP program is the open source [FileZilla](#), for Windows, Linux, and macOS 10.5 or newer (see [Figure 2-11](#)). Full instructions on how to use FileZilla are available on the [wiki](#).

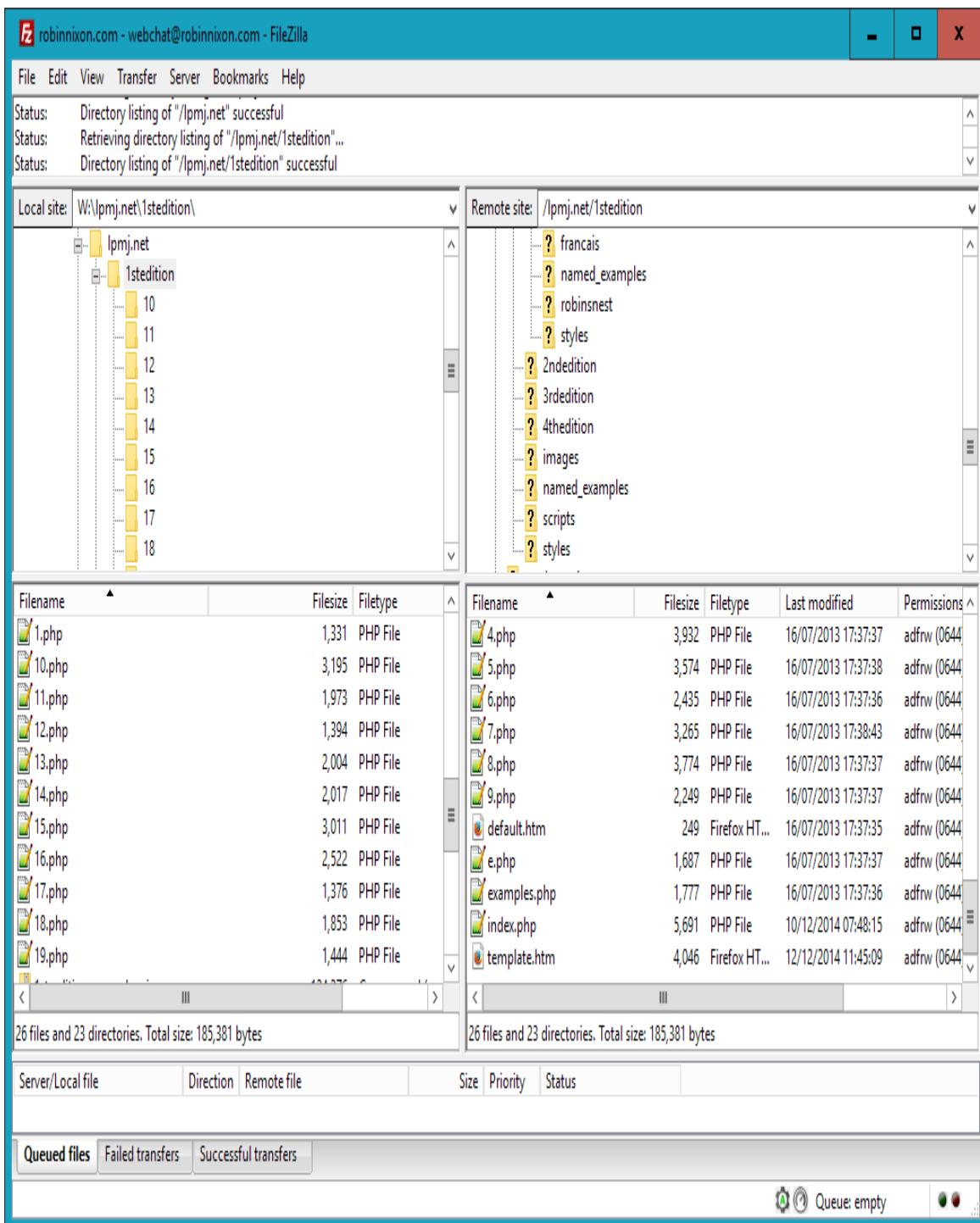


Figure 2-11. FileZilla is a full-featured FTP program

Of course, if you already have an FTP program, all the better—stick with what you know.

## Using a Program Editor

Although a plain-text editor works for editing HTML, PHP, and JavaScript, there have been some tremendous improvements in dedicated program editors, which now incorporate very handy features such as colored syntax highlighting. Today's program editors are smart and can show you where you have syntax errors before you even run a program. Once you've used a modern editor, you'll wonder how you ever managed without one.

There are a number of good programs available, but I have settled on Editra (see [Figure 2-12](#)), because it's free and available on macOS, Windows, and Linux/Unix, and it suits the way I program. At the time of writing, Editra development appears to have been discontinued, but you can download the last known fully working release for free at <https://editra.en.softonic.com>. Everyone has different programming styles and preferences, though, so if you don't get on with it, there are plenty more program editors available to choose from—or you may wish to go directly for an integrated development environment (IDE), as described in the following section.

The screenshot shows the Editra v0.6.99 interface. The title bar reads "\*examples.php - file:///C:/Users/Robin/Desktop/examples.php - Editra v0.6.99". The menu bar includes File, Edit, View, Format, Settings, Tools, and Help. Below the menu is a toolbar with icons for new, open, save, cut, copy, paste, find, and search. The main window displays a code editor with the file \*examples.php. The code is a PHP script with line numbers from 1753 to 1773. It uses color coding for syntax: purple for strings and comments, red for keywords like if and return, and black for variables and operators. The code performs file operations, including reading files, calculating MD5 checksums, and comparing arrays.

```
1753 $contents = @file_get_contents($page);
1754 if (!$contents) return FALSE;
1755 
1756 $checksum = md5($contents);
1757 
1758 if (file_exists($datafile))
1759 {
1760     $rawfile = file_get_contents($datafile);
1761     $data = explode("\n", rtrim($rawfile));
1762     $left = array_map("PU_F1", $data);
1763     $right = array_map("PU_F2", $data);
1764     $exists = -1;
1765 
1766     for ($j = 0 ; $j < count($left) ; ++$j)
1767     {
1768         if ($left[$j] == $page)
1769         {
1770             $exists = $j;
1771             if ($right[$j] == $checksum) return 0;
1772         }
1773     }
1774 }
```

Figure 2-12. Program editors (like Editra, pictured here) are superior to plain-text editors

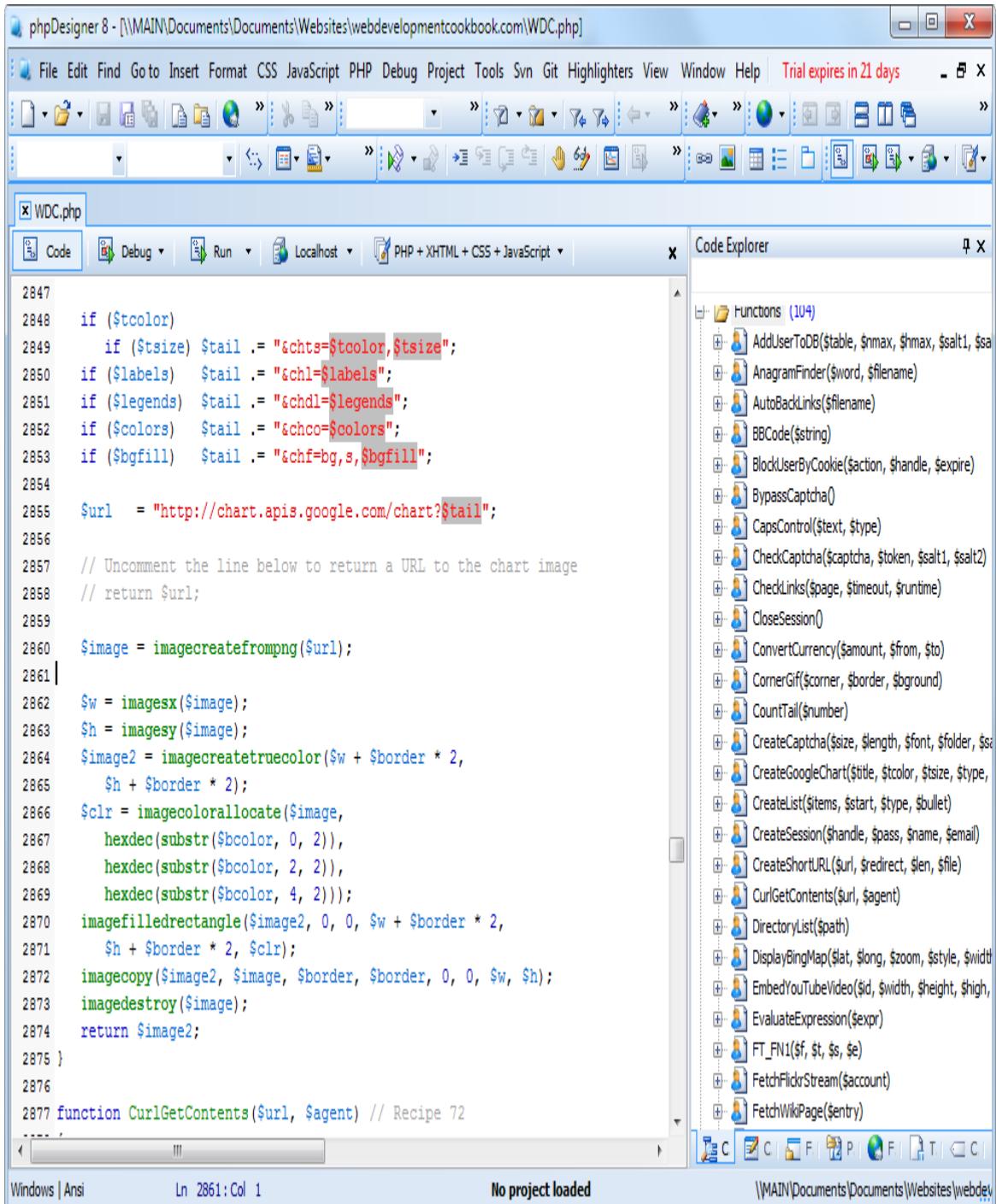
As you can see from Figure 2-12, Editra highlights the syntax appropriately, using colors to help clarify what's going on. What's more, you can place the cursor next to brackets or braces, and Editra will highlight the matching ones so that you can check whether you have too many or too few. In fact, Editra does a lot more in addition, which you will discover and enjoy as you use it.

Again, if you have a different preferred program editor, use that; it's always a good idea to use programs you're already familiar with.

## Using an IDE

As good as dedicated program editors can be for your programming productivity, their utility pales into insignificance when compared to integrated development environments, which offer many additional features such as in-editor debugging and program testing, as well as function descriptions and much more.

**Figure 2-13** shows the popular phpDesigner IDE with a PHP program loaded into the main frame, and the righthand Code Explorer listing the various classes, functions, and variables that it uses.



*Figure 2-13. When you're using an IDE such as phpDesigner, PHP development becomes much quicker and easier*

When developing with an IDE, you can set breakpoints and then run all (or portions) of your code, which will stop at the breakpoints and provide you with information about the program's current state.

As an aid to learning programming, the examples in this book can be entered into an IDE and run there and then, without the need to call up your web browser. There are several IDEs available for different platforms, **Table 2-1** lists some of the most popular free PHP IDEs, along with their download URLs.

*Table 2-1. A selection of Free PHP IDEs*

IDE	Download URL	Windows	macOS	Linux
Eclipse PDT	<a href="http://eclipse.org/pdt/downloads/">http://eclipse.org/pdt/downloads/</a>	✓	✓	✓
Komodo IDE	<a href="http://activestate.com/Products/komodo_ide">http://activestate.com/Products/komodo_ide</a>	✓	✓	✓
NetBeans	<a href="http://www.netbeans.org">http://www.netbeans.org</a>	✓	✓	✓
PHPeclipse	<a href="https://sourceforge.net/projects/phpeclipse/">https://sourceforge.net/projects/phpeclipse/</a>	✓	✓	✓

Choosing an IDE can be a very personal thing, so if you intend to use one, I advise you to download a couple or more to try them out first; they all either have trial versions or are free to use, so it won't cost you anything.

You should take the time to install a program editor or IDE you are comfortable with now; you'll then be ready to try out the examples in the coming chapters.

Armed with these tools, you are now ready to move on to **Chapter 3**, where we'll start exploring PHP in further depth and find out how to get HTML and PHP to work together, as well as how the PHP language itself is structured. But before moving on, I suggest you test your new knowledge with the following questions.

## Questions

1. What is the difference between a WAMP, a MAMP, and a LAMP?
2. What do the IP address 127.0.0.1 and the URL <http://localhost> have in common?
3. What is the purpose of an FTP program?

4. Name the main disadvantage of working on a remote web server.
5. Why is it better to use a program editor instead of a plain-text editor?

See “[Chapter 2 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 3. Introduction to PHP

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In [Chapter 1](#), I explained that PHP is the language that you use to make the server generate dynamic output—output that is potentially different each time a browser requests a page. In this chapter, you’ll start learning this simple but powerful language; it will be the topic of the following chapters up through [Chapter 7](#).

I encourage you to develop your PHP code using one of the IDEs listed in [Chapter 2](#). It will help you catch typos and speed up learning tremendously in comparison to a less feature-rich editor.

Many of these development environments will let you run the PHP code and see the output discussed in this chapter. I’ll also show you how to embed the PHP in an HTML file so that you can see what the output looks like in a web page (the way your users will ultimately see it). But that step, as thrilling as it may be at first, isn’t really important at this stage.

In production, your web pages will be a combination of PHP, HTML, JavaScript, and some MySQL statements laid out using CSS. Furthermore, each page can lead to other pages to provide users with ways to click

through links and fill out forms. We can avoid all that complexity while learning each language, though. Focus for now on just writing PHP code and making sure that you get the output you expect—or at least that you understand the output you actually get!

## Incorporating PHP Within HTML

By default, PHP documents end with the extension *.php*. When a web server encounters this extension in a requested file, it automatically passes it to the PHP processor. Of course, web servers are highly configurable, and some web developers choose to force files ending with *.htm* or *.html* to also get parsed by the PHP processor, usually because they want to hide their use of PHP.

### DOWNLOADING THIS BOOK'S EXAMPLE FILES

Remember that to save you typing them in, and to ensure that they have been pre-tested and are working, all the examples from this book can be downloaded from the following URL:

[github.com/RobinNixon/lpmj6](https://github.com/RobinNixon/lpmj6)

Your PHP program is responsible for passing back a clean file suitable for display in a web browser. At its very simplest, a PHP document will output only HTML. To prove this, you can take any normal HTML document and save it as a PHP document (for example, saving *index.html* as *index.php*), and it will display identically to the original.

To trigger the PHP commands, you need to learn a new tag. Here is the first part:

```
<?php
```

The first thing you may notice is that the tag has not been closed. This is because entire sections of PHP can be placed inside this tag, and they finish

only when the closing part is encountered, which looks like this:

```
?>
```

A small PHP “Hello World” program might look like [Example 3-1](#).

### Example 3-1. Invoking PHP

---

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

Use of this tag can be quite flexible. Some programmers open the tag at the start of a document and close it right at the end, outputting any HTML directly from PHP commands. Others, however, choose to insert only the smallest possible fragments of PHP within these tags wherever dynamic scripting is required, leaving the rest of the document in standard HTML.

The latter type of programmer generally argues that their style of coding results in faster code, while the former says that the speed increase is so minimal that it doesn’t justify the additional complexity of dropping in and out of PHP many times in a single document.

As you learn more, you will surely discover your preferred style of PHP development, but for the sake of making the examples in this book easier to follow, I have adopted the approach of keeping the number of transfers between PHP and HTML to a minimum—generally only once or twice in a document.

By the way, there is a slight variation to the PHP syntax. If you browse the internet for PHP examples, you may also encounter code where the opening and closing syntax looks like this:

```
<?  
    echo "Hello world";  
?>
```

Although it's not as obvious that the PHP parser is being called, this is a valid, alternative syntax that also usually works. But I discourage its use, as it is incompatible with XML and is now deprecated (meaning that it is no longer recommended and support could be removed in future versions).

### NOTE

If you have only PHP code in a file, you may omit the closing `?>`. This can be a good practice, as it will ensure that you have no excess whitespace leaking from your PHP files (especially important when you're writing object-oriented code).

## This Book's Examples

To save you the time it would take to type them all in, all the examples from this book have been archived at GitHub. You can download the archive to your computer by visiting the following URL:

[github.com/RobinNixon/lpmj6](https://github.com/RobinNixon/lpmj6)

In addition to listing all the examples by chapter and example number (such as `example3-1.php`), the provided archive also contains an additional directory called `named_examples` in which you'll find all the examples I suggest you save using a specific filename (such as the upcoming [Example 3-4](#), which should be saved as `test1.php`).

## The Structure of PHP

We're going to cover quite a lot of ground in this section. It's not too difficult, but I recommend that you work your way through it carefully, as it sets the foundation for everything else in this book. As always, there are some useful questions at the end of the chapter that you can use to test how much you've learned.

## Using Comments

There are two ways in which you can add comments to your PHP code. The first turns a single line into a comment by preceding it with a pair of forward slashes:

```
// This is a comment
```

This version of the comment feature is a great way to temporarily remove a line of code from a program that is giving you errors. For example, you could use such a comment to hide a debugging line of code until you need it, like this:

```
// echo "X equals $x";
```

You can also use this type of comment directly after a line of code to describe its action, like this:

```
$x += 10; // Increment $x by 10
```

When you need multiple-line comments, there's a second type of comment, which looks like [Example 3-2](#).

### Example 3-2. A multiline comment

---

```
<?php
/* This is a section
   of multiline comments
   which will not be
   interpreted */
?>
```

You can use the /\* and \*/ pairs of characters to open and close comments almost anywhere you like inside your code. Most, if not all, programmers use this construct to temporarily comment out entire sections of code that

do not work or that, for one reason or another, they do not wish to be interpreted.

### WARNING

A common error is to use `/*` and `*/` to comment out a large section of code that already contains a commented-out section that uses those characters. You can't nest comments this way; the PHP interpreter won't know where a comment ends and will display an error message. However, if you use a program editor or IDE with syntax highlighting, this type of error is easier to spot.

## Basic Syntax

PHP is quite a simple language with roots in C and Perl, yet it looks more like Java. It is also very flexible, but there are a few rules that you need to learn about its syntax and structure.

### Semicolons

You may have noticed in the previous examples that the PHP commands ended with a semicolon, like this:

```
$x += 10;
```

Probably the most common cause of errors you will encounter with PHP is forgetting this semicolon. This causes PHP to treat multiple statements like one statement, which it is unable to understand, prompting it to produce a `Parse error` message.

### The \$ symbol

The `$` symbol has come to be used in many different ways by different programming languages. For example, in the BASIC language, it was used to terminate variable names to denote them as strings.

In PHP, however, you must place a `$` in front of *all* variables. This is required to make the PHP parser faster, as it instantly knows whenever it

comes across a variable. Whether your variables are numbers, strings, or arrays, they should all look something like those in [Example 3-3](#).

### Example 3-3. Three different types of variable assignment

---

```
<?php  
    $mycounter = 1;  
    $mystring = "Hello";  
    $myarray = array("One", "Two", "Three");  
?>
```

And really that's pretty much all the syntax that you have to remember. Unlike languages such as Python, which are very strict about how you indent and lay out your code, PHP leaves you completely free to use (or not use) all the indenting and spacing you like. In fact, sensible use of whitespace is generally encouraged (along with comprehensive commenting) to help you understand your code when you come back to it. It also helps other programmers when they have to maintain your code.

## Variables

There's a simple metaphor that will help you understand what PHP variables are all about. Just think of them as little (or big) matchboxes! That's right—matchboxes that you've painted over and written names on.

### String variables

Imagine you have a matchbox on which you have written the word *username*. You then write *Fred Smith* on a piece of paper and place it into the box (see [Figure 3-1](#)). Well, that's the same process as assigning a string value to a variable, like this:

```
$username = "Fred Smith";
```

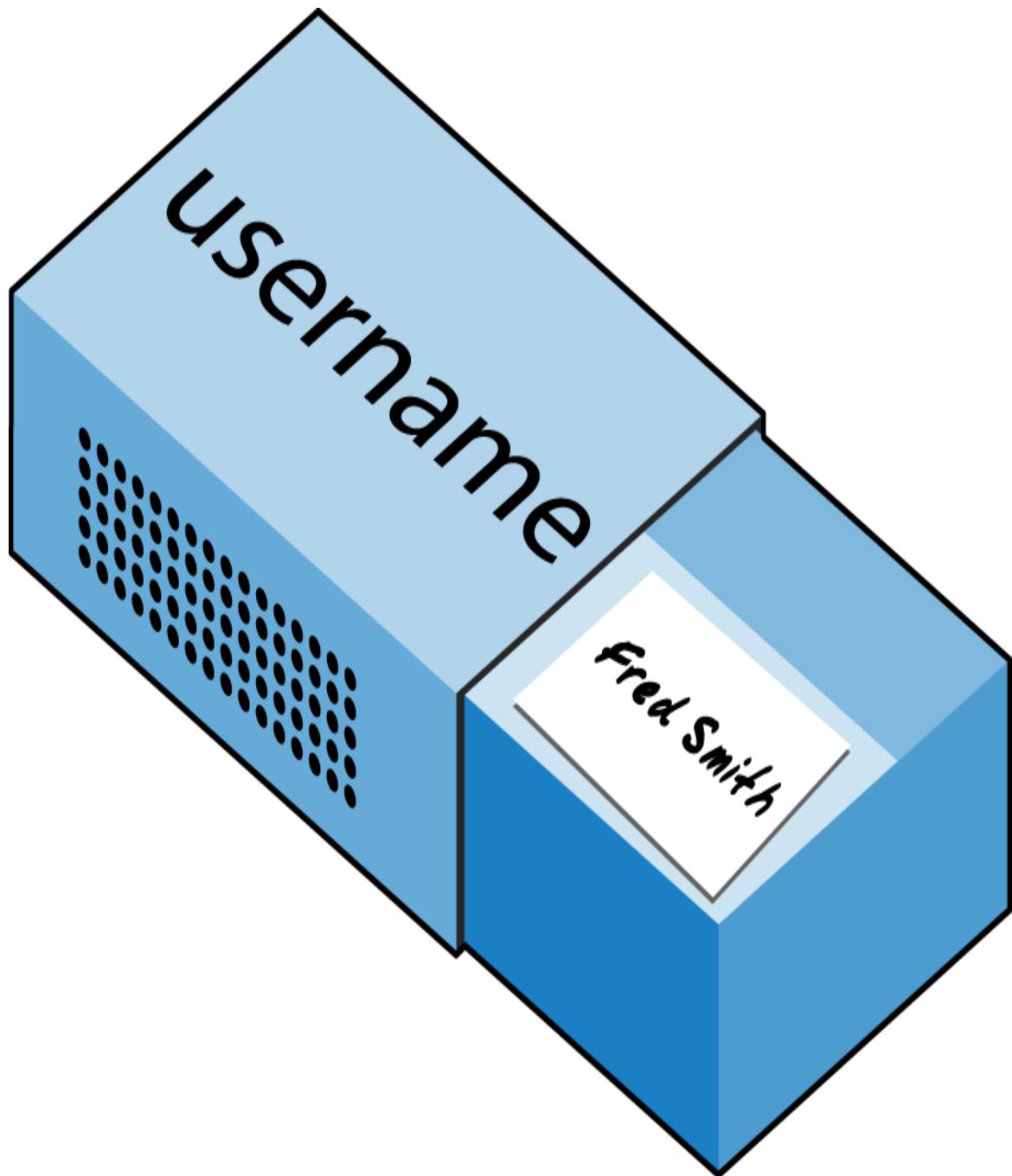


Figure 3-1. You can think of variables as matchboxes containing items

The quotation marks indicate that “Fred Smith” is a *string* of characters. You must enclose each string in either quotation marks or apostrophes (single quotes), although there is a subtle difference between the two types of quote, which is explained later. When you want to see what’s in the box,

you open it, take the piece of paper out, and read it. In PHP, doing so looks like this (which displays the contents of the variable to screen):

```
echo $username;
```

Or you can assign it to another variable (photocopy the paper and place the copy in another matchbox), like this:

```
$current_user = $username;
```

If you are keen to start trying out PHP for yourself, you could enter the examples in this chapter into an IDE (as recommended at the end of [Chapter 2](#)) to see instant results, or you could enter the code in [Example 3-4](#) into a program editor and save it to your server's document root directory (also discussed in [Chapter 2](#)) as *test1.php*.

#### *Example 3-4. Your first PHP program*

---

```
<?php // test1.php
$username = "Fred Smith";
echo $username;
echo "<br>";
$current_user = $username;
echo $current_user;
?>
```

Now you can call it up by entering the following into your browser's address bar:

<http://localhost/test1.php>

## NOTE

In the unlikely event that during the installation of your web server (as detailed in [Chapter 2](#)) you changed the port assigned to the server to anything other than 80, then you must place that port number within the URL in this and all other examples in this book. So, for example, if you changed the port to 8080, the preceding URL would become this:

```
http://localhost:8080/test1.php
```

I won't mention this again, so just remember to use the port number (if required) when trying examples or writing your own code.

The result of running this code should be two occurrences of the name *Fred Smith*, the first of which is the result of the `echo $username` command and the second of the `echo $current_user` command.

## Numeric variables

Variables don't have to contain just strings—they can contain numbers too. If we return to the matchbox analogy, to store the number 17 in the variable `$count`, the equivalent would be placing, say, 17 beads in a matchbox on which you have written the word *count*:

```
$count = 17;
```

You could also use a floating-point number (containing a decimal point). The syntax is the same:

```
$count = 17.5;
```

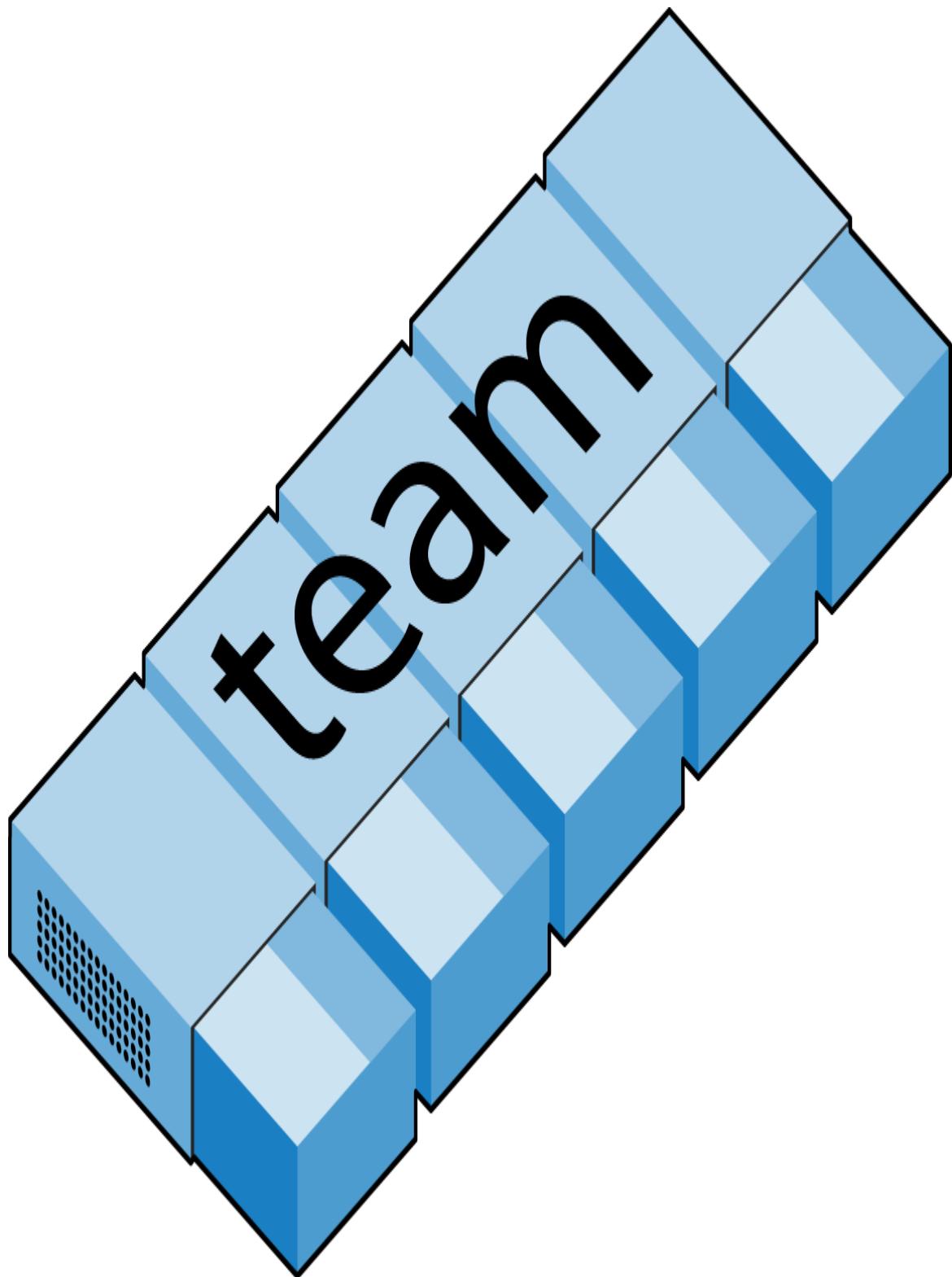
To see the contents of the matchbox, you would simply open it and count the beads. In PHP, you would assign the value of `$count` to another variable or perhaps just echo it to the web browser.

## Arrays

So what are arrays? Well, you can think of them as several matchboxes glued together. For example, let's say we want to store the player names for a five-person soccer team in an array called `$team`. To do this, we could glue five matchboxes side by side and write down the names of all the players on separate pieces of paper, placing one in each matchbox.

Across the top of the whole matchbox assembly we would write the word *team* (see [Figure 3-2](#)). The equivalent of this in PHP would be the following:

```
$team = array('Bill', 'Mary', 'Mike', 'Chris', 'Anne');
```



*Figure 3-2. An array is like several matchboxes glued together*

This syntax is more complicated than the other examples you've seen so far. The array-building code consists of the following construct:

```
array();
```

with five strings inside. Each string is enclosed in apostrophes, and strings must be separated with commas.

If we then wanted to know who player 4 is, we could use this command:

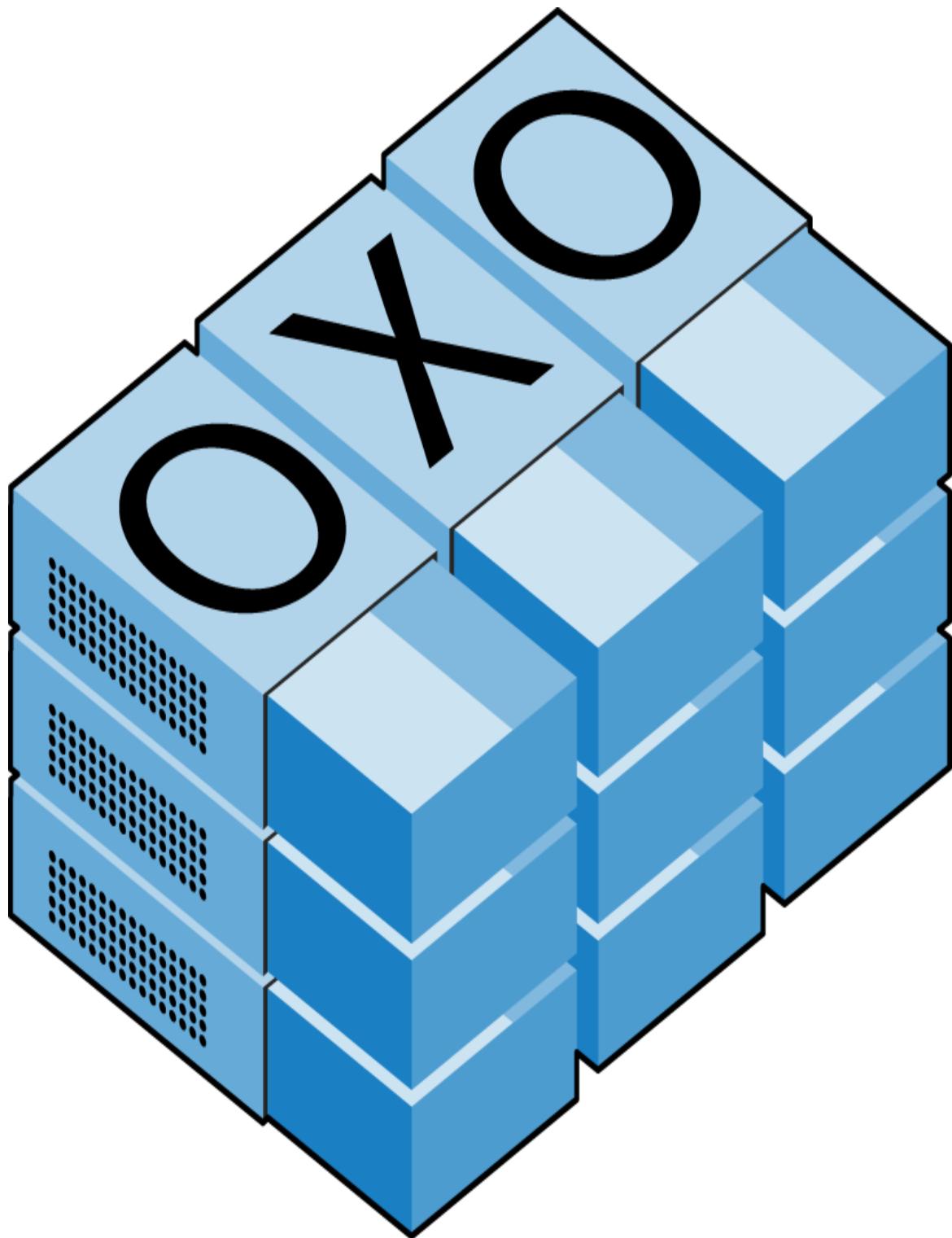
```
echo $team[3]; // Displays the name Chris
```

The reason the previous statement has the number 3, not 4, is because the first element of a PHP array is actually the zeroth element, so the player numbers will therefore be 0 through 4.

## Two-dimensional arrays

There's a lot more you can do with arrays. For example, instead of being single-dimensional lines of matchboxes, they can be two-dimensional matrixes or can even have more dimensions.

As an example of a two-dimensional array, let's say we want to keep track of a game of tic-tac-toe, which requires a data structure of nine cells arranged in a  $3 \times 3$  square. To represent this with matchboxes, imagine nine of them glued to each other in a matrix of three rows by three columns (see [Figure 3-3](#)).



*Figure 3-3. A multidimensional array simulated with matchboxes*

You can now place a piece of paper with either an *x* or an *o* on it in the correct matchbox for each move played. To do this in PHP code, you have

to set up an array containing three more arrays, as in [Example 3-5](#), in which the array is set up with a game already in progress.

*Example 3-5. Defining a two-dimensional array*

---

```
<?php  
$oxo = array(array('x', ' ', 'o'),  
             array('o', 'o', 'x'),  
             array('x', 'o', ' '));  
?>
```

Once again, we've moved up a step in complexity, but it's easy to understand if you grasp the basic array syntax. There are three `array()` constructs nested inside the outer `array()` construct. We've filled each row with an array consisting of just one character: an `x`, an `o`, or a blank space. (We use a blank space so that all the cells will be the same width when they are displayed.)

To then return the third element in the second row of this array, you would use the following PHP command, which will display an `x`:

```
echo $oxo[1][2];
```

**NOTE**

Remember that array indexes (pointers at elements within an array) start from zero, not one, so the `[1]` in the previous command refers to the second of the three arrays, and the `[2]` references the third position within that array. This command will return the contents of the matchbox three along and two down.

As mentioned, we can support arrays with even more dimensions by simply creating more arrays within arrays. However, we will not be covering arrays of more than two dimensions in this book.

And don't worry if you're still having difficulty coming to grips with using arrays, as the subject is explained in detail in [Chapter 6](#).

## Variable-naming rules

When creating PHP variables, you must follow these four rules:

- Variable names, after the dollar sign, must start with a letter of the alphabet or the `_` (underscore) character.
- Variable names can contain only the characters `a-z`, `A-Z`, `0-9`, and `_` (underscore).
- Variable names may not contain spaces. If a variable name must comprise more than one word, a good idea is to separate the words with the `_` (underscore) character (e.g., `$user_name`).
- Variable names are case-sensitive. The variable `$High_Score` is not the same as the variable `$high_score`.

### NOTE

To allow extended ASCII characters that include accents, PHP also supports the bytes from 127 through 255 in variable names. But unless your code will be maintained only by programmers who are used to those characters, it's probably best to avoid them, because programmers using English keyboards will have difficulty accessing them.

## Operators

*Operators* let you specify mathematical operations to perform, such as addition, subtraction, multiplication, and division. But several other types of operators exist too, such as the string, comparison, and logical operators. Math in PHP looks a lot like plain arithmetic—for instance, the following statement outputs 8:

```
echo 6 + 2;
```

Before moving on to learn what PHP can do for you, take a moment to learn about the various operators it provides.

## Arithmetic operators

Arithmetic operators do what you would expect—they are used to perform mathematics. You can use them for the main four operations (add, subtract, multiply, and divide) as well as to find a modulus (the remainder after a division) and to increment or decrement a value (see [Table 3-1](#)).

*Table 3-1. Arithmetic operators*

Operator	Description	Example
+	Addition	<code>\$j + 1</code>
-	Subtraction	<code>\$j - 6</code>
*	Multiplication	<code>\$j * 11</code>
/	Division	<code>\$j / 4</code>
%	Modulus (the remainder after a division is performed)	<code>\$j % 9</code>
++	Increment	<code>++\$j</code>
--	Decrement	<code>--\$j</code>
**	Exponentiation (or power)	<code>\$j**2</code>

## Assignment operators

These operators assign values to variables. They start with the very simple `=` and move on to `+=`, `-=`, and so on (see [Table 3-2](#)). The operator `+=` adds the value on the right side to the variable on the left, instead of totally replacing the value on the left. Thus, if `$count` starts with the value 5, the statement:

```
$count += 1;
```

sets `$count` to 6, just like the more familiar assignment statement:

```
$count = $count + 1;
```

*Table 3-2. Assignment operators*

Operator	Example	Equivalent to
=	\$j = 15	\$j = 15
+=	\$j += 5	\$j = \$j + 5
-=	\$j -= 3	\$j = \$j - 3
*=	\$j *= 8	\$j = \$j * 8
/=	\$j /= 16	\$j = \$j / 16
.=	\$j .= \$k	\$j = \$j . \$k
%=	\$j %= 4	\$j = \$j % 4

## Comparison operators

Comparison operators are generally used inside a construct such as an **if** statement in which you need to compare two items. For example, you may wish to know whether a variable you have been incrementing has reached a specific value, or whether another variable is less than a set value, and so on (see [Table 3-3](#)).

*Table 3-3. Comparison operators*

Operator	Description	Example
==	Is <i>equal</i> to	\$j == 4
!=	Is <i>not equal</i> to	\$j != 21
>	Is <i>greater than</i>	\$j > 3
<	Is <i>less than</i>	\$j < 100
>=	Is <i>greater than or equal</i> to	\$j >= 15
<=	Is <i>less than or equal</i> to	\$j <= 8
<>	Is <i>not equal</i> to	\$j <> 23
==>	Is <i>identical</i> to	\$j ==> "987"
!==>	Is <i>not identical</i> to	\$j !=> "1.2e3"

Note the difference between `=` and `==`. The first is an assignment operator, and the second is a comparison operator. Even advanced programmers can sometimes mix up the two when coding hurriedly, so be careful.

## Logical operators

If you haven't used them before, logical operators may at first seem a little daunting. But just think of them the way you would use logic in English. For example, you might say to yourself, "If the time is later than 12 p.m. and earlier than 2 p.m., have lunch." In PHP, the code for this might look something like the following (using military time):

```
if ($hour > 12 && $hour < 14) dolunch();
```

Here we have moved the set of instructions for actually going to lunch into a function that we will have to create later called `dolunch`.

As the previous example shows, you generally use a logical operator to combine the results of two of the comparison operators shown in the previous section. A logical operator can also be input to another logical operator: "If the time is later than 12 p.m. and earlier than 2 p.m., or if the smell of a roast is permeating the hallway and there are plates on the table." As a rule, if something has a TRUE or FALSE value, it can be input to a logical operator. A logical operator takes two true or false inputs and produces a true or false result.

Table 3-4 shows the logical operators.

Table 3-4. Logical operators

Operator	Description	Example
<code>&amp;&amp;</code>	<code>And</code>	<code>\$j == 3 &amp;&amp; \$k == 2</code>
<code>and</code>	Low-precedence <code>and</code>	<code>\$j == 3 and \$k == 2</code>
<code>  </code>	<code>Or</code>	<code>\$j &lt; 5    \$j &gt; 10</code>
<code>or</code>	Low-precedence <code>or</code>	<code>\$j &lt; 5 or \$j &gt; 10</code>
<code>!</code>	<code>Not</code>	<code>! (\$j == \$k)</code>
<code>xor</code>	<code>Exclusive or</code>	<code>\$j xor \$k</code>

Note that `&&` is usually interchangeable with `and`; the same is true for `||` and `or`. However, because `and` and `or` have a lower precedence you should avoid using them except when they are the only option, as in the following statement, which *must* use the `or` operator (`||` cannot be used to force a second statement to execute if the first fails):

```
$html = file_get_contents($site) or die("Cannot download from $site");
```

The most unusual of these operators is `xor`, which stands for *exclusive or* and returns a TRUE value if either value is TRUE, but a FALSE value if both inputs are TRUE or both inputs are FALSE. To understand this, imagine that you want to concoct your own cleaner for household items. Ammonia makes a good cleaner, and so does bleach, so you want your cleaner to have one of these. But the cleaner must not have both, because the combination is hazardous. In PHP, you could represent this as follows:

```
$ingredient = $ammonia xor $bleach;
```

In this example, if either `$ammonia` or `$bleach` is TRUE, `$ingredient` will also be set to TRUE. But if both are TRUE or both are FALSE, `$ingredient` will be set to FALSE.

## Variable Assignment

The syntax to assign a value to a variable is always `variable = value`. Or, to reassign the value to another variable, it is `other_variable = variable`.

There are also a couple of other assignment operators that you will find useful. For example, we've already seen this:

```
$x += 10;
```

which tells the PHP parser to add the value on the right (in this instance, the value `10`) to the variable `$x`. Likewise, we could subtract as follows:

```
$y -= 10;
```

## Variable incrementing and decrementing

Adding or subtracting 1 is such a common operation that PHP provides special operators for it. You can use one of the following in place of the `+=` and `-=` operators:

```
++$x;  
--$y;
```

In conjunction with a test (an `if` statement), you could use the following code:

```
if (++$x == 10) echo $x;
```

This tells PHP to *first* increment the value of `$x` and then to test whether it has the value `10` and, if it does, to output its value. But you can also require PHP to increment (or, as in the following example, decrement) a variable *after* it has tested the value, like this:

```
if ($y-- == 0) echo $y;
```

which gives a subtly different result. Suppose `$y` starts out as `0` before the statement is executed. The comparison will return a TRUE result, but `$y` will be set to `-1` after the comparison is made. So what will the `echo` statement display: `0` or `-1`? Try to guess, and then try out the statement in a PHP processor to confirm. Because this combination of statements is confusing, it should be taken as just an educational example and not as a guide to good programming style.

In short, a variable is incremented or decremented before the test if the operator is placed before the variable, whereas the variable is incremented or decremented after the test if the operator is placed after the variable.

By the way, the correct answer to the previous question is that the `echo` statement will display the result `-1`, because `$y` was decremented right after it was accessed in the `if` statement, and before the `echo` statement.

## String concatenation

*Concatenation* is a somewhat arcane term for putting something after another thing. So, string concatenation uses the period (.) to append one string of characters to another. The simplest way to do this is as follows:

```
echo "You have " . $msgs . " messages.;"
```

Assuming that the variable `$msgs` is set to the value `5`, the output from this line of code will be the following:

**You have 5 messages.**

Just as you can add a value to a numeric variable with the `+=` operator, you can append one string to another using `.=`, like this:

```
$bulletin .= $newsflash;
```

In this case, if `$bulletin` contains a news bulletin and `$newsflash` has a news flash, the command appends the news flash to the news bulletin so that `$bulletin` now comprises both strings of text.

## String types

PHP supports two types of strings that are denoted by the type of quotation mark that you use. If you wish to assign a literal string, preserving the exact contents, you should use single quotation marks (apostrophes), like this:

```
$info = 'Preface variables with a $ like this: $variable';
```

In this case, every character within the single-quoted string is assigned to `$info`. If you had used double quotes, PHP would have attempted to evaluate `$variable` as a variable.

On the other hand, when you want to include the value of a variable inside a string, you do so by using double-quoted strings:

```
echo "This week $count people have viewed your profile";
```

As you will realize, this syntax also offers a simpler option to concatenation in which you don't need to use a period, or close and reopen quotes, to append one string to another. This is called *variable substitution*, and some programmers use it extensively whereas others don't use it at all.

## Escaping characters

Sometimes a string needs to contain characters with special meanings that might be interpreted incorrectly. For example, the following line of code will not work, because the second quotation mark encountered in the word *spelling's* will tell the PHP parser that the string's end has been reached. Consequently, the rest of the line will be rejected as an error:

```
$text = 'My spelling's atroshus'; // Erroneous syntax
```

To correct this, you can add a backslash directly before the offending quotation mark to tell PHP to treat the character literally and not to interpret it:

```
$text = 'My spelling\'s still atroshus';
```

And you can perform this trick in almost all situations in which PHP would otherwise return an error by trying to interpret a character. For example, the following double-quoted string will be correctly assigned:

```
$text = "She wrote upon it, \"Return to sender\";";
```

Additionally, you can use escape characters to insert various special characters into strings, such as tabs, newlines, and carriage returns. These are represented, as you might guess, by \t, \n, and \r. Here is an example using tabs to lay out a heading—it is included here merely to illustrate escapes, because in web pages there are always better ways to do layout:

```
$heading = "Date\tName\tPayment";
```

These special backslash-preceded characters work only in double-quoted strings. In single-quoted strings, the preceding string would be displayed with the ugly \t sequences instead of tabs. Within single-quoted strings, only the escaped apostrophe (\') and escaped backslash itself (\\\) are recognized as escaped characters.

## Multiple-Line Commands

There are times when you need to output quite a lot of text from PHP, and using several `echo` (or `print`) statements would be time-consuming and messy. To overcome this, PHP offers two conveniences. The first is just to

put multiple lines between quotes, as in [Example 3-6](#). Variables can also be assigned, as in [Example 3-7](#).

*Example 3-6. A multiline string echo statement*

---

```
<?php
$author = "Steve Ballmer";

echo "Developers, developers, developers, developers, developers,
developers, developers, developers, developers!

- $author .";
?>
```

*Example 3-7. A multiline string assignment*

---

```
<?php
$author = "Bill Gates";

$text = "Measuring programming progress by lines of code is like
Measuring aircraft building progress by weight.

- $author .";
?>
```

PHP also offers a multiline sequence using the <<< operator—commonly referred to as a *here-document* or *heredoc*—as a way of specifying a string literal, preserving the line breaks and other whitespace (including indentation) in the text. Its use can be seen in [Example 3-8](#).

*Example 3-8. Alternative multiline echo statement*

---

```
<?php
$author = "Brian W. Kernighan";

echo <<<_END
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it.
```

```
- $author.  
_END;  
?>
```

This code tells PHP to output everything between the two \_END tags as if it were a double-quoted string (except that quotes in a heredoc do not need to be escaped). This means it's possible, for example, for a developer to write entire sections of HTML directly into PHP code and then just replace specific dynamic parts with PHP variables.

It is important to remember that the closing \_END; *must* appear right at the start of a new line and it must be the *only* thing on that line—not even a comment is allowed to be added after it (nor even a single space). Once you have closed a multiline block, you are free to use the same tag name again.

#### NOTE

Remember: using the <<<\_END...\_END; heredoc construct, you don't have to add \n linefeed characters to send a linefeed—just press Return and start a new line. Also, unlike in either a double-quote- or single-quote-delimited string, you are free to use all the single and double quotes you like within a heredoc, without escaping them by preceding them with a backslash (\).

**Example 3-9** shows how to use the same syntax to assign multiple lines to a variable.

#### Example 3-9. A multiline string variable assignment

```
<?php  
$author = "Scott Adams";  
  
$out = <<<_END  
Normal people believe that if it ain't broke, don't fix it.  
Engineers believe that if it ain't broke, it doesn't have enough  
features yet.  
  
- $author.  
_END;
```

```
echo $out;  
?>
```

The variable `$out` will then be populated with the contents between the two tags. If you were appending, rather than assigning, you could also have used `.=` in place of `=` to append the string to `$out`.

Be careful not to place a semicolon directly after the first occurrence of `_END`, as that would terminate the multiline block before it had even started and cause a `Parse error` message. The only place for the semicolon is after the terminating `_END` tag, although it is safe to use semicolons within the block as normal text characters.

By the way, the `_END` tag is simply one I chose for these examples because it is unlikely to be used anywhere else in PHP code and is therefore unique. You can use any tag you like, such as `_SECTION1` or `_OUTPUT` and so on. Also, to help differentiate tags such as this from variables or functions, the general practice is to preface them with an underscore, but you don't have to use one if you choose not to.

#### NOTE

Laying out text over multiple lines is usually just a convenience to make your PHP code easier to read, because once it is displayed in a web page, HTML formatting rules take over and whitespace is suppressed (but `$author` in our example will still be replaced with the variable's value).

So, for example, if you load these multiline output examples into a browser, they will *not* display over several lines, because all browsers treat newlines just like spaces. However, if you use the browser's View Source feature, you will find that the newlines are correctly placed, and that PHP preserved the line breaks.

## Variable Typing

PHP is a very loosely typed language. This means that variables do not have to be declared before they are used, and that PHP always converts variables to the type required by their context when they are accessed.

For example, you can create a multiple-digit number and extract the *n*th digit from it simply by assuming it to be a string. In [Example 3-10](#), the numbers 12345 and 67890 are multiplied together, returning a result of 838102050, which is then placed in the variable \$number.

*Example 3-10. Automatic conversion from a number to a string*

---

```
<?php  
$number = 12345 * 67890;  
echo substr($number, 3, 1);  
?>
```

At the point of the assignment, \$number is a numeric variable. But on the second line, a call is placed to the PHP function substr, which asks for one character to be returned from \$number, starting at the fourth position (remember that PHP offsets start from zero). To do this, PHP turns \$number into a nine-character string, so that substr can access it and return the character, which in this case is 1.

The same goes for turning a string into a number, and so on. In [Example 3-11](#), the variable \$pi is set to a string value, which is then automatically turned into a floating-point number in the third line by the equation for calculating a circle's area, which outputs the value 78.5398175.

*Example 3-11. Automatically converting a string to a number*

---

```
<?php  
$pi      = "3.1415927";  
$radius = 5;  
echo $pi * ($radius * $radius);  
?>
```

In practice, what this all means is that you don't have to worry too much about your variable types. Just assign them values that make sense to you, and PHP will convert them if necessary. Then, when you want to retrieve values, just ask for them—for example, with an echo statement.

## Constants

*Constants* are similar to variables, holding information to be accessed later, except that they are what they sound like—constant. In other words, once you have defined one, its value is set for the remainder of the program and cannot be altered.

One example of a use for a constant is to hold the location of your server root (the folder with the main files of your website). You would define such a constant like this:

```
define("ROOT_LOCATION", "/usr/local/www/");
```

Then, to read the contents of the variable, you just refer to it like a regular variable (but it isn't preceded by a dollar sign):

```
$directory = ROOT_LOCATION;
```

Now, whenever you need to run your PHP code on a different server with a different folder configuration, you have only a single line of code to change.

### NOTE

The main two things you have to remember about constants are that they must *not* be prefaced with a \$ (unlike regular variables), and that you can define them only using the `define` function.

It is generally considered a good practice to use only uppercase letters for constant variable names, especially if other people will also read your code.

## Predefined Constants

PHP comes ready-made with dozens of predefined constants that you won't generally use as a beginner. However, there are a few—known as the *magic*

*constants*—that you will find useful. The names of the magic constants always have two underscores at the beginning and two at the end, so that you won't accidentally try to name one of your own constants with a name that is already taken. They are detailed in [Table 3-5](#). The concepts referred to in the table will be introduced in future chapters.

*Table 3-5. PHP's magic constants*

Mag ic con stan t	Description
<code>__LINE__</code>	The current line number of the file.
<code>__FILE__</code>	The full path and filename of the file. If used inside an <code>include</code> , the name of the included file is returned. Some operating systems allow aliases for directories, called <i>symbolic links</i> ; in <code>__FILE__</code> these are always changed to the actual directories.
<code>__DIR__</code>	The directory of the file. (Added in PHP 5.3.0.) If used inside an <code>include</code> , the directory of the included file is returned. This is equivalent to <code>dirname(__FILE__)</code> . This directory name does not have a trailing slash unless it is the root directory.
<code>__FUNCTION__</code>	The function name. (Added in PHP 4.3.0.) As of PHP 5, returns the function name as it was declared (case-sensitive). In PHP 4, its value is always lowercase.
<code>__CLASS__</code>	The class name. (Added in PHP 4.3.0.) As of PHP 5, returns the class name as it was declared (case-sensitive). In PHP 4, its value is always lowercased.
<code>__METHOD__</code>	The class method name. (Added in PHP 5.0.0.) The method name is returned as it was declared (case-sensitive).
<code>__NAMESPACE__</code>	The name of the current namespace. (Added in PHP 5.3.0.) This constant is defined at compile time (case-sensitive).

One handy use of these variables is for debugging, when you need to insert a line of code to see whether the program flow reaches it:

```
echo "This is line " . __LINE__ . " of file " . __FILE__;
```

This prints the current program line in the current file (including the path) to the web browser.

## The Difference Between the echo and print Commands

So far, you have seen the `echo` command used in a number of different ways to output text from the server to your browser. In some cases, a string literal has been output. In others, strings have first been concatenated or variables have been evaluated. I've also shown output spread over multiple lines.

But there is an alternative to `echo` that you can use: `print`. The two commands are quite similar, but `print` is a function-like construct that takes a single parameter and has a return value (which is always 1), whereas `echo` is purely a PHP language construct. Since both commands are constructs, neither requires parentheses.

By and large, the `echo` command usually will be a tad faster than `print`, because it doesn't set a return value. On the other hand, because it isn't implemented like a function, `echo` cannot be used as part of a more complex expression, whereas `print` can. Here's an example to output whether the value of a variable is TRUE or FALSE using `print`—something you could not perform in the same manner with `echo`, because it would display a `Parse error` message:

```
$b ? print "TRUE" : print "FALSE";
```

The question mark is simply a way of interrogating whether variable `$b` is TRUE or FALSE. Whichever command is on the left of the following colon is executed if `$b` is TRUE, whereas the command to the right of the colon is executed if `$b` is FALSE.

Generally, though, the examples in this book use `echo`, and I recommend that you do so as well until you reach such a point in your PHP development that you discover the need for using `print`.

## Functions

*Functions* separate sections of code that perform a particular task. For example, maybe you often need to look up a date and return it in a certain format. That would be a good example to turn into a function. The code doing it might be only three lines long, but if you have to paste it into your program a dozen times, you're making your program unnecessarily large and complex if you don't use a function. And if you decide to change the date format later, putting it in a function means having to change it in only one place.

Placing code into a function not only shortens your program and makes it more readable, but also adds extra functionality (pun intended), because functions can be passed parameters to make them perform differently. They can also return values to the calling code.

To create a function, declare it in the manner shown in [Example 3-12](#).

---

*Example 3-12. A simple function declaration*

```
<?php
function longdate($timestamp)
{
    return date("l F jS Y", $timestamp);
}
?>
```

This function returns a date in the format *Sunday May 2nd 2021*. Any number of parameters can be passed between the initial parentheses; we have chosen to accept just one. The curly braces enclose all the code that is executed when you later call the function. Note that the first letter within the `date` function call in this example is a lowercase letter L, not to be confused with the number 1.

To output today's date using this function, place the following call in your code:

```
echo longdate(time());
```

If you need to print out the date 17 days ago, you now just have to issue the following call:

```
echo longdate(time() - 17 * 24 * 60 * 60);
```

which passes to `longdate` the current time less the number of seconds since 17 days ago ( $17 \text{ days} \times 24 \text{ hours} \times 60 \text{ minutes} \times 60 \text{ seconds}$ ).

Functions can also accept multiple parameters and return multiple results, using techniques that I'll introduce over the following chapters.

## Variable Scope

If you have a very long program, it's quite possible that you could start to run out of good variable names, but with PHP you can decide the *scope* of a variable. In other words, you can, for example, tell it that you want the variable `$temp` to be used only inside a particular function and to forget it was ever used when the function returns. In fact, this is the default scope for PHP variables.

Alternatively, you could inform PHP that a variable is global in scope and thus can be accessed by every other part of your program.

### Local variables

*Local variables* are variables that are created within, and can be accessed only by, a function. They are generally temporary variables that are used to store partially processed results prior to the function's return.

One set of local variables is the list of arguments to a function. In the previous section, we defined a function that accepted a parameter named

`$timestamp`. This is meaningful only in the body of the function; you can't get or set its value outside the function.

For another example of a local variable, take another look at the `longdate` function, which is modified slightly in [Example 3-13](#).

*Example 3-13. An expanded version of the longdate function*

---

```
<?php
function longdate($timestamp)
{
    $temp = date("l F jS Y", $timestamp);
    return "The date is $temp";
}
?>
```

Here we have assigned the value returned by the `date` function to the temporary variable `$temp`, which is then inserted into the string returned by the function. As soon as the function returns, the `$temp` variable and its contents disappear, as if they had never been used at all.

Now, to see the effects of variable scope, let's look at some similar code in [Example 3-14](#). Here `$temp` has been created *before* we call the `longdate` function.

*Example 3-14. This attempt to access \$temp in function longdate will fail*

---

```
<?php
$temp = "The date is ";
echo longdate(time());

function longdate($timestamp)
{
    return $temp . date("l F jS Y", $timestamp);
}
?>
```

However, because `$temp` was neither created within the `longdate` function nor passed to it as a parameter, `longdate` cannot access it. Therefore, this code snippet outputs only the date, not the preceding text. In fact,

depending on how PHP is configured, it may first display the error message **Notice: Undefined variable: temp**, something you don't want your users to see.

The reason for this is that, by default, variables created within a function are local to that function, and variables created outside of any functions can be accessed only by nonfunction code.

Some ways to repair [Example 3-14](#) appear in Examples [3-15](#) and [3-16](#).

*Example 3-15. Rewriting to refer to \$temp within its local scope fixes the problem*

---

```
<?php
    $temp = "The date is ";
    echo $temp . longdate(time());

    function longdate($timestamp)
    {
        return date("l F jS Y", $timestamp);
    }
?>
```

[Example 3-15](#) moves the reference to `$temp` out of the function. The reference appears in the same scope where the variable was defined.

*Example 3-16. An alternative solution: passing \$temp as an argument*

---

```
<?php
    $temp = "The date is ";
    echo longdate($temp, time());

    function longdate($text, $timestamp)
    {
        return $text . date("l F jS Y", $timestamp);
    }
?>
```

The solution in [Example 3-16](#) passes `$temp` to the `longdate` function as an extra argument. `longdate` reads it into a temporary variable that it creates

called `$text` and outputs the desired result.

### NOTE

Forgetting the scope of a variable is a common programming error, so remembering how variable scope works will help you debug some quite obscure problems. Suffice it to say that unless you have declared a variable otherwise, its scope is limited to being local: either to the current function, or to the code outside of any functions, depending on whether it was first created or accessed inside or outside a function.

## Global variables

There are cases when you need a variable to have *global* scope, because you want all your code to be able to access it. Also, some data may be large and complex, and you don't want to keep passing it as arguments to functions.

To access variables from global scope, add the keyword `global`. Let's assume that you have a way of logging your users into your website and want all your code to know whether it is interacting with a logged-in user or a guest. One way to do this is to use the `global` keyword before a variable such as `$is_logged_in`:

```
global $is_logged_in;
```

Now your login function simply has to set that variable to 1 upon a successful login attempt, or 0 upon failure. Because the scope of the variable is set to global, every line of code in your program can access it.

You should use variables given global access with caution, though. I recommend that you create them only when you absolutely cannot find another way of achieving the result you desire. In general, programs that are broken into small parts and segregated data are less buggy and easier to maintain. If you have a thousand-line program (and some day you will) in which you discover that a global variable has the wrong value at some point, how long will it take you to find the code that set it incorrectly?

Also, if you have too many variables with global scope, you run the risk of using one of those names again locally, or at least thinking you have used it locally, when in fact it has already been declared as global. All manner of strange bugs can arise from such situations.

### NOTE

Sometimes I adopt the convention of making all variable names that require global access uppercase (just as it's recommended that constants should be uppercase) so that I can see at a glance the scope of a variable.

## Static variables

In the section “[Local variables](#)”, I mentioned that the value of a local variable is wiped out when the function ends. If a function runs many times, it starts with a fresh copy of the variable and the previous setting has no effect.

Here’s an interesting case. What if you have a local variable inside a function that you don’t want any other parts of your code to have access to, but you would also like to keep its value for the next time the function is called? Why? Perhaps because you want a counter to track how many times a function is called. The solution is to declare a *static* variable, as shown in [Example 3-17](#).

*Example 3-17. A function using a static variable*

---

```
<?php
    function test()
{
    static $count = 0;
    echo $count;
    $count++;
}
?>
```

Here, the very first line of the function `test` creates a static variable called `$count` and initializes it to a value of `0`. The next line outputs the variable's value; the final one increments it.

The next time the function is called, because `$count` has already been declared, the first line of the function is skipped. Then the previously incremented value of `$count` is displayed before the variable is again incremented.

If you plan to use static variables, you should note that you cannot assign the result of an expression in their definitions. They can be initialized only with predetermined values (see [Example 3-18](#)).

#### Example 3-18. Allowed and disallowed static variable declarations

---

```
<?php
    static $int = 0;          // Allowed
    static $int = 1+2;        // Correct (as of PHP 5.6)
    static $int = sqrt(144); // Disallowed
?>
```

## Superglobal variables

Starting with PHP 4.1.0, several predefined variables are available. These are known as *superglobal variables*, which means that they are provided by the PHP environment but are global within the program, accessible absolutely everywhere.

These superglobals contain lots of useful information about the currently running program and its environment (see [Table 3-6](#)). They are structured as associative arrays, a topic discussed in [Chapter 6](#).

*Table 3-6. PHP's superglobal variables*

Superglobal name	Contents
<code>\$GLOBALS</code>	All variables that are currently defined in the global scope of the script. The variable names are the keys of the array.
<code>\$_SERVER</code>	Information such as headers, paths, and locations of scripts. The entries in this array are created by the web server, and there is no guarantee that every web server will provide any or all of these.
<code>\$_GET</code>	Variables passed to the current script via the HTTP GET method.
<code>\$_POST</code>	Variables passed to the current script via the HTTP POST method.
<code>\$_FILES</code>	Items uploaded to the current script via the HTTP POST method.
<code>\$_COOKIE</code>	Variables passed to the current script via HTTP cookies.
<code>\$_SESSION</code>	Session variables available to the current script.
<code>\$_REQUEST</code>	Contents of information passed from the browser; by default, <code>\$_GET</code> , <code>\$_POST</code> , and <code>\$_COOKIE</code> .
<code>\$_ENV</code>	Variables passed to the current script via the environment method.

All of the superglobals (except for `$GLOBALS`) are named with a single initial underscore and only capital letters; therefore, you should avoid naming your own variables in this manner to avoid potential confusion.

To illustrate how you use them, let's look at a common example. Among the many nuggets of information supplied by superglobal variables is the URL of the page that referred the user to the current web page. This referring page information can be accessed like this:

```
$came_from = $_SERVER['HTTP_REFERER'];
```

It's that simple. Oh, and if the user came straight to your web page, such as by typing its URL directly into a browser, `$came_from` will be set to an

empty string.

## Superglobals and security

A word of caution is in order before you start using superglobal variables, because they are often used by hackers trying to find exploits to break into your website. What they do is load up `$_POST`, `$_GET`, or other superglobals with malicious code, such as Unix or MySQL commands that can damage or display sensitive data if you naively access them.

Therefore, you should always sanitize superglobals before using them. One way to do this is via the PHP `htmlentities` function. It converts all characters into HTML entities. For example, less-than and greater-than characters (< and >) are transformed into the strings `&lt;` and `&gt;`; so that they are rendered harmless, as are all quotes and backslashes, and so on.

Therefore, a much better way to access `$_SERVER` (and other superglobals) is:

```
$came_from = htmlentities($_SERVER['HTTP_REFERER']);
```

### WARNING

Using the `htmlentities` function for sanitization is an important practice in any circumstance where user or other third-party data is being processed for output, not just with superglobals.

This chapter has provided you with a solid introduction to using PHP. In [Chapter 4](#), you'll start using what you've learned to build expressions and control program flow—in other words, do some actual programming.

But before moving on, I recommend that you test yourself with some (if not all) of the following questions to ensure that you have fully digested the contents of this chapter.

# Questions

1. What tag is used to invoke PHP to start interpreting program code?  
And what is the short form of the tag?
2. What are the two types of comment tags?
3. Which character must be placed at the end of every PHP statement?
4. Which symbol is used to preface all PHP variables?
5. What can a variable store?
6. What is the difference between `$variable = 1` and `$variable == 1`?
7. Why do you suppose that an underscore is allowed in variable names (`$current_user`), whereas hyphens are not (`$current-user`)?
8. Are variable names case-sensitive?
9. Can you use spaces in variable names?
10. How do you convert one variable type to another (say, a string to a number)?
11. What is the difference between `+$j` and `$j++`?
12. Are the operators `&&` and `and` interchangeable?
13. How can you create a multiline `echo` or assignment?
14. Can you redefine a constant?
15. How do you escape a quotation mark?
16. What is the difference between the `echo` and `print` commands?
17. What is the purpose of functions?

18. How can you make a variable accessible to all parts of a PHP program?
19. If you generate data within a function, what are a couple of ways to convey the data to the rest of the program?
20. What is the result of combining a string with a number?

See “[Chapter 3 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 4. Expressions and Control Flow in PHP

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

The previous chapter introduced several topics in passing that this chapter covers more fully, such as making choices (branching) and creating complex expressions. In the previous chapter, I wanted to focus on the most basic syntax and operations in PHP, but I couldn’t avoid touching on more advanced topics. Now I can fill in the background that you need to use these powerful PHP features properly.

In this chapter, you will get a thorough grounding in how PHP programming works in practice and how to control the flow of the program.

## Expressions

Let’s start with the most fundamental part of any programming language: *expressions*.

An expression is a combination of values, variables, operators, and functions that results in a value. It's familiar to anyone who has taken high-school algebra. Here's an example:

$$y = 3 (|2x| + 4)$$

Which in PHP would be:

```
$y = 3 * (abs(2 * $x) + 4);
```

The value returned ( $y$  in this mathematical statement, or  $\$y$  in the PHP) can be a number, a string, or a *Boolean value* (named after George Boole, a 19th-century English mathematician and philosopher). By now, you should be familiar with the first two value types, but I'll explain the third.

## TRUE or FALSE?

A basic Boolean value can be either TRUE or FALSE. For example, the expression  $20 > 9$  (20 is greater than 9) is TRUE, and the expression  $5 == 6$  (5 is equal to 6) is FALSE. (You can combine such operations using other classic Boolean operators such as AND, OR, and XOR, which are covered later in this chapter.)

### NOTE

Note that I am using uppercase letters for the names TRUE and FALSE. This is because they are predefined constants in PHP. You can use the lowercase versions if you prefer, as they are also predefined. In fact, the lowercase versions are more stable, because PHP does not allow you to redefine them; the uppercase ones may be redefined, which is something you should bear in mind if you import third-party code.

PHP doesn't actually print the predefined constants if you ask it to do so as in [Example 4-1](#). For each line, the example prints out a letter followed by a colon and a predefined constant. PHP arbitrarily assigns a numerical value

of 1 to TRUE, so 1 is displayed after **a:** when the example runs. Even more mysteriously, because **b:** evaluates to FALSE, it does not show any value. In PHP the constant FALSE is defined as NULL, another predefined constant that denotes nothing.

*Example 4-1. Outputting the values of TRUE and FALSE*

---

```
<?php // test2.php
echo "a: [" . TRUE . "]<br>";
echo "b: [" . FALSE . "]<br>";
?>
```

The <br> tags are there to create line breaks and thus separate the output into two lines in HTML. Here is the output:

```
a: [1]
b: []
```

Turning to Boolean expressions, [Example 4-2](#) shows some simple expressions: the two I mentioned earlier, plus a couple more.

*Example 4-2. Four simple Boolean expressions*

---

```
<?php
echo "a: [" . (20 > 9) . "]<br>";
echo "b: [" . (5 == 6) . "]<br>";
echo "c: [" . (1 == 0) . "]<br>";
echo "d: [" . (1 == 1) . "]<br>";
?>
```

The output from this code is:

```
a: [1]
b: []
c: []
d: [1]
```

By the way, in some languages FALSE may be defined as 0 or even -1, so it's worth checking on its definition in each language you use. Luckily, Boolean expressions are usually buried in other code, so you don't normally have to worry about what TRUE and FALSE look like internally. In fact, even those names rarely appear in code.

## Literals and Variables

These are the most basic elements of programming, and the building blocks of expressions. A *literal* simply means something that evaluates to itself, such as the number 73 or the string "Hello". A variable, which we've already seen has a name beginning with a dollar sign, evaluates to the value that has been assigned to it. The simplest expression is just a single literal or variable, because both return a value.

**Example 4-3** shows three literals and two variables, all of which return values, albeit of different types.

*Example 4-3. Literals and variables*

---

```
<?php
    $myname = "Brian";
    $myage  = 37;

    echo "a: " . 73      . "<br>"; // Numeric literal
    echo "b: " . "Hello" . "<br>"; // String literal
    echo "c: " . FALSE   . "<br>"; // Constant literal
    echo "d: " . $myname . "<br>"; // String variable
    echo "e: " . $myage  . "<br>"; // Numeric variable
?>
```

And, as you'd expect, you see a return value from all of these with the exception of c:, which evaluates to FALSE, returning nothing in the following output:

```
a: 73
b: Hello
c:
```

```
d: Brian  
e: 37
```

In conjunction with operators, it's possible to create more complex expressions that evaluate to useful results.

Programmers combine expressions with other language constructs, such as the assignment operators we saw earlier, to form *statements*. [Example 4-4](#) shows two statements. The first assigns the result of the expression `366 - $day_number` to the variable `$days_to_new_year`, and the second outputs a friendly message only if the expression `$days_to_new_year < 30` evaluates to TRUE.

#### Example 4-4. An expression and a statement

---

```
<?php  
$days_to_new_year = 366 - $day_number; // Expression  
  
if ($days_to_new_year < 30)  
{  
    echo "Not long now till new year"; // Statement  
}  
?>
```

## Operators

PHP offers a lot of powerful operators of different types—arithmetic, string, logical, assignment, comparison, and more (see [Table 4-1](#)).

*Table 4-1. PHP operator types*

Operator	Description	Example
Arithmetic	Basic mathematics	<code>\$a + \$b</code>
Array	Array union	<code>\$a + \$b</code>
Assignment	Assign values	<code>\$a = \$b + 23</code>
Bitwise	Manipulate bits within bytes	<code>12 ^ 9</code>
Comparison	Compare two values	<code>\$a &lt; \$b</code>
Execution	Execute contents of backticks	<code>`ls -al`</code>
Increment/decrement	Add or subtract 1	<code>\$a++</code>
Logical	Boolean	<code>\$a and \$b</code>
String	Concatenation	<code>\$a . \$b</code>

Each operator takes a different number of operands:

- *Unary* operators, such as incrementing (`$a++`) or negation (`!$a`), take a single operand.
- *Binary* operators, which represent the bulk of PHP operators (including addition, subtraction, multiplication, and division), take two operands.
- The one *ternary* operator, which takes the form `expr ? x : y`, requires three operands. It's a terse, single-line `if` statement that returns `x` if `expr` is TRUE and `y` if `expr` is FALSE.

## Operator Precedence

If all operators had the same precedence, they would be processed in the order in which they are encountered. In fact, many operators do have the same precedence. Take a look at [Example 4-5](#).

### Example 4-5. Three equivalent expressions

`1 + 2 + 3 - 4 + 5`

```
2 - 4 + 5 + 3 + 1  
5 + 2 - 4 + 1 + 3
```

Here you will see that although the numbers (and their preceding operators) have been moved around, the result of each expression is the value 7, because the plus and minus operators have the same precedence. We can try the same thing with multiplication and division (see [Example 4-6](#)).

*Example 4-6. Three expressions that are also equivalent*

---

```
1 * 2 * 3 / 4 * 5  
2 / 4 * 5 * 3 * 1  
5 * 2 / 4 * 1 * 3
```

Here the resulting value is always 7.5. But things change when we mix operators with *different* precedences in an expression, as in [Example 4-7](#).

*Example 4-7. Three expressions using operators of mixed precedence*

---

```
1 + 2 * 3 - 4 * 5  
2 - 4 * 5 * 3 + 1  
5 + 2 - 4 + 1 * 3
```

If there were no operator precedence, these three expressions would evaluate to 25, -29, and 12, respectively. But because multiplication and division take precedence over addition and subtraction, the expressions are evaluated as if there were parentheses around these parts of the expressions, just like mathematical notation (see [Example 4-8](#)).

*Example 4-8. Three expressions showing implied parentheses*

---

```
1 + (2 * 3) - (4 * 5)  
2 - (4 * 5 * 3) + 1  
5 + 2 - 4 + (1 * 3)
```

PHP evaluates the subexpressions within parentheses first to derive the semi-completed expressions in [Example 4-9](#).

*Example 4-9. After evaluating the subexpressions in parentheses*

---

```
1 + (6) - (20)
2 - (60) + 1
5 + 2 - 4 + (3)
```

The final results of these expressions are  $-13$ ,  $-57$ , and  $6$ , respectively (quite different from the results of  $25$ ,  $-29$ , and  $12$  that we would have seen had there been no operator precedence).

Of course, you can override the default operator precedence by inserting your own parentheses and forcing whatever order you want (see [Example 4-10](#)).

*Example 4-10. Forcing left-to-right evaluation*

---

```
((1 + 2) * 3 - 4) * 5
(2 - 4) * 5 * 3 + 1
(5 + 2 - 4 + 1) * 3
```

With parentheses correctly inserted, we now see the values  $25$ ,  $-29$ , and  $12$ , respectively.

[Table 4-2](#) lists PHP's operators in order of precedence from high to low.

*Table 4-2. The precedence of PHP operators (high to low)*

Operator(s)	Type
( )	Parentheses
<code>++ --</code>	Increment/decrement
!	Logical
<code>* / %</code>	Arithmetic
<code>+ - .</code>	Arithmetic and string
<code>&lt;&lt; &gt;&gt;</code>	Bitwise
<code>&lt; &lt;= &gt; &gt;= &lt;&gt;</code>	Comparison
<code>== != === !==</code>	Comparison
<code>&amp;</code>	Bitwise (and references)
<code>^</code>	Bitwise
<code> </code>	Bitwise
<code>&amp;&amp;</code>	Logical
<code>  </code>	Logical
<code>? :</code>	Ternary
<code>= += -= *= /= .= %= &amp;= != ^= &lt;&lt;= &gt;&gt;=</code>	Assignment
<code>and</code>	Logical
<code>xor</code>	Logical
<code>or</code>	Logical

The order in this table is not arbitrary, but carefully designed so that the most common and intuitive precedences are the ones you can get without parentheses. For instance, you can separate two comparisons with an `and` or `or` and get what you'd expect.

## Associativity

We've been looking at processing expressions from left to right, except where operator precedence is in effect. But some operators require

processing from right to left, and this direction of processing is called the operator's *associativity*. For some operators, there is no associativity.

Associativity (as detailed in [Table 4-3](#)) becomes important in cases in which you do not explicitly force precedence, so you need to be aware of the default actions of operators.

*Table 4-3. Operator associativity*

Operator	Description	Associativity
< <= >= == != === !== <>	Comparison	None
!	Logical NOT	Right
~	Bitwise NOT	Right
++ --	Increment and decrement	Right
(int)	Cast to an integer	Right
(double) (float) (real)	Cast to a floating-point number	Right
(string)	Cast to a string	Right
(array)	Cast to an array	Right
(object)	Cast to an object	Right
@	Inhibit error reporting	Right
= += -= *= /=	Assignment	Right
.= %= &=  = ^= <<= >>=	Assignment	Right
+	Addition and unary plus	Left
-	Subtraction and negation	Left
*	Multiplication	Left
/	Division	Left
%	Modulus	Left
.	String concatenation	Left
<< >> & ^	Bitwise	Left
?:	Ternary	Left
&& and or xor	Logical	Left
,	Separator	Left

For example, let's take a look at the assignment operator in [Example 4-11](#), where three variables are all set to the value 0.

#### *Example 4-11. A multiple-assignment statement*

---

```
<?php  
    $level = $score = $time = 0;  
?>
```

This multiple assignment is possible only if the rightmost part of the expression is evaluated first, and then processing continues in a right-to-left direction.

#### NOTE

As a newcomer to PHP, you should avoid the potential pitfalls of operator associativity by always nesting your subexpressions within parentheses to force the order of evaluation. This will also help other programmers who may have to maintain your code to understand what is happening.

## Relational Operators

Relational operators answer questions such as “Does this variable have a value of zero?” and “Which variable has a greater value?” These operators test two operands and return a Boolean result of either TRUE or FALSE. There are three types of relational operators: *equality*, *comparison*, and *logical*.

### Equality

As we've already seen a few times in this chapter, the equality operator is == (two equals signs). It is important not to confuse it with the = (single equals sign) assignment operator. In [Example 4-12](#), the first statement assigns a value and the second tests it for equality.

#### *Example 4-12. Assigning a value and testing for equality*

---

```
<?php  
$month = "March";  
  
if ($month == "March") echo "It's springtime";  
?>
```

As you see, by returning either TRUE or FALSE, the equality operator enables you to test for conditions using, for example, an `if` statement. But that's not the whole story, because PHP is a loosely typed language. If the two operands of an equality expression are of different types, PHP will convert them to whatever type makes the best sense to it. A rarely used *identity* operator, which consists of three equals signs in a row, can be used to compare items without doing conversion.

For example, any strings composed entirely of numbers will be converted to numbers whenever compared with a number. In [Example 4-13](#), `$a` and `$b` are two different strings, and we would therefore expect neither of the `if` statements to output a result.

#### Example 4-13. The equality and identity operators

---

```
<?php  
$a = "1000";  
$b = "+1000";  
  
if ($a == $b) echo "1";  
if ($a === $b) echo "2";  
?>
```

However, if you run the example, you will see that it outputs the number 1, which means that the first `if` statement evaluated to TRUE. This is because both strings were first converted to numbers, and `1000` is the same numerical value as `+1000`. In contrast, the second `if` statement uses the identity operator, so it compares `$a` and `$b` as strings, sees that they are different, and thus doesn't output anything.

As with forcing operator precedence, whenever you have any doubt about how PHP will convert operand types, you can use the identity operator to

turn this behavior off.

In the same way that you can use the equality operator to test for operands being equal, you can test for them *not* being equal using !=, the inequality operator. Take a look at [Example 4-14](#), which is a rewrite of [Example 4-13](#), in which the equality and identity operators have been replaced with their inverses.

#### Example 4-14. The inequality and not-identical operators

---

```
<?php
$a = "1000";
$b = "+1000";

if ($a != $b) echo "1";
if ($a !== $b) echo "2";
?>
```

And, as you might expect, the first `if` statement does not output the number 1, because the code is asking whether `$a` and `$b` are *not* equal to each other numerically.

Instead, this code outputs the number 2, because the second `if` statement is asking whether `$a` and `$b` are *not* identical to each other in their actual string type, and the answer is TRUE; they are not the same.

## Comparison operators

Using comparison operators, you can test for more than just equality and inequality. PHP also gives you > (is greater than), < (is less than), >= (is greater than or equal to), and <= (is less than or equal to) to play with.

[Example 4-15](#) shows these in use.

#### Example 4-15. The four comparison operators

---

```
<?php
$a = 2; $b = 3;

if ($a > $b) echo "$a is greater than $b<br>";
```

```

if ($a < $b) echo "$a is less than $b<br>";
if ($a >= $b) echo "$a is greater than or equal to $b<br>";
if ($a <= $b) echo "$a is less than or equal to $b<br>";
?>

```

In this example, where `$a` is 2 and `$b` is 3, the following is output:

```

2 is less than 3
2 is less than or equal to 3

```

Try this example yourself, altering the values of `$a` and `$b`, to see the results. Try setting them to the same value and see what happens.

## Logical operators

Logical operators produce true or false results, and therefore are also known as *Boolean operators*. There are four of them (see [Table 4-4](#)).

*Table 4-4. The logical operators*

Logical operator	Description
AND	TRUE if both operands are TRUE
OR	TRUE if either operand is TRUE
XOR	TRUE if one of the two operands is TRUE
! (NOT)	TRUE if the operand is FALSE, or FALSE if the operand is TRUE

You can see these operators used in [Example 4-16](#). Note that the `!` symbol is required by PHP in place of NOT. Furthermore, the operators can be lower- or uppercase.

*Example 4-16. The logical operators in use*

```

<?php
$a = 1; $b = 0;

echo ($a AND $b) . "<br>";
echo ($a or $b) . "<br>";

```

```
echo ($a XOR $b) . "<br>";
echo !$a . "<br>";
?>
```

Line by line, this example outputs nothing, 1, 1, and nothing, meaning that only the second and third `echo` statements evaluate as TRUE. (Remember that NULL—or nothing—represents a value of FALSE.) This is because the AND statement requires both operands to be TRUE if it is going to return a value of TRUE, while the fourth statement performs a NOT on the value of `$a`, turning it from TRUE (a value of 1) to FALSE. If you wish to experiment with this, try out the code, giving `$a` and `$b` varying values of 1 and 0.

#### NOTE

When coding, remember that AND and OR have lower precedence than the other versions of the operators, `&&` and `||`.

The OR operator can cause unintentional problems in `if` statements, because the second operand will not be evaluated if the first is evaluated as TRUE. In [Example 4-17](#), the function `getnext` will never be called if `$finished` has a value of 1.

#### Example 4-17. A statement using the OR operator

```
<?php
if ($finished == 1 OR getnext() == 1) exit;
?>
```

If you need `getnext` to be called at each `if` statement, you could rewrite the code as has been done in [Example 4-18](#).

#### Example 4-18. The if...OR statement modified to ensure calling of getnext

```
<?php
$gn = getnext();
```

```
if ($finished == 1 OR $gn == 1) exit;  
?>
```

In this case, the code executes the `getnext` function and stores the value returned in `$gn` before executing the `if` statement.

### NOTE

Another solution is to switch the two clauses to make sure that `getnext` is executed, as it will then appear first in the expression.

**Table 4-5** shows all the possible variations of using the logical operators. You should also note that `!TRUE` equals `FALSE`, and `!FALSE` equals `TRUE`.

*Table 4-5. All possible PHP logical expressions*

Inputs		Operators and results		
a	b	AND	OR	XOR
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

## Conditionals

*Conditionals* alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways.

Conditionals are central to creating dynamic web pages—the goal of using PHP in the first place—because they make it easy to render different output each time a page is viewed.

I'll present three basic conditionals in this section: the `if` statement, the `switch` statement, and the `? operator`. In addition, looping conditionals

(which we'll get to shortly) execute code over and over until a condition is met.

## The if Statement

One way of thinking about program flow is to imagine it as a single-lane highway that you are driving along. It's pretty much a straight line, but now and then you encounter various signs telling you where to go.

In the case of an `if` statement, you could imagine coming across a detour sign that you have to follow if a certain condition is TRUE. If so, you drive off and follow the detour until you return to the main road and then continue on your way in your original direction. Or, if the condition isn't TRUE, you ignore the detour and carry on driving (see [Figure 4-1](#)).

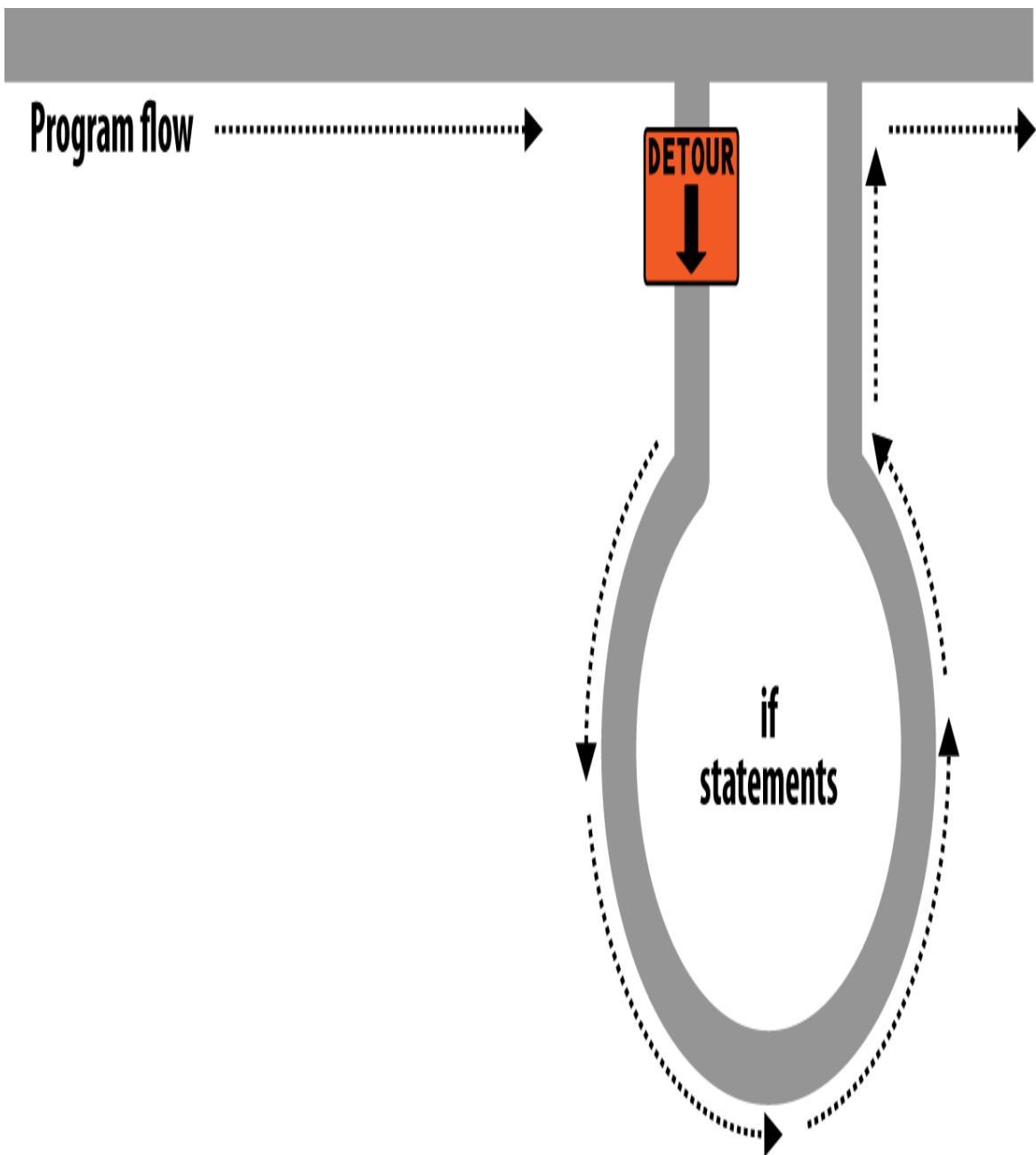


Figure 4-1. Program flow is like a single-lane highway

The contents of the `if` condition can be any valid PHP expression, including tests for equality, comparison expressions, tests for `0` and `NULL`, and even functions (either built-in functions or ones that you write).

The actions to take when an `if` condition is `TRUE` are generally placed inside curly braces (`{ }`). You can ignore the braces if you have only a single statement to execute, but if you always use curly braces, you'll avoid

having to hunt down difficult-to-trace bugs, such as when you add an extra line to a condition and it doesn't get evaluated due to the lack of braces.

### NOTE

A notorious security vulnerability known as the “goto fail” bug haunted the Secure Sockets Layer (SSL) code in Apple’s products for many years because a programmer had forgotten the curly braces around an `if` statement, causing a function to sometimes report a successful connection when that may not actually have always been the case. This allowed a malicious attacker to get a secure certificate to be accepted when it should have been rejected. If in doubt, place braces around your `if` statements.

Note that for brevity and clarity, however, many of the examples in this book ignore this suggestion and omit the braces for single statements.

In [Example 4-19](#), imagine that it is the end of the month and all your bills have been paid, so you are performing some bank account maintenance.

*Example 4-19. An if statement with curly braces*

---

```
<?php
  if ($bank_balance < 100)
  {
    $money      = 1000;
    $bank_balance += $money;
  }
?>
```

In this example, you are checking your balance to see whether it is less than \$100 (or whatever your currency is). If so, you pay yourself \$1,000 and then add it to the balance. (If only making money were that simple!)

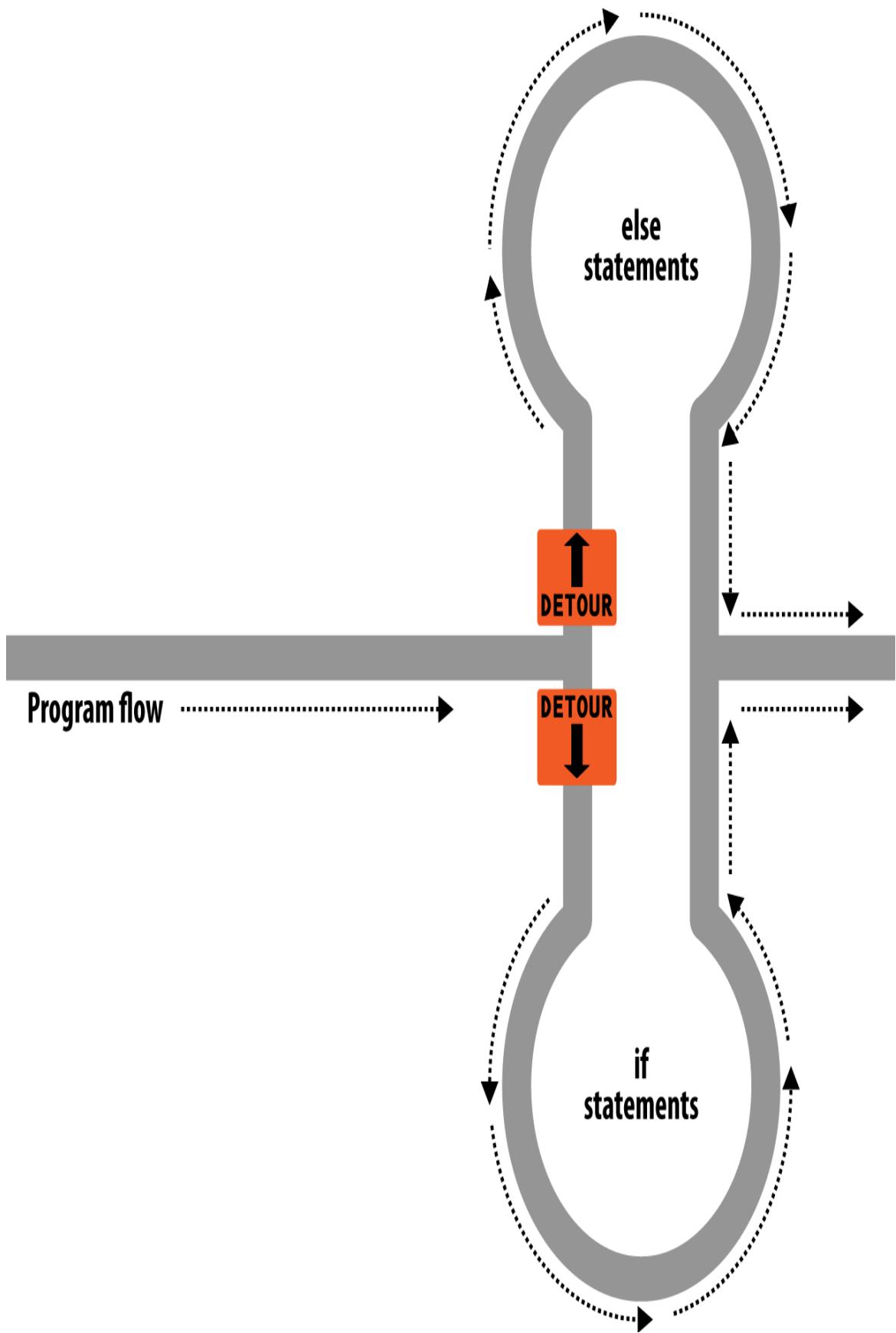
If the bank balance is \$100 or greater, the conditional statements are ignored and program flow skips to the next line (not shown).

In this book, opening curly braces generally start on a new line. Some people like to place the first curly brace to the right of the conditional expression; others start a new line with it. Either of these is fine, because

PHP allows you to set out your whitespace characters (spaces, newlines, and tabs) any way you choose. However, you will find your code easier to read and debug if you indent each level of conditionals with a tab.

## The `else` Statement

Sometimes when a conditional is not TRUE, you may not want to continue on to the main program code immediately but might wish to do something else instead. This is where the `else` statement comes in. With it, you can set up a second detour on your highway, as in [Figure 4-2](#).



*Figure 4-2. The highway now has an if detour and an else detour*

With an **if...else** statement, the first conditional statement is executed if the condition is TRUE. But if it's FALSE, the second one is executed. One of the two choices *must* be executed. Under no circumstance can both (or neither) be executed. **Example 4-20** shows the use of the **if...else** structure.

#### Example 4-20. An if...else statement with curly braces

---

```
<?php
    if ($bank_balance < 100)
    {
        $money      = 1000;
        $bank_balance += $money;
    }
    else
    {
        $savings    += 50;
        $bank_balance -= 50;
    }
?>
```

In this example, if you've ascertained that you have \$100 or more in the bank the **else** statement is executed, placing some of this money into your savings account.

As with **if** statements, if your **else** has only one conditional statement, you can opt to leave out the curly braces. (Curly braces are always recommended, though. First, they make the code easier to understand. Second, they let you easily add more statements to the branch later.)

## The **elseif** Statement

There are also times when you want a number of different possibilities to occur, based upon a sequence of conditions. You can achieve this using the **elseif** statement. As you might imagine, it is like an **else** statement, except that you place a further conditional expression prior to the

conditional code. In [Example 4-21](#), you can see a complete `if...elseif...else` construct.

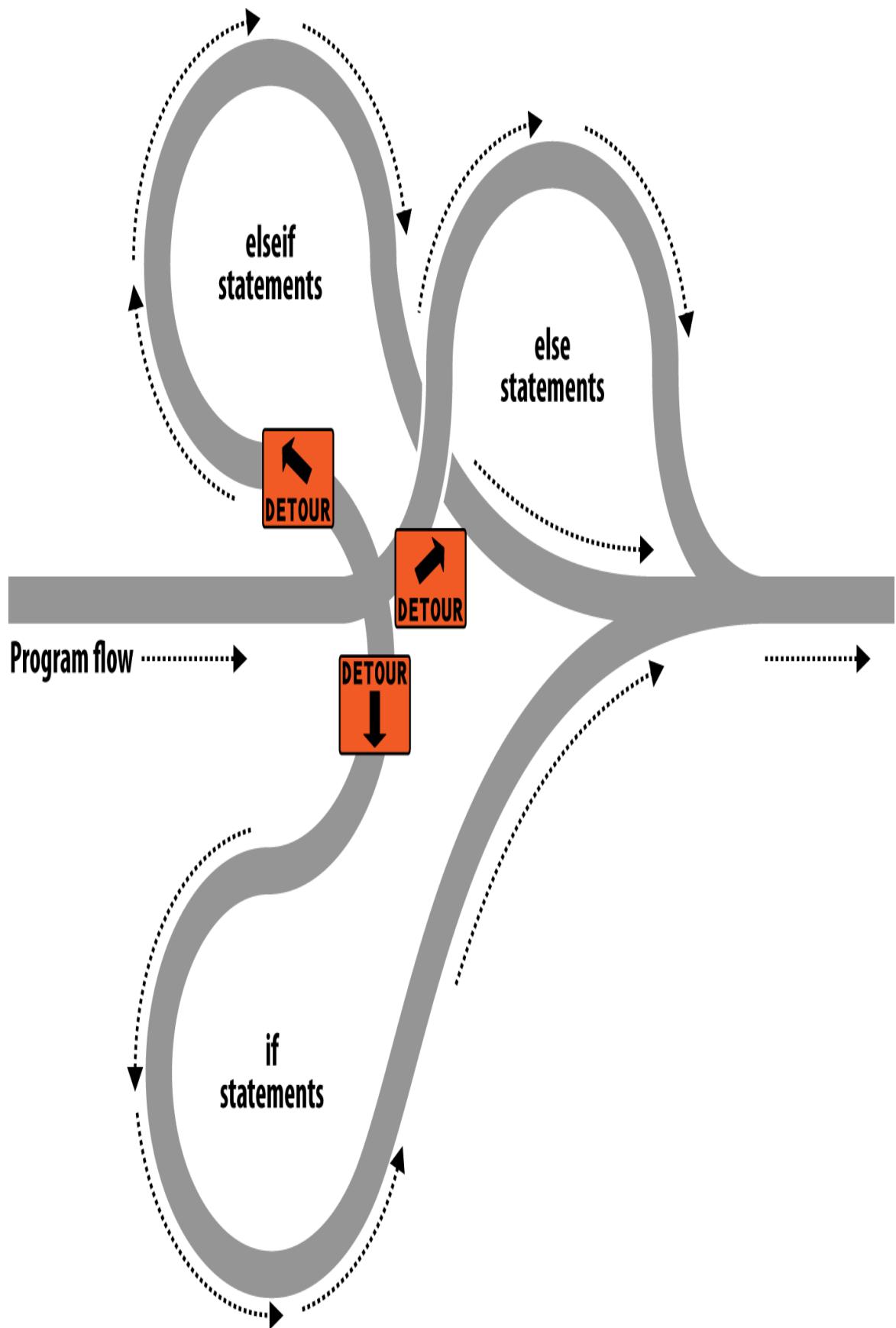
*Example 4-21. An if...elseif...else statement with curly braces*

---

```
<?php
    if ($bank_balance < 100)
    {
        $money      = 1000;
        $bank_balance += $money;
    }
    elseif ($bank_balance > 200)
    {
        $savings    += 100;
        $bank_balance -= 100;
    }
    else
    {
        $savings    += 50;
        $bank_balance -= 50;
    }
?>
```

In the example, an `elseif` statement has been inserted between the `if` and `else` statements. It checks whether your bank balance exceeds \$200 and, if so, decides that you can afford to save \$100 of it this month.

Although I'm starting to stretch the metaphor a bit too far, you can imagine this as a multiway set of detours (see [Figure 4-3](#)).



*Figure 4-3. The highway with if, elseif, and else detours*

### NOTE

An `else` statement closes either an `if...else` or an `if...elseif...else` statement. You can leave out a final `else` if it is not required, but you cannot have one before an `elseif`; neither can you have an `elseif` before an `if` statement.

You may have as many `elseif` statements as you like. But as the number of `elseif` statements increases, you would probably be better advised to consider a `switch` statement if it fits your needs. We'll look at that next.

## The switch Statement

The `switch` statement is useful where one variable, or the result of an expression, can have multiple values, each of which should trigger a different activity.

For example, consider a PHP-driven menu system that passes a single string to the main menu code according to what the user requests. Let's say the options are Home, About, News, Login, and Links, and we set the variable `$page` to one of these, according to the user's input.

If we write the code for this using `if...elseif...else`, it might look like **Example 4-22**.

---

### *Example 4-22. A multiple-line if...elseif...else statement*

```
<?php
    if      ($page == "Home") echo "You selected Home";
    elseif ($page == "About") echo "You selected About";
    elseif ($page == "News") echo "You selected News";
    elseif ($page == "Login") echo "You selected Login";
    elseif ($page == "Links") echo "You selected Links";
    else                  echo "Unrecognized selection";
?>
```

If we use a `switch` statement, the code might look like [Example 4-23](#).

### Example 4-23. A switch statement

---

```
<?php
switch ($page)
{
    case "Home":
        echo "You selected Home";
        break;
    case "About":
        echo "You selected About";
        break;
    case "News":
        echo "You selected News";
        break;
    case "Login":
        echo "You selected Login";
        break;
    case "Links":
        echo "You selected Links";
        break;
}
?>
```

As you can see, `$page` is mentioned only once at the start of the `switch` statement. Thereafter, the `case` command checks for matches. When one occurs, the matching conditional statement is executed. Of course, in a real program you would have code here to display or jump to a page, rather than simply telling the user what was selected.

#### NOTE

With `switch` statements, you do not use curly braces inside `case` commands. Instead, they commence with a colon and end with the `break` statement. The entire list of cases in the `switch` statement is enclosed in a set of curly braces, though.

## Breaking out

If you wish to break out of the `switch` statement because a condition has been fulfilled, use the `break` command. This command tells PHP to exit the `switch` and jump to the following statement.

If you were to leave out the `break` commands in [Example 4-23](#) and the case of `Home` evaluated to be TRUE, all five cases would then be executed. Or, if `$page` had the value `News`, all the `case` commands from then on would execute. This is deliberate and allows for some advanced programming, but generally you should always remember to issue a `break` command every time a set of `case` conditionals has finished executing. In fact, leaving out the `break` statement is a common error.

## Default action

A typical requirement in `switch` statements is to fall back on a default action if none of the `case` conditions are met. For example, in the case of the menu code in [Example 4-23](#), you could add the code in [Example 4-24](#) immediately before the final curly brace.

*Example 4-24. A default statement to add to [Example 4-23](#)*

---

```
default:  
echo "Unrecognized selection";  
break;
```

This replicates the effect of the `else` statement in [Example 4-22](#).

Although a `break` command is not required here because the default is the final sub-statement and program flow will automatically continue to the closing curly brace, should you decide to place the `default` statement higher up, it would definitely need a `break` command to prevent program flow from dropping into the following statements. Generally, the safest practice is to always include the `break` command.

## Alternative syntax

If you prefer, you may replace the first curly brace in a `switch` statement with a single colon and the final curly brace with an `endswitch` command,

as in [Example 4-25](#). However, this approach is not commonly used and is mentioned here only in case you encounter it in third-party code.

*Example 4-25. Alternate switch statement syntax*

---

```
<?php
switch ($page):
    case "Home":
        echo "You selected Home";
        break;

    // etc

    case "Links":
        echo "You selected Links";
        break;
endswitch;
?>
```

## The ? Operator

One way of avoiding the verbosity of `if` and `else` statements is to use the more compact ternary operator, `?`, which is unusual in that it takes three operands rather than the typical two.

We briefly came across this in [Chapter 3](#) in the discussion about the difference between the `print` and `echo` statements as an example of an operator type that works well with `print` but not `echo`.

The `?` operator is passed an expression that it must evaluate, along with two statements to execute: one for when the expression evaluates to `TRUE`, the other for when it is `FALSE`. [Example 4-26](#) shows code we might use for writing a warning about the fuel level of a car to its digital dashboard.

*Example 4-26. Using the ? operator*

---

```
<?php
echo $fuel <= 1 ? "Fill tank now" : "There's enough fuel";
?>
```

In this statement, if there is one gallon or less of fuel (in other words, `$fuel` is set to 1 or less), the string `Fill tank` now is returned to the preceding `echo` statement. Otherwise, the string `There's enough fuel` is returned. You can also assign the value returned in a `?`  statement to a variable (see [Example 4-27](#)).

*Example 4-27. Assigning a ? conditional result to a variable*

---

```
<?php  
    $enough = $fuel <= 1 ? FALSE : TRUE;  
?>
```

Here, `$enough` will be assigned the value `TRUE` only when there is more than a gallon of fuel; otherwise, it is assigned the value `FALSE`.

If you find the `?` operator confusing, you are free to stick to `if` statements, but you should be familiar with the operator because you'll see it in other people's code. It can be hard to read, because it often mixes multiple occurrences of the same variable. For instance, code such as the following is quite popular:

```
$saved = $saved >= $new ? $saved : $new;
```

If you take it apart carefully, you can figure out what this code does:

```
$saved =           // Set the value of $saved to...  
    $saved >= $new // Check $saved against $new  
?             // Yes, comparison is true...  
    $saved        // ... so assign the current value of $saved  
:             // No, comparison is false...  
    $new;         // ... so assign the value of $new
```

It's a concise way to keep track of the largest value that you've seen as a program progresses. You save the largest value in `$saved` and compare it to `$new` each time you get a new value. Programmers familiar with the `?` operator find it more convenient than `if` statements for such short

comparisons. When not used for writing compact code, it is typically used to make some decision inline, such as when you are testing whether a variable is set before passing it to a function.

## Looping

One of the great things about computers is that they can repeat calculating tasks quickly and tirelessly. Often you may want a program to repeat the same sequence of code again and again until something happens, such as a user inputting a value or reaching a natural end. PHP's loop structures provide the perfect way to do this.

To picture how this works, look at [Figure 4-4](#). It is much the same as the highway metaphor used to illustrate `if` statements, except the detour also has a loop section that—once a vehicle has entered it—can be exited only under the right program conditions.

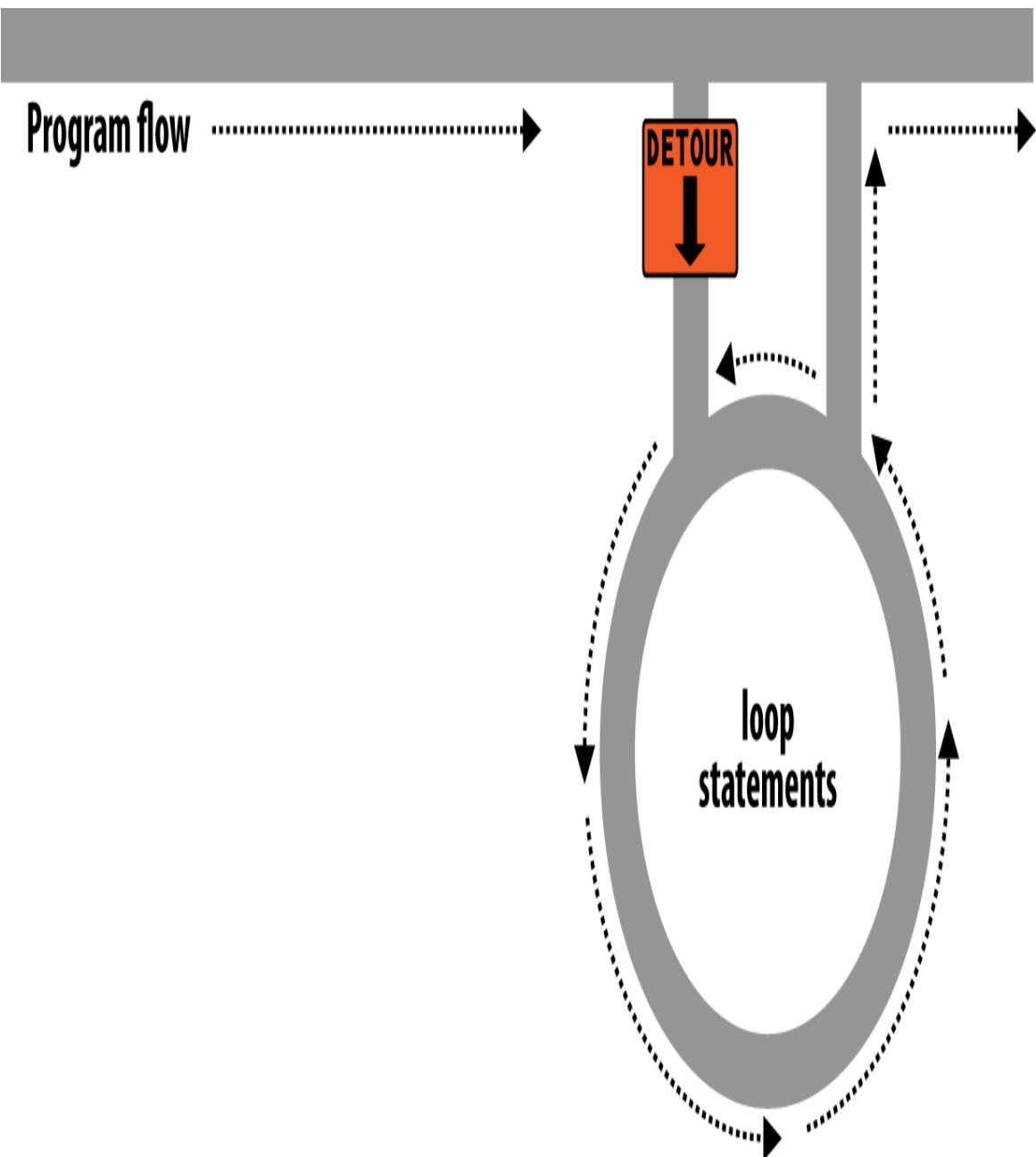


Figure 4-4. Imagining a loop as part of a program highway layout

## while Loops

Let's turn the digital car dashboard in [Example 4-26](#) into a loop that continuously checks the fuel level as you drive, using a `while` loop ([Example 4-28](#)).

[Example 4-28. A while loop](#)

```
<?php
$fuel = 10;

while ($fuel > 1)
{
    // Keep driving...
    echo "There's enough fuel";
}
?>
```

Actually, you might prefer to keep a green light lit rather than output text, but the point is that whatever positive indication you wish to make about the level of fuel is placed inside the `while` loop. By the way, if you try this example for yourself, note that it will keep printing the string until you click the Stop button in your browser.

### NOTE

As with `if` statements, you will notice that curly braces are required to hold the statements inside the `while` statements, unless there's only one.

For another example of a `while` loop that displays the 12 times table, see [Example 4-29](#).

*Example 4-29. A while loop to print the 12 times table*

---

```
<?php
$count = 1;

while ($count <= 12)
{
    echo "$count times 12 is " . $count * 12 . "<br>";
    ++$count;
}
?>
```

Here the variable `$count` is initialized to a value of 1, and then a `while` loop starts with the comparative expression `$count <= 12`. This loop will

continue executing until the variable is greater than 12. The output from this code is as follows:

```
1 times 12 is 12
2 times 12 is 24
3 times 12 is 36
and so on...
```

Inside the loop, a string is printed along with the value of `$count` multiplied by 12. For neatness, this is followed with a `<br>` tag to force a new line. Then `$count` is incremented, ready for the final curly brace that tells PHP to return to the start of the loop.

At this point, `$count` is again tested to see whether it is greater than 12. It isn't, but it now has the value 2, and after another 11 times around the loop, it will have the value 13. When that happens, the code within the `while` loop is skipped and execution passes to the code following the loop, which, in this case, is the end of the program.

If the `++$count` statement (which could equally have been `$count++`) had not been there, this loop would be like the first one in this section. It would never end, and only the result of `1 * 12` would be printed over and over.

But there is a much neater way this loop can be written, which I think you will like. Take a look at [Example 4-30](#).

*Example 4-30. A shortened version of [Example 4-29](#)*

---

```
<?php
$count = 0;

while (++$count <= 12)
    echo "$count times 12 is " . $count * 12 . "<br>";
?>
```

In this example, it was possible to move the `++$count` statement from the statements inside the `while` loop into the conditional expression of the loop. What now happens is that PHP encounters the variable `$count` at the start

of each iteration of the loop and, noticing that it is prefaced with the increment operator, first increments the variable and only then compares it to the value 12. You can therefore see that \$count now has to be initialized to 0, not 1, because it is incremented as soon as the loop is entered. If you keep the initialization at 1, only results between 2 and 12 will be output.

## do...while Loops

A slight variation to the `while` loop is the `do...while` loop, used when you want a block of code to be executed at least once and made conditional only after that. [Example 4-31](#) shows a modified version of the code for the 12 times table that uses such a loop.

*Example 4-31. A do...while loop for printing the 12 times table*

---

```
<?php
$count = 1;
do
    echo "$count times 12 is " . $count * 12 . "<br>";
    while (++$count <= 12);
?>
```

Notice how we are back to initializing \$count to 1 (rather than 0) because of the loop's `echo` statement being executed before we have an opportunity to increment the variable. Other than that, though, the code looks pretty similar.

Of course, if you have more than a single statement inside a `do...while` loop, remember to use curly braces, as in [Example 4-32](#).

*Example 4-32. Expanding Example 4-31 to use curly braces*

---

```
<?php
$count = 1;

do {
    echo "$count times 12 is " . $count * 12;
    echo "<br>";
```

```
    } while (++$count <= 12);
?>
```

## for Loops

The final kind of loop statement, the `for` loop, is also the most powerful, as it combines the abilities to set up variables as you enter the loop, test for conditions while iterating loops, and modify variables after each iteration.

**Example 4-33** shows how to write the multiplication table program with a `for` loop.

*Example 4-33. Outputting the 12 times table from a for loop*

---

```
<?php
for ($count = 1 ; $count <= 12 ; ++$count)
    echo "$count times 12 is " . $count * 12 . "<br>";
?>
```

See how the code has been reduced to a single `for` statement containing a single conditional statement? Here's what is going on. Each `for` statement takes three parameters:

- An initialization expression
- A condition expression
- A modification expression

These are separated by semicolons like this: `for (expr1 ; expr2 ; expr3)`. At the start of the first iteration of the loop, the initialization expression is executed. In the case of the times table code, `$count` is initialized to the value 1. Then, each time around the loop, the condition expression (in this case, `$count <= 12`) is tested, and the loop is entered only if the condition is TRUE. Finally, at the end of each iteration, the modification expression is executed. In the case of the times table code, the variable `$count` is incremented.

All this structure neatly removes any requirement to place the controls for a loop within its body, freeing it up just for the statements you want the loop to perform.

Remember to use curly braces with a `for` loop if it will contain more than one statement, as in [Example 4-34](#).

*Example 4-34. The for loop from [Example 4-33](#) with added curly braces*

---

```
<?php
for ($count = 1 ; $count <= 12 ; ++$count)
{
    echo "$count times 12 is " . $count * 12;
    echo "<br>";
}
?>
```

Let's compare when to use `for` and `while` loops. The `for` loop is explicitly designed around a single value that changes on a regular basis. Usually you have a value that increments, as when you are passed a list of user choices and want to process each choice in turn. But you can transform the variable any way you like. A more complex form of the `for` statement even lets you perform multiple operations in each of the three parameters:

```
for ($i = 1, $j = 1 ; $i + $j < 10 ; $i++ , $j++)
{
    // ...
}
```

That's complicated and not recommended for first-time users, though. The key is to distinguish commas from semicolons. The three parameters must be separated by semicolons. Within each parameter, multiple statements can be separated by commas. Thus, in the previous example, the first and third parameters each contain two statements:

```
$i = 1, $j = 1 // Initialize $i and $j
```

```
$i + $j < 10      // Terminating condition
$i++ , $j++       // Modify $i and $j at the end of each iteration
```

The main thing to take from this example is that you must separate the three parameter sections with semicolons, not commas (which should be used only to separate statements within a parameter section).

So, when is a `while` statement more appropriate than a `for` statement? When your condition doesn't depend on a simple, regular change to a variable. For instance, if you want to check for some special input or error and end the loop when it occurs, use a `while` statement.

## Breaking Out of a Loop

Just as you saw how to break out of a `switch` statement, you can also break out of a `for` loop (or any loop) using the same `break` command. This step can be necessary when, for example, one of your statements returns an error and the loop cannot continue executing safely.

One case in which this might occur is when writing a file returns an error, possibly because the disk is full (see [Example 4-35](#)).

*Example 4-35. Writing a file using a for loop with error trapping*

---

```
<?php
$fp = fopen("text.txt", 'wb');

for ($j = 0 ; $j < 100 ; ++$j)
{
    $written = fwrite($fp, "data");

    if ($written == FALSE) break;
}

fclose($fp);
?>
```

This is the most complicated piece of code that you have seen so far, but you're ready for it. We'll look into the file-handling commands in a

[Chapter 7](#), but for now all you need to know is that the first line opens the file `text.txt` for writing in binary mode, and then returns a pointer to the file in the variable `$fp`, which is used later to refer to the open file.

The loop then iterates 100 times (from 0 to 99), writing the string `data` to the file. After each write, the variable `$written` is assigned a value by the `fwrite` function representing the number of characters correctly written. But if there is an error, the `fwrite` function assigns the value `FALSE`.

The behavior of `fwrite` makes it easy for the code to check the variable `$written` to see whether it is set to `FALSE` and, if so, to break out of the loop to the following statement that closes the file.

If you are looking to improve the code, you can simplify the line:

```
if ($written == FALSE) break;
```

using the NOT operator, like this:

```
if (!$written) break;
```

In fact, the pair of inner loop statements can be shortened to a single statement:

```
if (!fwrite($fp, "data")) break;
```

In other words, you can eliminate the `$written` variable, because it existed only to check the value returned from `fwrite`. You can instead test the return value directly.

The `break` command is even more powerful than you might think, because if you have code nested more than one layer deep that you need to break out of, you can follow the `break` command with a number to indicate how many levels to break out of:

```
break 2;
```

## The continue Statement

The `continue` statement is a little like a `break` statement, except that it instructs PHP to stop processing the current iteration of the loop and move right to its next iteration. So, instead of breaking out of the whole loop, PHP exits only the current iteration.

This approach can be useful in cases where you know there is no point continuing execution within the current loop and you want to save processor cycles or prevent an error from occurring by moving right along to the next iteration of the loop. In [Example 4-36](#), a `continue` statement is used to prevent a division-by-zero error from being issued when the variable `$j` has a value of `0`.

*Example 4-36. Trapping division-by-zero errors using continue*

---

```
<?php
$j = 11;

while ($j > -10)
{
    $j--;

    if ($j == 0) continue;

    echo (10 / $j) . "<br>";
}
?>
```

For all values of `$j` between `10` and `-10`, with the exception of `0`, the result of calculating `10` divided by `$j` is displayed. But for the case of `$j` being `0`, the `continue` statement is issued and execution skips immediately to the next iteration of the loop.

## Implicit and Explicit Casting

PHP is a loosely typed language that allows you to declare a variable and its type simply by using it. It also automatically converts values from one type to another whenever required. This is called *implicit casting*.

However, at times PHP's implicit casting may not be what you want. In [Example 4-37](#), note that the inputs to the division are integers. By default, PHP converts the output to floating point so it can give the most precise value—4.66 recurring.

*Example 4-37. This expression returns a floating-point number*

---

```
<?php
    $a = 56;
    $b = 12;
    $c = $a / $b;

    echo $c;
?>
```

But what if we had wanted `$c` to be an integer instead? There are various ways we could achieve this, one of which is to force the result of `$a / $b` to be cast to an integer value using the integer cast type (`int`), like this:

```
$c = (int) ($a / $b);
```

This is called *explicit casting*. Note that in order to ensure that the value of the entire expression is cast to an integer, we place the expression within parentheses. Otherwise, only the variable `$a` would have been cast to an integer—a pointless exercise, as the division by `$b` would still have returned a floating-point number.

You can explicitly cast variables and literals to the types shown in [Table 4-6](#).

*Table 4-6. PHP's cast types*

Cast type	Description
(int) (integer)	Cast to an integer by dropping the decimal portion.
(bool) (boolean)	Cast to a Boolean.
(float) (double) (real)	Cast to a floating-point number.
(string)	Cast to a string.
(array)	Cast to an array.
(object)	Cast to an object.

#### NOTE

You can usually avoid having to use a cast by calling one of PHP's built-in functions. For example, to obtain an integer value, you could use the `intval` function. As with some other sections in this book, this section is here mainly to help you understand third-party code that you may encounter from time to time.

## PHP Dynamic Linking

Because PHP is a programming language, and the output from it can be completely different for each user, it's possible for an entire website to run from a single PHP web page. Each time the user clicks on something, the details can be sent back to the same web page, which decides what to do next according to the various cookies and/or other session details it may have stored.

But although it is possible to build an entire website this way, it's not recommended, because your source code will grow and grow and start to become unwieldy, as it has to account for every possible action a user could take.

Instead, it's much more sensible to split your website development into different parts. For example, one distinct process is signing up for a website, along with all the checking this entails to validate an email address, determine whether a username is already taken, and so on.

A second module might be one that logs users in before handing them off to the main part of your website. Then you might have a messaging module with the facility for users to leave comments, a module containing links and useful information, another to allow uploading of images, and more.

As long as you have created a way to track your user through your website by means of cookies or session variables (both of which we'll look at more closely in later chapters), you can split up your website into sensible sections of PHP code, each one self-contained, and therefore treat yourself to a much easier future, developing each new feature and maintaining old ones. If you have a team, different people can work on different modules, so that each programmer needs to learn just one part of the program thoroughly.

## Dynamic Linking in Action

One of the more popular PHP-driven applications on the web today is the blogging platform WordPress (see [Figure 4-5](#)). As a blogger or a blog reader, you might not realize it, but every major section has been given its own main PHP file, and a whole raft of generic, shared functions have been placed in separate files that are included by the main PHP pages as necessary.

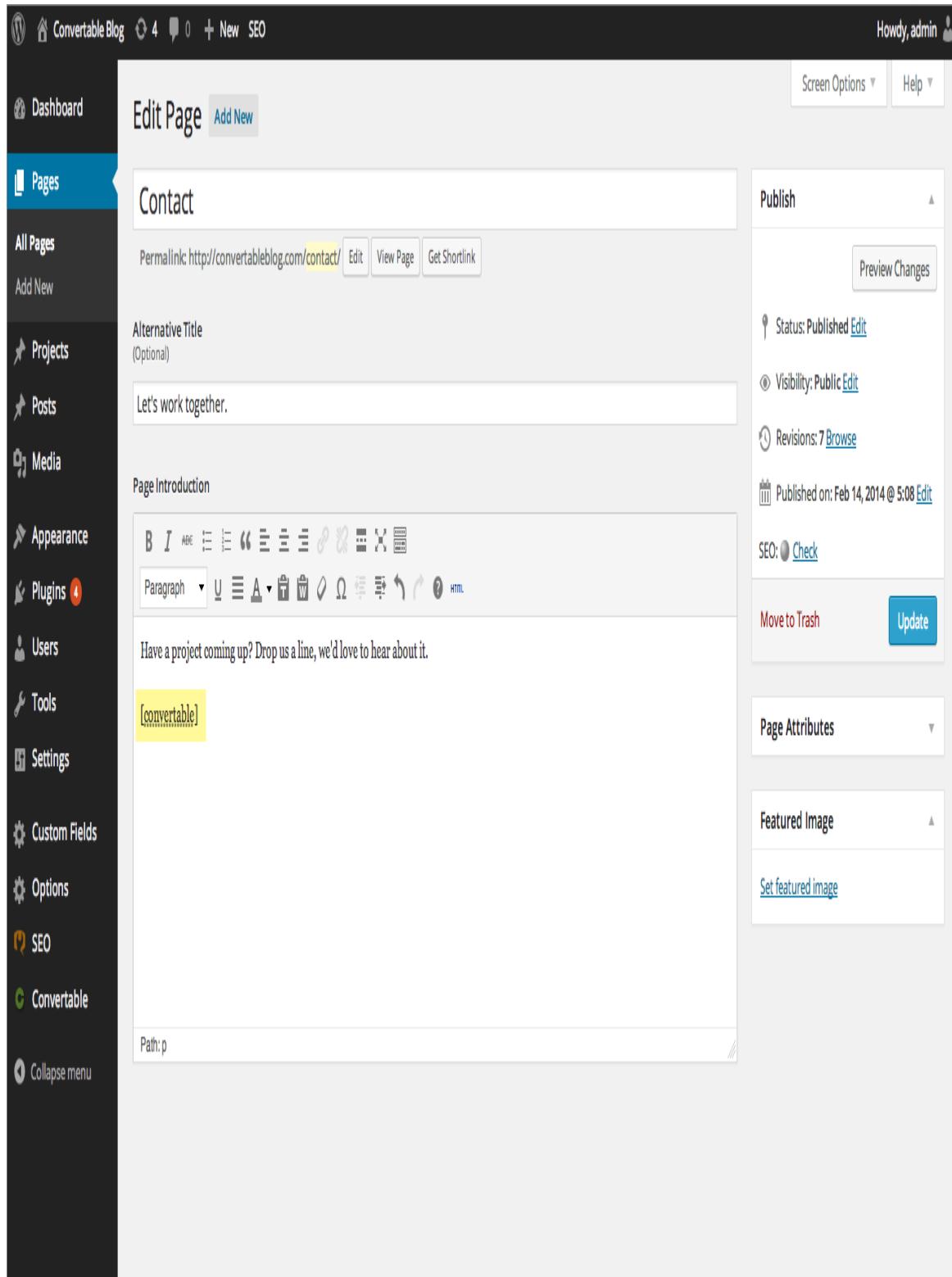


Figure 4-5. The WordPress blogging platform's dashboard

The whole platform is held together with behind-the-scenes session tracking, so that you hardly know when you are transitioning from one subsection to another. Therefore, a web developer who wants to tweak WordPress can easily find the particular file they need, modify it, and test and debug it without messing around with unconnected parts of the program. Next time you use WordPress, keep an eye on your browser's address bar, particularly if you are managing a blog, and you'll notice some of the different PHP files that it uses.

This chapter has covered quite a lot of ground, and by now you should be able to put together your own small PHP programs. But before you do, and before proceeding with the following chapter on functions and objects, you may wish to test your new knowledge by answering the following questions.

## Questions

1. What actual underlying values are represented by TRUE and FALSE?
2. What are the simplest two forms of expressions?
3. What is the difference between unary, binary, and ternary operators?
4. What is the best way to force your own operator precedence?
5. What is meant by *operator associativity*?
6. When would you use the `==` (identity) operator?
7. Name the three conditional statement types.
8. What command can you use to skip the current iteration of a loop and move on to the next one?
9. Why is a `for` loop more powerful than a `while` loop?

10. How do `if` and `while` statements interpret conditional expressions of different data types?

See “[Chapter 4 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 5. PHP Functions and Objects

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

The basic requirements of any programming language include somewhere to store data, a means of directing program flow, and a few bits and pieces such as expression evaluation, file management, and text output. PHP has all these, plus tools like `else` and `elseif` to make life easier. But even with all these in your toolkit, programming can be clumsy and tedious, especially if you have to rewrite portions of very similar code each time you need them.

That’s where functions and objects come in. As you might guess, a *function* is a set of statements that performs a particular function and—optionally—returns a value. You can pull out a section of code that you have used more than once, place it into a function, and call the function by name when you want the code.

Functions have many advantages over contiguous, inline code. For example, they:

- Involve less typing
- Reduce syntax and other programming errors
- Decrease the loading time of program files
- Decrease execution time, because each function is compiled only once, no matter how often you call it
- Accept arguments and can therefore be used for general as well as specific cases

Objects take this concept a step further. An *object* incorporates one or more functions, and the data they use, into a single structure called a *class*.

In this chapter, you'll learn all about using functions, from defining and calling them to passing arguments back and forth. With that knowledge under your belt, you'll start creating functions and using them in your own objects (where they will be referred to as *methods*).

### NOTE

It is now highly unusual (and definitely not recommended) to use any version of PHP lower than 5.4. Therefore, this chapter assumes that this release is the bare minimum version you will be working with. Generally I would recommend version 5.6, or the new version 7.0 or 7.1 (there is no version 6). You can select any of these from the AMPPS control panel, as described in [Chapter 2](#).

## PHP Functions

PHP comes with hundreds of ready-made, built-in functions, making it a very rich language. To use a function, call it by name. For example, you can see the `date` function in action here:

```
echo date("l"); // Displays the day of the week
```

The parentheses tell PHP that you're referring to a function. Otherwise, it thinks you're referring to a constant.

Functions can take any number of arguments, including zero. For example, `phpinfo`, as shown next, displays lots of information about the current installation of PHP and requires no argument. The result of calling this function can be seen in [Figure 5-1](#).

```
phpinfo();
```

A screenshot of a web browser window titled "phpinfo()". The address bar shows "localhost/test.php". The page content starts with "PHP Version 5.5.38" and features a large "php" logo. Below this is a table with various system configuration details:

System	Windows NT BOOTH 10.0 build 16299 (Windows 10) i586
Build Date	Jul 20 2016 11:08:49
Compiler	MSVC11 (Visual C++ 2012)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--disable-isapi" "--enable-debug-pack" "--without-mssql" "--without-pdo-mssql" "--without-pi3web" "--with-pdo-oci=C:\php-sdk\oracle\x86\instantclient10\sdk,shared" "--with-oci8=C:\php-sdk\oracle\x86\instantclient10\sdk,shared" "--with-oci8-11g=C:\php-sdk\oracle\x86\instantclient11\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet-shared" "--with-mcrypt=static" "--disable-static-analyze" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\WINDOWS

Figure 5-1. The output of PHP's built-in `phpinfo` function

## NOTE

The `phpinfo` function is extremely useful for obtaining information about your current PHP installation, but that information could also be very useful to potential hackers. Therefore, never leave a call to this function in any web-ready code.

Some of the built-in functions that use one or more arguments appear in **Example 5-1**.

### *Example 5-1. Three string functions*

---

```
<?php
    echo strrev(" .dlrow olleH"); // Reverse string
    echo str_repeat("Hip ", 2);   // Repeat string
    echo strtoupper("hooray!");   // String to uppercase
?>
```

This example uses three string functions to output the following text:

**Hello world. Hip Hip HOORAY!**

As you can see, the `strrev` function reversed the order of characters in the string, `str_repeat` repeated the string "Hip " twice (as required by the second argument), and `strtoupper` converted "hooray!" to uppercase.

## Defining a Function

The general syntax for a function is as follows:

```
function function_name([parameter [, ...]])
{
    // Statements
}
```

The first line of the syntax indicates the following:

- A definition starts with the word `function`.

- A name follows, which must start with a letter or underscore, followed by any number of letters, numbers, or underscores.
- The parentheses are required.
- One or more parameters, separated by commas, are optional (as indicated by the square brackets).

Function names are case-insensitive, so all of the following strings can refer to the `print` function: `PRINT`, `Print`, and `PrInT`.

The opening curly brace starts the statements that will execute when you call the function; a matching curly brace must close it. These statements may include one or more `return` statements, which force the function to cease execution and return to the calling code. If a value is attached to the `return` statement, the calling code can retrieve it, as we'll see next.

## Returning a Value

Let's take a look at a simple function to convert a person's full name to lowercase and then capitalize the first letter of each part of the name.

We've already seen an example of PHP's built-in `strtoupper` function in [Example 5-1](#). For our current function, we'll use its counterpart, `strtolower`:

```
$lowered = strtolower("aNY # of Letters and Punctuation you WANT");

echo $lowered;
```

The output of this experiment is as follows:

**any # of letters and punctuation you want**

We don't want names all lowercase, though; we want the first letter of each part of the name capitalized. (We're not going to deal with subtle cases such

as Mary-Ann or Jo-En-Lai for this example.) Luckily, PHP also provides a `ucfirst` function that sets the first character of a string to uppercase:

```
$ucfixed = ucfirst("any # of letters and punctuation you want");  
  
echo $ucfixed;
```

The output is as follows:

### **Any # of letters and punctuation you want**

Now we can do our first bit of program design: to get a word with its initial letter capitalized, we call `strtolower` on the string first, and then `ucfirst`. The way to do this is to nest a call to `strtolower` within `ucfirst`. Let's see why, because it's important to understand the order in which code is evaluated.

Say you make a simple call to the `print` function:

```
print(5-8);
```

The expression `5-8` is evaluated first, and the output is `-3`. (As you saw in the previous chapter, PHP converts the result to a string in order to display it.) If the expression contains a function, that function is evaluated first as well:

```
print(abs(5-8));
```

PHP is doing several things in executing that short statement:

1. Evaluate `5-8` to produce `-3`.
2. Use the `abs` function to turn `-3` into `3`.

3. Convert the result to a string and output it using the `print` function.

It all works because PHP evaluates each element from the inside out. The same procedure is in operation when we call the following:

```
ucfirst(strtowlower("aNY # of Letters and Punctuation you WANT"))
```

PHP passes our string to `strtowlower` and then to `ucfirst`, producing (as we've already seen when we played with the functions separately):

**Any # of letters and punctuation you want**

Now let's define a function (shown in [Example 5-2](#)) that takes three names and makes each one lowercase, with an initial capital letter.

#### Example 5-2. Cleaning up a full name

---

```
<?php
echo fix_names("WILLIAM", "henry", "gatES");

function fix_names($n1, $n2, $n3)
{
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));

    return $n1 . " " . $n2 . " " . $n3;
}
?>
```

You may well find yourself writing this type of code, because users often leave their Caps Lock key on, accidentally insert capital letters in the wrong places, and even forget capitals altogether. The output from this example is shown here:

**William Henry Gates**

## Returning an Array

We just saw a function returning a single value. There are also ways of getting multiple values from a function.

The first method is to return them within an array. As you saw in [Chapter 3](#), an array is like a bunch of variables stuck together in a row. [Example 5-3](#) shows how you can use an array to return function values.

*Example 5-3. Returning multiple values in an array*

---

```
<?php
$names = fix_names("WILLIAM", "henry", "gatES");
echo $names[0] . " " . $names[1] . " " . $names[2];

function fix_names($n1, $n2, $n3)
{
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));

    return array($n1, $n2, $n3);
}
?>
```

This method has the benefit of keeping all three names separate, rather than concatenating them into a single string, so you can refer to any user simply by first or last name without having to extract either name from the returned string.

## Passing Arguments by Reference

In PHP versions prior to 5.3, you used to be able to preface a variable with the & symbol at the time of calling a function (for example, `increment(&$myvar);`) to tell the parser to pass a reference to the variable, not the variable's value. This granted a function access to the variable (allowing different values to be written back to it).

## CAUTION

Call-time pass-by-reference was deprecated in PHP 5.3 and removed in PHP 5.4. You should therefore not use this feature other than on legacy websites, and even there you are recommended to rewrite code that passes by reference, because it will halt with a fatal error on newer versions of PHP.

However, *within* a function definition, you may continue to access arguments by reference. This concept can be hard to get your head around, so let's go back to the matchbox metaphor from [Chapter 3](#).

Imagine that, instead of taking a piece of paper out of a matchbox, reading it, copying what's on it onto another piece of paper, putting the original back, and passing the copy to a function (phew!), you could simply attach a piece of thread to the original piece of paper and pass one end of it to the function (see [Figure 5-2](#)).

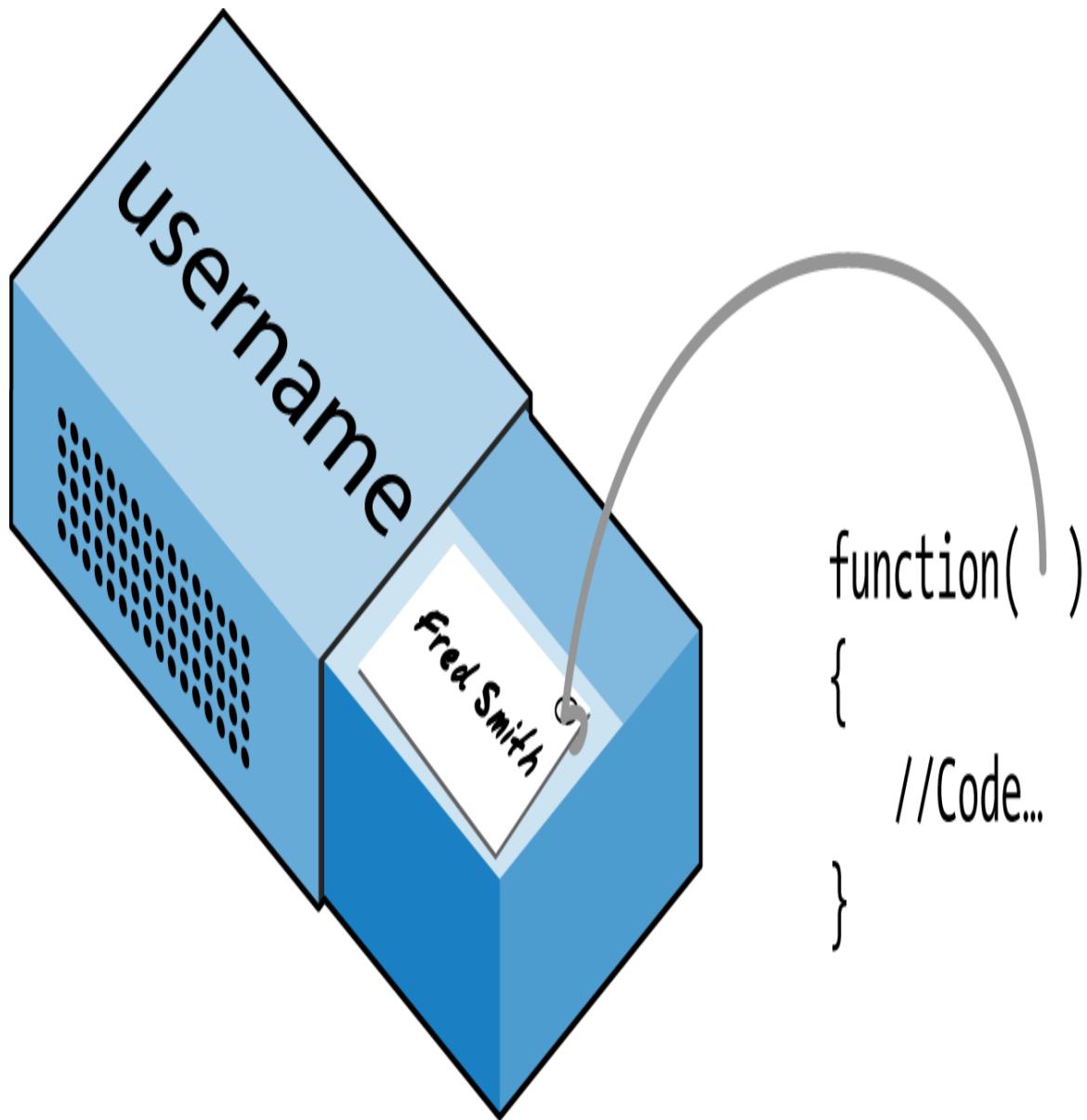


Figure 5-2. Imagining a reference as a thread attached to a variable

Now the function can follow the thread to find the data to be accessed. This avoids all the overhead of creating a copy of the variable just for the function's use. What's more, the function can now modify the variable's value.

This means you can rewrite [Example 5-3](#) to pass references to all the parameters, and then the function can modify these directly (see [Example 5-4](#)).

[Example 5-4. Passing values to a function by reference](#)

```

<?php
$a1 = "WILLIAM";
$a2 = "henry";
$a3 = "gatES";

echo $a1 . " " . $a2 . " " . $a3 . "<br>";
fix_names($a1, $a2, $a3);
echo $a1 . " " . $a2 . " " . $a3;

function fix_names(&$n1, &$n2, &$n3)
{
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));
}
?>

```

Rather than passing strings directly to the function, you first assign them to variables and print them out to see their “before” values. Then you call the function as before, but within the function definition you place a & symbol in front of each parameter to be passed by reference.

Now the variables `$n1`, `$n2`, and `$n3` are attached to “threads” that lead to the values of `$a1`, `$a2`, and `$a3`. In other words, there is one group of values, but two sets of variable names are allowed to access them.

Therefore, the function `fix_names` only has to assign new values to `$n1`, `$n2`, and `$n3` to update the values of `$a1`, `$a2`, and `$a3`. The output from this code is:

**WILLIAM henry gatES**  
**William Henry Gates**

As you see, both of the `echo` statements use only the values of `$a1`, `$a2`, and `$a3`.

## Returning Global Variables

The better way to give a function access to an externally created variable is by declaring it to have global access from within the function. The `global`

keyword followed by the variable name gives every part of your code full access to it (see [Example 5-5](#)).

#### *Example 5-5. Returning values in global variables*

---

```
<?php
    $a1 = "WILLIAM";
    $a2 = "henry";
    $a3 = "gatES";

    echo $a1 . " " . $a2 . " " . $a3 . "<br>";
    fix_names();
    echo $a1 . " " . $a2 . " " . $a3;

    function fix_names()
    {
        global $a1; $a1 = ucfirst(strtolower($a1));
        global $a2; $a2 = ucfirst(strtolower($a2));
        global $a3; $a3 = ucfirst(strtolower($a3));
    }
?>
```

Now you don't have to pass parameters to the function, and it doesn't have to accept them. Once declared, these variables retain global access and are available to the rest of your program, including its functions.

## Recap of Variable Scope

A quick reminder of what you know from [Chapter 3](#):

- *Local variables* are accessible just from the part of your code where you define them. If they're outside of a function, they can be accessed by all code outside of functions, classes, and so on. If a variable is inside a function, only that function can access the variable, and its value is lost when the function returns.
- *Global variables* are accessible from all parts of your code, whether within or outside of functions.

- *Static variables* are accessible only within the function that declared them but retain their value over multiple calls.

## Including and Requiring Files

As you progress in your use of PHP programming, you are likely to start building a library of functions that you think you will need again. You'll also probably start using libraries created by other programmers.

There's no need to copy and paste these functions into your code. You can save them in separate files and use commands to pull them in. There are two commands to perform this action: `include` and `require`.

### The `include` Statement

Using `include`, you can tell PHP to fetch a particular file and load all its contents. It's as if you pasted the included file into the current file at the insertion point. [Example 5-6](#) shows how you would include a file called `library.php`.

#### Example 5-6. Including a PHP file

---

```
<?php
    include "library.php";

    // Your code goes here
?>
```

### Using `include_once`

Each time you issue the `include` directive, it includes the requested file again, even if you've already inserted it. For instance, suppose that `library.php` contains a lot of useful functions, so you include it in your file, but you also include another library that includes `library.php`. Through nesting, you've inadvertently included `library.php` twice. This will produce error messages, because you're trying to define the same constant or

function multiple times. So, you should use `include_once` instead (see [Example 5-7](#)).

*Example 5-7. Including a PHP file only once*

---

```
<?php  
    include_once "library.php";  
  
    // Your code goes here  
?>
```

Then, any further attempts to include the same file (with `include` or `include_once`) will be ignored. To determine whether the requested file has already been executed, the absolute file path is matched after all relative paths are resolved and the file is found in your `include` path.

**NOTE**

In general, it's probably best to stick with `include_once` and ignore the basic `include` statement. That way, you will never have the problem of files being included multiple times.

## Using `require` and `require_once`

A potential problem with `include` and `include_once` is that PHP will only *attempt* to include the requested file. Program execution continues even if the file is not found.

When it is absolutely essential to include a file, `require` it. For the same reasons I gave for using `include_once`, I recommend that you generally stick with `require_once` whenever you need to `require` a file (see [Example 5-8](#)).

*Example 5-8. Requiring a PHP file only once*

---

```
<?php  
    require_once "library.php";
```

```
// Your code goes here  
?>
```

## PHP Version Compatibility

PHP is in an ongoing process of development, and there are multiple versions. If you need to check whether a particular function is available to your code, you can use the `function_exists` function, which checks all predefined and user-created functions.

**Example 5-9** checks for `array_combine`, a function specific to only some versions of PHP.

*Example 5-9. Checking for a function's existence*

---

```
<?php  
if (function_exists("array_combine"))  
{  
    echo "Function exists";  
}  
else  
{  
    echo "Function does not exist - better write our own";  
}  
?>
```

Using code such as this, you can take advantage of features in newer versions of PHP and yet still have your code run on earlier versions, as long as you replicate any features that are missing. Your functions may be slower than the built-in ones, but at least your code will be much more portable.

You can also use the `phpversion` function to determine which version of PHP your code is running on. The returned result will be similar to the following, depending on the version:

**8.0.0**

## PHP Objects

In much the same way that functions represent a huge increase in programming power over the early days of computing, where sometimes the best program navigation available was a very basic GOTO or GOSUB statement, *object-oriented programming* (OOP) takes the use of functions to a whole new level.

Once you get the hang of condensing reusable bits of code into functions, it's not that great a leap to consider bundling the functions and their data into objects.

Let's take a social networking site that has many parts. One handles all user functions—that is, code to enable new users to sign up and existing users to modify their details. In standard PHP, you might create a few functions to handle this and embed some calls to the MySQL database to keep track of all the users.

Imagine how much easier it would be to create an object to represent the current user. To do this, you could create a class, perhaps called `User`, that would contain all the code required for handling users and all the variables needed for manipulating the data within the class. Then, whenever you need to manipulate a user's data, you could simply create a new object with the `User` class.

You could treat this new object as if it were the actual user. For example, you could pass the object a name, password, and email address; ask it whether such a user already exists; and, if not, have it create a new user with those attributes. You could even have an instant messaging object, or one for managing whether two users are friends.

## Terminology

When creating a program to use objects, you need to design a composite of data and code called a *class*. Each new object based on this class is called an *instance* (or *occurrence*) of that class.

The data associated with an object is called its *properties*; the functions it uses are called *methods*. In defining a class, you supply the names of its properties and the code for its methods. See [Figure 5-3](#) for a jukebox metaphor for an object. Think of the CDs that it holds in the carousel as its properties; the method of playing them is to press buttons on the front panel. There is also a slot for inserting coins (the method used to activate the object), and the laser disc reader (the method used to retrieve the music, or properties, from the CDs).



Figure 5-3. A jukebox: a great example of a self-contained object

When you're creating objects, it is best to use *encapsulation*, or writing a class in such a way that only its methods can be used to manipulate its properties. In other words, you deny outside code direct access to its data. The methods you supply are known as the object's *interface*.

This approach makes debugging easy: you have to fix faulty code only within a class. Additionally, when you want to upgrade a program, if you have used proper encapsulation and maintained the same interface, you can simply develop new replacement classes, debug them fully, and then swap them in for the old ones. If they don't work, you can swap the old ones back in to immediately fix the problem before further debugging the new classes.

Once you have created a class, you may find that you need another class that is similar to it but not quite the same. The quick and easy thing to do is to define a new class using *inheritance*. When you do this, your new class has all the properties of the one it has inherited from. The original class is now called the *superclass*, and the new one is the *subclass* (or *derived class*).

In our jukebox example, if you invent a new jukebox that can play a video along with the music, you can inherit all the properties and methods from the original jukebox superclass and add some new properties (videos) and new methods (a movie player).

An excellent benefit of this system is that if you improve the speed or any other aspect of the superclass, its subclasses will receive the same benefit.

## Declaring a Class

Before you can use an object, you must define a class with the `class` keyword. Class definitions contain the class name (which is case-sensitive), its properties, and its methods. [Example 5-10](#) defines the class `User` with two properties, which are `$name` and `$password` (indicated by the `public` keyword—see “[Property and Method Scope](#)”). It also creates a new instance (called `$object`) of this class.

*Example 5-10. Declaring a class and examining an object*

---

```
<?php  
$object = new User;  
print_r($object);
```

```
class User
{
    public $name, $password;

    function save_user()
    {
        echo "Save User code goes here";
    }
}

?>
```

Here I have also used an invaluable function called `print_r`. It asks PHP to display information about a variable in human-readable form. (The `_r` stands for *human-readable*.) In the case of the new object `$object`, it displays the following:

```
User Object
(
    [name]    =>
    [password] =>
)
```

However, a browser compresses all the whitespace, so the output in a browser is slightly harder to read:

**User Object ( [name] => [password] => )**

In any case, the output says that `$object` is a user-defined object that has the properties `name` and `password`.

## Creating an Object

To create an object with a specified class, use the `new` keyword, like this: `$object = new Class`. Here are a couple of ways in which we could do this:

```
$object = new User;
$temp   = new User('name', 'password');
```

On the first line, we simply assign an object to the `User` class. In the second, we pass parameters to the call.

A class may require or prohibit arguments; it may also allow arguments without explicitly requiring them.

## Accessing Objects

Let's add a few lines to [Example 5-10](#) and check the results. [Example 5-11](#) extends the previous code by setting object properties and calling a method.

*Example 5-11. Creating and interacting with an object*

---

```
<?php
    $object = new User;
    print_r($object); echo "<br>";

    $object->name      = "Joe";
    $object->password  = "mypass";
    print_r($object); echo "<br>";

    $object->save_user();

class User
{
    public $name, $password;

    function save_user()
    {
        echo "Save User code goes here";
    }
}
?>
```

As you can see, the syntax for accessing an object's property is `$object->property`. Likewise, you call a method like this: `$object->method()`.

You should note that the example `property` and `method` do not have `$` signs in front of them. If you were to preface them with `$` signs, the code would not work, as it would try to reference the value inside a variable. For

example, the expression `$object->$property` would attempt to look up the value assigned to a variable named `$property` (let's say that value is the string `brown`) and then attempt to reference the property `$object->brown`. If `$property` is undefined, an attempt to reference `$object->NULL` would occur and cause an error.

When looked at using a browser's View Source facility, the output from [Example 5-11](#) is as follows:

```
User Object
(
    [name]      =>
    [password]  =>
)
User Object
(
    [name]      => Joe
    [password]  => mypass
)
Save User code goes here
```

Again, `print_r` shows its utility by providing the contents of `$object` before and after property assignment. From now on, I'll omit `print_r` statements, but if you are working along with this book on your development server, you can put some in to see exactly what is happening.

You can also see that the code in the method `save_user` was executed via the call to that method. It printed the string reminding us to create some code.

#### NOTE

You can place functions and class definitions anywhere in your code, before or after statements that use them. Generally, though, it is considered good practice to place them toward the end of a file.

## Cloning Objects

Once you have created an object, it is passed by reference when you pass it as a parameter. In the matchbox metaphor, this is like keeping several threads attached to an object stored in a matchbox, so that you can follow any attached thread to access it.

In other words, making object assignments does not copy objects in their entirety. You'll see how this works in [Example 5-12](#), where we define a very simple `User` class with no methods and only the property `name`.

### Example 5-12. Copying an object?

---

```
<?php
    $object1      = new User();
    $object1->name = "Alice";
    $object2      = $object1;
    $object2->name = "Amy";

    echo "object1 name = " . $object1->name . "<br>";
    echo "object2 name = " . $object2->name;

    class User
    {
        public $name;
    }
?>
```

Here, we first create the object `$object1` and assign the value `Alice` to the `name` property. Then we create `$object2`, assigning it the value of `$object1`, and assign the value `Amy` just to the `name` property of `$object2` —or so we might think. But this code outputs the following:

```
object1 name = Amy
object2 name = Amy
```

What has happened? Both `$object1` and `$object2` refer to the *same* object, so changing the `name` property of `$object2` to `Amy` also sets that property for `$object1`.

To avoid this confusion, you can use the `clone` operator, which creates a new instance of the class and copies the property values from the original

instance to the new instance. [Example 5-13](#) illustrates this usage.

### Example 5-13. Cloning an object

---

```
<?php
    $object1      = new User();
    $object1->name = "Alice";
    $object2      = clone $object1;
    $object2->name = "Amy";

    echo "object1 name = " . $object1->name . "<br>";
    echo "object2 name = " . $object2->name;

    class User
    {
        public $name;
    }
?>
```

Voilà! The output from this code is what we initially wanted:

```
object1 name = Alice
object2 name = Amy
```

## Constructors

When creating a new object, you can pass a list of arguments to the class being called. These are passed to a special method within the class, called the *constructor*, which initializes various properties.

To do this you use the function name `__construct` (that is, `construct` preceded by two underscore characters), as in [Example 5-14](#).

### Example 5-14. Creating a constructor method

---

```
<?php
    class User
    {
        function __construct($param1, $param2)
        {
            // Constructor statements go here
```

```
    public $username = "Guest";
}
?>
```

## Constructors

You also have the ability to create *destructor* methods. This ability is useful when code has made the last reference to an object or when a script reaches the end. [Example 5-15](#) shows how to create a destructor method. The destructor can do clean-up such as releasing a connection to a database or some other resource that you reserved within the class. Because you reserved the resource within the class, you have to release it here, or it will stick around indefinitely. Many system-wide problems are caused by programs reserving resources and forgetting to release them.

[Example 5-15. Creating a destructor method](#)

---

```
<?php
class User
{
    function __destruct()
    {
        // Destructor code goes here
    }
}
?>
```

## Writing Methods

As you have seen, declaring a method is similar to declaring a function, but there are a few differences. For example, method names beginning with a double underscore (\_) are reserved, and you should not create any of this form.

You also have access to a special variable called `$this`, which can be used to access the current object's properties. To see how it works, take a look at [Example 5-16](#), which contains a different method from the `User` class definition called `get_password`.

### Example 5-16. Using the variable \$this in a method

---

```
<?php
class User
{
    public $name, $password;

    function get_password()
    {
        return $this->password;
    }
}
?>
```

`get_password` uses the `$this` variable to access the current object and then return the value of that object's `password` property. Note how the preceding `$` of the property `$password` is omitted when we use the `->` operator. Leaving the `$` in place is a typical error you may run into, particularly when you first use this feature.

Here's how you would use the class defined in [Example 5-16](#):

```
$object      = new User;
)object->password = "secret";

echo $object->get_password();
```

This code prints the password `secret`.

## Declaring Properties

It is not necessary to explicitly declare properties within classes, as they can be implicitly defined when first used. To illustrate this, in [Example 5-17](#) the class `User` has no properties and no methods but is legal code.

### Example 5-17. Defining a property implicitly

---

```
<?php
```

```

$object1      = new User();
$object1->name = "Alice";

echo $object1->name;

class User {}
?>

```

This code correctly outputs the string `Alice` without a problem, because PHP implicitly declares the property `$object1->name` for you. But this kind of programming can lead to bugs that are infuriatingly difficult to discover, because `name` was declared from outside the class.

To help yourself and anyone else who will maintain your code, I advise that you get into the habit of always declaring your properties explicitly within classes. You'll be glad you did.

Also, when you declare a property within a class, you may assign a default value to it. The value you use must be a constant and not the result of a function or expression. [Example 5-18](#) shows a few valid and invalid assignments.

#### *Example 5-18. Valid and invalid property declarations*

---

```

<?php
class Test
{
    public $name  = "Paul Smith"; // Valid
    public $age   = 42;           // Valid
    public $time  = time();       // Invalid - calls a function
    public $score = $level * 2;   // Invalid - uses an expression
}
?>

```

## Declaring Constants

In the same way that you can create a global constant with the `define` function, you can define constants inside classes. The generally accepted practice is to use uppercase letters to make them stand out, as in [Example 5-19](#).

### Example 5-19. Defining constants within a class

---

```
<?php
    Translate::lookup();

class Translate
{
    const ENGLISH = 0;
    const SPANISH = 1;
    const FRENCH = 2;
    const GERMAN = 3;
    // ...

    static function lookup()
    {
        echo self::SPANISH;
    }
}

?>
```

You can reference constants directly, using the `self` keyword and double colon operator. Note that this code calls the class directly, using the double colon operator at line 1, without creating an instance of it first. As you would expect, the value printed when you run this code is 1.

Remember that once you define a constant, you can't change it.

## Property and Method Scope

PHP provides three keywords for controlling the scope of properties and methods (*members*):

*public*

Public members can be referenced anywhere, including by other classes and instances of the object. This is the default when variables are declared with the `var` or `public` keywords, or when a variable is implicitly declared the first time it is used. The keywords `var` and `public` are interchangeable because, although deprecated, `var` is

retained for compatibility with previous versions of PHP. Methods are assumed to be `public` by default.

### `protected`

These members can be referenced only by the object's class methods and those of any subclasses.

### `private`

These members can be referenced only by methods within the same class—not by subclasses.

Here's how to decide which you need to use:

- Use `public` when outside code *should* access this member and extending classes *should* also inherit it.
- Use `protected` when outside code *should not* access this member but extending classes *should* inherit it.
- Use `private` when outside code *should not* access this member and extending classes also *should not* inherit it.

**Example 5-20** illustrates the use of these keywords.

#### Example 5-20. Changing property and method scope

```
<?php
class Example
{
    var $name = "Michael"; // Same as public but deprecated
    public $age = 23;      // Public property
    protected $usercount; // Protected property

    private function admin() // Private method
    {
        // Admin code goes here
    }
}
?>
```

## Static Methods

You can define a method as `static`, which means that it is called on a class, not on an object. A static method has no access to any object properties and is created and accessed as in [Example 5-21](#).

### Example 5-21. Creating and accessing a static method

---

```
<?php
User::pwd_string();

class User
{
    static function pwd_string()
    {
        echo "Please enter your password";
    }
}

?>
```

Note how we call the class itself, along with the static method, using a double colon (also known as the *scope resolution operator*), not `->`. Static functions are useful for performing actions relating to the class itself, but not to specific instances of the class. You can see another example of a static method in [Example 5-19](#).

### NOTE

If you try to access `$this->property`, or other object properties from within a static function, you will receive an error message.

## Static Properties

Most data and methods apply to instances of a class. For example, in a `User` class, you want to do such things as set a particular user's password or check when the user has been registered. These facts and operations apply separately to each user and therefore use instance-specific properties and methods.

But occasionally you'll want to maintain data about a whole class. For instance, to report how many users are registered, you will store a variable that applies to the whole `User` class. PHP provides static properties and methods for such data.

As shown briefly in [Example 5-21](#), declaring members of a class `static` makes them accessible without an instantiation of the class. A property declared `static` cannot be directly accessed within an instance of a class, but a static method can.

[Example 5-22](#) defines a class called `Test` with a static property and a public method.

#### *Example 5-22. Defining a class with a static property*

---

```
<?php
$temp = new Test();

echo "Test A: " . Test::$static_property . "<br>";
echo "Test B: " . $temp->get_sp() . "<br>";
echo "Test C: " . $temp->static_property . "<br>";

class Test
{
    static $static_property = "I'm static";

    function get_sp()
    {
        return self::$static_property;
    }
}
?>
```

When you run this code, it returns the following output:

**Test A: I'm static**

**Test B: I'm static**

**Notice: Undefined property: Test::\$static\_property**

**Test C:**

This example shows that the property `$static_property` could be directly referenced from the class itself via the double colon operator in Test A. Also, Test B could obtain its value by calling the `get_sp` method of the object `$temp`, created from class `Test`. But Test C failed, because the static property `$static_property` was not accessible to the object `$temp`.

Note how the method `get_sp` accesses `$static_property` using the keyword `self`. This is how a static property or constant can be directly accessed within a class.

## Inheritance

Once you have written a class, you can derive subclasses from it. This can save lots of painstaking code rewriting: you can take a class similar to the one you need to write, extend it to a subclass, and just modify the parts that are different. You achieve this using the `extends` keyword.

In [Example 5-23](#), the class `Subscriber` is declared a subclass of `User` by means of the `extends` keyword.

*Example 5-23. Inheriting and extending a class*

---

```
<?php
    $object      = new Subscriber;
    $object->name   = "Fred";
    $object->password = "pword";
    $object->phone   = "012 345 6789";
    $object->email    = "fred@bloggs.com";
    $object->display();

    class User
    {
        public $name, $password;

        function save_user()
        {
            echo "Save User code goes here";
        }
    }

    class Subscriber extends User
```

```

{
    public $phone, $email;

    function display()
    {
        echo "Name: " . $this->name . "<br>";
        echo "Pass: " . $this->password . "<br>";
        echo "Phone: " . $this->phone . "<br>";
        echo "Email: " . $this->email;
    }
}
?>

```

The original `User` class has two properties, `$name` and `$password`, and a method to save the current user to the database. `Subscriber` extends this class by adding an additional two properties, `$phone` and `$email`, and includes a method of displaying the properties of the current object using the variable `$this`, which refers to the current values of the object being accessed. The output from this code is as follows:

**Name: Fred**  
**Pass: pword**  
**Phone: 012 345 6789**  
**Email: fred@bloggs.com**

## The parent keyword

If you write a method in a subclass with the same name as one in its parent class, its statements will override those of the parent class. Sometimes this is not the behavior you want, and you need to access the parent's method. To do this, you can use the `parent` operator, as in [Example 5-24](#).

*Example 5-24. Overriding a method and using the parent operator*

---

```

<?php
$object = new Son;
$object->test();
$object->test2();

class Dad
{

```

```

function test()
{
    echo "[Class Dad] I am your Father<br>";
}
}

class Son extends Dad
{
    function test()
    {
        echo "[Class Son] I am Luke<br>";
    }

    function test2()
    {
        parent::test();
    }
}
?>

```

This code creates a class called `Dad` and a subclass called `Son` that inherits its properties and methods, and then overrides the method `test`. Therefore, when line 2 calls the method `test`, the new method is executed. The only way to execute the overridden `test` method in the `Dad` class is to use the `parent` operator, as shown in function `test2` of class `Son`. The code outputs the following:

**[Class Son] I am Luke**  
**[Class Dad] I am your Father**

If you wish to ensure that your code calls a method from the current class, you can use the `self` keyword, like this:

```
self::method();
```

## Subclass constructors

When you extend a class and declare your own constructor, you should be aware that PHP will not automatically call the constructor method of the parent class. If you want to be certain that all initialization code is executed, subclasses should always call the parent constructors, as in [Example 5-25](#).

### Example 5-25. Calling the parent class constructor

---

```
<?php
$object = new Tiger();

echo "Tigers have...<br>";
echo "Fur: " . $object->fur . "<br>";
echo "Stripes: " . $object->stripes;

class Wildcat
{
    public $fur; // Wildcats have fur

    function __construct()
    {
        $this->fur = "TRUE";
    }
}

class Tiger extends Wildcat
{
    public $stripes; // Tigers have stripes

    function __construct()
    {
        parent::__construct(); // Call parent constructor first
        $this->stripes = "TRUE";
    }
}
?>
```

This example takes advantage of inheritance in the typical manner. The `Wildcat` class has created the property `$fur`, which we'd like to reuse, so we create the `Tiger` class to inherit `$fur` and additionally create another property, `$stripes`. To verify that both constructors have been called, the program outputs the following:

**Tigers have...**  
**Fur: TRUE**  
**Stripes: TRUE**

## Final methods

When you wish to prevent a subclass from overriding a superclass method, you can use the `final` keyword. [Example 5-26](#) shows how.

*Example 5-26. Creating a final method*

---

```
<?php
class User
{
    final function copyright()
    {
        echo "This class was written by Joe Smith";
    }
}
?>
```

Once you have digested the contents of this chapter, you should have a strong feel for what PHP can do for you. You should be able to use functions with ease and, if you wish, write object-oriented code. In [Chapter 6](#), we'll finish off our initial exploration of PHP by looking at the workings of PHP arrays.

## Questions

1. What is the main benefit of using a function?
2. How many values can a function return?
3. What is the difference between accessing a variable by name and by reference?
4. What is the meaning of *scope* in PHP?
5. How can you incorporate one PHP file within another?
6. How is an object different from a function?
7. How do you create a new object in PHP?

8. What syntax would you use to create a subclass from an existing one?
9. How can you cause an object to be initialized when you create it?
10. Why is it a good idea to explicitly declare properties within a class?

See “[Chapter 5 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 6. PHP Arrays

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In [Chapter 3](#), I gave a very brief introduction to PHP’s arrays—just enough for a little taste of their power. In this chapter, I’ll show you many more things that you can do with arrays, some of which—if you have ever used a strongly typed language such as C—may surprise you with their elegance and simplicity.

Arrays are an example of what has made PHP so popular. Not only do they remove the tedium of writing code to deal with complicated data structures, but they also provide numerous ways to access data while remaining amazingly fast.

## Basic Access

We’ve already looked at arrays as if they were clusters of matchboxes glued together. Another way to think of an array is like a string of beads, with the beads representing variables that can be numbers, strings, or even other arrays. They are like bead strings because each element has its own location

and (with the exception of the first and last ones) each has other elements on either side.

Some arrays are referenced by numeric indexes; others allow alphanumeric identifiers. Built-in functions let you sort them, add or remove sections, and walk through them to handle each item through a special kind of loop. And by placing one or more arrays inside another, you can create arrays of two, three, or any number of dimensions.

## Numerically Indexed Arrays

Let's assume that you've been tasked with creating a simple website for a local office supply company and you're currently working on the section devoted to paper. One way to manage the various items of stock in this category would be to place them in a numeric array. You can see the simplest way of doing so in [Example 6-1](#).

### Example 6-1. Adding items to an array

---

```
<?php
$paper[] = "Copier";
$paper[] = "Inkjet";
$paper[] = "Laser";
$paper[] = "Photo";

print_r($paper);
?>
```

In this example, each time you assign a value to the array `$paper`, the first empty location within that array is used to store the value, and a pointer internal to PHP is incremented to point to the next free location, ready for future insertions. The familiar `print_r` function (which prints out the contents of a variable, array, or object) is used to verify that the array has been correctly populated. It prints out the following:

```
Array
(
    [0] => Copier
    [1] => Inkjet
```

```
[2] => Laser  
[3] => Photo  
)
```

The previous code could also have been written as shown in [Example 6-2](#), where the exact location of each item within the array is specified. But, as you can see, that approach requires extra typing and makes your code harder to maintain if you want to insert supplies into or remove them from the array. So, unless you wish to specify a different order, it's usually better to simply let PHP handle the actual location numbers.

*Example 6-2. Adding items to an array using explicit locations*

---

```
<?php  
$paper[0] = "Copier";  
$paper[1] = "Inkjet";  
$paper[2] = "Laser";  
$paper[3] = "Photo";  
  
print_r($paper);  
?>
```

The output from these examples is identical, but you are not likely to use `print_r` in a developed website, so [Example 6-3](#) shows how you might print out the various types of paper the website offers using a `for` loop.

*Example 6-3. Adding items to an array and retrieving them*

---

```
<?php  
$paper[] = "Copier";  
$paper[] = "Inkjet";  
$paper[] = "Laser";  
$paper[] = "Photo";  
  
for ($j = 0 ; $j < 4 ; ++$j)  
    echo "$j: $paper[$j]<br>";  
?>
```

This example prints out the following:

- 0: Copier**
- 1: Inkjet**
- 2: Laser**
- 3: Photo**

So far, you've seen a couple of ways in which you can add items to an array and one way of referencing them. PHP offers many more, which I'll get to shortly. But first, we'll look at another type of array.

## Associative Arrays

Keeping track of array elements by index works just fine, but can require extra work in terms of remembering which number refers to which product. It can also make code hard for other programmers to follow.

This is where associative arrays come into their own. Using them, you can reference the items in an array by name rather than by number. [Example 6-4](#) expands on the previous code by giving each element in the array an identifying name and a longer, more explanatory string value.

*Example 6-4. Adding items to an associative array and retrieving them*

---

```
<?php
$paper['copier'] = "Copier & Multipurpose";
$paper['inkjet'] = "Inkjet Printer";
$paper['laser'] = "Laser Printer";
$paper['photo'] = "Photographic Paper";

echo $paper['laser'];
?>
```

In place of a number (which doesn't convey any useful information, aside from the position of the item in the array), each item now has a unique name that you can use to reference it elsewhere, as with the `echo` statement—which simply prints out `Laser Printer`. The names (`copier`, `inkjet`, and so on) are called *indexes* or *keys*, and the items assigned to them (such as `Laser Printer`) are called *values*.

This very powerful feature of PHP is often used when you are extracting information from XML and HTML. For example, an HTML parser such as those used by a search engine could place all the elements of a web page into an associative array whose names reflect the page's structure:

```
$html['title'] = "My web page";
$html['body'] = "... body of web page ...";
```

The program would also probably break down all the links found within a page into another array, and all the headings and subheadings into another. When you use associative rather than numeric arrays, the code to refer to all of these items is easy to write and debug.

## Assignment Using the array Keyword

So far, you've seen how to assign values to arrays by just adding new items one at a time. Whether you specify keys, specify numeric identifiers, or let PHP assign numeric identifiers implicitly, this is a long-winded approach. A more compact and faster assignment method uses the `array` keyword.

**Example 6-5** shows both a numeric and an associative array assigned using this method.

*Example 6-5. Adding items to an array using the array keyword*

---

```
<?php
$p1 = array("Copier", "Inkjet", "Laser", "Photo");

echo "p1 element: " . $p1[2] . "<br>";

$p2 = array('copier' => "Copier & Multipurpose",
            'inkjet' => "Inkjet Printer",
            'laser' => "Laser Printer",
            'photo' => "Photographic Paper");

echo "p2 element: " . $p2['inkjet'] . "<br>";
?>
```

The first half of this snippet assigns the old, shortened product descriptions to the array `$p1`. There are four items, so they will occupy slots 0 through 3. Therefore, the `echo` statement prints out the following:

### **p1 element: Laser**

The second half assigns associative identifiers and accompanying longer product descriptions to the array `$p2` using the format `key => value`. The use of `=>` is similar to the regular `=` assignment operator, except that you are assigning a value to an *index* and not to a *variable*. The index is then inextricably linked with that value, unless it is assigned a new value. The `echo` command therefore prints out this:

### **p2 element: Inkjet Printer**

You can verify that `$p1` and `$p2` are different types of array, because both of the following commands, when appended to the code, will cause an `Undefined index` or `Undefined offset` error, as the array identifier for each is incorrect:

```
echo $p1['inkjet']; // Undefined index
echo $p2[3]; // Undefined offset
```

## **The foreach...as Loop**

The creators of PHP have gone to great lengths to make the language easy to use. So, not content with the loop structures already provided, they added another one especially for arrays: the `foreach...as` loop. Using it, you can step through all the items in an array, one at a time, and do something with them.

The process starts with the first item and ends with the last one, so you don't even have to know how many items there are in an array. [Example 6-6](#) shows how `foreach...as` can be used to rewrite [Example 6-3](#).

*Example 6-6. Walking through a numeric array using foreach...as*

---

```

<?php
$paper = array("Copier", "Inkjet", "Laser", "Photo");
$j = 0;

foreach($paper as $item)
{
    echo "$j: $item<br>";
    ++$j;
}
?>

```

When PHP encounters a **foreach** statement, it takes the first item of the array and places it in the variable following the **as** keyword; and each time control flow returns to the **foreach**, the next array element is placed in the **as** keyword. In this case, the variable **\$item** is set to each of the four values in turn in the array **\$paper**. Once all values have been used, execution of the loop ends. The output from this code is exactly the same as in [Example 6-3](#).

Now let's see how **foreach** works with an associative array by taking a look at [Example 6-7](#), which is a rewrite of the second half of [Example 6-5](#).

#### *Example 6-7. Walking through an associative array using foreach...as*

```

<?php
$paper = array('copier' => "Copier & Multipurpose",
               'inkjet' => "Inkjet Printer",
               'laser'   => "Laser Printer",
               'photo'   => "Photographic Paper");

foreach($paper as $item => $description)
    echo "$item: $description<br>";
?>

```

Remember that associative arrays do not require numeric indexes, so the variable **\$j** is not used in this example. Instead, each item of the array **\$paper** is fed into the key/value pair of variables **\$item** and **\$description**, from which they are printed out. The displayed result of this code is as follows:

**copier: Copier & Multipurpose**

**inkjet: Inkjet Printer**

**laser: Laser Printer**

**photo: Photographic Paper**

Prior to version 7.2 of PHP, as an alternative syntax to `foreach...as`, you could use the `list` function in conjunction with the `each` function.

However, `each` was then deprecated and therefore is not recommended for use because it may be removed in a future version. This is a bit of a nightmare for PHP programmers with legacy code to update, especially as the `each` function is extremely useful. Therefore, I have written a replacement for `each` called `myEach`, which works identically and will allow you to easily update old code, as in [Example 6-8](#).

[Example 6-8. Walking through an associative array using myEach and list](#)

```
<?php
$paper = array('copier' => "Copier & Multipurpose",
               'inkjet' => "Inkjet Printer",
               'laser'   => "Laser Printer",
               'photo'   => "Photographic Paper");

while (list($item, $description) = myEach($paper))
    echo "$item: $description<br>";

function myEach(&$array) // Replacement for the deprecated each function
{
    $key    = key($array);
    $result = ($key === null) ? false :
              [$key, current($array), 'key', 'value' => current($array)];
    next($array);
    return $result;
}
?>
```

In this example, a `while` loop is set up and will continue looping until the `myEach` function (equivalent to the old PHP `each` function) returns a value of FALSE. The `myEach` function acts like `foreach` in that it returns an array containing a key/value pair from the array `$paper` and then moves its built-

in pointer to the next pair in that array. When there are no more pairs to return, `myEach` returns FALSE.

The `list` function takes an array as its argument (in this case, the key/value pair returned by the function `myEach`) and then assigns the values of the array to the variables listed within parentheses.

You can see how `list` works a little more clearly in [Example 6-9](#), where an array is created out of the two strings `Alice` and `Bob` and then passed to the `list` function, which assigns those strings as values to the variables `$a` and `$b`.

#### *Example 6-9. Using the list function*

---

```
<?php
    list($a, $b) = array('Alice', 'Bob');
    echo "a=$a b=$b";
?>
```

The output from this code is as follows:

**a=Alice b=Bob**

So, you can take your pick when walking through arrays. Use `foreach...as` to create a loop that extracts values to the variable following the `as`, or use the `myEach` function and create your own looping system.

## Multidimensional Arrays

A simple design feature in PHP's array syntax makes it possible to create arrays of more than one dimension. In fact, they can be as many dimensions as you like (although it's a rare application that goes further than three).

That feature is the ability to include an entire array as a part of another one, and to be able to keep doing so, just like the old rhyme: "Big fleas have little fleas upon their backs to bite 'em. Little fleas have lesser fleas, add flea, ad infinitum."

Let's look at how this works by taking the associative array in the previous example and extending it; see [Example 6-10](#).

*Example 6-10. Creating a multidimensional associative array*

---

```
<?php
$products = array(
    'paper' => array(
        'copier' => "Copier & Multipurpose",
        'inkjet' => "Inkjet Printer",
        'laser' => "Laser Printer",
        'photo' => "Photographic Paper"),
    'pens' => array(
        'ball' => "Ball Point",
        'hilite' => "Highlighters",
        'marker' => "Markers"),
    'misc' => array(
        'tape' => "Sticky Tape",
        'glue' => "Adhesives",
        'clips' => "Paperclips"
    )
);

echo "<pre>";
foreach($products as $section => $items)
    foreach($items as $key => $value)
        echo "$section:\t$key\t($value)<br>";

echo "</pre>";
?>
```

To make things clearer now that the code is starting to grow, I've renamed some of the elements. For example, because the previous array `$paper` is now just a subsection of a larger array, the main array is now called `$products`. Within this array, there are three items—`paper`, `pens`, and `misc`—each of which contains another array with key/value pairs.

If necessary, these subarrays could have contained even further arrays. For example, under `ball` there might be many different types and colors of ballpoint pens available in the online store. But for now, I've restricted the code to a depth of just two.

Once the array data has been assigned, I use a pair of nested `foreach...as` loops to print out the various values. The outer loop extracts the main sections from the top level of the array, and the inner loop extracts the key/value pairs for the categories within each section.

As long as you remember that each level of the array works the same way (it's a key/value pair), you can easily write code to access any element at any level.

The `echo` statement makes use of the PHP escape character `\t`, which outputs a tab. Although tabs are not normally significant to the web browser, I let them be used for layout by using the `<pre>...</pre>` tags, which tell the web browser to format the text as preformatted and monospaced, and *not* to ignore whitespace characters such as tabs and line feeds. The output from this code looks like the following:

```
paper: copier  (Copier & Multipurpose)
paper: inkjet   (Inkjet Printer)
paper: laser    (Laser Printer)
paper: photo     (Photographic Paper)
pens: ball      (Ball Point)
pens: hilite    (Highlighters)
pens: marker    (Markers)
misc: tape      (Sticky Tape)
misc: glue      (Adhesives)
misc: clips     (Paperclips)
```

You can directly access a particular element of the array by using square brackets:

```
echo $products['misc']['glue'];
```

This outputs the value **Adhesives**.

You can also create numeric multidimensional arrays that are accessed directly by indexes rather than by alphanumeric identifiers. **Example 6-11** creates the board for a chess game with the pieces in their starting positions.

*Example 6-11. Creating a multidimensional numeric array*

---

```
<?php
$chessboard = array(
    array('r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'),
    array('p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'),
    array('_', '_', '_', '_', '_', '_', '_', '_'),
    array('P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'),
    array('R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R')
);

echo "<pre>";

foreach($chessboard as $row)
{
    foreach ($row as $piece)
        echo "$piece ";

    echo "<br>";
}

echo "</pre>";
?>
```

In this example, the lowercase letters represent black pieces, and the uppercase white. The key is `r` = rook, `n` = knight, `b` = bishop, `k` = king, `q` = queen, and `p` = pawn. Again, a pair of nested `foreach...as` loops walks through the array and displays its contents. The outer loop processes each row into the variable `$row`, which itself is an array, because the `$chessboard` array uses a subarray for each row. This loop has two statements within it, so curly braces enclose them.

The inner loop then processes each square in a row, outputting the character (`$piece`) stored in it, followed by a space (to square up the printout). This loop has a single statement, so curly braces are not required to enclose it. The `<pre>` and `</pre>` tags ensure that the output displays correctly, like this:

```
r n b q k b n r  
p p p p p p p p
```

```
P P P P P P P P  
R N B Q K B N R
```

You can also directly access any element within this array by using square brackets:

```
echo $chessboard[7][3];
```

This statement outputs the uppercase letter `Q`, the eighth element down and the fourth along (remember that array indexes start at 0, not 1).

## Using Array Functions

You've already seen the `list` and `each` functions, but PHP comes with numerous other functions for handling arrays. You can find the full list in the [documentation](#). However, some of these functions are so fundamental that it's worth taking the time to look at them here.

### `is_array`

Arrays and variables share the same namespace. This means that you cannot have a string variable called `$fred` and an array also called `$fred`. If

you're in doubt and your code needs to check whether a variable is an array, you can use the `is_array` function, like this:

```
echo (is_array($fred)) ? "Is an array" : "Is not an array";
```

Note that if `$fred` has not yet been assigned a value, an **Undefined variable** message will be generated.

## count

Although the `each` function and `foreach...as` loop structure are excellent ways to walk through an array's contents, sometimes you need to know exactly how many elements there are in your array, particularly if you will be referencing them directly. To count all the elements in the top level of an array, use a command such as this:

```
echo count($fred);
```

Should you wish to know how many elements there are altogether in a multidimensional array, you can use a statement such as the following:

```
echo count($fred, 1);
```

The second parameter is optional and sets the mode to use. It should be either 0 to limit counting to only the top level, or 1 to force recursive counting of all subarray elements too.

## sort

Sorting is so common that PHP provides a built-in function for it. In its simplest form, you would use it like this:

```
sort($fred);
```

It is important to remember that, unlike some other functions, `sort` will act directly on the supplied array rather than returning a new array of sorted elements. It returns TRUE on success and FALSE on error and also supports a few flags—the main two that you might wish to use force items to be sorted either numerically or as strings, like this:

```
sort($fred, SORT_NUMERIC);
sort($fred, SORT_STRING);
```

You can also sort an array in reverse order using the `rsort` function, like this:

```
rsort($fred, SORT_NUMERIC);
rsort($fred, SORT_STRING);
```

## shuffle

There may be times when you need the elements of an array to be put in random order, such as when you're creating a game of playing cards:

```
shuffle($cards);
```

Like `sort`, `shuffle` acts directly on the supplied array and returns TRUE on success or FALSE on error.

## explode

`explode` is a very useful function with which you can take a string containing several items separated by a single character (or string of characters) and then place each of these items into an array. One handy example is to split up a sentence into an array containing all its words, as in [Example 6-12](#).

*Example 6-12. Exploding a string into an array using spaces*

---

```
<?php
$temp = explode(' ', "This is a sentence with seven words");
print_r($temp);
?>
```

This example prints out the following (on a single line when viewed in a browser):

```
Array
(
    [0] => This
    [1] => is
    [2] => a
    [3] => sentence
    [4] => with
    [5] => seven
    [6] => words
)
```

The first parameter, the delimiter, need not be a space or even a single character. [Example 6-13](#) shows a slight variation.

*Example 6-13. Exploding a string delimited with \*\*\* into an array*

---

```
<?php
$temp = explode('***', "A***sentence***with***asterisks");
print_r($temp);
?>
```

The code in [Example 6-13](#) prints out the following:

```
Array
(
    [0] => A
    [1] => sentence
    [2] => with
    [3] => asterisks
)
```

## extract

Sometimes it can be convenient to turn the key/value pairs from an array into PHP variables. One such time might be when you are processing the `$_GET` or `$_POST` variables sent to a PHP script by a form.

When a form is submitted over the web, the web server unpacks the variables into a global array for the PHP script. If the variables were sent using the GET method, they will be placed in an associative array called `$_GET`; if they were sent using POST, they will be placed in an associative array called `$_POST`.

You could, of course, walk through such associative arrays in the manner shown in the examples so far. However, sometimes you just want to store the values sent into variables for later use. In this case, you can have PHP do the job automatically:

```
extract($_GET);
```

So, if the query string parameter `q` is sent to a PHP script along with the associated value `Hi there`, a new variable called `$q` will be created and assigned that value.

Be careful with this approach, though, because if any extracted variables conflict with ones that you have already defined, your existing values will be overwritten. To avoid this possibility, you can use one of the many additional parameters available to this function, like this:

```
extract($_GET, EXTR_PREFIX_ALL, 'fromget');
```

In this case, all the new variables will begin with the given prefix string followed by an underscore, so `$q` will become `$fromget_q`. I strongly recommend that you use this version of the function when handling the `$_GET` and `$_POST` arrays, or any other array whose keys could be controlled by the user, because malicious users could submit keys chosen

deliberately to overwrite commonly used variable names and compromise your website.

## compact

At times you may want to use **compact**, the inverse of **extract**, to create an array from variables and their values. [Example 6-14](#) shows how you might use this function.

### *Example 6-14. Using the compact function*

---

```
<?php
    $fname      = "Doctor";
    $sname      = "Who";
    $planet     = "Gallifrey";
    $system     = "Gridlock";
    $constellation = "Kasterborous";

    $contact = compact('fname', 'sname', 'planet', 'system', 'constellation');

    print_r($contact);
?>
```

The result of running [Example 6-14](#) is as follows:

```
Array
(
    [fname] => Doctor
    [sname] => Who
    [planet] => Gallifrey
    [system] => Gridlock
    [constellation] => Kasterborous
)
```

Note how **compact** requires the variable names to be supplied in quotes, not preceded by a \$ symbol. This is because **compact** is looking for a list of variable names, not their values.

Another use of this function is for debugging, when you wish to quickly view several variables and their values, as in [Example 6-15](#).

### *Example 6-15. Using compact to help with debugging*

---

```
<?php
$j      = 23;
$temp   = "Hello";
$address = "1 Old Street";
$age    = 61;

print_r(compact(explode(' ', 'j temp address age')));
?>
```

This works by using the `explode` function to extract all the words from the string into an array, which is then passed to the `compact` function, which in turn returns an array to `print_r`, which finally shows its contents.

If you copy and paste the `print_r` line of code, you only need to alter the variables named there for a quick printout of a group of variables' values. In this example, the output is shown here:

```
Array
(
    [j] => 23
    [temp] => Hello
    [address] => 1 Old Street
    [age] => 61
)
```

## **reset**

When the `foreach...as` construct or the `each` function walks through an array, it keeps an internal PHP pointer that makes a note of which element of the array it should return next. If your code ever needs to return to the start of an array, you can issue `reset`, which also returns the value of that element. Examples of how to use this function are as follows:

```
reset($fred);           // Throw away return value
$item = reset($fred); // Keep first element of the array in $item
```

## end

As with `reset`, you can move PHP's internal array pointer to the final element in an array using the `end` function, which also returns the value of the element, and can be used as in these examples:

```
end($fred);
$item = end($fred);
```

This chapter concludes your basic introduction to PHP, and you should now be able to write quite complex programs using the skills you have learned. In the next chapter, we'll look at using PHP for common, practical tasks.

## Questions

1. What is the difference between a numeric and an associative array?
2. What is the main benefit of the `array` keyword?
3. What is the difference between `foreach` and `each`?
4. How can you create a multidimensional array?
5. How can you determine the number of elements in an array?
6. What is the purpose of the `explode` function?
7. How can you set PHP's internal pointer into an array back to the first element of the array?

See “[Chapter 6 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 7. Practical PHP

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

The previous chapters went over the elements of the PHP language. This chapter builds on your new programming skills to teach you how to perform some common but important practical tasks. You will learn the best ways to handle strings in order to achieve clear and concise code that displays in web browsers exactly how you want it to, including advanced date and time management. You’ll also find out how to create and otherwise modify files, including those uploaded by users.

## Using `printf`

You’ve already seen the `print` and `echo` functions, which simply output text to the browser. But a much more powerful function, `printf`, controls the format of the output by letting you put special formatting characters in a string. For each formatting character, `printf` expects you to pass an argument that it will display using that format. For instance, the following example uses the `%d` conversion specifier to display the value 3 in decimal:

```
printf("There are %d items in your basket", 3);
```

If you replace the %d with %b, the value 3 will be displayed in binary (11). **Table 7-1** shows the conversion specifiers supported.

*Table 7-1. The printf conversion specifiers*

Specifier	Conversion action on argument arg	Example (for an arg of 123)
%	Display a % character (no arg required)	%
b	Display arg as a binary integer	1111011
c	Display ASCII character for arg	{
d	Display arg as a signed decimal integer	123
e	Display arg using scientific notation	1.23000e+2
f	Display arg as floating point	123.000000
o	Display arg as an octal integer	173
s	Display arg as a string	123
u	Display arg as an unsigned decimal	123
x	Display arg in lowercase hexadecimal	7b
X	Display arg in uppercase hexadecimal	7B

You can have as many specifiers as you like in a `printf` function, as long as you pass a matching number of arguments and as long as each specifier is prefaced by a % symbol. Therefore, the following code is valid, and will output "My name is Simon. I'm 33 years old, which is 21 in hexadecimal":

```
printf("My name is %s. I'm %d years old, which is %X in hexadecimal",
      'Simon', 33, 33);
```

If you leave out any arguments, you will receive a parse error informing you that a right bracket, ), was unexpectedly encountered.

A more practical example of `printf` sets colors in HTML using decimal values. For example, suppose you know you want a color that has a triplet value of 65 red, 127 green, and 245 blue, but don't want to convert this to hexadecimal yourself. Here's an easy solution:

```
printf("<span style='color:#%X%X%X'>Hello</span>", 65, 127, 245);
```

Check the format of the color specification between the apostrophes (' ') carefully. First comes the pound, or hash, sign (#) expected by the color specification. Then come three %X format specifiers, one for each of your numbers. The resulting output from this command is as follows:

```
<span style='color:#417FF5'>Hello</span>
```

Usually, you'll find it convenient to use variables or expressions as arguments to `printf`. For instance, if you stored values for your colors in the three variables \$r, \$g, and \$b, you could create a darker color with this:

```
printf("<span style='color:#%X%X%X'>Hello</span>", $r-20, $g-20, $b-20);
```

## Precision Setting

Not only can you specify a conversion type, but you can also set the precision of the displayed result. For example, amounts of currency are usually displayed with only two digits of precision. However, after a calculation, a value may have a greater precision than this, such as 123.42 / 12, which results in 10.285. To ensure that such values are correctly stored internally, but displayed with only two digits of precision, you can insert the string ".2" between the % symbol and the conversion specifier:

```
printf("The result is: %.2f", 123.42 / 12);
```

The output from this command is as follows:

**The result is \$10.29**

But you actually have even more control than that, because you can also specify whether to pad output with either zeros or spaces by prefacing the specifier with certain values. [Example 7-1](#) shows four possible combinations.

### *Example 7-1. Precision setting*

---

```
<?php
echo "<pre>"; // Enables viewing of the spaces

// Pad to 15 spaces
printf("The result is $%15f\n", 123.42 / 12);

// Pad to 15 spaces, fill with zeros
printf("The result is $%015f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision
printf("The result is $%15.2f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision, fill with zeros
printf("The result is $%015.2f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision, fill with # symbol
printf("The result is $%'#15.2f\n", 123.42 / 12);
?>
```

The output from this example looks like this:

```
The result is $      10.285000
The result is $00000010.285000
The result is $      10.29
The result is $000000000010.29
The result is $#####10.29
```

The way it works is simple if you go from right to left (see [Table 7-2](#)). Notice that:

- The rightmost character is the conversion specifier: in this case, `f` for floating point.
- Just before the conversion specifier, if there is a period and a number together, then the precision of the output is specified as the value of the number.
- Regardless of whether there's a precision specifier, if there is a number, then that represents the number of characters to which the output should be padded. In the previous example, this is 15 characters. If the output is already equal to or greater than the padding length, then this argument is ignored.
- The leftmost parameter allowed after the `%` symbol is a `0`, which is ignored unless a padding value has been set, in which case the output is padded with zeros instead of spaces. If a pad character other than zero or a space is required, you can use any one of your choice as long as you preface it with a single quotation mark, like this: `'#`.
- On the left is the `%` symbol, which starts the conversion.

*Table 7-2. Conversion specifier components*

Start conversion	Pad character	Number of pad characters	Display precision	Conversion specifier	Example
<code>%</code>		15		<code>f</code>	<code>10.285000</code>
<code>%</code>	<code>0</code>	15	.2	<code>f</code>	<code>00000000000 10.29</code>
<code>%</code>	<code>'#</code>	15	.4	<code>f</code>	<code>#####1 0.2850</code>

## String Padding

You can also pad strings to required lengths (as you can with numbers), select different padding characters, and even choose between left and right justification. [Example 7-2](#) shows various examples.

## Example 7-2. String padding

---

```
<?php
echo "<pre>"; // Enables viewing of the spaces

$h = 'Rasmus';

printf("[%s]\n", $h); // Standard string output
printf("[%12s]\n", $h); // Right justify with spaces to width 12
printf(["%-12s"]\n, $h); // Left justify with spaces
printf(["%012s"]\n, $h); // Pad with zeros
printf(["%'#12s"]\n\n", $h); // Use the custom padding character '#'

$d = 'Rasmus Lerdorf'; // The original creator of PHP

printf(["%12.8s"]\n, $d); // Right justify, cutoff of 8 characters
printf(["%-12.12s"]\n, $d); // Left justify, cutoff of 12 characters
printf(["%- '@12.10s"]\n, $d); // Left justify, pad with '@', cutoff 10 chars
?>
```

Note how for purposes of layout in a web page, I've used the `<pre>` HTML tag to preserve all the spaces and the `\n` newline character after each of the lines to be displayed. The output from this example is as follows:

```
[Rasmus]
[      Rasmus]
[Rasmus      ]
[000000Rasmus]
[#####Rasmus]

[   Rasmus L]
[Rasmus Lero]
[Rasmus Ler@@]
```

When you specify a padding value, strings of a length equal to or greater than that value will be ignored, *unless* a cutoff value is given that shortens the strings back to less than the padding value.

Table 7-3 shows the components available to string conversion specifiers.

*Table 7-3. String conversion specifier components*

Start conversi on	Left/right t justify	Padding character	Number of pad characters	Cu tof f	Conversio n specifier	(using "Rasmus")
%				s		[Rasmus]
%	-		10	s		[Rasmus ]
%		'#	8	.4	s	[#####Rasm]

## Using sprintf

Often, you don't want to output the result of a conversion but need it to use elsewhere in your code. This is where the `sprintf` function comes in. With it, you can send the output to another variable rather than to the browser.

You might use it to make a conversion, as in the following example, which returns the hexadecimal string value for the RGB color group 65, 127, 245 in `$hexstring`:

```
$hexstring = sprintf("%X%X%X", 65, 127, 245);
```

Or you may wish to store output ready to display later on:

```
$out = sprintf("The result is: %.2f", 123.42 / 12);
echo $out;
```

## Date and Time Functions

To keep track of the date and time, PHP uses standard Unix timestamps, which are simply the number of seconds since the start of January 1, 1970. To determine the current timestamp, you can use the `time` function:

```
echo time();
```

Because the value is stored as seconds, to obtain the timestamp for this time next week, you would use the following, which adds 7 days  $\times$  24 hours  $\times$  60 minutes  $\times$  60 seconds to the returned value:

```
echo time() + 7 * 24 * 60 * 60;
```

If you wish to create a timestamp for a given date, you can use the `mktime` function. Its output is the timestamp **1669852800** for the first second of the first minute of the first hour of the first day of December in the year 2022:

```
echo mktime(0, 0, 0, 12, 1, 2022);
```

The parameters to pass are, in order from left to right:

- The number of the hour (0–23)
- The number of the minute (0–59)
- The number of seconds (0–59)
- The number of the month (1–12)
- The number of the day (1–31)
- The year (1970–2038, or 1901–2038 with PHP 5.1.0+ on 32-bit signed systems)

## NOTE

You may ask why you are limited to the years 1970 through 2038. Well, it's because the original developers of Unix chose the start of the year 1970 as the base date that no programmer should need to go before!

Luckily, as of version 5.1.0, PHP supports systems using a signed 32-bit integer for the timestamp, and dates from 1901 to 2038 are allowed on them. However, that introduces a problem even worse than the original one, because the Unix designers also decided that nobody would still be using Unix after about 70 years or so and therefore believed they could get away with storing the timestamp as a 32-bit value—which will run out on January 19, 2038!

This will create what has come to be known as the Y2K38 bug (much like the millennium bug, which was caused by storing years as two-digit values, and which also had to be fixed). PHP introduced the `DateTime` class in version 5.2 to overcome this issue, but it will work only on 64-bit architecture, which most computers will be these days (but do check before you use it).

To display the date, use the `date` function, which supports a plethora of formatting options enabling you to display the date any way you wish. The format is as follows:

```
date($format, $timestamp);
```

The parameter `$format` should be a string containing formatting specifiers as detailed in [Table 7-4](#), and `$timestamp` should be a Unix timestamp. For the complete list of specifiers, please see the [documentation](#). The following command will output the current date and time in the format "Thursday February 17th, 2022 - 1:38pm":

```
echo date("l F jS, Y - g:ia", time());
```

*Table 7-4. The major date function format specifiers*

Format Description	Returned value
<b>Day specifiers</b>	
d Day of month, two digits, with leading zeros	01 to 31
D Day of the week, three letters	Mon to Sun
j Day of month, no leading zeros	1 to 31
l Day of week, full names	Sunday to Saturday
N Day of week, numeric, Monday to Sunday	1 to 7
S Suffix for day of month (useful with specifier j) st, nd, rd, or th	
w Day of week, numeric, Sunday to Saturday	0 to 6
z Day of year	0 to 365
<b>Week specifier</b>	
W Week number of year	01 to 52
<b>Month specifiers</b>	
F Month name	January to December
m Month number with leading zeros	01 to 12
M Month name, three letters	Jan to Dec
n Month number, no leading zeros	1 to 12
t Number of days in given month	28 to 31
<b>Year specifiers</b>	
L Leap year	1 = Yes, 0 = No
y Year, 2 digits	00 to 99
Y Year, 4 digits	0000 to 9999
<b>Time specifiers</b>	
a Before or after midday, lowercase	am or pm
A Before or after midday, uppercase	AM or PM
g Hour of day, 12-hour format, no leading zeros	1 to 12
G Hour of day, 24-hour format, no leading zeros	0 to 23
h Hour of day, 12-hour format, with leading zeros	01 to 12
H Hour of day, 24-hour format, with leading zeros	00 to 23

Format	Description	Returned value
i	Minutes, with leading zeros	00 to 59
s	Seconds, with leading zeros	00 to 59

## Date Constants

There are a number of useful constants that you can use with the `date` command to return the date in specific formats. For example, `date(DATE_RSS)` returns the current date and time in the valid format for an RSS feed. Some of the more commonly used constants are as follows:

### *DATE\_ATOM*

This is the format for Atom feeds. The PHP format is "Y-m-d\TH:i:sP" and example output is "2025-05-15T12:00:00+00:00".

### *DATE\_COOKIE*

This is the format for cookies set from a web server or JavaScript. The PHP format is "l, d-M-y H:i:s T" and example output is "Thursday, 15-May-25 12:00:00 UTC".

### *DATE\_RSS*

This is the format for RSS feeds. The PHP format is "D, d M Y H:i:s O" and example output is "Thu, 15 May 2025 12:00:00 UTC".

### *DATE\_W3C*

This is the format for the World Wide Web Consortium. The PHP format is "Y-m-d\TH:i:sP" and example output is "2025-05-15T12:00:00+00:00".

The complete list can be found in the [documentation](#).

## Using checkdate

You've seen how to display a valid date in a variety of formats. But how can you check whether a user has submitted a valid date to your program? The answer is to pass the month, day, and year to the `checkdate` function, which returns a value of TRUE if the date is valid, or FALSE if it is not.

For example, if September 31 of any year is input, it will always be an invalid date. **Example 7-3** shows code that you could use for this. As it stands, it will find the given date invalid.

*Example 7-3. Checking for the validity of a date*

---

```
<?php
$month = 9;      // September (only has 30 days)
$day   = 31;     // 31st
$year  = 2025;  // 2025

if (checkdate($month, $day, $year)) echo "Date is valid";
else echo "Date is invalid";
?>
```

## File Handling

Powerful as it is, MySQL is not the only (or necessarily the best) way to store all data on a web server. Sometimes it can be quicker and more convenient to directly access files on the hard disk. Cases in which you might need to do this are when modifying images such as uploaded user avatars, or with log files that you wish to process.

First, though, a note about file naming: if you are writing code that may be used on various PHP installations, there is no way of knowing whether these systems are case-sensitive. For example, Windows and macOS filenames are not case-sensitive, but Linux and Unix ones are. Therefore, you should always assume that the system is case-sensitive and stick to a convention such as all-lowercase filenames.

## Checking Whether a File Exists

To determine whether a file already exists, you can use the `file_exists` function, which returns either TRUE or FALSE and is used like this:

```
if (file_exists("testfile.txt")) echo "File exists";
```

## Creating a File

At this point, `testfile.txt` doesn't exist, so let's create it and write a few lines to it. Type [Example 7-4](#) and save it as `testfile.php`.

*Example 7-4. Creating a simple text file*

---

```
<?php // testfile.php
$fh = fopen("testfile.txt", 'w') or die("Failed to create file");

$text = <<< _END
Line 1
Line 2
Line 3
_END;

fwrite($fh, $text) or die("Could not write to file");
fclose($fh);
echo "File 'testfile.txt' written successfully";
?>
```

Should a program call the `die` function, the open file will be automatically closed as part of terminating the program.

When you run this in a browser, all being well, you will receive the message `File 'testfile.txt' written successfully`. If you receive an error message, your hard disk may be full or, more likely, you may not have permission to create or write to the file, in which case you should modify the attributes of the destination folder according to your operating system. Otherwise, the file `testfile.txt` should now be residing in the same folder in which you saved the `testfile.php` program. Try opening the file in a text or program editor—the contents will look like this:

```
Line 1  
Line 2  
Line 3
```

This simple example shows the sequence that all file handling takes:

1. Always start by opening the file. You do this through a call to `fopen`.
2. Then you can call other functions; here we write to the file (`fwrite`), but you can also read from an existing file (`fclose` or `fgets`) and do other things.
3. Finish by closing the file (`fclose`). Although the program does this for you when it ends, you should clean up by closing the file when you're finished.

Every open file requires a file resource so that PHP can access and manage it. The preceding example sets the variable `$fh` (which I chose to stand for *file handle*) to the value returned by the `fopen` function. Thereafter, each file-handling function that accesses the opened file, such as `fwrite` or `fclose`, must be passed `$fh` as a parameter to identify the file being accessed. Don't worry about the content of the `$fh` variable; it's a number PHP uses to refer to internal information about the file—you just pass the variable to other functions.

Upon failure, `FALSE` will be returned by `fopen`. The previous example shows a simple way to capture and respond to the failure: it calls the `die` function to end the program and give the user an error message. A web application would never abort in this crude way (you would create a web page with an error message instead), but this is fine for our testing purposes.

Notice the second parameter to the `fopen` call. It is simply the character `w`, which tells the function to open the file for writing. The function creates the file if it doesn't already exist. Be careful when playing around with these functions: if the file already exists, the `w` mode parameter causes the `fopen` call to delete the old contents (even if you don't write anything new!).

There are several different mode parameters that can be used here, as detailed in [Table 7-5](#). The modes that include a + symbol are further explained in the section “[Updating Files](#)”.

*Table 7-5. The supported fopen modes*

Mode	Action	Description
'r'	Read from file's beginning	Open for reading only; place the file pointer at the beginning of the file. Return FALSE if the file doesn't already exist.
'r+'	Read from file's beginning and allow writing	Open for reading and writing; place the file pointer at the beginning of the file. Return FALSE if the file doesn't already exist.
'w'	Write from file's beginning and truncate file	Open for writing only; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.
'w+'	Write from file's beginning, truncate file, and allow reading	Open for reading and writing; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.
'a'	Append to file's end	Open for writing only; place the file pointer at the end of the file. If the file doesn't exist, attempt to create it.
'a+'	Append to file's end and allow reading	Open for reading and writing; place the file pointer at the end of the file. If the file doesn't exist, attempt to create it.

## Reading from Files

The easiest way to read from a text file is to grab a whole line through `fgets` (think of the final `s` as standing for *string*), as in [Example 7-5](#).

Example 7-5. Reading a file with fgets

```
<?php
$fh = fopen("testfile.txt", 'r') or
die("File does not exist or you lack permission to open it");

$line = fgets($fh);
fclose($fh);
```

```
echo $line;  
?>
```

If you created the file as shown in [Example 7-4](#), you'll get the first line:

### Line 1

You can retrieve multiple lines or portions of lines through the `fread` function, as in [Example 7-6](#).

#### Example 7-6. Reading a file with fread

---

```
<?php  
$fh = fopen("testfile.txt", 'r') or  
die("File does not exist or you lack permission to open it");  
  
$text = fread($fh, 3);  
fclose($fh);  
echo $text;  
?>
```

I've requested three characters in the `fread` call, so the program displays this:

### Lin

The `fread` function is commonly used with binary data. If you use it on text data that spans more than one line, remember to count newline characters.

## Copying Files

Let's try out the PHP `copy` function to create a clone of `testfile.txt`. Type [Example 7-7](#), save it as `copyfile.php`, and then call up the program in your browser.

#### Example 7-7. Copying a file

---

```
<?php // copyfile.php
    copy('testfile.txt', 'testfile2.txt') or die("Could not copy file");
    echo "File successfully copied to 'testfile2.txt'";
?>
```

If you check your folder again, you'll see that you now have the new file *testfile2.txt* in it. By the way, if you don't want your programs to exit on a failed copy attempt, you could try the alternate syntax in [Example 7-8](#). This uses the ! (NOT) operator as a quick and easy shorthand. Placed in front of an expression, it applies the NOT operator to it, so the equivalent statement here in English would begin “If not able to copy...”

*Example 7-8. Alternate syntax for copying a file*

---

```
<?php // copyfile2.php
    if (!copy('testfile.txt', 'testfile2.txt')) echo "Could not copy file";
    else echo "File successfully copied to 'testfile2.txt'";
?>
```

## Moving a File

To move a file, rename it with the `rename` function, as in [Example 7-9](#).

*Example 7-9. Moving a file*

---

```
<?php // movefile.php
    if (!rename('testfile2.txt', 'testfile2.new'))
        echo "Could not rename file";
    else echo "File successfully renamed to 'testfile2.new'";
?>
```

You can use the `rename` function on directories, too. To avoid any warning messages if the original file doesn't exist, you can call the `file_exists` function first to check.

## Deleting a File

Deleting a file is just a matter of using the `unlink` function to remove it from the filesystem, as in [Example 7-10](#).

#### *Example 7-10. Deleting a file*

---

```
<?php // deletefile.php
if (!unlink('testfile2.new')) echo "Could not delete file";
else echo "File 'testfile2.new' successfully deleted";
?>
```

#### **WARNING**

Whenever you access files on your hard disk directly, you must also always ensure that it is impossible for your filesystem to be compromised. For example, if you are deleting a file based on user input, you must make absolutely certain it is a file that can be safely deleted and that the user is allowed to delete it.

As with moving a file, a warning message will be displayed if the file doesn't exist, which you can avoid by using `file_exists` to first check for its existence before calling `unlink`.

## Updating Files

Often, you will want to add more data to a saved file, which you can do in many ways. You can use one of the append write modes (see [Table 7-5](#)), or you can simply open a file for reading and writing with one of the other modes that supports writing, and move the file pointer to the correct place within the file that you wish to write to or read from.

The *file pointer* is the position within a file at which the next file access will take place, whether it's a read or a write. It is not the same as the *file handle* (as stored in the variable `$fh` in [Example 7-4](#)), which contains details about the file being accessed.

You can see this in action by typing [Example 7-11](#) and saving it as `update.php`. Then call it up in your browser.

### Example 7-11. Updating a file

```
<?php // update.php
$fh = fopen("testfile.txt", 'r+') or die("Failed to open file");
$text = fgets($fh);

fseek($fh, 0, SEEK_END);
fwrite($fh, "$text") or die("Could not write to file");
fclose($fh);

echo "File 'testfile.txt' successfully updated";
?>
```

This program opens *testfile.txt* for both reading and writing by setting the mode with '*r+*', which puts the file pointer right at the start. It then uses the `fgets` function to read in a single line from the file (up to the first line feed). After that, the `fseek` function is called to move the file pointer right to the file end, at which point the line of text that was extracted from the start of the file (stored in `$text`) is then appended to the file's end and the file is closed. The resulting file now looks like this:

```
Line 1
Line 2
Line 3
Line 1
```

The first line has successfully been copied and then appended to the file's end.

As used here, in addition to the `$fh` file handle, the `fseek` function was passed two other parameters, `0` and `SEEK_END`. `SEEK_END` tells the function to move the file pointer to the end of the file, and `0` tells it how many positions it should then be moved backward from that point. In the case of [Example 7-11](#), a value of `0` is used because the pointer is required to remain at the file's end.

There are two other seek options available to the `fseek` function: `SEEK_SET` and `SEEK_CUR`. The `SEEK_SET` option tells the function to set the file pointer

to the exact position given by the preceding parameter. Thus, the following example moves the file pointer to position 18:

```
fseek($fh, 18, SEEK_SET);
```

SEEK\_CUR sets the file pointer to the current position *plus* the value of the given offset. Therefore, if the file pointer is currently at position 18, the following call will move it to position 23:

```
fseek($fh, 5, SEEK_CUR);
```

Although this is not recommended unless you have very specific reasons for it, it is even possible to use text files such as this (but with fixed line lengths) as simple flat file databases. Your program can then use `fseek` to move back and forth within such a file to retrieve, update, and add new records. You can also delete records by overwriting them with zero characters, and so on.

## Locking Files for Multiple Accesses

Web programs are often called by many users at the same time. If more than one person tries to write to a file simultaneously, it can become corrupted. And if one person writes to it while another is reading from it, the file is all right, but the person reading it can get odd results. To handle simultaneous users, you must use the file-locking `flock` function. This function queues up all other requests to access a file until your program releases the lock. So, whenever your programs use write access on files that may be accessed concurrently by multiple users, you should also add file locking to them, as in [Example 7-12](#), which is an updated version of [Example 7-11](#).

*Example 7-12. Updating a file with file locking*

---

```
<?php  
$fh    = fopen("testfile.txt", 'r+') or die("Failed to open file");
```

```

$text = fgets($fh);

if (flock($fh, LOCK_EX))
{
    fseek($fh, 0, SEEK_END);
    fwrite($fh, "$text") or die("Could not write to file");
    flock($fh, LOCK_UN);
}

fclose($fh);
echo "File 'testfile.txt' successfully updated";
?>

```

There is a trick to file locking to preserve the best possible response time for your website visitors: perform it directly before a change you make to a file, and then unlock it immediately afterward. Having a file locked for any longer than this will slow down your application unnecessarily. This is why the calls to `flock` in [Example 7-12](#) are directly before and after the `fwrite` call.

The first call to `flock` sets an exclusive file lock on the file referred to by `$fh` using the `LOCK_EX` parameter:

```
flock($fh, LOCK_EX);
```

From this point onward, no other processes can write to (or even read from) the file until you release the lock by using the `LOCK_UN` parameter, like this:

```
flock($fh, LOCK_UN);
```

As soon as the lock is released, other processes are again allowed access to the file. This is one reason why you should reseek to the point you wish to access in a file each time you need to read or write data—another process could have changed the file since the last access.

However, did you notice that the call to request an exclusive lock is nested as part of an `if` statement? This is because `flock` is not supported on all

systems; thus, it is wise to check whether you successfully secured a lock, just in case one could not be obtained.

Something else you must consider is that `flock` is what is known as an *advisory* lock. This means that it locks out only other processes that call the function. If you have any code that goes right in and modifies files without implementing `flock` file locking, it will always override the locking and could wreak havoc on your files.

By the way, implementing file locking and then accidentally leaving it out in one section of code can lead to an extremely hard-to-locate bug.

### WARNING

`flock` will not work on NFS and many other networked filesystems. Also, when using a multithreaded server like ISAPI, you may not be able to rely on `flock` to protect files against other PHP scripts running in parallel threads of the same server instance. Additionally, `flock` is not supported on any system using the old FAT filesystem, such as older versions of Windows.

If in doubt, you can try making a quick lock on a test file at the start of a program to see whether you can obtain a lock on the file. Don't forget to unlock it (and maybe delete it if not needed) after checking.

Also remember that any call to the `die` function automatically unlocks a lock and closes the file as part of ending the program.

## Reading an Entire File

A handy function for reading in an entire file without having to use file handles is `file_get_contents`. It's very easy to use, as you can see in [Example 7-13](#).

### Example 7-13. Using `file_get_contents`

```
<?php
    echo "<pre>"; // Enables display of line feeds
    echo file_get_contents("testfile.txt");
    echo "</pre>"; // Terminates <pre> tag
?>
```

But the function is actually a lot more useful than that, because you can also use it to fetch a file from a server across the internet, as in [Example 7-14](#), which requests the HTML from the O'Reilly home page, and then displays it as if the user had surfed to the page itself. The result will be similar to [Figure 7-1](#).

*Example 7-14. Grabbing the O'Reilly home page*

---

```
<?php  
    echo file_get_contents("http://oreilly.com");  
?>
```

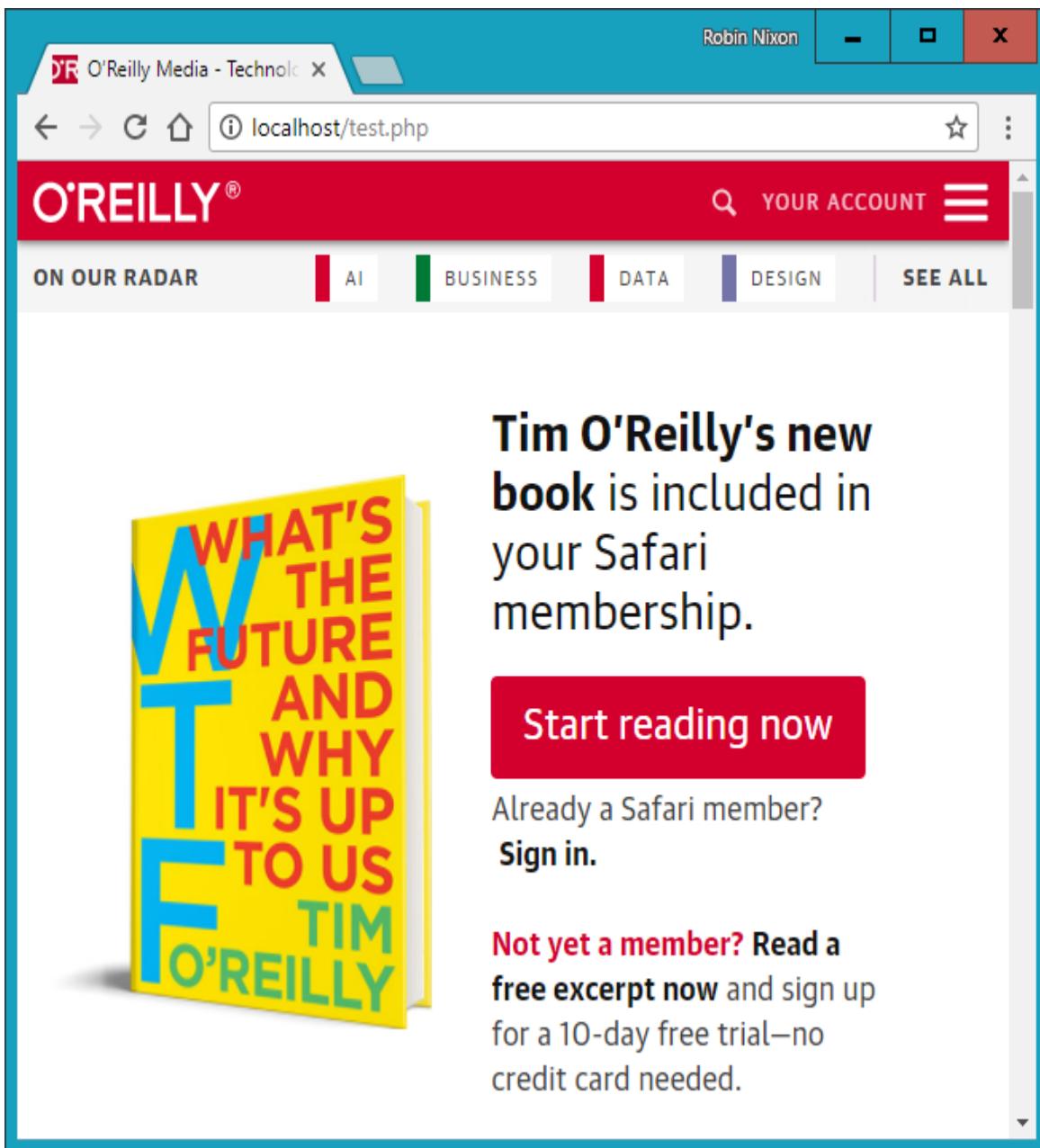


Figure 7-1. The O'Reilly home page grabbed with `file_get_contents`

## Uploading Files

Uploading files to a web server is a subject that seems daunting to many people, but it actually couldn't be much easier. All you need to do to upload a file from a form is choose a special type of encoding called `multipart/form-data`, and your browser will handle the rest. To see how this works, type the program in [Example 7-15](#) and save it as `upload.php`.

When you run it, you'll see a form in your browser that lets you upload a file of your choice.

Example 7-15. Image uploader upload.php

---

```
<?php // upload.php
echo <<<_END
<html><head><title>PHP Form Upload</title></head><body>
<form method='post' action='upload.php' enctype='multipart/form-data'>
Select File: <input type='file' name='filename' size='10'>
<input type='submit' value='Upload'>
</form>
_END;

if ($_FILES)
{
    $name = $_FILES['filename']['name'];
    move_uploaded_file($_FILES['filename']['tmp_name'], $name);
    echo "Uploaded image '$name'<br><img src='$name'>";
}

echo "</body></html>";
?>
```

Let's examine this program a section at a time. The first line of the multiline `echo` statement starts an HTML document, displays the title, and then starts the document's body.

Next we come to the form, which selects the POST method of form submission, sets the target for posted data to the program *upload.php* (the program itself), and tells the web browser that the data posted should be encoded via the content type of `multipart/form-data`.

With the form set up, the next lines display the prompt `Select File:` and then request two inputs. The first request is for a file; it uses an input type of `file`, a name of `filename`, and an input field with a width of 10 characters. The second requested input is just a submit button that is given the label `Upload` (which replaces the default button text of `Submit Query`). And then the form is closed.

This short program shows a common technique in web programming in which a single program is called twice: once when the user first visits a page, and again when the user presses the submit button.

The PHP code to receive the uploaded data is fairly simple, because all uploaded files are placed into the associative system array `$_FILES`. Therefore, a quick check to see whether `$_FILES` contains anything is sufficient to determine whether the user has uploaded a file. This is done with the statement `if ( $_FILES )`.

The first time the user visits the page, before uploading a file, `$_FILES` is empty, so the program skips this block of code. When the user uploads a file, the program runs again and discovers an element in the `$_FILES` array.

Once the program realizes that a file was uploaded, the actual name, as read from the uploading computer, is retrieved and placed into the variable `$name`. Now all that's necessary is to move the uploaded file from the temporary location in which PHP stored it to a more permanent one. We do this using the `move_uploaded_file` function, passing it the original name of the file, with which it is saved to the current directory.

Finally, the uploaded image is displayed within an `IMG` tag, and the result should look like [Figure 7-2](#).

### WARNING

If you run this program and receive a warning message such as `Permission denied` for the `move_uploaded_file` function call, then you may not have the correct permissions set for the folder in which the program is running.

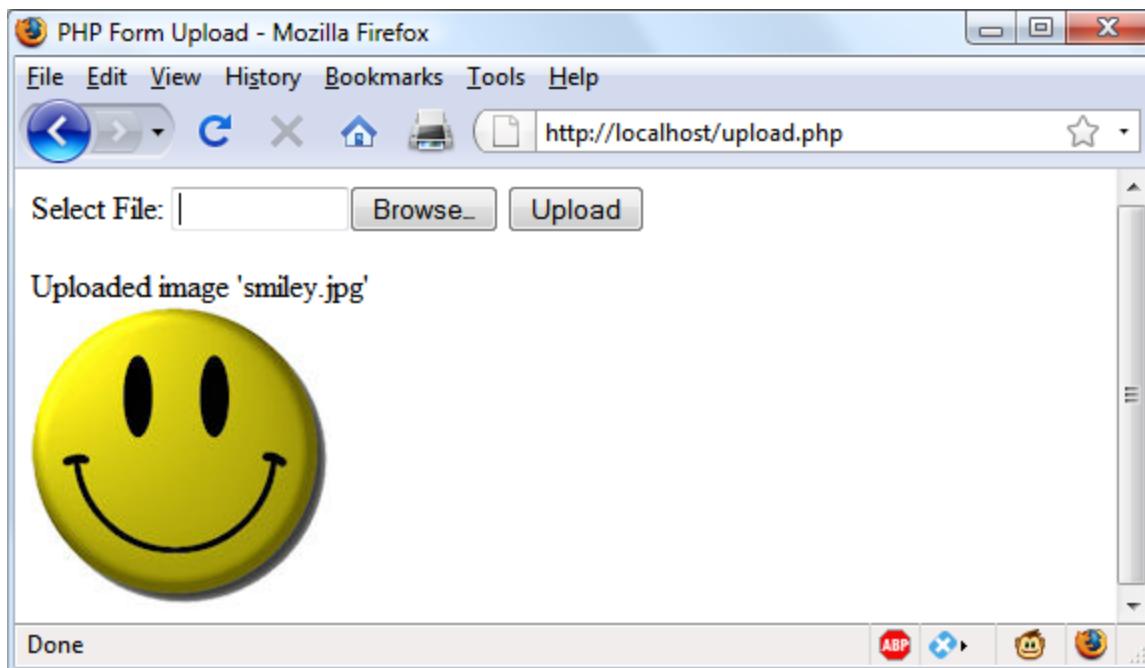


Figure 7-2. Uploading an image as form data

## Using `$_FILES`

Five things are stored in the `$_FILES` array when a file is uploaded, as shown in [Table 7-6](#) (where *file* is the file upload field name supplied by the submitting form).

Table 7-6. The contents of the `$_FILES` array

Array element	Contents
<code>\$_FILES['file']['name']</code>	The name of the uploaded file (e.g., <i>smiley.jpg</i> )
<code>\$_FILES['file']['type']</code>	The content type of the file (e.g., <i>image/jpeg</i> )
<code>\$_FILES['file']['size']</code>	The file's size in bytes
<code>\$_FILES['file']['tmp_name']</code>	The name of the temporary file stored on the server
<code>\$_FILES['file']['error']</code>	The error code resulting from the file upload

Content types used to be known as *MIME* (Multipurpose Internet Mail Extension) types, but because their use later expanded to the whole internet, now they are often called *internet media types*. [Table 7-7](#) shows some of the more frequently used types that turn up in `$_FILES['file']['type']`.

*Table 7-7. Some common internet media content types*

application/pdf	image/gif	multipart/form-data	text/xml
application/zip	image/jpeg	text/css	video/mpeg
audio/mpeg	image/png	text/html	video/mp4
audio/x-wav	image/tiff	text/plain	video/quicktime

## Validation

I hope it now goes without saying (although I'll do so anyway) that form data validation is of the utmost importance, due to the possibility of users attempting to hack into your server.

In addition to maliciously formed input data, some of the things you also have to check are whether a file was actually received and, if so, whether the right type of data was sent.

Taking all these things into account, **Example 7-16**, *upload2.php*, is a more secure rewrite of *upload.php*.

---

### *Example 7-16. A more secure version of upload.php*

---

```
<?php // upload2.php
echo <<<_END
<html><head><title>PHP Form Upload</title></head><body>
<form method='post' action='upload2.php' enctype='multipart/form-data'>
Select a JPG, GIF, PNG or TIF File:
<input type='file' name='filename' size='10'>
<input type='submit' value='Upload'></form>
_END;

if ($_FILES)
{
    $name = $_FILES['filename']['name'];

    switch($_FILES['filename']['type'])
    {
        case 'image/jpeg': $ext = 'jpg'; break;
        case 'image/gif': $ext = 'gif'; break;
        case 'image/png': $ext = 'png'; break;
        case 'image/tiff': $ext = 'tif'; break;
    }
}
```

```

    default:           $ext = '';
}
if ($ext)
{
    $n = "image.$ext";
    move_uploaded_file($_FILES['filename']['tmp_name'], $n);
    echo "Uploaded image '$name' as '$n':<br>";
    echo "<img src='$n'>";
}
else echo "'$name' is not an accepted image file";
}
else echo "No image has been uploaded";

echo "</body></html>";
?>

```

The non-HTML section of code has been expanded from the half-dozen lines of [Example 7-15](#) to more than 20 lines, starting at `if ($_FILES)`.

As with the previous version, this `if` line checks whether any data was actually posted, but there is now a matching `else` near the bottom of the program that echoes a message to the screen when nothing has been uploaded.

Within the `if` statement, the variable `$name` is assigned the value of the filename as retrieved from the uploading computer (just as before), but this time we won't rely on the user having sent us valid data. Instead, a `switch` statement checks the uploaded content type against the four types of image this program supports. If a match is made, the variable `$ext` is set to the three-letter file extension for that type. Should no match be found, the file uploaded was not of an accepted type and the variable `$ext` is set to the empty string "".

The next section of code then checks the variable `$ext` to see whether it contains a string and, if so, creates a new filename called `$n` with the base name *image* and the extension stored in `$ext`. This means that the program has full control over the name of the file to be created, as it can be only one of *image.jpg*, *image.gif*, *image.png*, or *image.tif*.

Safe in the knowledge that the program has not been compromised, the rest of the PHP code is much the same as in the previous version. It moves the

uploaded temporary image to its new location and then displays it, while also displaying the old and new image names.

### NOTE

Don't worry about having to delete the temporary file that PHP creates during the upload process, because if the file has not been moved or renamed, it will be automatically removed when the program exits.

After the `if` statement, there is a matching `else`, which is executed only if an unsupported image type was uploaded (in which case it displays an appropriate error message).

When you write your own file-uploading routines, I strongly advise you to use a similar approach and have prechosen names and locations for uploaded files. That way, no attempts to add pathnames and other malicious data to the variables you use can get through. If this means that more than one user could end up having a file uploaded with the same name, you could prefix such files with their user's usernames, or save them to individually created folders for each user.

But if you must use a supplied filename, you should sanitize it by allowing only alphanumeric characters and the period, which you can do with the following command, using a regular expression (see Chapter 18) to perform a search and replace on `$name`:

```
$name = preg_replace("/[^A-Za-z0-9.]/", "", $name);
```

This leaves only the characters A–Z, a–z, 0–9, and periods in the string `$name`, and strips out everything else.

Even better, to ensure that your program will work on all systems, regardless of whether they are case-sensitive or case-insensitive, you should probably use the following command instead, which changes all uppercase characters to lowercase at the same time:

```
$name = strtolower(preg_replace("[^A-Za-z0-9.]", "", $name));
```

## NOTE

Sometimes you may encounter the media type of `image/pjpeg`, which indicates a progressive JPEG, but you can safely add this to your code as an alias of `image/jpeg`, like this:

```
case 'image/pjpeg':
case 'image/jpeg': $ext = 'jpg'; break;
```

# System Calls

Sometimes PHP will not have the function you need to perform a certain action, but the operating system it is running on may. In such cases, you can use the `exec` system call to do the job.

For example, to quickly view the contents of the current directory, you can use a program such as [Example 7-17](#). If you are on a Windows system, it will run as is using the Windows `dir` command. On Linux, Unix, or macOS, comment out or remove the first line and uncomment the second to use the `ls` system command. You may wish to type this program, save it as `exec.php`, and call it up in your browser.

---

### *Example 7-17. Executing a system command*

```
<?php // exec.php
$cmd = "dir"; // Windows
// $cmd = "ls"; // Linux, Unix & Mac

exec(escapeshellcmd($cmd), $output, $status);

if ($status) echo "Exec command failed";
else
{
    echo "<pre>";
```

```

foreach($output as $line) echo htmlspecialchars("$line\n");
echo "</pre>";
}
?>

```

The `htmlspecialchars` function is called to turn any special characters returned by the system into ones that HTML can understand and properly display, neatening the output. Depending on the system you are using, the result of running this program will look something like this (from a Windows `dir` command):

```

Volume in drive C is Hard Disk
Volume Serial Number is DC63-0E29

Directory of C:\Program Files (x86)\Ampps\www

11/04/2025  11:58    <DIR>          .
11/04/2025  11:58    <DIR>          ..
28/01/2025  16:45    <DIR>          5th_edition_examples
08/01/2025  10:34    <DIR>          cgi-bin
08/01/2025  10:34    <DIR>          error
29/01/2025  16:18            1,150 favicon.ico
                           1 File(s)   1,150 bytes
                           5 Dir(s)  1,611,387,486,208 bytes free

```

`exec` takes three arguments:

- The command itself (in the previous case, `$cmd`)
- An array in which the system will put the output from the command (in the previous case, `$output`)
- A variable to contain the returned status of the call (which, in the previous case, is `$status`)

If you wish, you can omit the `$output` and `$status` parameters, but you will not know the output created by the call or even whether it completed successfully.

You should also note the use of the `escapeshellcmd` function. It is a good habit to always use this when issuing an `exec` call, because it sanitizes the command string, preventing the execution of arbitrary commands, should you supply user input to the call.

### WARNING

The system call functions are typically disabled on shared web hosts, as they pose a security risk. You should always try to solve your problems within PHP if you can, and go to the system directly only if it is really necessary. Also, going to the system is relatively slow, and you need to code two implementations if your application is expected to run on both Windows and Linux/Unix systems.

## XHTML or HTML5?

Because XHTML documents need to be well formed, you can parse them using standard XML parsers—unlike HTML, which requires a lenient HTML-specific parser (which, thankfully, most popular web browsers are). For this reason, XHTML never really caught on, and when the time came to devise a new standard, the World Wide Web Consortium chose to support HTML5 rather than the newer XHTML2 standard.

HTML5 has some of the features of both HTML4 and XHTML, but is much simpler to use and less strict to validate—and, happily, there is now just a single document type you need to place at the head of an HTML5 document (instead of the variety of strict, transitional, and frameset types previously required):

```
<!DOCTYPE html>
```

Just the simple word `html` is sufficient to tell the browser that your web page is designed for HTML5 and, because all the latest versions of the most popular browsers have been supporting most of the HTML5 specification since 2011 or so, this document type is generally the only one you need, unless you choose to cater to older browsers.

For all intents and purposes, when writing HTML documents, web developers can safely ignore the old XHTML document types and syntax (such as using `<br />` instead of the simpler `<br>` tag). But if you find yourself having to cater to a very old browser or an unusual application that relies on XHTML, then you can get more information on how to do that at <http://xhtml.com>.

## Questions

1. Which `printf` conversion specifier would you use to display a floating-point number?
2. What `printf` statement could be used to take the input string "Happy Birthday" and output the string "\*\*Happy"?
3. To send the output from `printf` to a variable instead of to a browser, what alternative function would you use?
4. How would you create a Unix timestamp for 7:11 a.m. on May 2, 2016?
5. Which file access mode would you use with `fopen` to open a file in write and read mode, with the file truncated and the file pointer at the start?
6. What is the PHP command for deleting the file `file.txt`?
7. Which PHP function is used to read in an entire file in one go, even from across the web?
8. Which PHP superglobal variable holds the details on uploaded files?
9. Which PHP function enables the running of system commands?
10. Which of the following tag styles is preferred in HTML5: `<hr>` or `<hr />`?

See “[Chapter 7 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 8. Introduction to MySQL

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

With well over 10 million installations, MySQL is probably the most popular database management system for web servers. Developed in the mid-1990s, it’s now a mature technology that powers many of today’s most-visited internet destinations.

One reason for its success must be that, like PHP, it’s free to use. But it’s also extremely powerful and exceptionally fast—it can run on even the most basic of hardware, and it hardly puts a dent in system resources.

MySQL is also highly scalable, which means that it can grow with your website (the latest benchmarks are kept [up to date online](#)).

## MySQL Basics

A *database* is a structured collection of records or data stored in a computer system and organized in such a way that it can be quickly searched and

information can be rapidly retrieved.

The *SQL* in MySQL stands for *Structured Query Language*. This language is loosely based on English and also used in other databases such as Oracle and Microsoft SQL Server. It is designed to allow simple requests from a database via commands such as:

```
SELECT title FROM publications WHERE author = 'Charles Dickens';
```

A MySQL database contains one or more *tables*, each of which contains *records* or *rows*. Within these rows are various *columns* or *fields* that contain the data itself. **Table 8-1** shows the contents of an example database of five publications detailing the author, title, type, and year of publication.

*Table 8-1. Example of a simple database*

Author	Title	Type	Year
Mark Twain	The Adventures of Tom Sawyer	Fiction	1876
Jane Austen	Pride and Prejudice	Fiction	1811
Charles Darwin	The Origin of Species	Non-fiction	1856
Charles Dickens	The Old Curiosity Shop	Fiction	1841
William Shakespeare	Romeo and Juliet	Play	1594

Each row in the table is the same as a row in a MySQL table, a column in the table corresponds to a column in MySQL, and each element within a row is the same as a MySQL field.

To uniquely identify this database, I'll refer to it as the *publications* database in the examples that follow. And, as you will have observed, all these publications are considered to be classics of literature, so I'll call the table within the database that holds the details *classics*.

## Summary of Database Terms

The main terms you need to acquaint yourself with for now are as follows:

*Database*

The overall container for a collection of MySQL data

*Table*

A sub-container within a database that stores the actual data

*Row*

A single record within a table, which may contain several fields

*Column*

The name of a field within a row

I should note that I'm not trying to reproduce the precise terminology used in academic literature about relational databases, but just to provide simple, everyday terms to help you quickly grasp basic concepts and get started with a database.

## Accessing MySQL via the Command Line

There are three main ways you can interact with MySQL: using a command line, via a web interface such as phpMyAdmin, and through a programming language like PHP. We'll start doing the third of these in Chapter 11 , but for now, let's look at the first two.

### Starting the Command-Line Interface

The following sections describe relevant instructions for Windows, macOS, and Linux.

#### Windows users

If you installed AMPPS (as explained in [Chapter 2](#)) in the usual way, you will be able to access the MySQL executable from the following directory:

```
C:\Program Files\Ampps\mysql\bin
```

### NOTE

If you installed AMPPS in any other place, you will need to use that directory instead, such as the following for 32-bit installations of AMPPS:

```
C:\Program Files (x86)\Ampps\mysql\bin
```

By default, the initial MySQL user is *root*, and it will have a default password of *mysql*. So, to enter MySQL's command-line interface, select Start → Run, enter **CMD** into the Run box, and press Return. This will call up a Windows command prompt. From there, enter the following (making any appropriate changes as just discussed):

```
cd C:\\"Program Files\Ampps\mysql\bin"
mysql -u root -pmysql
```

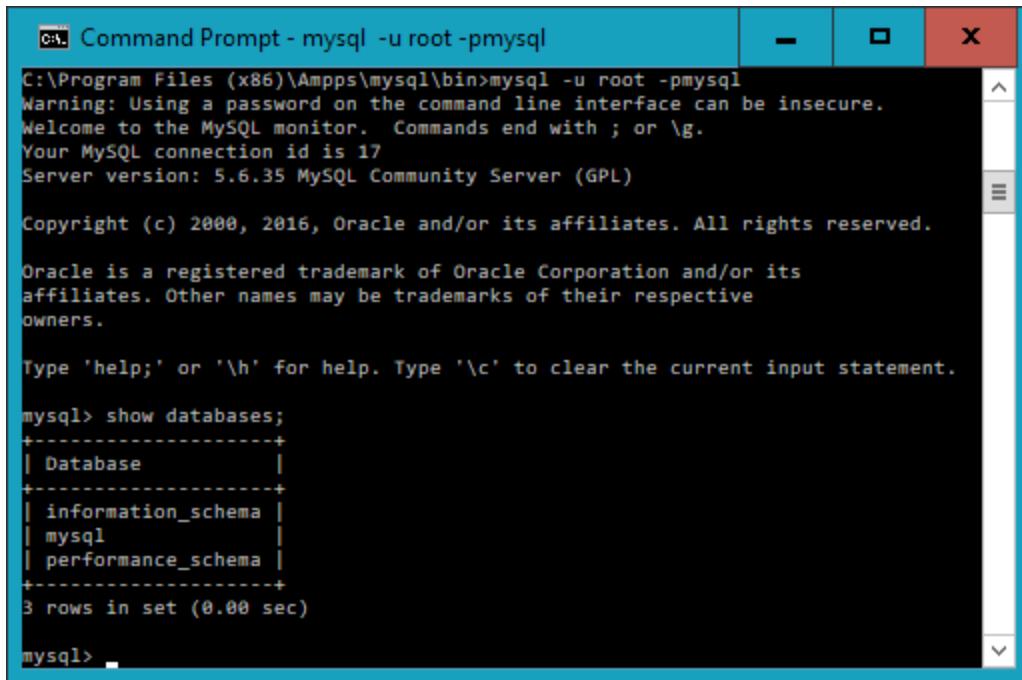
The first command changes to the MySQL directory, and the second tells MySQL to log you in as user *root*, with the password *mysql*. You will now be logged into MySQL and can start entering commands.

If you are using Windows PowerShell (rather than Command Prompt), it will not load commands from the current location as you must explicitly specify where to load a program from, in which case you would, instead, enter the following (note the preceding **.** before the **mysql** command):

```
cd C:\\"Program Files\Ampps\mysql\bin"
./mysql -u root -pmysql
```

To be sure everything is working as it should be, enter the following—the results should be similar to [Figure 8-1](#):

```
SHOW databases;
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt - mysql -u root -pmysql". The window displays the MySQL monitor interface. It starts with a warning about using a password on the command line. It then shows the MySQL connection ID (17), server version (5.6.35 MySQL Community Server (GPL)), copyright information (Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.), and trademarks (Oracle is a registered trademark of Oracle Corporation and/or its affiliates). It also provides help instructions. Finally, it executes the command "mysql> show databases;" which lists three databases: "information\_schema", "mysql", and "performance\_schema". The output ends with "3 rows in set (0.00 sec)".

```
c:\Program Files (x86)\Ampps\mysql\bin>mysql -u root -pmysql
Warning: Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.6.35 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
+-----+
3 rows in set (0.00 sec)

mysql>
```

Figure 8-1. Accessing MySQL from a Windows command prompt

You are now ready to move on to the next section, “[Using the Command-Line Interface](#)”.

## macOS users

To proceed with this chapter, you should have installed AMPPS as detailed in [Chapter 2](#). You should also have the web server running and the MySQL server started.

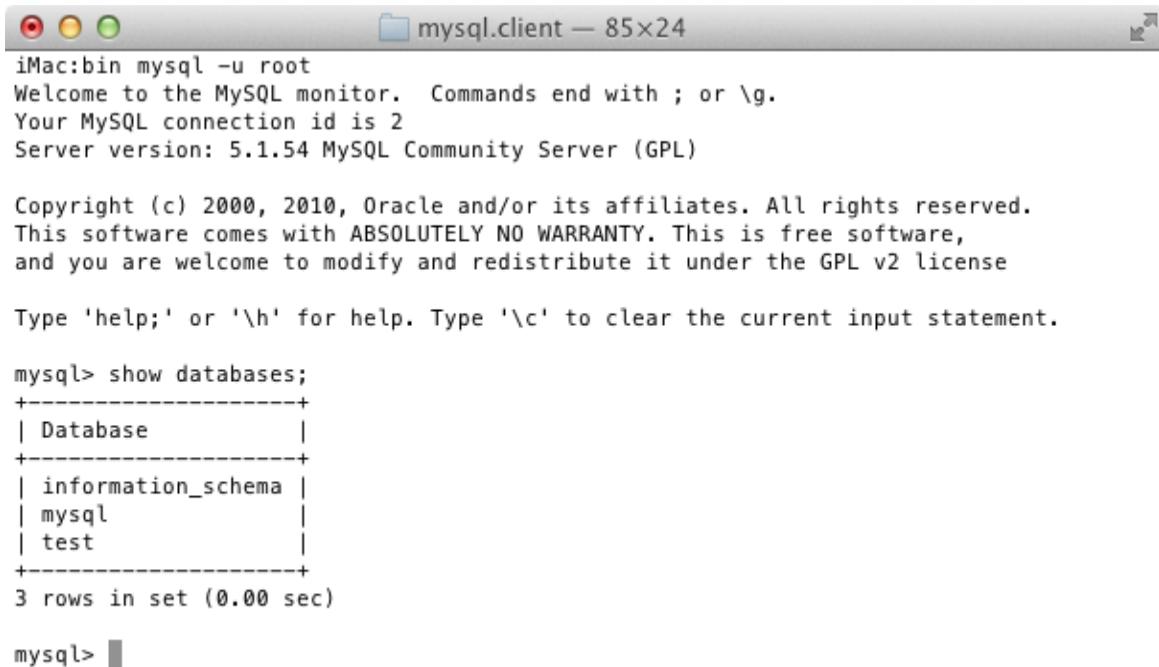
To enter the MySQL command-line interface, start the Terminal program (which should be available in Finder → Utilities). Then call up the MySQL program, which will have been installed in the directory `/Applications/ampps/mysql/bin`.

By default, the initial MySQL user is *root*, and it will have a password of *mysql*. So, to start the program, type the following:

```
/Applications/ampss/mysql/bin/mysql -u root -pmysql
```

This command tells MySQL to log you in as user *root* using the password *mysql*. To verify that all is well, type the following (Figure 8-2 should be the result):

```
SHOW databases;
```



```
iMac:bin mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.1.54 MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| test           |
+-----+
3 rows in set (0.00 sec)

mysql> █
```

Figure 8-2. Accessing MySQL from the macOS Terminal program

If you receive an error such as `Can't connect to local MySQL server through socket`, you will need to first start the MySQL server as described in [Chapter 2](#).

You should now be ready to move on to the next section, “[Using the Command-Line Interface](#)”.

## Linux users

On a system running a Unix-like operating system such as Linux, you will almost certainly already have PHP and MySQL installed and running, and you will be able to enter the examples in the next section (if not, you can follow the procedure outlined in [Chapter 2](#) to install AMPPS). First, you should type the following to log into your MySQL system:

```
mysql -u root -p
```

This tells MySQL to log you in as the user *root* and to request your password. If you have a password, enter it; otherwise, just press Return.

Once you are logged in, type the following to test the program—you should see something like [Figure 8-3](#) in response:

```
SHOW databases;
```

```
You may also use sysinstall(8) to re-enter the installation and
configuration utility. Edit /etc/motd to change this login announcement.

robnix# mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 4377812
Server version: mysql-server-5.0.51a

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| test           |
+-----+
3 rows in set (0.02 sec)

mysql> █
```

*Figure 8-3. Accessing MySQL using Linux*

If this procedure fails at any point, please refer to [Chapter 2](#) to ensure that you have MySQL properly installed. Otherwise, you should now be ready to move on to the next section, “[Using the Command-Line Interface](#)”.

## MySQL on a remote server

If you are accessing MySQL on a remote server, it will probably be a Linux/FreeBSD/Unix type of box, and you should connect to it via the secure SSH protocol (avoid using the insecure Telnet protocol at all costs). Once in there, you might find that things are a little different, depending on how the system administrator has set the server up—especially if it’s a shared hosting server. Therefore, you need to ensure that you have been given access to MySQL and that you have your username and password. Armed with these, you can then type the following, where *username* is the name supplied:

```
mysql -u username -p
```

Enter your password when prompted. You can then try the following command, which should result in something like [Figure 8-3](#):

```
SHOW databases;
```

There may be other databases already created, and the *test* database may not be there.

Bear in mind also that system administrators have ultimate control over everything and that you can encounter some unexpected setups. For example, you may find that you are required to preface all database names that you create with a unique identifying string to ensure that your names do not conflict with those of databases created by other users.

Therefore, if you have any problems, talk with your system administrator, who will get you sorted out. Just let the sysadmin know that you need a username and password. You should also ask for the ability to create new

databases or, at a minimum, to have at least one database created for you ready to use. You can then create all the tables you require within that database.

## Using the Command-Line Interface

From here on out, it makes no difference whether you are using Windows, macOS, or Linux to access MySQL directly, as all the commands used (and errors you may receive) are identical.

### The semicolon

Let's start with the basics. Did you notice the semicolon (;) at the end of the `SHOW databases;` command that you typed? The semicolon is used by MySQL to separate or end commands. If you forget to enter it, MySQL will issue a prompt and wait for you to do so. The required semicolon was made part of the syntax to let you enter multiple-line commands, which can be convenient because some commands get quite long. It also allows you to issue more than one command at a time by placing a semicolon after each one. The interpreter gets them all in a batch when you press the Enter (or Return) key and executes them in order.

#### NOTE

It's very common to receive a MySQL prompt instead of the results of your command; it means that you forgot the final semicolon. Just enter the semicolon and press the Enter key, and you'll get what you want.

There are six different prompts that MySQL may present you with (see [Table 8-2](#)), so you will always know where you are during a multiline input.

*Table 8-2. MySQL's six command prompts*

MySQL prompt	Meaning
mysql>	Ready and waiting for a command
->	Waiting for the next line of a command
'>	Waiting for the next line of a string started with a single quote
">	Waiting for the next line of a string started with a double quote
`>	Waiting for the next line of a string started with a backtick
/*>	Waiting for the next line of a comment started with /*

## Cancelling a command

If you are partway through entering a command and decide you don't wish to execute it after all, whatever you do, *don't press Ctrl-C!* That will close the program. Instead, you can enter \c and press Return. **Example 8-1** shows how to use the command.

### Example 8-1. Cancelling a line of input

```
meaningless gibberish to mysql \c
```

When you type that line, MySQL will ignore everything you typed and issue a new prompt. Without the \c, it would have displayed an error message. Be careful, though: if you have opened a string or comment, close it first before using the \c or MySQL will think the \c is just part of the string. **Example 8-2** shows the right way to do this.

### Example 8-2. Cancelling input from inside a string

```
this is "meaningless gibberish to mysql" \c
```

Also note that using \c after a semicolon will not cancel the preceding command, as it is then a new statement.

## MySQL Commands

You've already seen the `SHOW` command, which lists tables, databases, and many other items. The commands you'll use most often are listed in **Table 8-3**.

*Table 8-3. Common MySQL commands*

Command	Action
ALTER	Alter a database or table
BACKUP	Back up a table
\c	Cancel input
CREATE	Create a database
DELETE	Delete a row from a table
DESCRIBE	Describe a table's columns
DROP	Delete a database or table
EXIT (Ctrl-C)	Exit
GRANT	Change user privileges
HELP (\h, \?)	Display help
INSERT	Insert data
LOCK	Lock table(s)
QUIT (\q)	Same as EXIT
RENAME	Rename a table
SHOW	List details about an object
SOURCE	Execute a file
STATUS (\s)	Display the current status
TRUNCATE	Empty a table
UNLOCK	Unlock table(s)
UPDATE	Update an existing record
USE	Use a database

I'll cover most of these as we proceed, but first, you need to remember a couple of points about MySQL commands:

- SQL commands and keywords are case-insensitive. `CREATE`, `create`, and `CrEaTe` all mean the same thing. However, for the sake of clarity, you may prefer to use uppercase.
- Table names are case-sensitive on Linux and macOS, but case-insensitive on Windows. So, for the sake of portability, you should always choose a case and stick to it. The recommended style is to use lowercase for table names.

## Creating a database

If you are working on a remote server and have only a single user account and access to a single database that was created for you, move on to the section “[Creating a table](#)”. Otherwise, get the ball rolling by issuing the following command to create a new database called *publications*:

```
CREATE DATABASE publications;
```

A successful command will return a message that doesn't mean much yet—`Query OK, 1 row affected (0.00 sec)`—but will make sense soon. Now that you've created the database, you want to work with it, so issue the following command:

```
USE publications;
```

You should now see the message `Database changed` and will then be set to proceed with the following examples.

## Creating users

Now that you've seen how easy it is to use MySQL and created your first database, it's time to look at how you create users, as you probably won't

want to grant your PHP scripts root access to MySQL—it could cause a real headache should you get hacked.

To create a user, issue the `CREATE USER` command, which takes the following form (don't type this in; it's not an actual working command):

```
CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';
GRANT PRIVILEGES ON database.object TO 'username'@'hostname'
```

This should all look pretty straightforward, with the possible exception of the `database.object` part, which refers to the database itself and the objects it contains, such as tables (see [Table 8-4](#)).

*Table 8-4. Example parameters for the GRANT command*

Arguments	Meaning
<code>*.*</code>	All databases and all their objects
<code>database.*</code>	Only the database called <code>database</code> and all its objects
<code>database.object</code>	Only the database called <code>database</code> and its object called <code>object</code>

So, let's create a user who can access just the new *publications* database and all its objects, by entering the following commands (replacing the username *jim* and also the password *mypasswd* with ones of your choosing):

```
CREATE USER 'jim'@'localhost' IDENTIFIED BY 'password'
GRANT ALL ON publications.* TO 'jim'@'localhost'
```

What this does is allow the user *jim@localhost* full access to the *publications* database using the password *mypasswd*. You can test whether this step has worked by entering `quit` to exit and then rerunning MySQL the way you did before, but instead of entering `-u root -p`, type `-u jim -p`, or whatever username you created. See [Table 8-5](#) for the correct command for your operating system. Modify it as necessary if the *mysql* client program is installed in a different directory on your system.

*Table 8-5. Starting MySQL and logging in as  
jim@localhost*

OS	Example command
Windows	C:\\"Program Files\Ampps\mysql\bin\mysql" -u jim -p
macOS	/Applications/ampps/mysql/bin/mysql -u jim -p
Linux	mysql -u jim -p

All you have to do now is enter your password when prompted and you will be logged in. If you choose to, you can place your password immediately following the `-p` (without any spaces) to avoid having to enter it when prompted, but this is considered poor practice because if other people are logged into your system, there may be ways for them to look at the command you entered and find out your password.

#### NOTE

You can grant only privileges that you already have, and you must also have the privilege to issue GRANT commands. There are a whole range of privileges you can choose to grant if you are not granting all privileges. For further details on the GRANT command and the REVOKE command, which can remove privileges once granted, see the [documentation](#). Also be aware that if you create a new user but do not specify an IDENTIFIED BY clause, the user will have no password, a situation that is very insecure and should be avoided.

## Creating a table

At this point, you should now be logged into MySQL with ALL privileges granted for the database *publications* (or a database that was created for you), so you're ready to create your first table. Make sure the correct database is in use by typing the following (replacing *publications* with the name of your database if it is different):

```
USE publications;
```

Now enter the command in [Example 8-3](#) one line at a time.

### Example 8-3. Creating a table called classics

---

```
CREATE TABLE classics (
    author VARCHAR(128),
    title VARCHAR(128),
    type VARCHAR(16),
    year CHAR(4)) ENGINE InnoDB;
```

#### NOTE

The final two words in this command require a little explanation. MySQL can process queries in many different ways internally, and these different ways are supported by different *engines*. From version 5.6 onwards *InnoDB* is the default storage engine for MySQL, and we use it here because it supports FULLTEXT searches. So long as you have a relatively up-to-date version of MySQL, you can omit the `ENGINE InnoDB` section of the command when creating a table, but I have kept it in for now to emphasize that this is the engine being used.

If you are running a version of MySQL prior to 5.6, the InnoDB engine will not support FULLTEXT indexes, so you will have to replace InnoDB in the command with MyISAM to indicate that you want to use that engine (see “[Creating a FULLTEXT index](#)”).

InnoDB is generally more efficient and the recommended option. If you installed the AMPPS stack as detailed in [Chapter 2](#), you should have at least version 5.6.35 of MySQL.

#### NOTE

You could also issue the previous command on a single line, like this:

```
CREATE TABLE classics (author VARCHAR(128), title
    VARCHAR(128), type VARCHAR(16), year CHAR(4)) ENGINE
    InnoDB;
```

But MySQL commands can be long and complicated, so I recommend using the format shown in [Example 8-3](#) until you are comfortable with longer ones.

MySQL should then issue the response `Query OK, 0 rows affected`, along with how long it took to execute the command. If you see an error message instead, check your syntax carefully. Every parenthesis and comma counts, and typing errors are easy to make.

To check whether your new table has been created, type the following:

```
DESCRIBE classics;
```

All being well, you will see the sequence of commands and responses shown in [Example 8-4](#), where you should particularly note the table format displayed.

*Example 8-4. A MySQL session: creating and checking a new table*

---

```
mysql> USE publications;
Database changed
mysql> CREATE TABLE classics (
    -> author VARCHAR(128),
    -> title VARCHAR(128),
    -> type VARCHAR(16),
    -> year CHAR(4)) ENGINE InnoDB;
Query OK, 0 rows affected (0.03 sec)

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| author | varchar(128) | YES  |     | NULL    |       |
| title  | varchar(128) | YES  |     | NULL    |       |
| type   | varchar(16)  | YES  |     | NULL    |       |
| year   | char(4)     | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

The `DESCRIBE` command is an invaluable debugging aid when you need to ensure that you have correctly created a MySQL table. You can also use it to remind yourself about a table's field or column names and the types of data in each one. Let's look at each of the headings in detail:

### *Field*

The name of each field or column within a table

### *Type*

The type of data being stored in the field

### *Null*

Whether the field is allowed to contain a value of NULL

### *Key*

What type of key, if any, has been applied (*keys* or *indexes* in MySQL are quick ways to look up and search for data)

### *Default*

The default value that will be assigned to the field if no value is specified when a new row is created

### *Extra*

Additional information, such as whether a field is set to auto-increment

## Data Types

In [Example 8-3](#), you may have noticed that three of the table's fields were given the data type of VARCHAR, and one was given the type CHAR. The term VARCHAR stands for *VARiable length CHARacter string*, and the command takes a numeric value that tells MySQL the maximum length allowed for a string stored in this field.

Both CHAR and VARCHAR accept text strings and impose a limit on the size of the field. The difference is that every string in a CHAR field has the specified size. If you put in a smaller string, it is padded with spaces. A VARCHAR field does not pad the text; it lets the size of the field vary to fit the text that is inserted. But VARCHAR requires a small amount of overhead to keep track of

the size of each value. So, CHAR is slightly more efficient if the sizes are similar in all records, whereas VARCHAR is more efficient if sizes can vary a lot and get large. In addition, the overhead causes access to VARCHAR data to be slightly slower than to CHAR data.

Another feature of character and text columns, important for today's global web reach, is *character sets*. These assign particular binary values to particular characters. The character set you use for English is obviously different from the one you'd use for Russian. You can assign the character set to a character or text column when you create it.

VARCHAR is useful in our example, because it can accommodate author names and titles of different lengths, while helping MySQL plan the size of the database and perform lookups and searches more easily. Just be aware that if you ever attempt to assign a string value longer than the length allowed, it will be truncated to the maximum length declared in the table definition.

The `year` field, however, has predictable values, so instead of VARCHAR we use the more efficient CHAR(4) data type. The parameter of 4 allows for 4 bytes of data, supporting all years from -999 to 9999; a byte comprises 8 bits and can have the values 00000000 through 11111111, which are 0 to 255 in decimal.

You could, of course, just store two-digit values for the year, but if your data is going to still be needed in the following century, or may otherwise wrap around, it will have to be sanitized first—think of the “millennium bug” that would have caused dates beginning on January 1, 2000, to be treated as 1900 on many of the world’s biggest computer installations.

### NOTE

I didn't use the YEAR data type in the *classics* table because it supports only the years 0000 and 1901 through 2155. This is because MySQL stores the year in a single byte for reasons of efficiency, but it means that only 256 years are available, and the publication years of the titles in the *classics* table are well before 1901.

## The CHAR data type

**Table 8-6** lists the CHAR data types. Both types offer a parameter that sets the maximum (or exact) length of the string allowed in the field. As the table shows, each type has a built-in maximum number of bytes it can occupy.

*Table 8-6. MySQL’s CHAR data types*

Data type	Bytes used	Examples
CHAR( <i>n</i> )	Exactly <i>n</i> ( $\leq 255$ )	CHAR(5) “Hello” uses 5 bytes CHAR(57) “Goodbye” uses 57 bytes
VARCHAR( <i>n</i> )	Up to <i>n</i> ( $\leq 65535$ )	VARCHAR(7) “Hello” uses 5 bytes VARCHAR(100) “Goodbye” uses 7 bytes

## The BINARY data type

The BINARY data types (see **Table 8-7**) store strings of bytes that do not have an associated character set. For example, you might use the BINARY data type to store a GIF image.

*Table 8-7. MySQL’s BINARY data types*

Data type	Bytes used	Examples
BINARY( <i>n</i> )	Exactly <i>n</i> ( $\leq 255$ )	As CHAR but contains binary data
VARBINARY( <i>n</i> )	Up to <i>n</i> ( $\leq 65535$ )	As VARCHAR but contains binary data

## The TEXT data types

Character data can also be stored in one of the set of TEXT fields. The differences between these fields and VARCHAR fields are small:

- Prior to version 5.0.3, MySQL would remove leading and trailing spaces from VARCHAR fields.
- TEXT fields cannot have default values.

- MySQL indexes only the first  $n$  characters of a TEXT column (you specify  $n$  when you create the index).

What this means is that VARCHAR is the better and faster data type to use if you need to search the entire contents of a field. If you will never search more than a certain number of leading characters in a field, you should probably use a TEXT data type (see [Table 8-8](#)).

*Table 8-8. MySQL's TEXT data types*

Data type	Bytes used	Attributes
TINYTEXT( $n$ )	Up to $n$ ( $\leq 255$ )	Treated as a string with a character set
TEXT( $n$ )	Up to $n$ ( $\leq 65535$ )	Treated as a string with a character set
MEDIUMTEXT( $n$ )	Up to $n$ ( $\leq 1.67e+7$ )	Treated as a string with a character set
LONGTEXT( $n$ )	Up to $n$ ( $\leq 4.29e+9$ )	Treated as a string with a character set

The data types that have smaller maximums are also more efficient; therefore, you should use the one with the smallest maximum that you know is enough for any string you will be storing in the field.

## The BLOB data types

The term BLOB stands for *Binar Large OBject*, and therefore, as you would think, the BLOB data type is most useful for binary data in excess of 65,536 bytes in size. The main other difference between the BLOB and BINARY data types is that BLOBs cannot have default values. The BLOB data types are listed in [Table 8-9](#).

*Table 8-9. MySQL's BLOB data types*

Data type	Bytes used	Attributes
TINYBLOB( $n$ )	Up to $n$ ( $\leq 255$ )	Treated as binary data—no character set
BLOB( $n$ )	Up to $n$ ( $\leq 65535$ )	Treated as binary data—no character set
MEDIUMBLOB( $n$ )	Up to $n$ ( $\leq 1.67e+7$ )	Treated as binary data—no character set
LONGBLOB( $n$ )	Up to $n$ ( $\leq 4.29e+9$ )	Treated as binary data—no character set

## Numeric data types

MySQL supports various numeric data types, from a single byte up to double-precision floating-point numbers. Although the most memory that a numeric field can use up is 8 bytes, you are well advised to choose the smallest data type that will adequately handle the largest value you expect. This will help keep your databases small and quickly accessible.

**Table 8-10** lists the numeric data types supported by MySQL and the ranges of values they can contain. In case you are not acquainted with the terms, a *signed number* is one with a possible range from a minus value, through 0, to a positive one; and an *unsigned number* has a value ranging from 0 to a positive one. They can both hold the same number of values; just picture a signed number as being shifted halfway to the left so that half its values are negative and half are positive. Note that floating-point values (of any precision) may only be signed.

*Table 8-10. MySQL's numeric data types*

Data type	Bytes used	Minimum value		Maximum value	
		Signed	Unsigned	Signed	Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	-8.38e+6	0	8.38e+6	1.67e+7
INT / INTEGER	4	-2.15e+9	0	2.15e+9	4.29e+9
BIGINT	8	-9.22e+18	0	9.22e+18	1.84e+19
FLOAT	4	-3.40e+38	n/a	3.4e+38	n/a
DOUBLE / REAL	8	-1.80e+308	n/a	1.80e+308	n/a

To specify whether a data type is unsigned, use the **UNSIGNED** qualifier. The following example creates a table called *tablename* with a field in it called *fieldname* of the data type **UNSIGNED INTEGER**:

```
CREATE TABLE tablename (fieldname INT UNSIGNED);
```

When creating a numeric field, you can also pass an optional number as a parameter, like this:

```
CREATE TABLE tablename (fieldname INT(4));
```

But you must remember that, unlike with the BINARY and CHAR data types, this parameter does not indicate the number of bytes of storage to use. It may seem counterintuitive, but what the number actually represents is the display width of the data in the field when it is retrieved. It is commonly used with the ZEROFILL qualifier, like this:

```
CREATE TABLE tablename (fieldname INT(4) ZEROFILL);
```

What this does is cause any numbers with a width of less than four characters to be padded with one or more zeros, sufficient to make the display width of the field four characters long. When a field is already of the specified width or greater, no padding takes place.

## DATE and TIME types

The main remaining data types supported by MySQL relate to the date and time and can be seen in [Table 8-11](#).

*Table 8-11. MySQL's DATE and TIME data types*

Data type	Time/date format
DATETIME	'0000-00-00 00:00:00'
DATE	'0000-00-00'
TIMESTAMP	'0000-00-00 00:00:00'
TIME	'00:00:00'
YEAR	0000 (Only years 0000 and 1901–2155)

The DATETIME and TIMESTAMP data types display the same way. The main difference is that TIMESTAMP has a very narrow range (from the years 1970 through 2037), whereas DATETIME will hold just about any date you’re likely to specify, unless you’re interested in ancient history or science fiction.

TIMESTAMP is useful, however, because you can let MySQL set the value for you. If you don’t specify the value when adding a row, the current time is automatically inserted. You can also have MySQL update a TIMESTAMP column each time you change a row.

## **The AUTO\_INCREMENT attribute**

Sometimes you need to ensure that every row in your database is guaranteed to be unique. You could do this in your program by carefully checking the data you enter and making sure that there is at least one value that differs in any two rows, but this approach is error-prone and works only in certain circumstances. In the *classics* table, for instance, an author may appear multiple times. Likewise, the year of publication will also be frequently duplicated, and so on. It would be hard to guarantee that you have no duplicate rows.

The general solution is to use an extra column just for this purpose. In a while, we’ll look at using a publication’s ISBN (International Standard Book Number), but first I’d like to introduce the AUTO\_INCREMENT data type.

As its name implies, a column given this data type will set the value of its contents to that of the column entry in the previously inserted row, plus 1. **Example 8-5** shows how to add a new column called *id* to the table *classics* with auto-incrementing.

---

### *Example 8-5. Adding the auto-incrementing column id*

---

```
ALTER TABLE classics ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT KEY;
```

This is your introduction to the ALTER command, which is very similar to CREATE. ALTER operates on an existing table and can add, change, or delete columns. Our example adds a column named *id* with the following characteristics:

#### *INT UNSIGNED*

Makes the column take an integer large enough for us to store more than 4 billion records in the table.

#### *NOT NULL*

Ensures that every column has a value. Many programmers use NULL in a field to indicate that it doesn't have any value. But that would allow duplicates, which would violate the whole reason for this column's existence, so we disallow NULL values.

#### *AUTO\_INCREMENT*

Causes MySQL to set a unique value for this column in every row, as described earlier. We don't really have control over the value that this column will take in each row, but we don't care: all we care about is that we are guaranteed a unique value.

#### *KEY*

An auto-increment column is useful as a key, because you will tend to search for rows based on this column. This will be explained in the section “[Indexes](#)”.

Each entry in the column *id* will now have a unique number, with the first starting at 1 and the others counting upward from there. And whenever a new row is inserted, its *id* column will automatically be given the next number in the sequence.

Rather than applying the column retroactively, you could have included it by issuing the CREATE command in a slightly different format. In that case,

the command in [Example 8-3](#) would be replaced with [Example 8-6](#). Check the final line in particular.

#### *Example 8-6. Adding the auto-incrementing id column at table creation*

---

```
CREATE TABLE classics (
    author VARCHAR(128),
    title VARCHAR(128),
    type VARCHAR(16),
    year CHAR(4),
    id INT UNSIGNED NOT NULL AUTO_INCREMENT KEY) ENGINE InnoDB;
```

If you wish to check whether the column has been added, use the following command to view the table's columns and data types:

```
DESCRIBE classics;
```

Now that we've finished with it, the *id* column is no longer needed, so if you created it using [Example 8-5](#), you should now remove the column using the command in [Example 8-7](#).

#### *Example 8-7. Removing the id column*

---

```
ALTER TABLE classics DROP id;
```

### Adding data to a table

To add data to a table, use the `INSERT` command. Let's see this in action by populating the table *classics* with the data from [Table 8-1](#), using one form of the `INSERT` command repeatedly ([Example 8-8](#)).

#### *Example 8-8. Populating the classics table*

---

```
INSERT INTO classics(author, title, type, year)
VALUES('Mark Twain','The Adventures of Tom Sawyer','Fiction','1876');
INSERT INTO classics(author, title, type, year)
VALUES('Jane Austen','Pride and Prejudice','Fiction','1811');
```

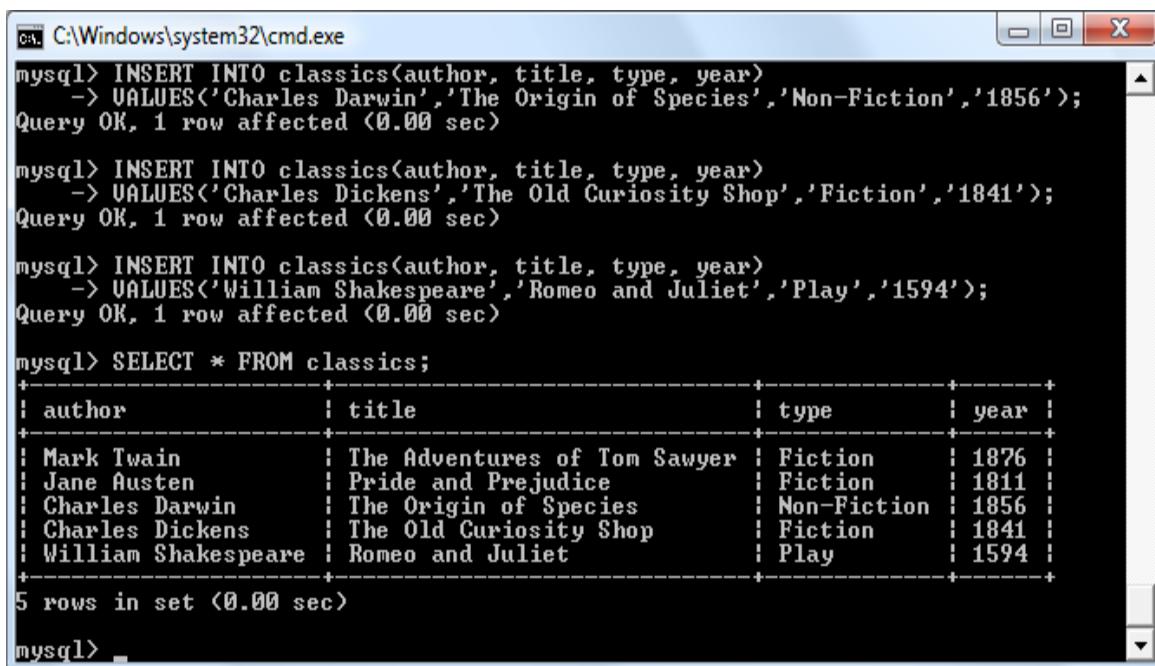
```

INSERT INTO classics(author, title, type, year)
VALUES('Charles Darwin', 'The Origin of Species', 'Non-Fiction', '1856');
INSERT INTO classics(author, title, type, year)
VALUES('Charles Dickens', 'The Old Curiosity Shop', 'Fiction', '1841');
INSERT INTO classics(author, title, type, year)
VALUES('William Shakespeare', 'Romeo and Juliet', 'Play', '1594');

```

After every second line, you should see a `Query OK` message. Once all lines have been entered, type the following command, which will display the table's contents. The result should look like [Figure 8-4](#):

```
SELECT * FROM classics;
```



The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". Inside, a MySQL session is running. The user has run three `INSERT` statements to add data to the `classics` table. After each insert, a `Query OK, 1 row affected <0.00 sec>` message is displayed. Finally, the user runs a `SELECT * FROM classics;` query, which returns the following table:

author	title	type	year
Mark Twain	The Adventures of Tom Sawyer	Fiction	1876
Jane Austen	Pride and Prejudice	Fiction	1811
Charles Darwin	The Origin of Species	Non-Fiction	1856
Charles Dickens	The Old Curiosity Shop	Fiction	1841
William Shakespeare	Romeo and Juliet	Play	1594

Below the table, the message "5 rows in set <0.00 sec>" is shown. The MySQL prompt "mysql>" is at the bottom.

*Figure 8-4. Populating the `classics` table and viewing its contents*

Don't worry about the `SELECT` command for now—we'll come to it in the section [“Querying a MySQL Database”](#). Suffice it to say that, as typed, it will display all the data you just entered.

Also, don't worry if you see the returned results in a different order as this is normal, because the order is unspecified at this point. Later in this chapter we will learn how to use `ORDER BY` to choose the order in which we wish results to be returned, but for now, they may appear in any order.

Let's go back and look at how we used the `INSERT` command. The first part, `INSERT INTO classics`, tells MySQL where to insert the following data. Then, within parentheses, the four column names are listed—*author*, *title*, *type*, and *year*—all separated by commas. This tells MySQL that these are the fields into which the data is to be inserted.

The second line of each `INSERT` command contains the keyword `VALUES` followed by four strings within parentheses, separated by commas. This supplies MySQL with the four values to be inserted into the four columns previously specified. (As always, my choice of where to break the lines was arbitrary.)

Each item of data will be inserted into the corresponding column, in a one-to-one correspondence. If you accidentally listed the columns in a different order from the data, the data would go into the wrong columns. Also, the number of columns must match the number of data items. (There are safer ways of using `INSERT`, which we'll see soon.)

## Renaming a table

Renaming a table, like any other change to the structure or meta-information about a table, is achieved via the `ALTER` command. So, for example, to change the name of the table *classics* to *pre1900*, you would use the following command:

```
ALTER TABLE classics RENAME pre1900;
```

If you tried that command, you should revert the table name by entering the following, so that later examples in this chapter will work as printed:

```
ALTER TABLE pre1900 RENAME classics;
```

## Changing the data type of a column

Changing a column's data type also makes use of the `ALTER` command, this time in conjunction with the `MODIFY` keyword. To change the data type of

the column *year* from CHAR(4) to SMALLINT (which requires only 2 bytes of storage and so will save disk space), enter the following:

```
ALTER TABLE classics MODIFY year SMALLINT;
```

When you do this, if the conversion of data type makes sense to MySQL, it will automatically change the data while keeping the meaning. In this case, it will change each string to a comparable integer, so long as the string is recognizable as referring to an integer.

## Adding a new column

Let's suppose that you have created a table and populated it with plenty of data, only to discover you need an additional column. Not to worry. Here's how to add the new column *pages*, which will be used to store the number of pages in a publication:

```
ALTER TABLE classics ADD pages SMALLINT UNSIGNED;
```

This adds the new column with the name *pages* using the UNSIGNED SMALLINT data type, sufficient to hold a value of up to 65,535—hopefully that's more than enough for any book ever published!

And, if you ask MySQL to describe the updated table by using the DESCRIBE command, as follows, you will see the change has been made (see [Figure 8-5](#)):

```
DESCRIBE classics;
```

```
C:\Windows\system32\cmd.exe
+-----+
| author | varchar(128) | YES | NULL |
| title  | varchar(128) | YES | NULL |
| type   | varchar(16)   | YES | NULL |
| year   | smallint(6)  | YES | NULL |
+-----+
4 rows in set (0.01 sec)

mysql> ALTER TABLE classics ADD pages SMALLINT UNSIGNED;
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> DESCRIBE classics;
+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+
| author | varchar(128) | YES | NULL |
| title  | varchar(128) | YES | NULL |
| type   | varchar(16)   | YES | NULL |
| year   | smallint(6)  | YES | NULL |
| pages  | smallint(5) unsigned | YES | NULL |
+-----+
5 rows in set (0.00 sec)

mysql>
```

Figure 8-5. Adding the new pages column and viewing the table

## Renaming a column

Looking again at [Figure 8-5](#), you may decide that having a column named *type* is confusing, because that is the name used by MySQL to identify data types. Again, no problem—let’s change its name to *category*, like this:

```
ALTER TABLE classics CHANGE type category VARCHAR(16);
```

Note the addition of `VARCHAR(16)` on the end of this command. That’s because the `CHANGE` keyword requires the data type to be specified, even if you don’t intend to change it, and `VARCHAR(16)` was the data type specified when that column was initially created as *type*.

## Removing a column

Actually, upon reflection, you might decide that the page count column *pages* isn’t actually all that useful for this particular database, so here’s how to remove that column by using the `DROP` keyword:

```
ALTER TABLE classics DROP pages;
```

## WARNING

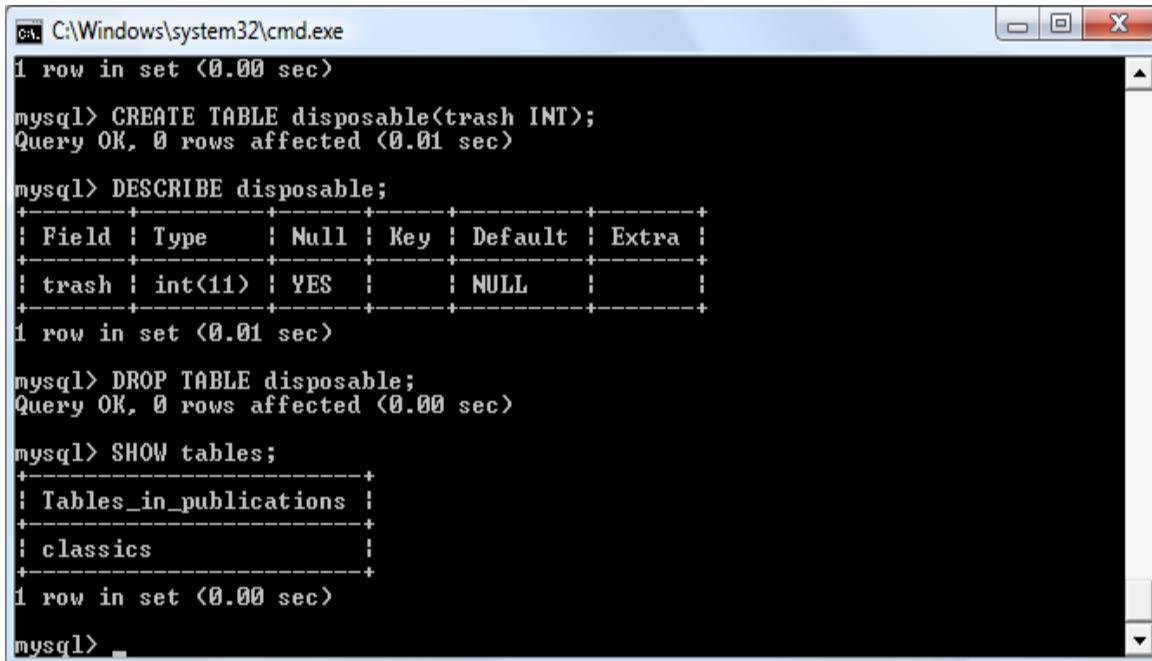
Remember that `DROP` is irreversible. You should always use it with caution, because you could inadvertently delete entire tables (and even databases) with it if you are not careful!

### Deleting a table

Deleting a table is very easy indeed. But, because I don't want you to have to reenter all the data for the *classics* table, let's quickly create a new table, verify its existence, and then delete it. You can do this by typing the commands in [Example 8-9](#). The result of these four commands should look like [Figure 8-6](#).

Example 8-9. Creating, viewing, and deleting a table

```
CREATE TABLE disposable(trash INT);
DESCRIBE disposable;
DROP TABLE disposable;
SHOW tables;
```



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The MySQL command-line interface is running within it. The session starts with creating a table named 'disposable' with one column 'trash' of type INT. It then describes the table to show the structure, which includes columns for Field, Type, Null, Key, Default, and Extra. Finally, it drops the table and shows the current list of tables in the database, which only contains 'Tables\_in\_publications' and 'classics'.

```
C:\Windows\system32\cmd.exe
1 row in set <0.00 sec>

mysql> CREATE TABLE disposable(trash INT);
Query OK, 0 rows affected <0.01 sec>

mysql> DESCRIBE disposable;
+-----+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| trash | int(11) | YES |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
1 row in set <0.01 sec>

mysql> DROP TABLE disposable;
Query OK, 0 rows affected <0.00 sec>

mysql> SHOW tables;
+-----+
| Tables_in_publications |
+-----+
| classics                |
+-----+
1 row in set <0.00 sec>

mysql>
```

*Figure 8-6. Creating, viewing, and deleting a table*

# Indexes

As things stand, the table *classics* works and can be searched without problem by MySQL—until it grows to more than a couple of hundred rows. At that point, database accesses will get slower and slower with every new row added, because MySQL has to search through every row whenever a query is issued. This is like searching through every book in a library whenever you need to look something up.

Of course, you don't have to search libraries that way, because they have either a card index system or, most likely, a database of their own. And the same goes for MySQL, because at the expense of a slight overhead in memory and disk space, you can create a “card index” for a table that MySQL will use to conduct lightning-fast searches.

## Creating an Index

The way to achieve fast searches is to add an *index*, either when creating a table or at any time afterward. But the decision is not so simple. For example, there are different index types, such as a regular INDEX, a PRIMARY KEY, or a FULLTEXT index. Also, you must decide which columns require an index, a judgment that requires you to predict whether you will be searching any of the data in each column. Indexes can get more complicated too, because you can combine multiple columns in one index. And even when you've decided that, you still have the option of reducing index size by limiting the amount of each column to be indexed.

If we imagine the searches that may be made on the *classics* table, it becomes apparent that all of the columns may need to be searched. However, if the *pages* column created in the section “[Adding a new column](#)” had not been deleted, it would probably not have needed an index, as most people would be unlikely to search for books by the number of pages they have. Anyway, go ahead and add an index to each of the columns, using the commands in [Example 8-10](#).

*Example 8-10. Adding indexes to the classics table*

---

```
ALTER TABLE classics ADD INDEX(author(20));
ALTER TABLE classics ADD INDEX(title(20));
ALTER TABLE classics ADD INDEX(category(4));
ALTER TABLE classics ADD INDEX(year);
DESCRIBE classics;
```

The first two commands create indexes on the *author* and *title* columns, limiting each index to only the first 20 characters. For instance, when MySQL indexes the following title:

The Adventures of Tom Sawyer

It will actually store in the index only the first 20 characters:

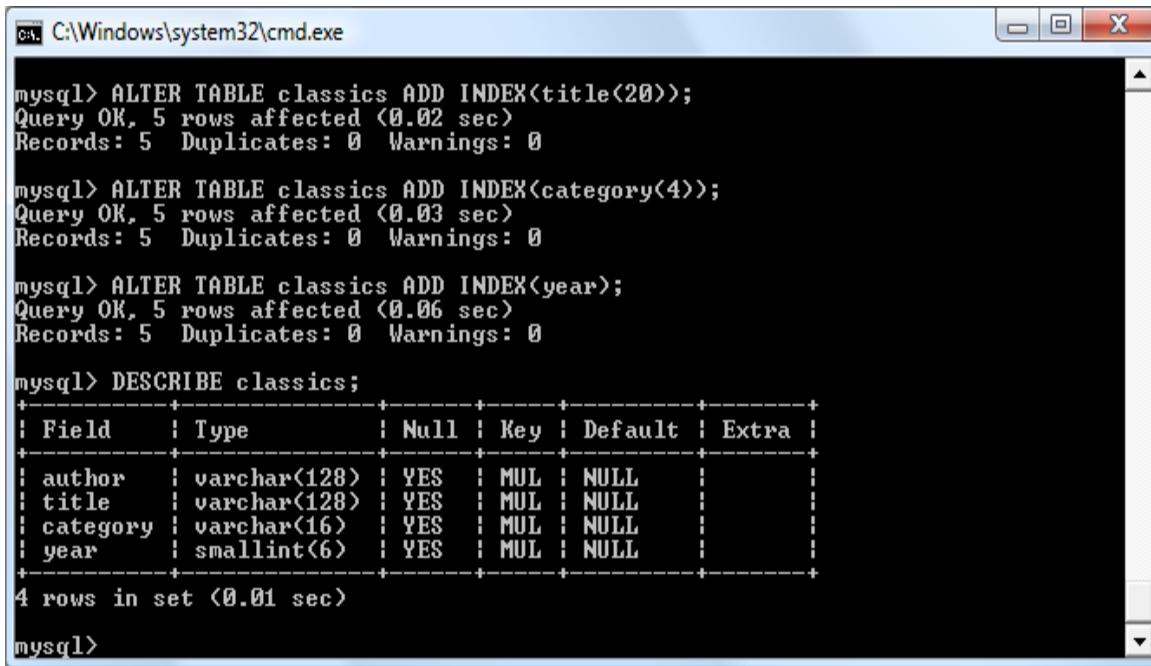
The Adventures of To

This is done to minimize the size of the index, and to optimize database access speed. I chose 20 because it's likely to be sufficient to ensure uniqueness for most strings in these columns. If MySQL finds two indexes with the same contents, it will have to waste time going to the table itself and checking the column that was indexed to find out which rows really matched.

With the *category* column, currently only the first character is required to identify a string as unique (F for Fiction, N for Non-fiction, and P for Play), but I chose an index of four characters to allow for future categories that may share the first three characters. You can also reindex this column later, when you have a more complete set of categories. And finally, I set no limit to the *year* column's index, because it has a clearly defined length of four characters.

The results of issuing these commands (and a DESCRIBE command to confirm that they worked) can be seen in [Figure 8-7](#), which shows the key MUL for each column. This key means that multiple occurrences of a value may occur within that column, which is exactly what we want, as authors

may appear many times, the same book title could be used by multiple authors, and so on.



```
C:\Windows\system32\cmd.exe
mysql> ALTER TABLE classics ADD INDEX(title<20>);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE classics ADD INDEX(category<4>);
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE classics ADD INDEX(year);
Query OK, 5 rows affected (0.06 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| author | varchar(128) | YES  | MUL  | NULL    |        |
| title  | varchar(128)  | YES  | MUL  | NULL    |        |
| category | varchar(16) | YES  | MUL  | NULL    |        |
| year   | smallint(6)  | YES  | MUL  | NULL    |        |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

mysql>
```

Figure 8-7. Adding indexes to the classics table

## Using CREATE INDEX

An alternative to using `ALTER TABLE` to add an index is to use the `CREATE INDEX` command. They are equivalent, except that `CREATE INDEX` cannot be used for creating a `PRIMARY KEY` (see the section “[Primary keys](#)”). The format of this command is shown in the second line of [Example 8-11](#).

[Example 8-11. These two commands are equivalent](#)

```
ALTER TABLE classics ADD INDEX(author<20>);
CREATE INDEX author ON classics (author<20>);
```

## Adding indexes when creating tables

You don’t have to wait until after creating a table to add indexes. In fact, doing so can be time-consuming, as adding an index to a large table can take a very long time. Therefore, let’s look at a command that creates the table `classics` with indexes already in place.

**Example 8-12** is a reworking of **Example 8-3** in which the indexes are created at the same time as the table. Note that to incorporate the modifications made in this chapter, this version uses the new column name *category* instead of *type* and sets the data type of *year* to `SMALLINT` instead of `CHAR(4)`. If you want to try it out without first deleting your current *classics* table, change the word *classics* in line 1 to something else like *classics1*, and then drop *classics1* after you have finished with it.

### *Example 8-12. Creating the table classics with indexes*

---

```
CREATE TABLE classics (
    author VARCHAR(128),
    title VARCHAR(128),
    category VARCHAR(16),
    year SMALLINT,
    INDEX(author(20)),
    INDEX(title(20)),
    INDEX(category(4)),
    INDEX(year)) ENGINE InnoDB;
```

## Primary keys

So far, you've created the table *classics* and ensured that MySQL can search it quickly by adding indexes, but there's still something missing. All the publications in the table can be searched, but there is no single unique key for each publication to enable instant accessing of a row. The importance of having a key with a unique value for each row will come up when we start to combine data from different tables.

The section “[The AUTO\\_INCREMENT attribute](#)” briefly introduced the idea of a primary key when creating the auto-incrementing column *id*, which could have been used as a primary key for this table. However, I wanted to reserve that task for a more appropriate column: the internationally recognized ISBN.

So let's go ahead and create a new column for this key. Now, bearing in mind that ISBNs are 13 characters long, you might think that the following command would do the job:

```
ALTER TABLE classics ADD isbn CHAR(13) PRIMARY KEY;
```

But it doesn't. If you try it, you'll get the error `Duplicate entry` for key 1. The reason is that the table is already populated with some data and this command is trying to add a column with the value `NULL` to each row, which is not allowed, as all values must be unique in any column having a primary key index. However, if there were no data already in the table, this command would work just fine, as would adding the primary key index upon table creation.

In our current situation, we have to be a bit sneaky and create the new column without an index, populate it with data, and then add the index retrospectively using the commands in [Example 8-13](#). Luckily, each of the years is unique in the current set of data, so we can use the `year` column to identify each row for updating. Note that this example uses the `UPDATE` command and `WHERE` keyword, which are explained in more detail in the section [“Querying a MySQL Database”](#).

*Example 8-13. Populating the `isbn` column with data and using a primary key*

---

```
ALTER TABLE classics ADD isbn CHAR(13);
UPDATE classics SET isbn='9781598184891' WHERE year='1876';
UPDATE classics SET isbn='9780582506206' WHERE year='1811';
UPDATE classics SET isbn='9780517123201' WHERE year='1856';
UPDATE classics SET isbn='9780099533474' WHERE year='1841';
UPDATE classics SET isbn='9780192814968' WHERE year='1594';
ALTER TABLE classics ADD PRIMARY KEY(isbn);
DESCRIBE classics;
```

Once you have typed these commands, the results should look like [Figure 8-8](#). Note that the keywords `PRIMARY KEY` replace the keyword `INDEX` in the `ALTER TABLE` syntax (compare Examples [8-10](#) and [8-13](#)).

```

C:\Windows\system32\cmd.exe
mysql> UPDATE classics SET isbn='9780099533474' WHERE year='1841';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE classics SET isbn='9780192814968' WHERE year='1594';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> ALTER TABLE classics ADD PRIMARY KEY (isbn);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra       |
+-----+-----+-----+-----+-----+
| author | varchar(128) | YES  | MUL  | NULL    |             |
| title  | varchar(128)  | YES  | MUL  | NULL    |             |
| category | varchar(16)   | YES  | MUL  | NULL    |             |
| year   | smallint(6)   | YES  | MUL  | NULL    |             |
| isbn   | char(13)      | NO   | PRI  |          |             |
+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql>

```

Figure 8-8. Retrospectively adding a primary key to the *classics* table

To have created a primary key when the table *classics* was created, you could have used the commands in [Example 8-14](#). Again, rename *classics* in line 1 to something else if you wish to try this example, and then delete the test table afterward.

#### Example 8-14. Creating the table *classics* with a primary key

```

CREATE TABLE classics (
  author VARCHAR(128),
  title VARCHAR(128),
  category VARCHAR(16),
  year SMALLINT,
  isbn CHAR(13),
  INDEX(author(20)),
  INDEX(title(20)),
  INDEX(category(4)),
  INDEX(year),
  PRIMARY KEY (isbn)) ENGINE InnoDB;

```

## Creating a FULLTEXT index

Unlike a regular index, MySQL's FULLTEXT allows super-fast searches of entire columns of text. It stores every word in every data string in a special

index that you can search using “natural language,” in a similar manner to using a search engine.

### NOTE

It’s not strictly true that MySQL stores *all* the words in a FULLTEXT index, because it has a built-in list of more than 500 words that it chooses to ignore because they are so common that they aren’t very helpful for searching anyway—so-called *stopwords*. This list includes *the*, *as*, *is*, *of*, and so on. The list helps MySQL run much more quickly when performing a FULLTEXT search and keeps database sizes down.

Here are some things that you should know about FULLTEXT indexes:

- Since MySQL 5.6, InnoDB tables can use FULLTEXT indexes, but prior to that FULLTEXT indexes could be used only with MyISAM tables. If you need to convert a table to MyISAM, you can usually use the MySQL command `ALTER TABLE tablename ENGINE = MyISAM;`.
- FULLTEXT indexes can be created for CHAR, VARCHAR, and TEXT columns only.
- A FULLTEXT index definition can be given in the `CREATE TABLE` statement when a table is created, or added later using `ALTER TABLE` (or `CREATE INDEX`).
- For large data sets, it is *much* faster to load your data into a table that has no FULLTEXT index and then create the index than to load data into a table that has an existing FULLTEXT index.

To create a FULLTEXT index, apply it to one or more records as in **Example 8-15**, which adds a FULLTEXT index to the pair of columns *author* and *title* in the *classics* table (this index is in addition to the ones already created and does not affect them).

*Example 8-15. Adding a FULLTEXT index to the table classics*

---

```
ALTER TABLE classics ADD FULLTEXT(author,title);
```

You can now perform FULLTEXT searches across this pair of columns. This feature could really come into its own if you could now add the entire text of these publications to the database (particularly as they're out of copyright protection) and they would be fully searchable. See the section “[MATCH...AGAINST](#)” for a description of searches using FULLTEXT.

### NOTE

If you find that MySQL is running slower than you think it should be when accessing your database, the problem is usually related to your indexes. Either you don't have an index where you need one, or the indexes are not optimally designed. Tweaking a table's indexes will often solve such a problem. Performance is beyond the scope of this book, but in [Chapter 9](#) I'll give you a few tips so you know what to look for.

## Querying a MySQL Database

So far, we've created a MySQL database and tables, populated them with data, and added indexes to make them fast to search. Now it's time to look at how these searches are performed, and the various commands and qualifiers available.

### SELECT

As you saw in [Figure 8-4](#), the SELECT command is used to extract data from a table. In that section, I used its simplest form to select all data and display it—something you will never want to do on anything but the smallest tables, because all the data will scroll by at an unreadable pace.

Alternatively, on Unix/Linux computers, you can tell MySQL to page output a screen at a time by issuing the command:

```
pager less;
```

This pipes output to the `less` program. To restore standard output and turn paging off, you can issue this command:

```
nopager;
```

Let's now examine `SELECT` in more detail. The basic syntax is:

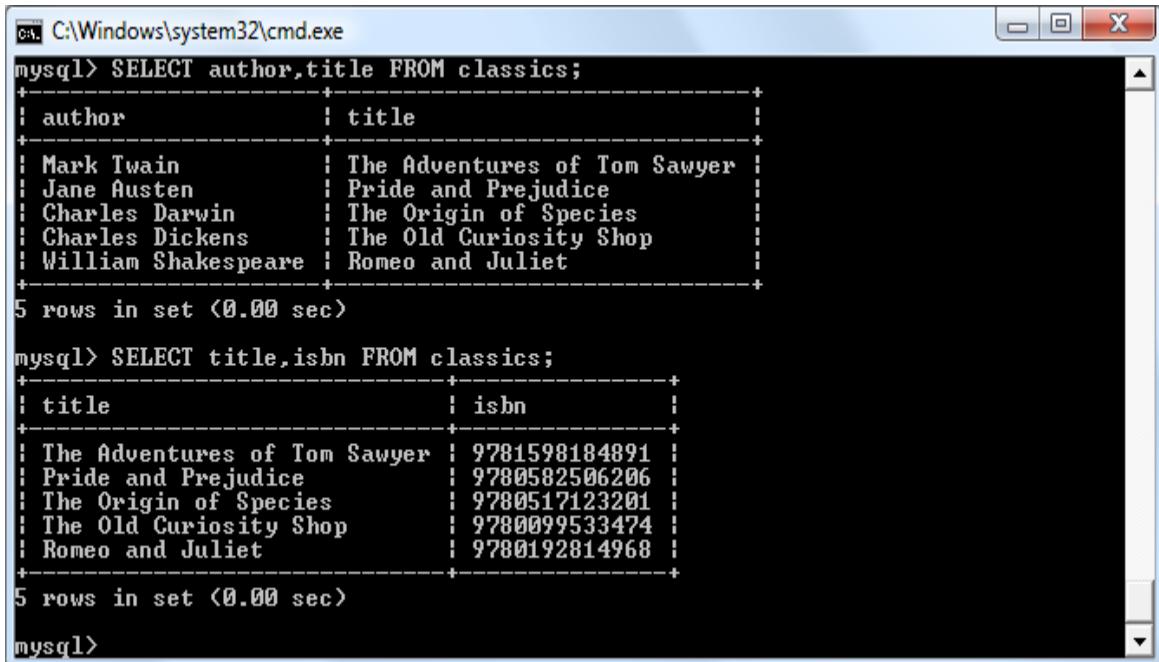
```
SELECT something FROM tablename;
```

The *something* can be an \* (asterisk) as you saw before, which means *every column*, or you can choose to select only certain columns. For instance, [Example 8-16](#) shows how to select just the *author* and *title* and just the *title* and *isbn*. The result of typing these commands can be seen in [Figure 8-9](#).

*Example 8-16. Two different SELECT statements*

---

```
SELECT author,title FROM classics;
SELECT title,isbn FROM classics;
```



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. It contains two MySQL SELECT statements. The first statement, 'SELECT author,title FROM classics;', returns five rows of data with columns 'author' and 'title'. The second statement, 'SELECT title,isbn FROM classics;', returns five rows of data with columns 'title' and 'isbn'. Both statements show 5 rows in set (0.00 sec) and end with 'mysql>'.

```
mysql> SELECT author,title FROM classics;
+-----+-----+
| author | title      |
+-----+-----+
| Mark Twain | The Adventures of Tom Sawyer |
| Jane Austen | Pride and Prejudice |
| Charles Darwin | The Origin of Species |
| Charles Dickens | The Old Curiosity Shop |
| William Shakespeare | Romeo and Juliet |
+-----+-----+
5 rows in set (0.00 sec)

mysql> SELECT title,isbn FROM classics;
+-----+-----+
| title      | isbn        |
+-----+-----+
| The Adventures of Tom Sawyer | 9781598184891 |
| Pride and Prejudice | 9780582506206 |
| The Origin of Species | 9780517123201 |
| The Old Curiosity Shop | 9780099533474 |
| Romeo and Juliet | 9780192814968 |
+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Figure 8-9. The output from two different SELECT statements

## SELECT COUNT

Another replacement for the *something* parameter is COUNT, which can be used in many ways. In [Example 8-17](#), it displays the number of rows in the table by passing \* as a parameter, which means *all rows*. As you'd expect, the result returned is 5, as there are five publications in the table.

### Example 8-17. Counting rows

```
SELECT COUNT(*) FROM classics;
```

## SELECT DISTINCT

The DISTINCT qualifier (and its synonym DISTINCTROW) allows you to weed out multiple entries when they contain the same data. For instance, suppose that you want a list of all authors in the table. If you select just the *author* column from a table containing multiple books by the same author, you'll normally see a long list with the same author names over and over. But by adding the DISTINCT keyword, you can show each author just once.

So, let's test that out by adding another row that repeats one of our existing authors ([Example 8-18](#)).

*Example 8-18. Duplicating data*

---

```
INSERT INTO classics(author, title, category, year, isbn)
VALUES('Charles Dickens','Little Dorrit','Fiction','1857', '9780141439969');
```

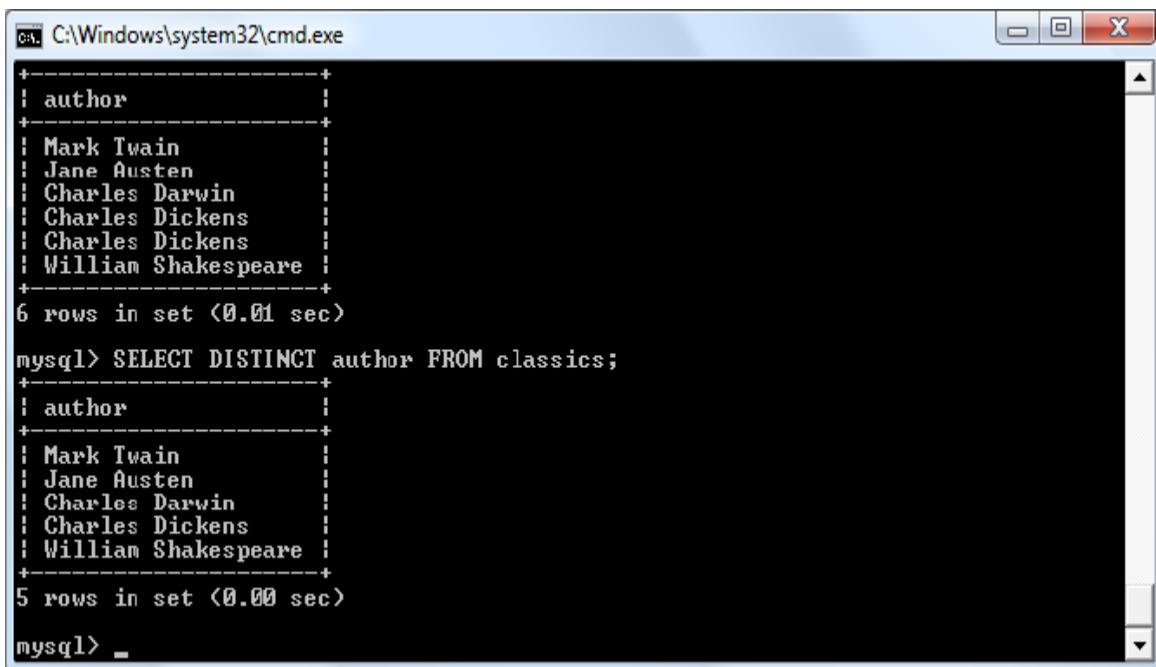
Now that Charles Dickens appears twice in the table, we can compare the results of using SELECT with and without the DISTINCT qualifier.

[Example 8-19](#) and [Figure 8-10](#) show that the simple SELECT lists Dickens twice, and the command with the DISTINCT qualifier shows him only once.

*Example 8-19. With and without the DISTINCT qualifier*

---

```
SELECT author FROM classics;
SELECT DISTINCT author FROM classics;
```



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. It contains the following MySQL session:

```
C:\Windows\system32\cmd.exe
+-----+
| author |
+-----+
| Mark Twain
| Jane Austen
| Charles Darwin
| Charles Dickens
| Charles Dickens
| William Shakespeare
+-----+
6 rows in set (0.01 sec)

mysql> SELECT DISTINCT author FROM classics;
+-----+
| author |
+-----+
| Mark Twain
| Jane Austen
| Charles Darwin
| Charles Dickens
| William Shakespeare
+-----+
5 rows in set (0.00 sec)

mysql> _
```

The first query, 'SELECT author FROM classics;', returns 6 rows, including two entries for 'Charles Dickens'. The second query, 'SELECT DISTINCT author FROM classics;', returns 5 rows, showing that the DISTINCT keyword has removed the duplicate entry for 'Charles Dickens'.

*Figure 8-10. Selecting data with and without DISTINCT*

## DELETE

When you need to remove a row from a table, use the `DELETE` command. Its syntax is similar to the `SELECT` command and allows you to narrow down the exact row or rows to delete using qualifiers such as `WHERE` and `LIMIT`.

Now that you've seen the effects of the `DISTINCT` qualifier, if you typed [Example 8-18](#), you should remove *Little Dorrit* by entering the commands in [Example 8-20](#).

#### Example 8-20. Removing the new entry

---

```
DELETE FROM classics WHERE title='Little Dorrit';
```

This example issues a `DELETE` command for all rows whose `title` column contains the string `Little Dorrit`.

The `WHERE` keyword is very powerful, and important to enter correctly; an error could lead a command to the wrong rows (or have no effect in cases where nothing matches the `WHERE` clause). So now we'll spend some time on that clause, which is the heart and soul of SQL.

## WHERE

The `WHERE` keyword enables you to narrow down queries by returning only those where a certain expression is true. [Example 8-20](#) returns only the rows where the column exactly matches the string `Little Dorrit`, using the equality operator `=`. [Example 8-21](#) shows a couple more examples of using `WHERE` with `=`.

#### Example 8-21. Using the WHERE keyword

---

```
SELECT author,title FROM classics WHERE author="Mark Twain";
SELECT author,title FROM classics WHERE isbn="9781598184891";
```

Given our current table, the two commands in [Example 8-21](#) display the same results. But we could easily add more books by Mark Twain, in which case the first line would display all the titles he wrote and the second line would continue (because we know the ISBN is unique) to display The

*Adventures of Tom Sawyer*. In other words, searches using a unique key are more predictable, and you'll see further evidence later of the value of unique and primary keys.

You can also do pattern matching for your searches using the LIKE qualifier, which allows searches on parts of strings. This qualifier should be used with a % character before or after some text. When placed before a keyword, % means *anything before*. After a keyword, it means *anything after*.

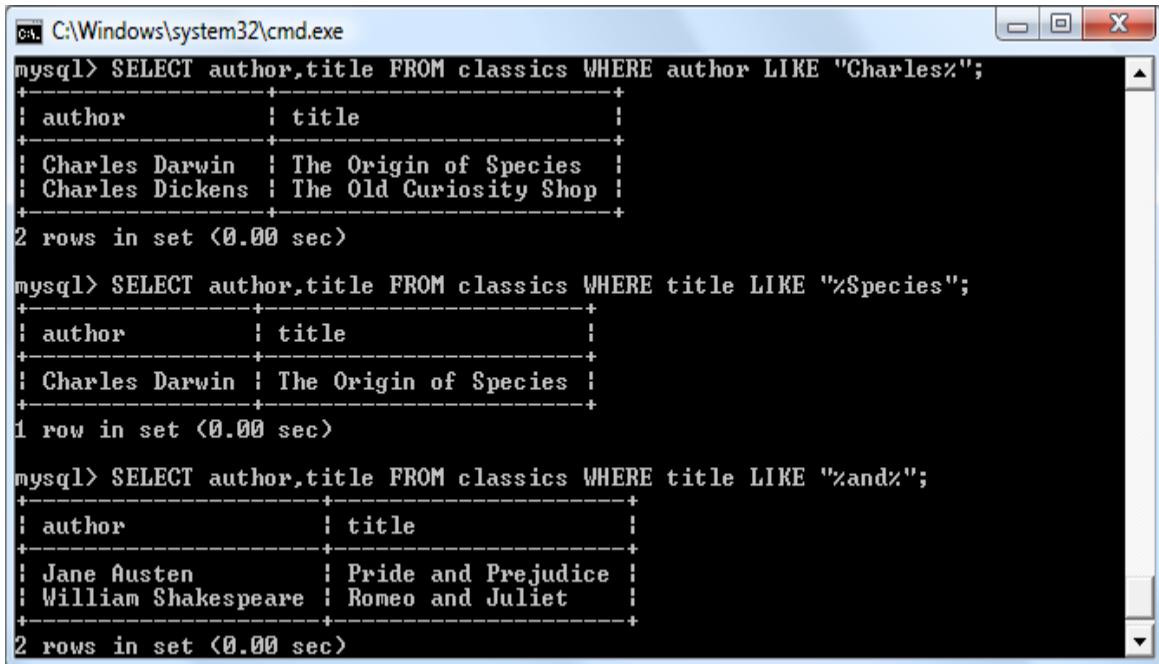
**Example 8-22** performs three different queries, one for the start of a string, one for the end, and one for anywhere in a string.

*Example 8-22. Using the LIKE qualifier*

---

```
SELECT author,title FROM classics WHERE author LIKE "Charles%";  
SELECT author,title FROM classics WHERE title LIKE "%Species";  
SELECT author,title FROM classics WHERE title LIKE "%and%";
```

You can see the results of these commands in [Figure 8-11](#). The first command outputs the publications by both Charles Darwin and Charles Dickens because the LIKE qualifier was set to return anything matching the string **Charles** followed by any other text. Then just **The Origin of Species** is returned, because it's the only row whose column ends with the string **Species**. Last, both **Pride and Prejudice** and **Romeo and Juliet** are returned, because they both matched the string **and** anywhere in the column. The % will also match if there is nothing in the position it occupies; in other words, it can match an empty string.



```
C:\Windows\system32\cmd.exe
mysql> SELECT author,title FROM classics WHERE author LIKE "Charles%";
```

author	title
Charles Darwin	The Origin of Species
Charles Dickens	The Old Curiosity Shop

```
2 rows in set <0.00 sec>
```

```
mysql> SELECT author,title FROM classics WHERE title LIKE "%Species";
```

author	title
Charles Darwin	The Origin of Species

```
1 row in set <0.00 sec>
```

```
mysql> SELECT author,title FROM classics WHERE title LIKE "%and%";
```

author	title
Jane Austen	Pride and Prejudice
William Shakespeare	Romeo and Juliet

```
2 rows in set <0.00 sec>
```

Figure 8-11. Using WHERE with the LIKE qualifier

## LIMIT

The LIMIT qualifier enables you to choose how many rows to return in a query, and where in the table to start returning them. When passed a single parameter, it tells MySQL to start at the beginning of the results and just return the number of rows given in that parameter. If you pass it two parameters, the first indicates the offset from the start of the results where MySQL should start the display, and the second indicates how many to return. You can think of the first parameter as saying, “Skip this number of results at the start.”

**Example 8-23** includes three commands. The first returns the first three rows from the table. The second returns two rows starting at position 1 (skipping the first row). The last command returns a single row starting at position 3 (skipping the first three rows). **Figure 8-12** shows the results of issuing these three commands.

*Example 8-23. Limiting the number of results returned*

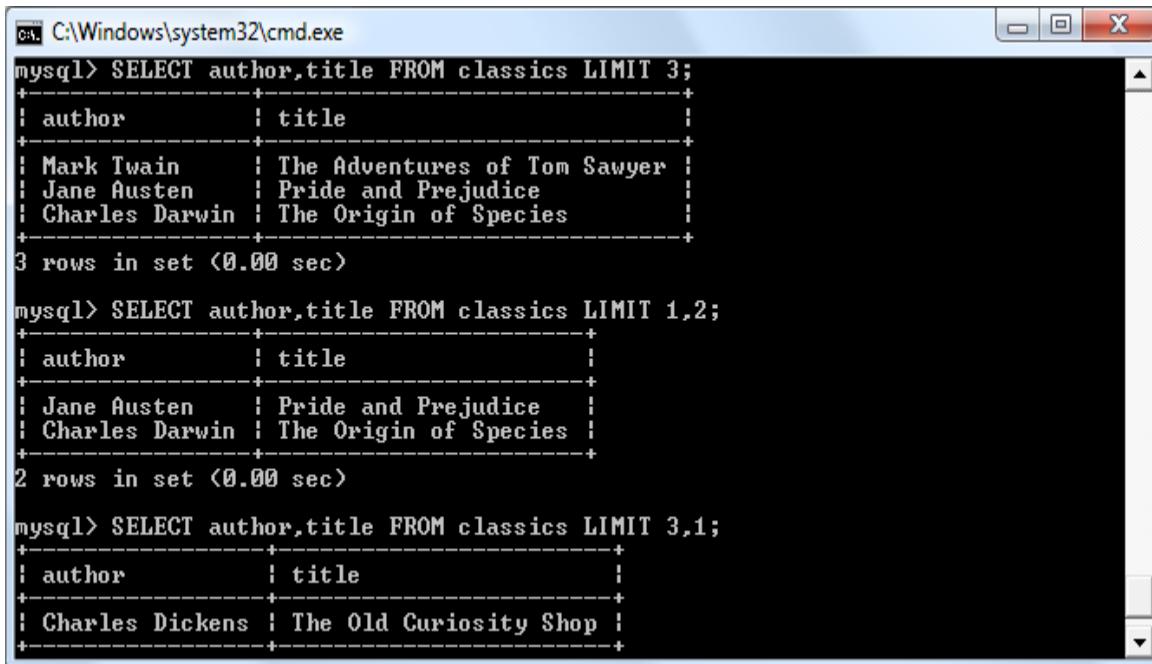
---

```
SELECT author,title FROM classics LIMIT 3;
```

```
SELECT author,title FROM classics LIMIT 1,2;
SELECT author,title FROM classics LIMIT 3,1;
```

## WARNING

Be careful with the `LIMIT` keyword, because offsets start at 0, but the number of rows to return starts at 1. So, `LIMIT 1,3` means return *three* rows starting from the *second* row. You could look at the first argument as stating how many rows to skip, so that in English the instruction would be “Return 3 rows, skipping the first 1.”



The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". It contains three MySQL queries demonstrating the use of the `LIMIT` clause:

```
mysql> SELECT author,title FROM classics LIMIT 3;
+-----+-----+
| author | title      |
+-----+-----+
| Mark Twain | The Adventures of Tom Sawyer |
| Jane Austen | Pride and Prejudice |
| Charles Darwin | The Origin of Species |
+-----+-----+
3 rows in set (0.00 sec)

mysql> SELECT author,title FROM classics LIMIT 1,2;
+-----+-----+
| author | title      |
+-----+-----+
| Jane Austen | Pride and Prejudice |
| Charles Darwin | The Origin of Species |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT author,title FROM classics LIMIT 3,1;
+-----+-----+
| author | title      |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop |
+-----+-----+
```

Figure 8-12. Restricting the rows returned with `LIMIT`

## MATCH...AGAINST

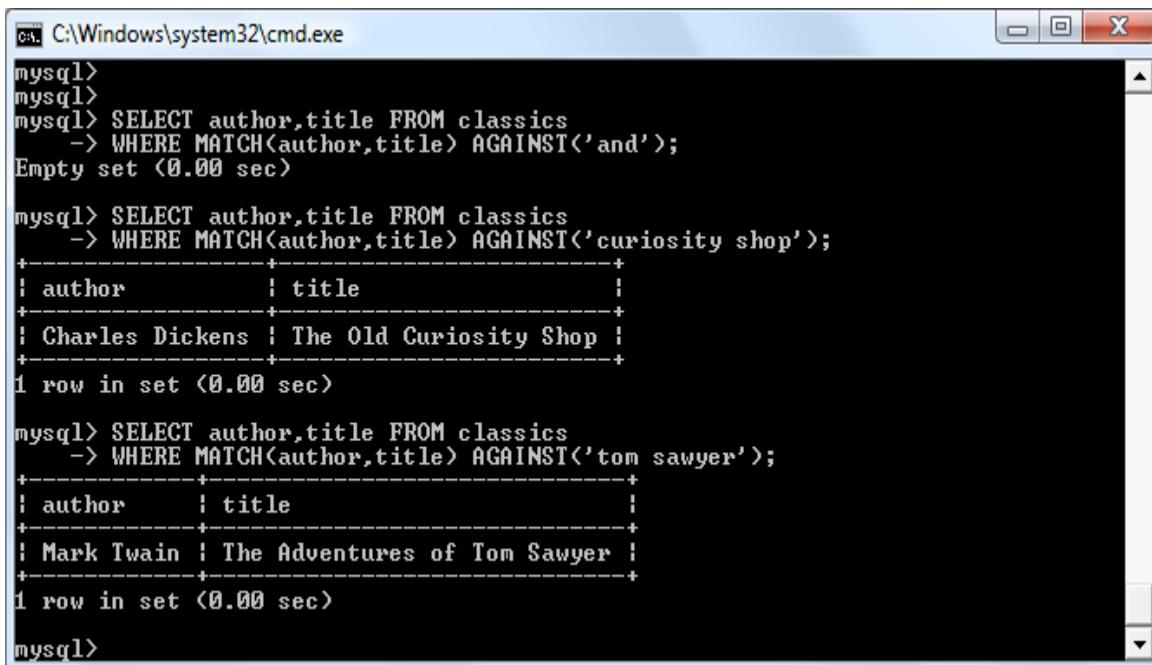
The `MATCH...AGAINST` construct can be used on columns that have been given a `FULLTEXT` index (see the section “[Creating a FULLTEXT index](#)”). With it, you can make natural-language searches as you would in an internet search engine. Unlike the use of `WHERE...=` or `WHERE...LIKE`, `MATCH...AGAINST` lets you enter multiple words in a search query and checks them against all words in the `FULLTEXT` columns. `FULLTEXT` indexes are case-insensitive, so it makes no difference what case is used in your queries.

Assuming that you have added a FULLTEXT index to the *author* and *title* columns, enter the three queries shown in [Example 8-24](#). The first asks for any rows that contain the word *and* to be returned. If you are using the MyISAM storage engine, then because *and* is a stopword in that engine, MySQL will ignore it and the query will always produce an empty set—no matter what is stored in the column. Otherwise, if you are using InnoDB *and* is an allowed word. The second query asks for any rows that contain both of the words *curiosity* and *shop* anywhere in them, in any order, to be returned. And the last query applies the same kind of search for the words *tom* and *sawyer*. [Figure 8-13](#) shows the results of these queries.

*Example 8-24. Using MATCH...AGAINST on FULLTEXT indexes*

---

```
SELECT author,title FROM classics
  WHERE MATCH(author,title) AGAINST('and');
SELECT author,title FROM classics
  WHERE MATCH(author,title) AGAINST('curiosity shop');
SELECT author,title FROM classics
  WHERE MATCH(author,title) AGAINST('tom sawyer');
```



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following MySQL session:

```
mysql>
mysql>
mysql> SELECT author,title FROM classics
    -> WHERE MATCH(author,title) AGAINST('and');
Empty set (0.00 sec)

mysql> SELECT author,title FROM classics
    -> WHERE MATCH(author,title) AGAINST('curiosity shop');
+-----+-----+
| author | title |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT author,title FROM classics
    -> WHERE MATCH(author,title) AGAINST('tom sawyer');
+-----+-----+
| author | title |
+-----+-----+
| Mark Twain | The Adventures of Tom Sawyer |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

*Figure 8-13. Using MATCH...AGAINST on FULLTEXT indexes*

## MATCH...AGAINST in Boolean mode

If you wish to give your MATCH...AGAINST queries even more power, use *Boolean mode*. This changes the effect of the standard FULLTEXT query so that it searches for any combination of search words, instead of requiring all search words to be in the text. The presence of a single word in a column causes the search to return the row.

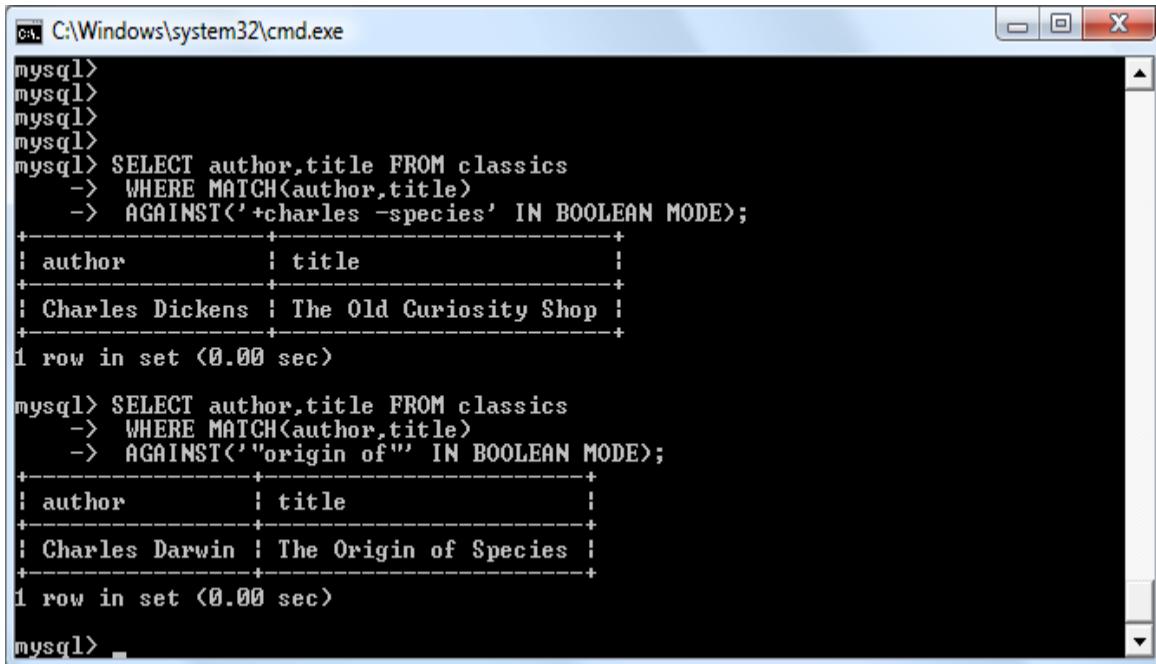
Boolean mode also allows you to preface search words with a + or - sign to indicate whether they must be included or excluded. If normal Boolean mode says, “Any of these words will do,” a plus sign means, “This word must be present; otherwise, don’t return the row.” A minus sign means, “This word must not be present; its presence disqualifies the row from being returned.”

**Example 8-25** illustrates Boolean mode through two queries. The first asks for all rows containing the word *charles* and not the word *species* to be returned. The second uses double quotes to request that all rows containing the exact phrase *origin of* be returned. **Figure 8-14** shows the results of these queries.

#### Example 8-25. Using MATCH...AGAINST in Boolean mode

---

```
SELECT author,title FROM classics
  WHERE MATCH(author,title)
    AGAINST('+charles -species' IN BOOLEAN MODE);
SELECT author,title FROM classics
  WHERE MATCH(author,title)
    AGAINST('"origin of"' IN BOOLEAN MODE);
```



```
C:\Windows\system32\cmd.exe
mysql>
mysql>
mysql>
mysql>
mysql> SELECT author,title FROM classics
-> WHERE MATCH(author,title)
-> AGAINST('+charles -species' IN BOOLEAN MODE);
+-----+-----+
| author | title |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT author,title FROM classics
-> WHERE MATCH(author,title)
-> AGAINST('"origin of"' IN BOOLEAN MODE);
+-----+-----+
| author | title |
+-----+-----+
| Charles Darwin | The Origin of Species |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Figure 8-14. Using MATCH...AGAINST in Boolean mode

As you would expect, the first request returns only *The Old Curiosity Shop* by Charles Dickens; any rows containing the word *species* have been excluded, so Charles Darwin's publication is ignored.

### NOTE

There is something of interest to note in the second query: the stopword *of* is part of the search string, but it is still used by the search because the double quotation marks override stopwords.

## UPDATE...SET

This construct allows you to update the contents of a field. If you wish to change the contents of one or more fields, you need to first narrow in on just the field or fields to be changed, in much the same way you use the SELECT command. [Example 8-26](#) shows the use of UPDATE...SET in two different ways. You can see the results in [Figure 8-15](#).

[Example 8-26. Using UPDATE...SET](#)

```

UPDATE classics SET author='Mark Twain (Samuel Langhorne Clemens)'
  WHERE author='Mark Twain';
UPDATE classics SET category='Classic Fiction'
  WHERE category='Fiction';

```

```

C:\Windows\system32\cmd.exe
mysql>
mysql> UPDATE classics SET author='Mark Twain (Samuel Langhorne Clemens)'
    ->   WHERE author='Mark Twain';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE classics SET category='Classic Fiction'
    ->   WHERE category='Fiction';
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0

mysql> SELECT author,category FROM classics;
+-----+-----+
| author          | category      |
+-----+-----+
| Mark Twain (Samuel Langhorne Clemens) | Classic Fiction
| Jane Austen      | Classic Fiction
| Charles Darwin    | Non-Fiction
| Charles Dickens   | Classic Fiction
| William Shakespeare | Play
+-----+-----+
5 rows in set (0.00 sec)

mysql>

```

Figure 8-15. Updating columns in the classics table

In the first query, Mark Twain's real name of Samuel Langhorne Clemens was appended to his pen name in parentheses, which affected only one row. The second query, however, affected three rows, because it changed all occurrences of the word *Fiction* in the *category* column to the term *Classic Fiction*.

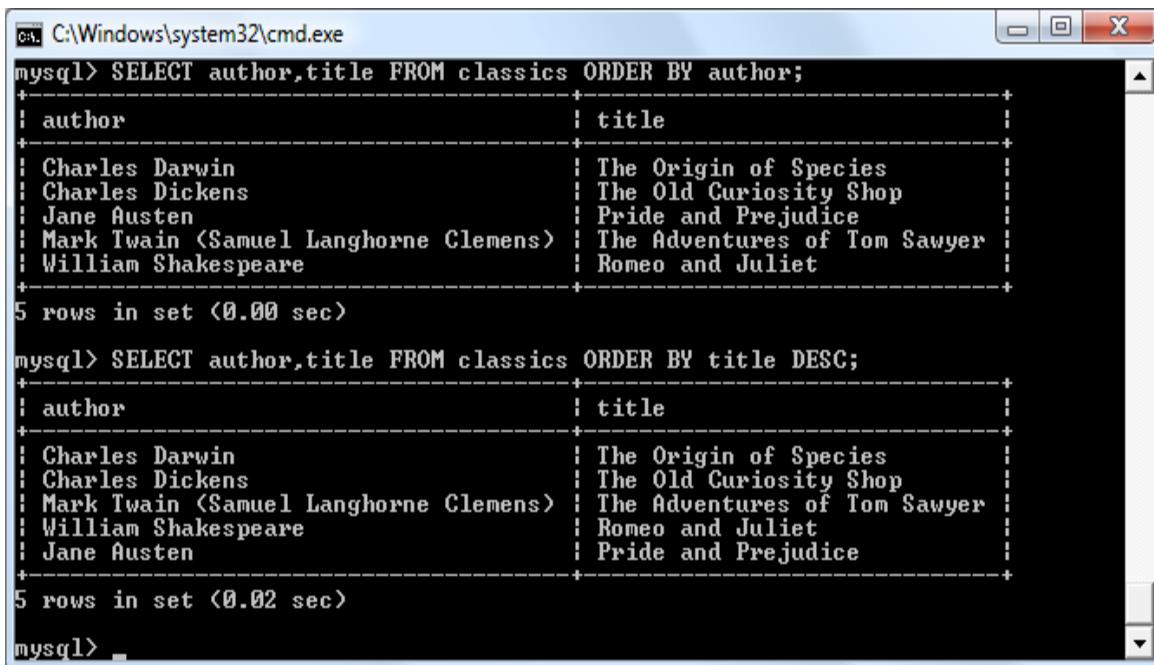
When performing an update, you can also make use of the qualifiers you have already seen, such as **LIMIT**, and the following **ORDER BY** and **GROUP BY** keywords.

## ORDER BY

**ORDER BY** sorts returned results by one or more columns in ascending or descending order. [Example 8-27](#) shows two such queries, the results of which can be seen in [Figure 8-16](#).

[Example 8-27. Using ORDER BY](#)

```
SELECT author,title FROM classics ORDER BY author;
SELECT author,title FROM classics ORDER BY title DESC;
```



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. It contains two MySQL queries and their results. The first query is 'SELECT author,title FROM classics ORDER BY author;'. The second query is 'SELECT author,title FROM classics ORDER BY title DESC;'. Both queries return 5 rows in less than a second. The results are displayed in a table format with 'author' in the first column and 'title' in the second column.

author	title
Charles Darwin	The Origin of Species
Charles Dickens	The Old Curiosity Shop
Jane Austen	Pride and Prejudice
Mark Twain <Samuel Langhorne Clemens>	The Adventures of Tom Sawyer
William Shakespeare	Romeo and Juliet

author	title
Charles Darwin	The Origin of Species
Charles Dickens	The Old Curiosity Shop
Mark Twain <Samuel Langhorne Clemens>	The Adventures of Tom Sawyer
William Shakespeare	Romeo and Juliet
Jane Austen	Pride and Prejudice

Figure 8-16. Sorting the results of requests

As you can see, the first query returns the publications by *author* in ascending alphabetical order (the default), and the second returns them by *title* in descending order.

If you wanted to sort all the rows by *author* and then by descending *year of publication* (to view the most recent first), you would issue the following query:

```
SELECT author,title,year FROM classics ORDER BY author,year DESC;
```

This shows that each ascending and descending qualifier applies to a single column. The DESC keyword applies only to the preceding column, *year*. Because you allow *author* to use the default sort order, it is sorted in ascending order. You could also have explicitly specified ascending order for that column, with the same results:

```
SELECT author,title,year FROM classics ORDER BY author ASC,year DESC;
```

## GROUP BY

In a similar fashion to ORDER BY, you can group results returned from queries using GROUP BY, which is good for retrieving information about a group of data. For example, if you want to know how many publications there are of each category in the *classics* table, you can issue the following query:

```
SELECT category,COUNT(author) FROM classics GROUP BY category;
```

which returns the following output:

```
+-----+-----+
| category | COUNT(author) |
+-----+-----+
| Classic Fiction |      3 |
| Non-Fiction    |      1 |
| Play           |      1 |
+-----+
3 rows in set (0.00 sec)
```

## Joining Tables Together

It is quite normal to maintain multiple tables within a database, each holding a different type of information. For example, consider the case of a *customers* table that needs to be able to be cross-referenced with publications purchased from the *classics* table. Enter the commands in **Example 8-28** to create this new table and populate it with three customers and their purchases. **Figure 8-17** shows the result.

*Example 8-28. Creating and populating the customers table*

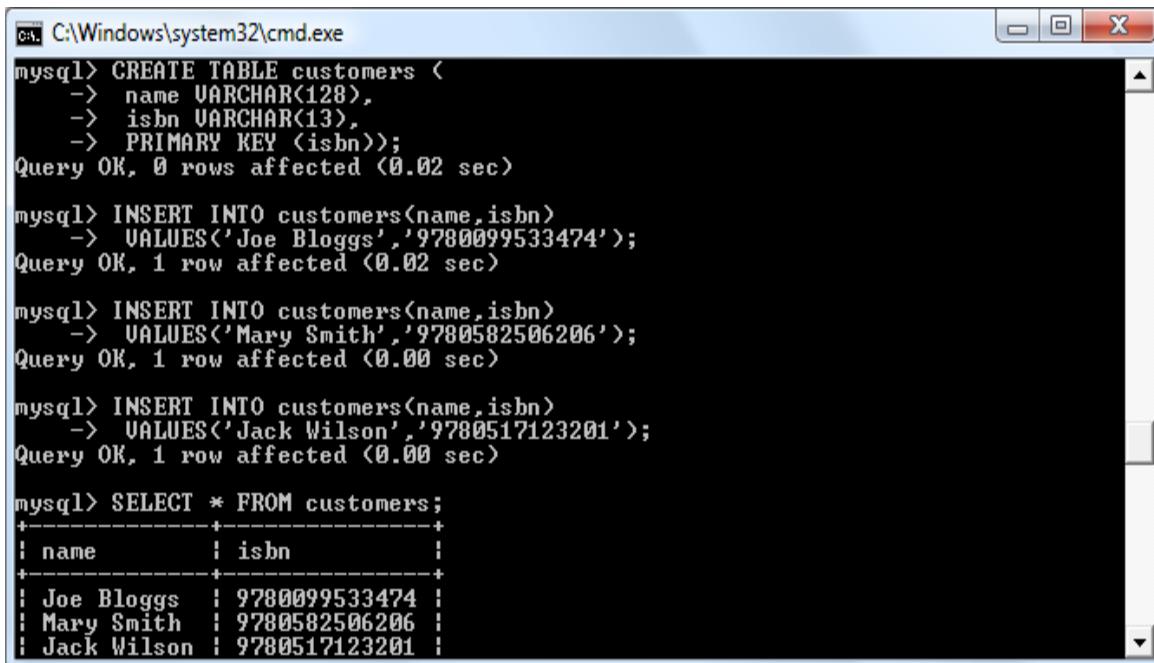
---

```
CREATE TABLE customers (
  name VARCHAR(128),
```

```

isbn VARCHAR(13),
PRIMARY KEY (isbn)) ENGINE InnoDB;
INSERT INTO customers(name,isbn)
VALUES('Joe Bloggs','9780099533474');
INSERT INTO customers(name,isbn)
VALUES('Mary Smith','9780582506206');
INSERT INTO customers(name,isbn)
VALUES('Jack Wilson','9780517123201');
SELECT * FROM customers;

```



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. Inside, MySQL commands are run:

```

mysql> CREATE TABLE customers (
    -> name VARCHAR(128),
    -> isbn VARCHAR(13),
    -> PRIMARY KEY (isbn));
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO customers(name,isbn)
    -> VALUES('Joe Bloggs','9780099533474');
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO customers(name,isbn)
    -> VALUES('Mary Smith','9780582506206');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO customers(name,isbn)
    -> VALUES('Jack Wilson','9780517123201');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM customers;
+-----+-----+
| name | isbn |
+-----+-----+
| Joe Bloggs | 9780099533474 |
| Mary Smith | 9780582506206 |
| Jack Wilson | 9780517123201 |

```

Figure 8-17. Creating the customers table

### NOTE

There's also a shortcut for inserting multiple rows of data, as in [Example 8-28](#), in which you can replace the three separate `INSERT INTO` queries with a single one listing the data to be inserted, separated by commas, like this:

```

INSERT INTO customers(name,isbn) VALUES
('Joe Bloggs','9780099533474'),
('Mary Smith','9780582506206'),
('Jack Wilson','9780517123201');

```

Of course, in a proper table containing customers' details there would also be addresses, phone numbers, email addresses, and so on, but they aren't necessary for this explanation. While creating the new table, you should have noticed that it has something in common with the *classics* table: a column called *isbn*. Because it has the same meaning in both tables (an ISBN refers to a book, and always the same book), we can use this column to tie the two tables together into a single query, as in [Example 8-29](#).

#### Example 8-29. Joining two tables into a single SELECT

---

```
SELECT name,author,title FROM customers,classics  
WHERE customers.isbn=classics.isbn;
```

The result of this operation is the following:

```
+-----+-----+-----+  
| name      | author        | title       |  
+-----+-----+-----+  
| Joe Bloggs | Charles Dickens | The Old Curiosity Shop |  
| Mary Smith | Jane Austen    | Pride and Prejudice |  
| Jack Wilson | Charles Darwin  | The Origin of Species |  
+-----+-----+-----+  
3 rows in set (0.00 sec)
```

See how this query has neatly linked the tables together to show the publications purchased from the *classics* table by the people in the *customers* table?

## NATURAL JOIN

Using NATURAL JOIN, you can save yourself some typing and make queries a little clearer. This kind of join takes two tables and automatically joins columns that have the same name. So, to achieve the same results as from [Example 8-29](#), you would enter the following:

```
SELECT name,author,title FROM customers NATURAL JOIN classics;
```

## JOIN...ON

If you wish to specify the column on which to join two tables, use the JOIN...ON construct, as follows, to achieve results identical to those of [Example 8-29](#):

```
SELECT name,author,title FROM customers
  JOIN classics ON customers.isbn=classics.isbn;
```

## Using AS

You can also save yourself some typing and improve query readability by creating aliases using the AS keyword. Simply follow a table name with AS and the alias to use. The following code, therefore, is also identical in action to [Example 8-29](#):

```
SELECT name,author,title from
  customers AS cust, classics AS class WHERE cust.isbn=class.isbn;
```

The result of this operation is the following:

```
+-----+-----+-----+
| name      | author        | title       |
+-----+-----+-----+
| Joe Bloggs | Charles Dickens | The Old Curiosity Shop |
| Mary Smith | Jane Austen    | Pride and Prejudice   |
| Jack Wilson | Charles Darwin  | The Origin of Species |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

You can also use AS to rename a column (whether or not joining tables), like this:

```
SELECT name AS customer FROM customers ORDER BY customer;
```

Which results in the following output:

```
+-----+
| customer      |
+-----+
| Jack Wilson  |
| Joe Bloggs   |
| Mary Smith   |
+-----+
3 rows in set (0.00 sec)
```

Aliases can be particularly useful when you have long queries that reference the same table names many times.

## Using Logical Operators

You can also use the logical operators AND, OR, and NOT in your MySQL WHERE queries to further narrow down your selections. [Example 8-30](#) shows one instance of each, but you can mix and match them in any way you need.

### Example 8-30. Using logical operators

---

```
SELECT author,title FROM classics WHERE
  author LIKE "Charles%" AND author LIKE "%Darwin";
SELECT author,title FROM classics WHERE
  author LIKE "%Mark Twain%" OR author LIKE "%Samuel Langhorne Clemens%";
SELECT author,title FROM classics WHERE
  author LIKE "Charles%" AND author NOT LIKE "%Darwin";
```

I've chosen the first query because Charles Darwin might be listed in some rows by his full name, Charles Robert Darwin. The query returns any publications for which the *author* column starts with *Charles* and ends with *Darwin*. The second query searches for publications written using either Mark Twain's pen name or his real name, Samuel Langhorne Clemens. The third query returns publications written by authors with the first name Charles but not the surname Darwin.

## MySQL Functions

You might wonder why anyone would want to use MySQL functions when PHP comes with a whole bunch of powerful functions of its own. The answer is very simple: the MySQL functions work on the data right there in the database. If you were to use PHP, you would first have to extract raw data from MySQL, manipulate it, and then perform the database query you wanted.

Having functions built into MySQL substantially reduces the time needed for performing complex queries, as well as their complexity. You can learn more about all the available **string** and **date/time** functions from the documentation.

## Accessing MySQL via phpMyAdmin

Although to use MySQL you have to learn these main commands and how they work, once you understand them, it can be much quicker and simpler to use a program such as *phpMyAdmin* to manage your databases and tables.

To do this, assuming you have installed AMPPS as described in [Chapter 2](#), type the following to open up the program (see [Figure 8-18](#)):

`http://localhost/phpmyadmin`

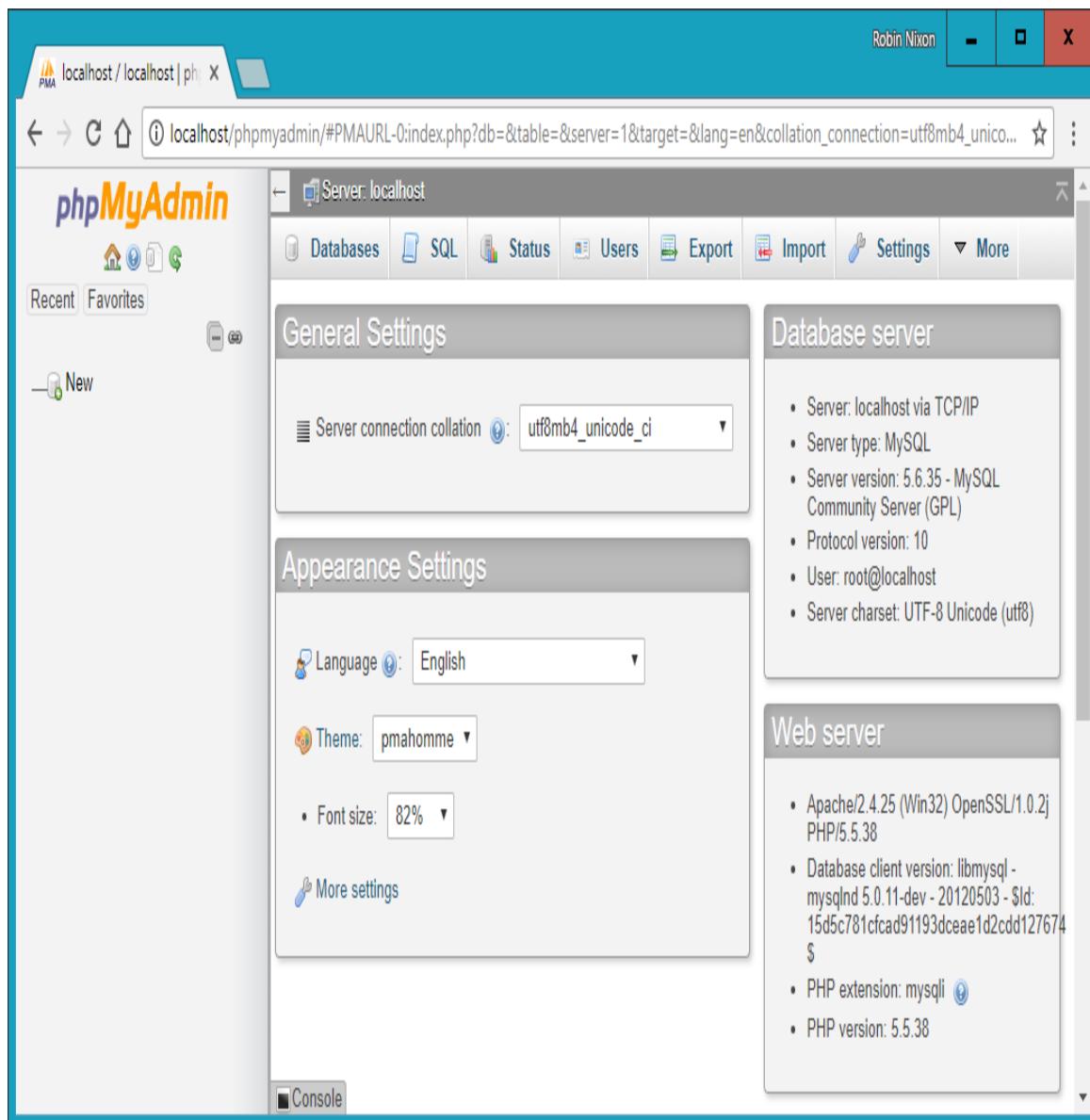


Figure 8-18. The phpMyAdmin main screen

In the left-hand pane of the main phpMyAdmin screen, you can click to select any tables you wish to work with (although none will be available until created). You can also click New to create a new database.

From here, you can perform all the main operations, such as creating new databases, adding tables, creating indexes, and much more. To find out more about phpMyAdmin, consult the [documentation](#).

If you worked with me through the examples in this chapter, congratulations—it has been quite a long journey. You've come all the way

from learning how to create a MySQL database, through issuing complex queries that combine multiple tables, to using Boolean operators and leveraging MySQL's various qualifiers.

In the next chapter, we'll start looking at how to approach efficient database design, advanced SQL techniques, and MySQL functions and transactions.

## Questions

1. What is the purpose of the semicolon in MySQL queries?
2. Which command would you use to view the available databases or tables?
3. How would you create a new MySQL user on the local host called *newuser* with a password of *newpass* and with access to everything in the database *newdatabase*?
4. How can you view the structure of a table?
5. What is the purpose of a MySQL index?
6. What benefit does a FULLTEXT index provide?
7. What is a stopword?
8. Both `SELECT DISTINCT` and `GROUP BY` cause the display to show only one output row for each value in a column, even if multiple rows contain that value. What are the main differences between `SELECT DISTINCT` and `GROUP BY`?
9. Using the `SELECT...WHERE` construct, how would you return only rows containing the word *Langhorne* somewhere in the *author* column of the *classics* table used in this chapter?
10. What needs to be defined in two tables to make it possible for you to join them together?

See “[Chapter 8 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 9. Mastering MySQL

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

Chapter 8 provided you with a good grounding in the practice of using relational databases with the Structured Query Language. You’ve learned about creating databases and the tables they comprise, as well as inserting, looking up, changing, and deleting data.

With that knowledge under your belt, we now need to look at how to design databases for maximum speed and efficiency. For example, how do you decide what data to place in which table? Well, over the years, a number of guidelines have been developed that—if you follow them—ensure that your databases will be efficient and capable of growing as you feed them more and more data.

## Database Design

It’s very important that you design a database correctly before you start to create it; otherwise, you are almost certainly going to have to go back and change it by splitting up some tables, merging others, and moving various

columns about in order to achieve sensible relationships that MySQL can easily use.

Sitting down with a sheet of paper and a pencil and writing down a selection of the queries that you think you and your users are likely to ask is an excellent starting point. In the case of an online bookstore's database, some of your questions could be:

- How many authors, books, and customers are in the database?
- Which author wrote a certain book?
- Which books were written by a certain author?
- What is the most expensive book?
- What is the best-selling book?
- Which books have not sold this year?
- Which books did a certain customer buy?
- Which books have been purchased along with the same other books?

Of course, there are many more queries that you could make on such a database, but even this small sample will begin to give you insights into how to lay out your tables. For example, books and ISBNs can probably be combined into one table, because they are closely linked (we'll examine some of the subtleties later). In contrast, books and customers should be in separate tables, because their connection is very loose. A customer can buy any book, and even multiple copies of a book, yet a book can be bought by many customers and be ignored by still more potential customers.

When you plan to do a lot of searches on something, it can often benefit by having its own table. And when couplings between things are loose, it's best to put them in separate tables.

Taking into account those simple rules of thumb, we can guess we'll need at least three tables to accommodate all these queries:

## *Authors*

There will be lots of searches for authors, many of whom have collaborated on titles, and many of whom will be featured in collections. Listing all the information about each author together, linked to that author, will produce optimal results for searches—hence an *Authors* table.

## *Books*

Many books appear in different editions. Sometimes they change publisher and sometimes they have the same titles as other, unrelated books. So, the links between books and authors are complicated enough to call for a separate table.

## *Customers*

It's even more clear why customers should get their own table, as they are free to purchase any book by any author.

# **Primary Keys: The Keys to Relational Databases**

Using the power of relational databases, we can define information for each author, book, and customer in just one place. Obviously, what interests us is the links between them—such as who wrote each book and who purchased it—but we can store that information just by making links between the three tables. I'll show you the basic principles, and then it just takes practice for it to feel natural.

The magic involves giving every author a unique identifier. We'll do the same for every book and for every customer. We saw the means of doing that in the previous chapter: the *primary key*. For a book, it makes sense to use the ISBN, although you then have to deal with multiple editions that have different ISBNs. For authors and customers, you can just assign arbitrary keys, which the `AUTO_INCREMENT` feature that you saw in the last chapter makes easy.

In short, every table will be designed around some object that you’re likely to search for a lot—an author, book, or customer, in this case—and that object will have a primary key. Don’t choose a key that could possibly have the same value for different objects. The ISBN is a rare case for which an industry has provided a primary key that you can rely on to be unique for each product. Most of the time, you’ll create an arbitrary key for this purpose, using AUTO\_INCREMENT.

## Normalization

The process of separating your data into tables and creating primary keys is called *normalization*. Its main goal is to make sure each piece of information appears in the database only once. Duplicating data is inefficient, because it makes databases larger than they need to be and therefore slows access. More importantly, the presence of duplicates creates a strong risk that you’ll update only one row of duplicated data, creating inconsistencies in a database and potentially causing serious errors.

For example, if you list the titles of books in the *Authors* table as well as the *Books* table, and you have to correct a typographic error in a title, you’ll have to search through both tables and make sure you make the same change every place the title is listed. It’s better to keep the title in one place and use the ISBN in other places.

But in the process of splitting a database into multiple tables, it’s important not to go too far and create more tables than is necessary, which would also lead to inefficient design and slower access.

Luckily, E. F. Codd, the inventor of the relational model, analyzed the concept of normalization and split it into three separate schemas called *First*, *Second*, and *Third Normal Form*. If you modify a database to satisfy each of these forms in order, you will ensure that your database is optimally balanced for fast access and minimum memory and disk space usage.

To see how the normalization process works, let’s start with the rather monstrous database in [Table 9-1](#), which shows a single table containing all

of the author names, book titles, and (fictional) customer details. You could consider it a first attempt at a table intended to keep track of which customers have ordered books. Obviously this is an inefficient design, because data is duplicated all over the place (duplications are highlighted), but it represents a starting point.

*Table 9-1. A highly inefficient design for a database table*

Author 1		Author 2		Title	ISBN	P ri c e	Custom er name	Customer address	Purcha se date
David Sklar	Adam Trachtenberg			PHP Cookboo k	0596 1010 15	.44 .9 .9	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Danny Goodman				Dynamic HTML	0596 5274 03	.59 .9 .9	Darren Ryder	<b>4758 Emily Drive, Richmond, VA 23219</b>	<b>Dec 19 2008</b>
Hugh E. William s	David Lane			PHP and MySQL	0596 0054 36	.44 .9 .5	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
David Sklar	Adam Trachtenberg			PHP Cookboo k	0596 1010 15	.44 .9 .9	Darren Ryder	<b>4758 Emily Drive, Richmond, VA 23219</b>	<b>Dec 19 2008</b>
Rasmus Lerdorf	Kevin Tatroe & Peter MacIntyre			Program ming PHP	0596 0068 15	.39 .9 .9	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

In the following three sections, we will examine this database design, and you'll see how we can improve it by removing the various duplicate entries and splitting the single table into multiple tables, each containing one type of data.

## First Normal Form

For a database to satisfy the *First Normal Form*, it must fulfill three requirements:

- There should be no repeating columns containing the same kind of data.
- All columns should contain a single value.
- There should be a primary key to uniquely identify each row.

Looking at these requirements in order, you should notice straightaway that both the *Author 1* and *Author 2* columns constitute repeating data types. So we already have a target column for pulling into a separate table, as the repeated *Author* columns violate Rule 1.

Second, there are three authors listed for the final book, *Programming PHP*. I've handled that by making Kevin Tatroe and Peter MacIntyre share the *Author 2* column, which violates Rule 2—yet another reason to transfer the *Author* details to a separate table.

However, Rule 3 is satisfied, because the primary key of ISBN has already been created.

**Table 9-2** shows the result of removing the *Author* columns from **Table 9-1**. Already it looks a lot less cluttered, although there remain duplications that are highlighted.

*Table 9-2. The result of stripping the Author columns from Table 9-1*

Title	ISBN	Pri ce	Customer name	Customer address	Purchase date
<i>PHP Cookbook</i>	0596101 015	44. 99	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Dynamic HTML	0596527 403	59. 99	Darren Ryder	<b>4758 Emily Drive, Richmond, VA 23219</b>	<b>Dec 19 2008</b>
PHP and MySQL	0596005 436	44. 95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
<i>PHP Cookbook</i>	0596101 015	44. 99	Darren Ryder	<b>4758 Emily Drive, Richmond, VA 23219</b>	<b>Dec 19 2008</b>
Programming PHP	0596006 815	39. 99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

The new *Authors* table shown in [Table 9-3](#) is small and simple. It just lists the ISBN of a title along with an author. If a title has more than one author, additional authors get their own rows. At first, you may feel ill at ease with this table, because you can't tell which author wrote which book. But don't worry: MySQL can quickly tell you. All you have to do is tell it which book you want information for, and MySQL will use its ISBN to search the *Authors* table in a matter of milliseconds.

*Table 9-3. The new Authors table*

ISBN	Author
0596101015	David Sklar
0596101015	Adam Trachtenberg
0596527403	Danny Goodman
0596005436	Hugh E. Williams
0596005436	David Lane
0596006815	Rasmus Lerdorf
0596006815	Kevin Tatroe
0596006815	Peter MacIntyre

As I mentioned earlier, the ISBN will be the primary key for the *Books* table, when we get around to creating that table. I mention that here in order to emphasize that the ISBN is not, however, the primary key for the *Authors* table. In the real world, the *Authors* table would deserve a primary key, too, so that each author would have a key to uniquely identify them.

So, in the *Authors* table, *ISBN* is just a column that—for the purposes of speeding up searches—we'll probably make a key, but not the primary key. In fact, it *cannot* be the primary key in this table, because it's not unique: the same ISBN appears multiple times whenever two or more authors have collaborated on a book.

Because we'll use it to link authors to books in another table, this column is called a *foreign* key.

## NOTE

Keys (also called *indexes*) have several purposes in MySQL. The fundamental reason for defining a key is to make searches faster. You've seen examples in [Chapter 8](#) in which keys are used in WHERE clauses for searching. But a key can also be useful to uniquely identify an item. Thus, a unique key is often used as a primary key in one table, and as a foreign key to link rows in that table to rows in another table.

## Second Normal Form

The First Normal Form deals with duplicate data (or redundancy) across multiple columns. The *Second Normal Form* is all about redundancy across multiple rows. To achieve Second Normal Form, your tables must already be in First Normal Form. Once this has been done, you achieve Second Normal Form by identifying columns whose data repeats in different places and then removing them to their own tables.

So, let's look again at [Table 9-2](#). Notice how Darren Ryder bought two books and therefore his details are duplicated. This tells us that the *Customer* columns need to be pulled into their own table. [Table 9-4](#) shows the result of removing the *Customer* columns from [Table 9-2](#).

*Table 9-4. The new Titles table*

ISBN	Title	Price
0596101015	PHP Cookbook	44.99
0596527403	Dynamic HTML	59.99
0596005436	PHP and MySQL	44.95
0596006815	Programming PHP	39.99

As you can see, all that's left in [Table 9-4](#) are the *ISBN*, *Title*, and *Price* columns for four unique books, so this now constitutes an efficient and self-contained table that satisfies the requirements of both the First and Second Normal Forms. Along the way, we've managed to reduce the information to data closely related to book titles. This table could also include years of publication, page counts, numbers of reprints, and so on, as these details are

also closely related. The only rule is that we can't put in any column that could have multiple values for a single book, because then we'd have to list the same book in multiple rows and would thus violate Second Normal Form. Restoring an *Author* column, for instance, would violate this normalization.

However, looking at the extracted *Customer* columns, now in [Table 9-5](#), we can see that there's still more normalization work to do, because Darren Ryder's details are still duplicated. And it could also be argued that First Normal Form Rule 2 (all columns should contain a single value) has not been properly complied with, because the addresses really need to be broken into separate columns for *Address*, *City*, *State*, and *Zip*.

*Table 9-5. The customer details from Table 9-2*

ISBN	Customer name	Customer address	Purchase date
059610101 5	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
059652740 3	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
059600543 6	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
059610101 5	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
059600681 5	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

What we have to do is split this table further to ensure that each customer's details are entered only once. Because the ISBN is not and cannot be used as a primary key to identify customers (or authors), a new key must be created.

[Table 9-6](#) is the result of normalizing the *Customers* table into both First and Second Normal Forms. Each customer now has a unique customer number called *CustNo* that is the table's primary key, and that will most likely have been created via `AUTO_INCREMENT`. All the parts of customer

addresses have also been separated into distinct columns to make them easily searchable and updateable.

*Table 9-6. The new Customers table*

CustNo	Name	Address	City	State	Zip
1	Emma Brown	1565 Rainbow Road	Los Angeles	CA	90014
2	Darren Ryder	4758 Emily Drive	Richmond	VA	23219
3	Earl B. Thurston	862 Gregory Lane	Frankfort	KY	40601
4	David Miller	3647 Cedar Lane	Waltham	MA	02154

At the same time, in order to normalize [Table 9-6](#), we had to remove the information on customer purchases, because otherwise, there would be multiple instances of customer details for each book purchased. Instead, the purchase data is now placed in a new table called *Purchases* (see [Table 9-7](#)).

*Table 9-7. The new Purchases table*

CustNo	ISBN	Date
1	0596101015	Mar 03 2009
2	0596527403	Dec 19 2008
2	0596101015	Dec 19 2008
3	0596005436	Jun 22 2009
4	0596006815	Jan 16 2009

Here the *CustNo* column from [Table 9-6](#) is reused as a key to tie the *Customers* and *Purchases* tables together. Because the *ISBN* column is also repeated here, this table can be linked with the *Authors* and *Titles* tables, too.

The *CustNo* column may be a useful key in the *Purchases* table, but it's not a primary key. A single customer can buy multiple books (and even multiple copies of one book), so the *CustNo* column is not a primary key. In

fact, the *Purchases* table has no primary key. That's all right, because we don't expect to need to keep track of unique purchases. If one customer buys two copies of the same book on the same day, we'll just allow two rows with the same information. For easy searching, we can define both *CustNo* and *ISBN* as keys—just not as primary keys.

### NOTE

There are now four tables, one more than the three we had initially assumed would be needed. We arrived at this decision through the normalization process, by methodically following the First and Second Normal Form rules, which made it plain that a fourth table called *Purchases* would also be required.

The tables we now have are *Authors* ([Table 9-3](#)), *Titles* ([Table 9-4](#)), *Customers* ([Table 9-6](#)), and *Purchases* ([Table 9-7](#)), and we can link each table to any other using either the *CustNo* or the *ISBN* key.

For example, to see which books Darren Ryder has purchased, you can look him up in [Table 9-6](#), the *Customers* table, where you will see his *CustNo* is 2. Armed with this number, you can now go to [Table 9-7](#), the *Purchases* table; looking at the *ISBN* column here, you will see that he purchased titles 0596527403 and 0596101015 on December 19, 2008. This looks like a lot of trouble for a human, but it's not so hard for MySQL.

To determine what these titles were, you can then refer to [Table 9-4](#), the *Titles* table, and see that the books he bought were *Dynamic HTML* and *PHP Cookbook*. Should you wish to know the authors of these books, you could also use the ISBNs you just looked up on [Table 9-3](#), the *Authors* table, and you would see that ISBN 0596527403, *Dynamic HTML*, was written by Danny Goodman, and that ISBN 0596101015, *PHP Cookbook*, was written by David Sklar and Adam Trachtenberg.

## Third Normal Form

Once you have a database that complies with both the First and Second Normal Forms, it is in pretty good shape and you might not have to modify

it any further. However, if you wish to be very strict with your database, you can ensure that it adheres to the *Third Normal Form*, which requires that data that is *not* directly dependent on the primary key but *is* dependent on another value in the table should also be moved into separate tables, according to the dependence.

For example, in [Table 9-6](#), the *Customers* table, it could be argued that the *State*, *City*, and *Zip* keys are not directly related to each customer, because many other people will have the same details in their addresses, too. However, they are directly related to each other, in that the *Address* relies on the *City*, and the *City* relies on the *State*.

Therefore, to satisfy Third Normal Form for [Table 9-6](#), you would need to split it into Tables [9-8](#) through [9-11](#).

*Table 9-8. Third Normal Form Customers table*

CustNo	Name	Address	Zip
1	Emma Brown	1565 Rainbow Road	90014
2	Darren Ryder	4758 Emily Drive	23219
3	Earl B. Thurston	862 Gregory Lane	40601
4	David Miller	3647 Cedar Lane	02154

*Table 9-9.*

*Third  
Normal  
Form Zip  
codes table*

<b>Zip</b>	<b>CityID</b>
90014	1234
23219	5678
40601	4321
02154	8765

*Table 9-10.*

*Third Normal  
Form Cities table*

<b>CityID</b>	<b>Name</b>	<b>StateID</b>
1234	Los Angeles	5
5678	Richmond	46
4321	Frankfort	17
8765	Waltham	21

*Table 9-11.*

*Third Normal Form States  
table*

<b>StateID</b>	<b>Name</b>	<b>Abbreviation</b>
5	California	CA
46	Virginia	VA
17	Kentucky	KY
21	Massachusetts	MA

So, how would you use this set of four tables instead of the single [Table 9-6](#)? Well, you would look up the *Zip code* in [Table 9-8](#), and then find the matching *CityID* in [Table 9-9](#). Given this information, you could look up the city *Name* in [Table 9-10](#) and then also find the *StateID*, which you could use in [Table 9-11](#) to look up the state's *Name*.

Although using the Third Normal Form in this way may seem like overkill, it can have advantages. For example, take a look at [Table 9-11](#), where it has been possible to include both a state's name and its two-letter abbreviation. It could also contain population details and other demographics, if you desired.

### NOTE

[Table 9-10](#) could also contain even more localized demographics that could be useful to you and/or your customers. By splitting up these pieces of data, you can make it easier to maintain your database in the future, should it be necessary to add columns.

Deciding whether to use the Third Normal Form can be tricky. Your evaluation should rest on what data you may need to add at a later date. If you are absolutely certain that the name and address of a customer is all that you will ever require, you probably will want to leave out this final normalization stage.

On the other hand, suppose you are writing a database for a large organization such as the US Postal Service. What would you do if a city were to be renamed? With a table such as [Table 9-6](#), you would need to perform a global search-and-replace on every instance of that city. But if you have your database set up according to the Third Normal Form, you would have to change only a single entry in [Table 9-10](#) for the change to be reflected throughout the entire database.

Therefore, I suggest that you ask yourself two questions to help you decide whether to perform a Third Normal Form normalization on any table:

- Is it likely that many new columns will need to be added to this table?
- Could any of this table's fields require a global update at any point?

If either of the answers is yes, you should probably consider performing this final stage of normalization.

## When Not to Use Normalization

Now that you know all about normalization, I'm going to tell you why you should throw these rules out of the window on high-traffic sites. That's right —you should never fully normalize your tables on sites that will cause MySQL to thrash.

Normalization requires spreading data across multiple tables, and this means making multiple calls to MySQL for each query. On a very popular site, if you have normalized tables, your database access will slow down considerably once you get above a few dozen concurrent users, because they will be creating hundreds of database accesses between them. In fact, I would go so far as to say you should *denormalize* any commonly looked-up data as much as you can.

You see, if you have data duplicated across your tables, you can substantially reduce the number of additional requests that need to be made, because most of the data you want is available in each table. This means that you can simply add an extra column to a query and that field will be available for all matching results.

Of course, you have to deal with the downsides previously mentioned, such as using up large amounts of disk space, and ensuring that you update every single duplicate copy of the data when it needs modifying.

Multiple updates can be computerized, though. MySQL provides a feature called *triggers* that make automatic changes to the database in response to changes you make. (Triggers are, however, beyond the scope of this book.) Another way to propagate redundant data is to set up a PHP program to run

regularly and keep all copies in sync. The program reads changes from a “master” table and updates all the others. (You’ll see how to access MySQL from PHP in the next chapter.)

However, until you are very experienced with MySQL, I recommend that you fully normalize all your tables (at least to First and Second Normal Form), as this will instil the habit and put you in good stead. Only when you actually start to see MySQL logjams should you consider looking at denormalization.

## Relationships

MySQL is called a *relational* database management system because its tables store not only data but the *relationships* among the data. There are three categories of relationships.

### One-to-One

A *one-to-one relationship* is like a (traditional) marriage: each item has a relationship to only one item of the other type. This is surprisingly rare. For instance, an author can write multiple books, a book can have multiple authors, and even an address can be associated with multiple customers. Perhaps the best example in this chapter so far of a one-to-one relationship is the relationship between the name of a state and its two-character abbreviation.

However, for the sake of argument, let’s assume that there can always be only one customer at any address. In such a case, the Customers–Addresses relationship in [Figure 9-1](#) is a one-to-one relationship: only one customer lives at each address, and each address can have only one customer.

*Table 9-8a (Customers)**Table 9-8b (Addresses)*

CustNo	Name	Address	Zip
1	Emma Brown .....	1565 Rainbow Road	90014
2	Darren Ryder .....	4758 Emily Drive	23219
3	Earl B. Thurston .....	862 Gregory Lane	40601
4	David Miller.....	3647 Cedar Lane	02154

*Figure 9-1. The Customers table, Table 9-8, split into two tables*

Usually, when two items have a one-to-one relationship, you just include them as columns in the same table. There are two reasons for splitting them into separate tables:

- You want to be prepared in case the relationship changes later and is no longer one-to-one.
- The table has a lot of columns, and you think that performance or maintenance would be improved by splitting it.

Of course, when you build your own databases in the real world, you will have to create one-to-many Customer–Address relationships (*one address, many customers*).

## One-to-Many

*One-to-many* (or *many-to-one*) relationships occur when one row in one table is linked to many rows in another table. You have already seen how Table 9-8 would take on a one-to-many relationship if multiple customers

were allowed at the same address, which is why it would have to be split up if that were the case.

So, looking at Table 9-8a within [Figure 9-1](#), you can see that it shares a one-to-many relationship with [Table 9-7](#) because there is only one of each customer in Table 9-8a. However [Table 9-7](#), the *Purchases* table, can (and does) contain more than one purchase from a given customer. Therefore, *one* customer has a relationship with *many* purchases.

You can see these two tables alongside each other in [Figure 9-2](#), where the dashed lines joining rows in each table start from a single row in the lefthand table but can connect to more than one row in the righthand table. This one-to-many relationship is also the preferred scheme to use when describing a many-to-one relationship, in which case you would normally swap the left and right tables to view them as a one-to-many relationship.

*Table 9-8a (Customers)*

CustNo	Name
1	Emma Brown .....
2	Darren Ryder .....
	(etc...)
3	Earl B. Thurston .....
4	David Miller .....

*Table 9-7. (Purchases)*

CustNo	ISBN	Date
1	0596101015	Mar 03 2009
2	0596527403	Dec 19 2008
	0596101015	Dec 19 2008
3	0596005436	Jun 22 2009
4	0596006815	Jan 16 2009

*Figure 9-2. Illustrating the relationship between two tables*

To represent a one-to-many relationship in a relational database, create a table for the “many” and a table for the “one.” The table for the “many” must contain a column that lists the primary key from the “one” table. Thus, the *Purchases* table will contain a column that lists the primary key of the customer.

## Many-to-Many

In a *many-to-many relationship*, many rows in one table are linked to many rows in another table. To create this relationship, add a third table containing the same key column from each of the other tables. This third table contains nothing else, as its sole purpose is to link up the other tables.

**Table 9-12** is just such a table. It was extracted from **Table 9-7**, the *Purchases* table, but omits the purchase date information. It contains a copy of the ISBN of every title sold, along with the customer number of each purchaser.

*Table 9-12. An intermediary table*

Cust No	ISBN
1	0596101015
2	0596527403
2	0596101015
3	0596005436
4	0596006815

With this intermediary table in place, you can traverse all the information in the database through a series of relations. You can take an address as a starting point and find out the authors of any books purchased by the customer living at that address.

For example, let’s suppose that you want to find out about purchases in the 23219 zip code. Look that zip code up in Table 9-8b, and you’ll find that customer number 2 has bought at least one item from the database. At this

point, you can use Table 9-8a to find out that customer's name, or use the new intermediary **Table 9-12** to see the book(s) purchased.

From here, you will find that two titles were purchased and can follow them back to **Table 9-4** to find the titles and prices of these books, or to **Table 9-3** to see who the authors were.

If it seems to you that this is really combining multiple one-to-many relationships, then you are absolutely correct. To illustrate, **Figure 9-3** brings three tables together.

<i>Columns from Table 9-8 (Customers)</i>		<i>Intermediary Table 9-12 (Customer/ISBN)</i>		<i>Columns from Table 9-4 (Titles)</i>	
Zip	CustNo	CustNo	ISBN	ISBN	Title
90014	1	1	0596101015	0596101015	PHP Cookbook
23219	2	2	0596101015	(etc...)	
(etc...)		2	0596527403	0596527403	Dynamic HTML
40601	3	3	0596005436	0596005436	PHP and MySQL
02154	4	4	0596006815	0596006815	Programming PHP

*Figure 9-3. Creating a many-to-many relationship via a third table*

Follow any zip code in the lefthand table to the associated customer IDs. From there, you can link to the middle table, which joins the left and right

tables by linking customer IDs and ISBNs. Now all you have to do is follow an ISBN over to the righthand table to see which book it relates to.

You can also use the intermediary table to work your way backward from book titles to zip codes. The *Titles* table can tell you the ISBN, which you can use in the middle table to find the ID numbers of customers who bought the book, and finally you can use the *Customers* table to match the customer ID numbers to the customers' zip codes.

## Databases and Anonymity

An interesting aspect of using relations is that you can accumulate a lot of information about some item—such as a customer—without actually knowing who that customer is. Note that in the previous example we went from customers' zip codes to customers' purchases, and back again, without finding out the name of a customer. Databases can be used to track people, but they can also be used to help preserve people's privacy while still finding useful information.

## Transactions

In some applications, it is vitally important that a sequence of queries runs in the correct order and that every single query successfully completes. For example, suppose that you are creating a sequence of queries to transfer funds from one bank account to another. You would not want either of the following events to occur:

- You add the funds to the second account, but when you try to subtract them from the first account, the update fails, and now both accounts have the funds.
- You subtract the funds from the first bank account, but the update request to add them to the second account fails, and the funds have disappeared into thin air.

As you can see, not only is the order of queries important in this type of transaction, but it is also vital that all parts of the transaction complete successfully. But how can you ensure this happens, because surely after a query has occurred, it cannot be undone? Do you have to keep track of all parts of a transaction and then undo them all one at a time if any one fails? The answer is absolutely not, because MySQL comes with powerful transaction-handling features to cover just these types of eventualities.

In addition, transactions allow concurrent access to a database by many users or programs at the same time. MySQL handles this seamlessly by ensuring that all transactions are queued and that users or programs take their turns and don't tread on each other's toes.

## Transaction Storage Engines

To be able to use MySQL's transaction facility, you have to be using MySQL's InnoDB storage engine (which is the default from version 5.5 onward). If you are not sure which version of MySQL your code will be running on, rather than assuming InnoDB is the default engine you can force its use when creating a table, as follows.

Create a table of bank accounts by typing the commands in [Example 9-1](#). (Remember that to do this, you will need access to the MySQL command line, and must also have already selected a suitable database in which to create this table.)

*Example 9-1. Creating a transaction-ready table*

---

```
CREATE TABLE accounts (
    number INT, balance FLOAT, PRIMARY KEY(number)
) ENGINE InnoDB;
DESCRIBE accounts;
```

The final line of this example displays the contents of the new table so you can ensure that it was correctly created. The output from it should look like this:

```
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| number | int(11) | NO   | PRI  | 0       |       |
| balance | float   | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Now let's create two rows within the table so that you can practice using transactions. Type the commands in [Example 9-2](#).

*Example 9-2. Populating the accounts table*

---

```
INSERT INTO accounts(number, balance) VALUES(12345, 1025.50);
INSERT INTO accounts(number, balance) VALUES(67890, 140.00);
SELECT * FROM accounts;
```

The third line displays the contents of the table to confirm that the rows were correctly inserted. The output should look like this:

```
+-----+
| number | balance |
+-----+
| 12345 | 1025.5 |
| 67890 |      140 |
+-----+
2 rows in set (0.00 sec)
```

With this table created and prepopulated, you are ready to start using transactions.

## Using BEGIN

Transactions in MySQL start with either a BEGIN or a START TRANSACTION statement. Type the commands in [Example 9-3](#) to send a transaction to MySQL.

*Example 9-3. A MySQL transaction*

---

```
BEGIN;
UPDATE accounts SET balance=balance+25.11 WHERE number=12345;
COMMIT;
SELECT * FROM accounts;
```

The result of this transaction is displayed by the final line, and should look like this:

```
+-----+-----+
| number | balance |
+-----+-----+
| 12345 | 1050.61 |
| 67890 |      140 |
+-----+-----+
2 rows in set (0.00 sec)
```

As you can see, the balance of account number 12345 was increased by 25.11 and is now 1050.61. You may also have noticed the COMMIT command in [Example 9-3](#), which is explained next.

## Using COMMIT

When you are satisfied that a series of queries in a transaction has successfully completed, issue a COMMIT command to commit all the changes to the database. Until it receives a COMMIT, MySQL considers all the changes you make to be merely temporary. This feature gives you the opportunity to cancel a transaction by not sending a COMMIT but issuing a ROLLBACK command instead.

## Using ROLLBACK

Using the ROLLBACK command, you can tell MySQL to forget all the queries made since the start of a transaction and to cancel the transaction. See this in action by entering the funds transfer transaction in [Example 9-4](#).

*Example 9-4. A funds transfer transaction*

---

```
BEGIN;
UPDATE accounts SET balance=balance-250 WHERE number=12345;
UPDATE accounts SET balance=balance+250 WHERE number=67890;
SELECT * FROM accounts;
```

Once you have entered these lines, you should see the following result:

```
+-----+-----+
| number | balance |
+-----+-----+
| 12345 | 800.61 |
| 67890 |      390 |
+-----+-----+
2 rows in set (0.00 sec)
```

The first bank account now has a value that is 250 less than before, and the second has been incremented by 250; you have transferred a value of 250 between them. But let's assume that something went wrong and you wish to undo this transaction. All you have to do is issue the commands in

### Example 9-5.

#### Example 9-5. Canceling a transaction using ROLLBACK

---

```
ROLLBACK;
SELECT * FROM accounts;
```

You should now see the following output, showing that the two accounts have had their previous balances restored, due to the entire transaction being cancelled via the ROLLBACK command:

```
+-----+-----+
| number | balance |
+-----+-----+
| 12345 | 1050.61 |
| 67890 |      140 |
+-----+-----+
2 rows in set (0.00 sec)
```

# Using EXPLAIN

MySQL comes with a powerful tool for investigating how the queries you issue to it are interpreted. Using EXPLAIN, you can get a snapshot of any query to find out whether you could issue it in a better or more efficient way. [Example 9-6](#) shows how to use it with the *accounts* table you created earlier.

## Example 9-6. Using the EXPLAIN command

---

```
EXPLAIN SELECT * FROM accounts WHERE number='12345';
```

The results of this EXPLAIN command should look like the following:

```
+-----+-----+-----+-----+
| id  | select_type | table   | partitions | type   | |
| possible_keys | key      | key_len | ref     | rows   | filtered |
| Extra |           |
+-----+-----+-----+-----+
| 1  | SIMPLE    | accounts | NULL      | const  | PRIMARY
| PRIMARY | 4        | const   | 1 | 100.00 | NULL  |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

The information that MySQL is giving you here is as follows:

### *select\_type*

The selection type is SIMPLE. If you were joining tables together, this would show the join type.

### *table*

The current table being queried is accounts.

### *type*

The query type is **const**. From the least efficient to the most efficient type, the possible values can be ALL, index, range, ref, eq\_ref, const, system, and NULL.

#### *possible\_keys*

There is a possible PRIMARY key, which means that accessing should be fast.

#### *key*

The key actually used is PRIMARY. This is good.

#### *key\_len*

The key length is 4. This is the number of bytes of the index that MySQL will use.

#### *ref*

The **ref** column displays which columns or constants are used with the key. In this case, a constant key is being used.

#### *rows*

The number of rows that need to be searched by this query is 1. This is good.

Whenever you have a query that seems to be taking longer than you think it should to execute, try using EXPLAIN to see where you can optimize it. You will discover which keys (if any) are being used, their lengths, and so on, and will be able to adjust your query or the design of your table(s) accordingly.

## NOTE

When you have finished experimenting with the temporary *accounts* table, you may wish to remove it by entering the following command:

```
DROP TABLE accounts;
```

## Backing Up and Restoring

Whatever kind of data you are storing in your database, it must have some value to you, even if it's only the cost of the time required for reentering it should the hard disk fail. Therefore, it's important that you keep backups to protect your investment. Also, there will be times when you have to migrate your database over to a new server; the best way to do this is usually to back it up first. It is also important that you test your backups from time to time to ensure that they are valid and will work if they need to be used.

Thankfully, backing up and restoring MySQL data is easy with the `mysqldump` command.

### Using mysqldump

With `mysqldump`, you can dump a database or collection of databases into one or more files containing all the instructions necessary to re-create all your tables and repopulate them with your data. This command can also generate files in CSV (comma-separated values) and other delimited text formats, or even in XML format. Its main drawback is that you must make sure that no one writes to a table while you're backing it up. There are various ways to do this, but the easiest is to shut down the MySQL server before running `mysqldump` and start up the server again after `mysqldump` finishes.

Alternatively, you can lock the tables you are backing up before running `mysqldump`. To lock tables for reading (as we want to read the data), from

the MySQL command line issue this command:

```
LOCK TABLES tablename1 READ, tablename2 READ ...
```

Then, to release the lock(s), enter the following:

```
UNLOCK TABLES;
```

By default, the output from `mysqldump` is simply printed out, but you can capture it in a file through the > redirect symbol.

The basic format of the `mysqldump` command is shown here:

```
mysqldump -u user -ppassword database
```

However, before you can dump the contents of a database, you must make sure that `mysqldump` is in your path, or else specify its location as part of your command. [Table 9-13](#) shows the likely locations of the program for the different installations and operating systems covered in [Chapter 2](#). If you have a different installation, it may be in a slightly different location.

*Table 9-13. Likely locations of mysqldump for different installations*

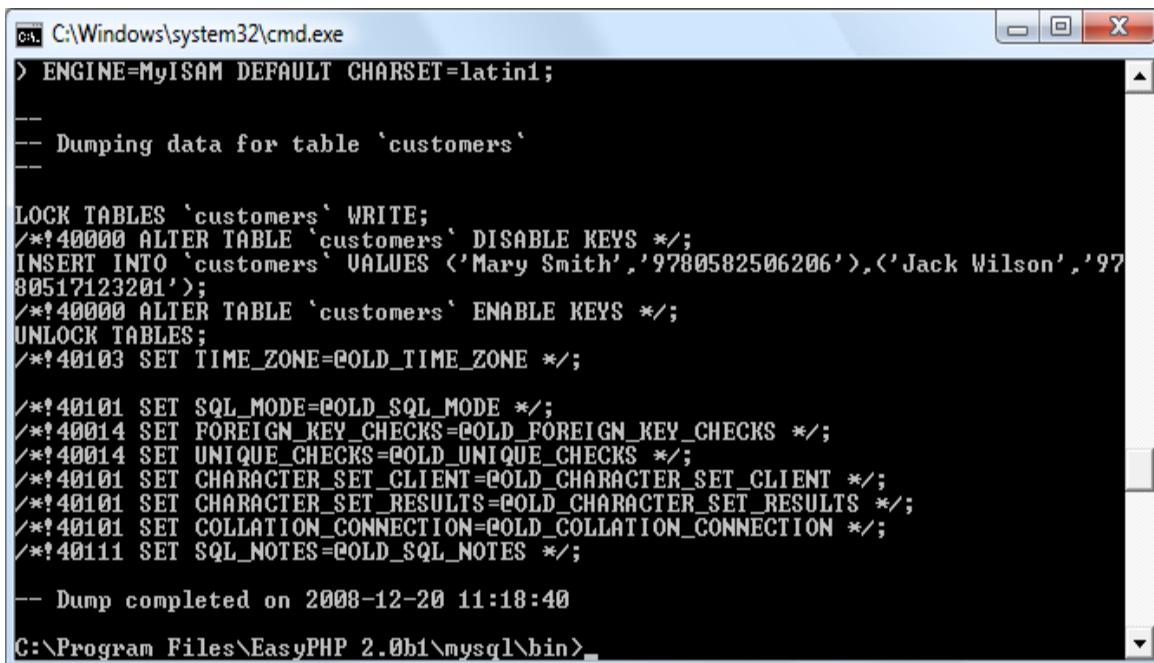
Operating system and program	Likely folder location
Windows AMPPS	C:\Program Files\Ampps\mysql\bin
macOS AMPPS	/Applications/ampps/mysql/bin
Linux AMPPS	/Applications/ampps/mysql/bin

So, to dump the contents of the *publications* database that you created in [Chapter 8](#) to the screen, first exit MySQL and then enter the command in [Example 9-7](#) (specifying the full path to `mysqldump` if necessary).

*Example 9-7. Dumping the publications database to screen*

```
mysqldump -u user -ppassword publications
```

Make sure that you replace *user* and *password* with the correct details for your installation of MySQL. If there is no password set for the user, you can omit that part of the command, but the `-u user` part is mandatory unless you have root access without a password and are executing as root (not recommended). The result of issuing this command will look something like [Figure 9-4](#).



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window contains the output of a mysqldump command. The output includes SQL code for creating a MyISAM table named 'customers' with two rows of data: ('Mary Smith', '9780582506206') and ('Jack Wilson', '9780582506201'). It also includes comments and SQL statements to disable keys, insert data, enable keys, unlock tables, and reset time zone. The command concludes with a note about the dump being completed on 2008-12-20 at 11:18:40.

```
> ENGINE=MyISAM DEFAULT CHARSET=latin1;
-- 
-- Dumping data for table 'customers'
--

LOCK TABLES `customers` WRITE;
/*!40000 ALTER TABLE `customers` DISABLE KEYS */;
INSERT INTO `customers` VALUES ('Mary Smith','9780582506206'),('Jack Wilson','9780582506201');
/*!40000 ALTER TABLE `customers` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2008-12-20 11:18:40
C:\Program Files\EasyPHP 2.0b1\mysql\bin>
```

*Figure 9-4. Dumping the publications database to the screen*

## Creating a Backup File

Now that you have `mysqldump` working, and have verified it outputs correctly to the screen, you can send the backup data directly to a file using the `>` redirect symbol. Assuming that you wish to call the backup file `publications.sql`, type the command in [Example 9-8](#) (remembering to replace *user* and *password* with the correct details).

[Example 9-8. Dumping the publications database to a file](#)

```
mysqldump -u user -p password publications > publications.sql
```

### NOTE

The command in [Example 9-8](#) stores the backup file into the current directory. If you need it to be saved elsewhere, you should insert a file path before the filename. You must also ensure that the directory you are backing up to has the right permissions set to allow the file to be written.

If you echo the backup file to screen or load it into a text editor, you will see that it comprises sequences of SQL commands such as the following:

```
DROP TABLE IF EXISTS 'classics';
CREATE TABLE 'classics' (
    'author' varchar(128) default NULL,
    'title' varchar(128) default NULL,
    'category' varchar(16) default NULL,
    'year' smallint(6) default NULL,
    'isbn' char(13) NOT NULL default '',
    PRIMARY KEY ('isbn'),
    KEY 'author' ('author'(20)),
    KEY 'title' ('title'(20)),
    KEY 'category' ('category'(4)),
    KEY 'year' ('year'),
    FULLTEXT KEY 'author_2' ('author','title')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

This is smart code that can be used to restore a database from a backup, even if it currently exists; it will first drop any tables that need to be re-created, thus avoiding potential MySQL errors.

## Backing up a single table

To back up only a single table from a database (such as the *classics* table from the *publications* database), you should first lock the table from within the MySQL command line, by issuing a command such as the following:

```
LOCK TABLES publications.classics READ;
```

This ensures that MySQL remains running for read purposes, but writes cannot be made. Then, while keeping the MySQL command line open, use another terminal window to issue the following command from the operating system command line:

```
mysqldump -u user -ppassword publications classics > classics.sql
```

You must now release the table lock by entering the following command from the MySQL command line in the first terminal window, which unlocks all tables that have been locked during the current session:

```
UNLOCK TABLES;
```

## Backing up all tables

If you want to back up all your MySQL databases at once (including the system databases such as *mysql*), you can use a command such as the one in **Example 9-9**, which would enable you to restore an entire MySQL database installation. Remember to use locking where required.

*Example 9-9. Dumping all the MySQL databases to file*

---

```
mysqldump -u user -ppassword --all-databases > all_databases.sql
```

### NOTE

Of course, there's a lot more than just a few lines of SQL code in backed-up database files. I recommend that you take a few minutes to examine a couple in order to familiarize yourself with the types of commands that appear in backup files and how they work.

## Restoring from a Backup File

To perform a restore from a file, call the `mysql` executable, passing it the file to restore from using the `<` symbol. So, to recover an entire database that you dumped using the `--all-databases` option, use a command such as that in [Example 9-10](#).

*Example 9-10. Restoring an entire set of databases*

---

```
mysql -u user -ppassword < all_databases.sql
```

To restore a single database, use the `-D` option followed by the name of the database, as in [Example 9-11](#), where the *publications* database is being restored from the backup made in [Example 9-8](#).

*Example 9-11. Restoring the publications database*

---

```
mysql -u user -ppassword -D publications < publications.sql
```

To restore a single table to a database, use a command such as that in [Example 9-12](#), where just the *classics* table is being restored to the *publications* database.

*Example 9-12. Restoring the classics table to the publications database*

---

```
mysql -u user -ppassword -D publications < classics.sql
```

## Dumping Data in CSV Format

As previously mentioned, the `mysqldump` program is very flexible and supports various types of output, such as the CSV format, which you might use to import data into a spreadsheet, among other purposes. [Example 9-13](#) shows how you can dump the data from the *classics* and *customers* tables in the *publications* database to the files *classics.txt* and *customers.txt* in the folder *c:/temp*. On macOS or Linux systems, you should modify the destination path to an existing folder.

### Example 9-13. Dumping data to CSV-format files

```
mysqldump -u user -ppassword --no-create-info --tab=c:/temp  
--fields-terminated-by=',' publications
```

This command is quite long and is shown here wrapped over two lines, but you must type it all as a single line. The result is the following:

```
Mark Twain (Samuel Langhorne Clemens)', 'The Adventures of Tom  
Sawyer',  
    'Classic Fiction', '1876', '9781598184891  
Jane Austen', 'Pride and Prejudice', 'Classic  
Fiction', '1811', '9780582506206  
Charles Darwin', 'The Origin of Species', 'Non-  
Fiction', '1856', '9780517123201  
Charles Dickens', 'The Old Curiosity Shop', 'Classic  
Fiction', '1841', '9780099533474  
William Shakespeare', 'Romeo and  
Juliet', 'Play', '1594', '9780192814968  
  
Mary Smith', '9780582506206  
Jack Wilson', '9780517123201
```

## Planning Your Backups

The golden rule to backing up is to do so as often as you find practical. The more valuable the data, the more often you should back it up, and the more copies you should make. If your database gets updated at least once a day, you should really back it up on a daily basis. If, on the other hand, it is not updated very often, you could probably get by with less frequent backups.

## NOTE

You should consider making multiple backups and storing them in different locations. If you have several servers, it is a simple matter to copy your backups between them. You would also be well advised to make physical backups on removable hard disks, thumb drives, CDs or DVDs, and so on, and to keep these in separate locations—preferably somewhere like a fireproof safe.

It's important to test restoring a database once in a while, too, to make sure your backups are done correctly. You also want to be familiar with restoring a database because you may have to do so when you are stressed and in a hurry, such as after a power failure that takes down the website. You can restore a database to a private server and run a few SQL commands to make sure the data is as you think it should be.

Once you've digested the contents of this chapter, you will be proficient in using both PHP and MySQL; the next chapter will show you how to bring these two technologies together.

## Questions

1. What does the word *relationship* mean in reference to a relational database?
2. What is the term for the process of removing duplicate data and optimizing tables?
3. What are the three rules of the First Normal Form?
4. How can you make a table satisfy the Second Normal Form?
5. What do you put in a column to tie together two tables that contain items having a one-to-many relationship?
6. How can you create a database with a many-to-many relationship?
7. What commands initiate and end a MySQL transaction?
8. What feature does MySQL provide to enable you to examine how a query will work in detail?

9. What command would you use to back up the database *publications* to a file called *publications.sql*?

See “[Chapter 9 Answers](#)” in [Appendix A](#) for the answers to these questions.

# Chapter 10. What's new in PHP 8 and MySQL 8

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

By the end of the second decade of the 21st century both PHP and MySQL had matured into their 8th versions, and can now be considered very mature products by software technology standards.

According to the [w3techs.com](https://w3techs.com) website, in 2020 PHP was now used in one way or another by almost 79% of all websites, a massive 70% ahead of its nearest rival, ASP.net, while [explore-group.com](https://explore-group.com) reported that in 2019 MySQL remained the most popular database in use on the web installed on 52% of websites. Although MySQL’s market share had slipped a little in recent years, it still remained a good 16% ahead of its nearest competitor, PostgreSQL, on 36%, and this looks like continuing to be the case in the foreseeable future.

With the latest version 8 releases of these two technologies, along with JavaScript, these mainstays of modern web development appear set to rule

the roost for many years to come. So let's take a look at what you'll find that's new in the latest versions of PHP and MySQL.

## PHP 8

Version 8 of PHP marks a major update and a huge milestone, bringing a number of new features to type, system, syntax, error handling, strings, object-oriented programming, and more.

Whilst minimising changes that could break or modify existing installations, the new features make PHP more powerful and easy to use than ever with Named Parameters, Just In Time (JIT) compilation, Attributes, and Constructor Properties, to bring major improvements and syntax changes, improved error handling, and changes and improvements in operators to help reduce the chances of overlooked bugs.

### PHP8 AND THE AMPPS STACK

At the time of writing, the AMPPS stack, which you are recommended to install in Chapter 2, comes with MySQL 8.0.18, but its version of PHP is still 7.3. During the course of publication it is anticipated that version 8 of PHP will have become included in the AMPPS stack, and you are recommended to let AMPPS handle its installation for you, especially if you are a beginner. But if it is not yet available in AMPPS, and you wish to start using PHP 8 immediately, you can always download the newer version directly from the [php.net](#) website, following the instructions provided there to install and use it.

## Named Parameters

In addition to traditional positional characters, PHP 8 allows named parameters in function calls, like this:

```
str_contains(needle: 'ian', haystack: 'Antidisestablishmentarianism');
```

This makes the function (or method) parameter names a part of the public API and allow you to pass input data into a function, based on their

argument names, instead of the argument order, substantially increasing code clarity. Incidentally, `str_contains` (explained below) is also new to PHP 8.

## Attributes

In PHP 8 attributes map to PHP class names, allowing the inclusion of meta data in classes, methods and variables. Previously you would have had to use DocBlock comments to infer them.

Attributes are a way to give one system details of another system, such as a plugin or an event system, and are simple classes annotated with the `# [Attribute]` attribute, which can be attached to classes, properties, methods, functions, class constants, or function/method parameters.

At run time Attributes do nothing and have no impact on the code. However, they are available to the reflection API, which allows other code to examine the attribute and take additional actions.

You can find a good explanation of Attributes (which are somewhat out of the remit of this book) at the following URL:

[wiki.php.net/rfc/attributes\\_v2](https://wiki.php.net/rfc/attributes_v2)

As long as they are single line, Attributes are interpreted as comments in older PHP versions and therefore are safely ignored.

## Constructor Properties

With PHP 8 you can now declare class properties right from the class constructor, saving a great deal of boilerplate coding. Take this code for example:

```
class newRecord
{
    public string $username;
    public string $email;

    public function __construct()
```

```
        string $username,
        string $email,
    ) {
    $this->username = $username;
    $this->email     = $email;
}
}
```

With Constructor Properties you can now reduce all of that to the following:

```
class newRecord
{
    public function __construct(
        public string $username,
        public string $email,
    ){}
}
```

This is a backwards-incompatible feature, so only use it where you know for sure that PHP 8 is installed.

## Just-In-Time Compilation

Just-In-Time (or JIT) Compilation is disabled by default. When enabled JIT compiles and caches native instructions to provide a performance boost to CPU-heavy applications.

You can enable JIT in the *php.ini* file like this:

```
opcache.enable      = 1
opcache.jit_buffer_size = 100M
opcache.jit          = tracing
```

Bear in mind that JIT is relatively new to PHP and that it currently makes debugging and profiling harder with the added layer. Also, there were issues with JIT right up to the day before initial release, so be wary that there may remain some undiscovered bugs in the system.

## Union Types

In PHP 8, type declarations can be extended with Union Types to declare more than one type (which also supports `false` as a special type for Boolean `false`) like this:

```
function parse_value(string|int|float): string|null {}
```

## Null-safe Operator

In the following, the `?->` null-safe operator will short-circuit the remainder of the section if it encounters a `null` value, and will return a `null` immediately without causing an error:

```
return $user->getAddress()->getCountry()->isoCode;
```

## Match Expressions

A `match` expression is like a `switch` block but it provides type-safe comparisons, supports a return value, does not require a `break` statement to break out, and also supports multiple matching values. So this rather cumbersome `switch` block:

```
switch ($country)
{
    case "UK":
    case "USA":
    case "Australia":
    default:
        $lang = "English";
        break;
    case "Spain":
        $lang= "Spanish";
        break;
    case "Germany":
    case "Austria":
        $lang = "German";
```

```
        break;  
    }  
  
Can be replaced with the following much simpler match expression:
```

```
$lang = match($country)  
{  
    "UK", "USA", "Australia", default => "English",  
    "Spain"                      => "Spanish",  
    "Germany", "Austria"          => "German",  
};
```

## New Functions

PHP 8 provides a number of new functions that offer greater functionality and improvements to the language, covering areas such as string handling, debugging and error handling

### `str_contains`

This function will return whether or not a string is contained within another string. It is a better function than `strpos`, because `strpos` returns the position of a string within another, or the value `false` if it's not found. But there's a potential problem because if the string is found at position zero and the value `0` is returned, this evaluates to `false` unless the strict comparison operator (`==`) is used instead of `=`.

Therefore, using `strpos` you need to write less than clear code such as this:

```
if (strpos('Once upon a time', 'Once') !== false)  
    echo 'Found';
```

Whereas code using `str_contains`, such as the following, is far clearer to understand when quickly scanning (and writing) code, and is less likely to lead to obscure bugs.

```
if (str_contains('Once upon a time', 'Once'))
```

```
echo 'Found';
```

The `str_contains` function is case-sensitive, so if you need to perform a case-insensitive check you should run both `$needle` and `$haystack` through a function to remove case first, such as `strtolower`, like this:

```
if (str_contains(strtolower('Once upon a time'),
                  strtolower('Once')))
echo 'Found';
```

Should you wish to use `str_contains` and also ensure your code is backwards compatible with older versions of PHP, you can employ the use of a polyfill (code that provides a feature your code expects to be natively available) to create your own version of the `str_contains` function (should it not already exist).

In the following just such a polyfill function for PHP 7+, the check for `$needle` being the empty string is there because PHP 8 considers the empty string to exist at every position within every string (even including within the null string), so this behaviour must be matched by the replacement function:

```
if (!function_exists('str_contains'))
{
    function str_contains(string $haystack, string $needle): bool
    {
        return $needle === '' || strpos($haystack, $needle) != false;
    }
}
```

## str\_starts\_with

This function provides a clearer means of checking whether one string starts with another. Previously you would probably use the `strpos` function and check whether it returns the value zero, but since we have already seen that 0 and `false` can become confused in certain situations, `str_starts_with` reduces this possibility substantially. You use it like this:

```
if (str_starts_with('In the beginning', 'In'))
echo 'Found';
```

Just as with `str_contains`, the test is made in a case-sensitive manner, so use a function such as `strtolower` on both strings to perform an insensitive test. A PHP 7+ polyfill for this function could be the following:

```
if (!function_exists('str_starts_with'))
{
    function str_starts_with(string $haystack, string $needle): bool
    {
        return $needle === '' || strpos($haystack, $needle) == 0;
    }
}
```

Since PHP 8 considers the null string to exist at every position in a string, this polyfill always returns `true` if `$needle` is the null string.

## str\_ends\_with

This function provides a clearer and simpler means of checking whether one string ends with another. Previously you would probably use the `substr` function, passing it the negative length of `$needle`, but `str_ends_with` makes this task far simpler. You use it like this:

```
if (str_ends_with('In the end', 'end'))
echo 'Found';
```

Just as with the other new string functions, the test is made in a case-sensitive manner, so use a function such as `strtolower` on both strings to perform an insensitive test. A PHP 7+ polyfill for this function could be the following:

```
if (!function_exists('str_ends_with'))
{
    function str_ends_with(string $haystack, string $needle): bool
```

```
{  
    return $needle === '' ||  
        $needle === substr($haystack, -strlen($needle));  
}  
}
```

If the second argument passed to `substr` is negative (as in this case) then the string is matched working backwards that number of characters from its end. Once again this polyfill always returns `true` if `$needle` is the null string. Also note the use of the `==` strict comparison operator to ensure an exact comparison is made between the two strings.

## fdiv

The new `fdiv` function is similar to the existing `fmod` and `intdiv` functions, but it allows for division by 0 without issuing a Division by Zero error, but returning one of `INF`, `-INF` or `NAN`, depending on the case.

For example, `intdiv(1, 0)` will issue a Division by Zero error, as will `1 % 0`, or simply `1 / 0`. But you can safely use `fdiv(1, 0)` and the result will be a float with the value `INF`, while `fdiv(-1, 0)` returns `-INF`.

Here's a PHP 7+ polyfill that you can use to make your code backwards compatible:

```
if (!function_exists('fdiv'))  
{  
    function fdiv(float $dividend, float $divisor): float  
    {  
        return @($dividend / $divisor);  
    }  
}
```

## get\_resource\_id

PHP 8 adds the new `get_resource_id` function that is similar to an (`int`) `$resource` cast to make it easier to retrieve a resource ID, but the return

type is checked to be both a resource and an integer, and is therefore type safe.

### **get\_debug\_type**

The `get_debug_type` function provides more consistent values than the existing `gettype` function and is best used to get detailed information on an unexpected variable in exception messages or logs, because it is more verbose and provides additional information.

### **preg\_last\_error\_msg**

PHP's `preg_` functions do not throw exceptions, so if there is an error you must retrieve any message using `preg_last_error` to get the error code. But if you want a friendly error message instead of just an unexplained code, in PHP 8 you can now call `preg_last_error_msg`. If there was no error then "No error" is returned.

As this book is aimed at beginner to intermediate level, I have only really scratched the surface of all the great new features in PHP 8, giving you a taste of the main ones that you can start using right away. However, if you are keen to learn absolutely everything about this milestone update, you can get full details at the following official web page:

[php.net/releases/8.0](https://php.net/releases/8.0)

## **MySQL 8**

MySQL 8 was first released in 2018 before the previous edition of this book was able to cover the features included in the update. So now, with it's most recent update (to 8.0.22) in late 2020, it's a good opportunity to catch up with all that MySQL 8 has to offer over earlier releases, such as better Unicode support, better JSON and document handling, geographic support, and window functions.

In this summary you'll get an overview of eight of the areas that have been improved, upgraded or added to in the latest version 8 release.

## **Updates to SQL**

With MySQL 8 now comes windows functions (also known as analytic functions) which are similar to grouped aggregate functions which collapse calculations on sets of rows into a single row, but a window function performs the aggregation for each row in the result set and are non-aggregate.

These 11 new functions (RANK, DENSE\_RANK, PERCENT\_RANK, CUME\_DIST, NTILE, ROW\_NUMBER, FIRST\_VALUE, LAST\_VALUE, NTH\_VALUE, LEAD and LAG) are fully documented at the following URL on the MySQL official website (shortened for ease of typing):

[tinyurl.com/mysql8wifuncs](http://tinyurl.com/mysql8wifuncs)

MySQL 8 also brings recursive Common Table Expressions, enhanced alternatives of NOWAIT and SKIP LOCKED in the SQL locking clause, Descendent Indexes, a GROUPING function and Optimizer Hints.

All these and more can be viewed on the MySQL website at this (shortened) URL:

[tinyurl.com/mysql8statements](http://tinyurl.com/mysql8statements)

## **JavaScript Object Notation (JSON)**

There are a number of new functions for handling JSON, while sorting and grouping of JSON values has been improved. As well as adding extended syntax for ranges in path expressions and improved sorting, there are new table, aggregation, merge and other functions.

Use of JSON in MySQL is beyond the remit of this book, but if it is an area of interest to you, the official documentation on all the new features can be found at the following (shortened) URL:

[tinyurl.com/mysql8json](http://tinyurl.com/mysql8json)

## **Geography Support**

MySQL 8 also brings GIS or Geography Support, including meta-data support for SRS (Spatial Reference System), SRS data types, indexes and functions. This means that MySQL can now (for example) calculate the distance between two points on the Earth's surface using their latitude and longitude coordinates, in any of the supported spatial reference systems.

For further details on how to access MySQL GIS you can refer to the following (shortened) URL on the official website:

[tinyurl.com/mysql8gis](http://tinyurl.com/mysql8gis)

## **Reliability**

MySQL is already extremely reliable, but it has been even further improved with version 8 by storing its meta-data in the InnoDB transactional storage engine, such that Users, Privileges and Dictionary tables now reside in InnoDB.

In MySQL 8 there is now only a single data dictionary, whereas in version 5.7 and earlier there were two data dictionaries (one for the server, and one for the InnoDB layer), which could get out of sync.

From version 8 the user is guaranteed that any DDL statement will either be executed fully, or not at all, preventing masters and slave servers from possibly getting out of sync.

## **Speed**

In MySQL the Information Schema has been sped up by up to 100 times by storing the tables as simple views on data dictionary tables in InnoDB. Additionally over 100 indexes on Performance Schema tables have been added to further speed up performance

## **Management**

With MySQL 8 you can now toggle an index between visible and invisible. And invisible index is not considered by the optimizer when a query plan is

created but the index is still maintained in the background, so it is easy to make it visible again, allowing you to decide whether an index is droppable.

Also users now have full control over Undo tablespaces and you can now persist global, dynamic server variables that would be lost upon server restart. Plus there's now an SQL RESTART command to allow remote management of a MySQL server over and SQL connection, and there's a new RENAME COLUMN command provided as an improvement over the previous ALTER TABLE ... CHANGE syntax.

For further details please refer to the official website at this (shortened) URL:

[tinyurl.com/mysql8serveradmin](http://tinyurl.com/mysql8serveradmin)

## Security

Security was definitely not left on the table when the new version 8 was planned, as there are many new improvements.

To start with the default authentication plugin has changed from `mysql_native_password` to `caching_sha2_password`, OpenSSL has been selected as the default TLS/SSL library in both the Enterprise and now the Community editions and is dynamically linked.

With MySQL 8 the Undo and Redo logs are now encrypted, and SQL roles have been implemented such that you can grant roles to users and privileges to roles. You can also use mandatory roles when new users are created. There is also now a password rotation policy configurable as global or user level with a securely stored password history.

In the authentication process a delay has been added in MySQL 8 based on consecutive unsuccessful login attempts to slow down brute force attempts at attack. Triggering of and the maximum length of delay is configurable.

For more information on MySQL and security please visit the official website at this (shortened) URL:

[tinyurl.com/mysql8security](http://tinyurl.com/mysql8security)

## Performance

And finally in this summary, we come to MySQL's overall performance, which is much improved with faster reads and writes, IO bound workloads and high contention workloads, plus you can now map user threads to CPUs to further optimize performance.

MySQL 8 scales better on heavy write workloads by a factor of up to 4 times compared with version 5.7 and offers even more significant increases for read/write workloads.

With MySQL you can use every storage device at it's highest ability and performance on high contention workloads (where transactions are queuing up to be granted a lock) is increased.

In all, the developers of MySQL 8 say that it is up to twice as fast, and you can read their reasoning and tips on how you can achieve this increase in your apps at the official website by following this (shortened) link:

[tinyurl.com/mysql8performance](http://tinyurl.com/mysql8performance)

## Questions

1. What does PHP 8 now allow you to do when declaring class properties?
2. What is the Null-safe operator, and what is it for?
3. How would you use a `match` expression in PHP 8, and why can it be better than the alternative?
4. What easy to use new function can you now use in PHP 8 to determine if one string exists within another?
5. In PHP 8 what is the best new way to make a floating point division calculation without causing a Division by Zero error?
6. What is a polyfill?

7. What is a simple new way in PHP 8 to see in plain English the most recent error generated by a call to one of the `preg_` functions?
8. By default what does MySQL 8 now use as its transactional storage engine?
9. In MySQL 8 what can you use instead of an `ALTER TABLE ... CHANGE TABLE` command to change the name of a column?
10. What is the default default authentication plugin in MySQL 8?

See “[Chapter 10 Answers](#)” in [Appendix A](#) for the answers to these questions.

# **Appendix A. Solutions to the Chapter Questions**

---

# Chapter 1 Answers

1. A web server (such as Apache), a server-side scripting language (PHP), a database (MySQL), and a client-side scripting language (JavaScript).
2. *HyperText Markup Language*: the web page itself, including text and markup tags.
3. Like nearly all database engines, MySQL accepts commands in *Structured Query Language* (SQL). SQL is the way that every user (including a PHP program) communicates with MySQL.
4. PHP runs on the server, whereas JavaScript runs on the client. PHP can communicate with the database to store and retrieve data, but it can't alter the user's web page quickly and dynamically. JavaScript has the opposite benefits and drawbacks.
5. *Cascading Style Sheets*: styling and layout rules applied to the elements in an HTML document.
6. Probably the most interesting new elements in HTML5 are `<audio>`, `<video>`, and `<canvas>`, although there are many others, such as `<article>`, `<summary>`, `<footer>`, and more.
- 7.

Some of these technologies are controlled by companies that accept bug reports and fix the errors like any software company. But open source software also depends on a community, so your bug report may be handled by any user who understands the code well enough. You may someday fix bugs in an open source tool yourself.

8.

It allows developers to concentrate on building the core functionality of a website or web app, passing on to the framework the task of making sure it always looks and runs its best, regardless of the platform (whether Linux, macOS, Windows, iOS, or Android), the dimensions of the screen, or the browser it finds itself running on.

## Chapter 2 Answers

1.

WAMP stands for *Windows, Apache, MySQL, and PHP*. The *M* in MAMP stands for *Mac* instead of Windows, and the *L* in LAMP stands for *Linux*. They all refer to a complete solution for hosting dynamic web pages.

2.

Both 127.0.0.1 and *http://localhost* are ways of referring to the local computer. When a WAMP or MAMP is properly configured, you can type either into a browser's address bar to call up the default page on the local server.

3.

FTP stands for *File Transfer Protocol*. An FTP program is used to transfer files back and forth between a client and a server.

4.

It is necessary to FTP files to a remote server in order to update them, which can substantially increase development time if this action is carried out many times in a session.

5.

Dedicated program editors are smart and can highlight problems in your code before you even run it.

## Chapter 3 Answers

1.

The tag used is `<?php...?>`. It can be shortened to `<?...?>`, but that is not recommended practice.

2.

You can use `//` for a single-line comment or `/*...*/` to span multiple lines.

3.

All PHP statements must end with a semicolon (`;`).

4.

With the exception of constants, all PHP variables must begin with `$`.

5.

A variable holds a value that can be a string, a number, or other data.

6.

`$variable = 1` is an assignment statement, whereas the `==` in `$variable == 1` is a comparison operator. Use `$variable = 1` to set the value of `$variable`. Use `$variable == 1` to find out later in the program whether `$variable` equals 1. If you mistakenly use `$variable = 1` where you meant to do a comparison, it will do two things you probably don't want: set `$variable` to 1 and return a `true` value all the time, no matter what its previous value was.

7.

In PHP, the hyphen is reserved for the subtraction, decrement, and negation operators. A construct like `$current-user` would be harder to

interpret if hyphens were also allowed in variable names, and in any case would lead programs to be ambiguous.

8.

Yes, variable names are case-sensitive. `$This_Variable` is not the same as `$this_variable`.

9.

You cannot use spaces in variable names, as this would confuse the PHP parser. Instead, try using the `_` (underscore).

10.

To convert one variable type to another, reference it and PHP will automatically convert it for you.

11.

There is no difference between `++$j` and `$j++` unless the value of `$j` is being tested, assigned to another variable, or passed as a parameter to a function. In such cases, `++$j` increments `$j` before the test or other operation is performed, whereas `$j++` performs the operation and then increments `$j`.

12.

Generally, the operators `&&` and `and` are interchangeable except where precedence is important, in which case `&&` has a high precedence, while `and` has a low one.

13.

You can use multiple lines within quotations marks or the `<<<_END..._END;` construct to create a multiline `echo` or assignment. In the latter case, the closing tag must be on a line by itself with nothing before or after it.

14.

You cannot redefine constants because, by definition, once defined they retain their value until the program terminates.

15.

You can use \ ' or \ " to escape either a single or double quote.

16.

The `echo` and `print` commands are similar in that they are both constructs, except that `print` behaves like a PHP function and takes a single argument, while `echo` can take multiple arguments.

17.

The purpose of functions is to separate discrete sections of code into their own self-contained sections that can be referenced by a single function name.

18.

You can make a variable accessible to all parts of a PHP program by declaring it as `global`.

19.

If you generate data within a function, you can convey the data to the rest of the program by returning a value or modifying a global variable.

20.

When you combine a string with a number, the result is another string.

## Chapter 4 Answers

1. In PHP, TRUE represents the value 1, and FALSE represents NULL, which can be thought of as “nothing” and is output as the empty string.
2. The simplest forms of expressions are literals (such as numbers and strings) and variables, which simply evaluate to themselves.
3. The difference between unary, binary, and ternary operators is the number of operands each requires (one, two, and three, respectively).
4. The best way to force your own operator precedence is to place parentheses around subexpressions to which you wish to give high precedence.
5. Operator associativity refers to the direction of processing (left to right, or right to left).
6. You use the identity operator when you wish to bypass PHP’s automatic operand type changing (also called *type casting*).
7. The three conditional statement types are `if`, `switch`, and the `?:` operator.
- 8.

To skip the current iteration of a loop and move on to the next one, use a `continue` statement.

9.

Loops using `for` statements are more powerful than `while` loops because they support two additional parameters to control the loop handling.

10.

Most conditional expressions in `if` and `while` statements are literals (or Booleans) and therefore trigger execution when they evaluate to TRUE. Numeric expressions trigger execution when they evaluate to a nonzero value. String expressions trigger execution when they evaluate to a nonempty string. A NULL value is evaluated as false and therefore does not trigger execution.

# Chapter 5 Answers

1.

Using functions avoids the need to copy or rewrite similar code sections many times over by combining sets of statements so that they can be called by a simple name.

2.

By default, a function can return a single value. But by utilizing arrays, references, and global variables, it can return any number of values.

3.

When you reference a variable by name, such as by assigning its value to another variable or by passing its value to a function, its value is copied. The original does not change when the copy is changed. But if you reference a variable, only a pointer (or reference) to its value is used, so that a single value is referenced by more than one name. Changing the value of the reference will change the original as well.

4.

Scope refers to which parts of a program can access a variable. For example, a variable of global scope can be accessed by all parts of a PHP program.

5.

To incorporate one file within another, you can use the `include` or `require` directives, or their safer variants, `include_once` and `require_once`.

6.

A function is a set of statements referenced by a name that can receive and return values. An object may contain zero or many functions (which

are then called methods) as well as variables (which are called properties), all combined in a single unit.

7.

To create a new object in PHP, use the `new` keyword like this:

```
$object = new Class;
```

8.

To create a subclass, use the `extends` keyword with syntax such as this:

```
class Subclass extends Parentclass ...
```

9.

To cause an object to be initialized when you create it, you can call a piece of initializing code by creating a constructor method called `__construct` within the class, and place your code there.

10.

Explicitly declaring properties within a class is unnecessary, as they will be implicitly declared upon first use. But it is considered good practice as it helps with code readability and debugging, and is especially useful to other people who may have to maintain your code.

# Chapter 6 Answers

1.

A numeric array can be indexed numerically using numbers or numeric variables. An associative array uses alphanumeric identifiers to index elements.

2.

The main benefit of the `array` keyword is that it enables you to assign several values at a time to an array without repeating the array name.

3.

Both the `each` function and the `foreach...as` loop construct return elements from an array; both start at the beginning and increment a pointer to make sure the next element is returned by the following call or iteration, and both return FALSE when the end of the array is reached. The difference is that the `each` function returns a single element, so it is usually wrapped in a loop. The `foreach...as` construct is already a loop, executing repeatedly until the array is exhausted or you explicitly break out of the loop.

4.

To create a multidimensional array, you need to assign additional arrays to elements of the main array.

5.

You can use the `count` function to count the number of elements in an array.

6.

The purpose of the `explode` function is to extract sections from a string that are separated by an identifier, such as extracting words separated by spaces within a sentence.

7.

To reset PHP's internal pointer into an array back to the first element, call the **reset** function.

# Chapter 7 Answers

1.

The conversion specifier you would use to display a floating-point number is %f.

2.

To take the input string "Happy Birthday" and output the string "\*\*Happy", you could use a `printf` statement such as this:

```
printf("%'*7.5s", "Happy Birthday");
```

3.

To send the output from `printf` to a variable instead of to a browser, you would use `sprintf` instead.

4.

To create a Unix timestamp for 7:11 a.m. on May 2nd, 2016, you could use the following command:

```
$timestamp = mktime(7, 11, 0, 5, 2, 2016);
```

5.

You would use the w+ file access mode with `fopen` to open a file in write and read mode, with the file truncated and the file pointer at the start.

6.

The PHP command for deleting the file *file.txt* is as follows:

```
unlink('file.txt');
```

7.

The PHP function **file\_get\_contents** is used to read in an entire file in one go. It will also read a file from across the internet if provided with a URL.

8.

The PHP superglobal associative array **\$\_FILES** contains the details about uploaded files.

9.

The PHP **exec** function enables the running of system commands.

10.

In HTML5, you can use either the XHTML style of tag (such as **<hr />**) or the standard HTML4 style (such as **<hr>**). It's entirely up to you or your company's coding style.

## Chapter 8 Answers

1.

The semicolon in MySQL separates or ends commands. If you forget to enter it, MySQL will issue a prompt and wait for you to enter it.

2.

To see the available databases, type `SHOW databases`. To see tables within a database that you are using, type `SHOW tables`. (These commands are case-insensitive.)

3.

To create this new user, use the `GRANT` command like this:

```
GRANT PRIVILEGES ON newdatabase.* TO 'newuser'@'localhost'  
IDENTIFIED BY 'newpassword';
```

4.

To view the structure of a table, type `DESCRIBE tablename`.

5.

The purpose of a MySQL index is to substantially decrease database access times by adding some metadata to the table about one or more key columns, which can then be quickly searched to locate rows within a table.

6.

A `FULLTEXT` index enables natural-language queries to find keywords, wherever they are in the `FULLTEXT` column(s), in much the same way as using a search engine.

7.

A stopword is a word that is so common that it is considered not worth including in a FULLTEXT index or using in searches. However, it is included in searches when it is part of a larger string bounded by double quotes.

8.

`SELECT DISTINCT` essentially affects only the display, choosing a single row and eliminating all the duplicates. `GROUP BY` does not eliminate rows, but combines all the rows that have the same value in the column. Therefore, `GROUP BY` is useful for performing an operation such as `COUNT` on groups of rows. `SELECT DISTINCT` is not useful for that purpose.

9.

To return only those rows containing the word *Langhorne* somewhere in the column *author* of the table *classics*, use a command such as this:

```
SELECT * FROM classics WHERE author LIKE "%Langhorne%";
```

10.

When you're joining two tables together they must share at least one common column, such as an ID number or, as in the case of the *classics* and *customers* tables, the *isbn* column.

# Chapter 9 Answers

1.

The term *relationship* refers to the connection between two pieces of data that have some association, such as a book and its author, or a book and the customer who bought the book. A relational database such as MySQL specializes in storing and retrieving such relations.

2.

The process of removing duplicate data and optimizing tables is called *normalization*.

3.

The three rules of First Normal Form are as follows:

- There should be no repeating columns containing the same kind of data.
- All columns should contain a single value.
- There should be a primary key to uniquely identify each row.

4.

To satisfy Second Normal Form, columns whose data repeats across multiple rows should be removed to their own tables.

5.

In a one-to-many relationship, the primary key from the table on the “one” side must be added as a separate column (a foreign key) to the table on the “many” side.

6.

To create a database with a many-to-many relationship, you create an intermediary table containing keys from two other tables. The other

tables can then reference each other via the third.

7.

To initiate a MySQL transaction, use the BEGIN or START TRANSACTION command. To terminate a transaction and cancel all actions, issue a ROLLBACK command. To terminate a transaction and commit all actions, issue a COMMIT command.

8.

To examine how a query will work in detail, you can use the EXPLAIN command.

9.

To back up the database *publications* to a file called *publications.sql*, you would use a command such as:

```
mysqldump -u user -ppassword publications > publications.sql
```

# Chapter 10 Answers

1.

When declaring class properties PHP 8 allows you to provide named parameters in function calls, allowing you to pass input data into a function, based on their argument names, instead of the argument order.

2.

The Null-safe operator is `?->`, it will short-circuit the remainder of a section if it encounters a `null` value, and will return a `null` immediately without causing an error.

3.

In PHP 8 a `match` expression can replace a `switch` block and can be better than `switch` because it provides type-safe comparisons, supports a return value, does not require a `break` statement to break out, and also supports multiple matching values.

4.

In PHP 8 you can now use the `str_contains` function to determine whether one string exists within another.

5.

The best new way to make a floating point division calculation without causing a Division by Zero error in PHP 8 is to use the `fdiv` function, which returns `INF`, `-INF` upon a division by zero.

6.

A polyfill is code that provides a feature your code expects to be natively available, and which you include in a program to allow your code to continue functioning correctly if it doesn't exist.

7.

In PHP 8, to see the most recent error in plain English that was generated by a call to one of the `preg_` functions you can call the `preg_last_error_msg` function.

8.

By default MySQL 8 now uses InnoDB as its transactional storage engine?

9.

In MySQL 8 to change the name of a column you can use the RENAME COLUMN command instead of using an `ALTER TABLE ... CHANGE TABLE` command.

10.

The default default authentication plugin in MySQL 8 is `caching_sha2_password`.

# Chapter 11 Answers

1. To connect to a MySQL database with PDO, call the `pdo` method, passing the attributes, username, password, and options. A connection object will be returned on success.
2. To submit a query to MySQL using PDO, ensure you have first created a connection object to a database and then call its `query` method, passing the query string.
3. The `PDO::FETCH_NUM` style of the `fetch` method can be used to return a row as an array indexed by column number.
4. You can manually close a PDO connection by assigning the value `null` to the PDO object used to connect to the database.
5. When adding a row to a table with an `AUTO_INCREMENT` column, you should pass the value `null` to that column.
6. To escape special characters in strings, you can call the `quote` method of a PDO connection object, passing it the string to be escaped. Of course, for security, using prepared statements will serve you best.
7. The absolute best way to ensure database security when accessing it is to use placeholders.

## Chapter 12 Answers

1. The associative arrays used to pass submitted form data to PHP are `$_GET` for the GET method and `$_POST` for the POST method.
2. The difference between a text box and a text area is that although they both accept text for form input, a text box is a single line, whereas a text area can be multiple lines and include word wrapping.
3. To offer three mutually exclusive choices in a web form, you should use radio buttons, because checkboxes allow multiple selections.
4. You can submit a group of selections from a web form using a single field name by using an array name with square brackets, such as `choices[]`, instead of a regular field name. Each value is then placed into the array, whose length will be the number of elements submitted.
5. To submit a form field without the user seeing it, place it in a hidden field using the attribute `type="hidden"`.
6. You can encapsulate a form element and supporting text or graphics, making the entire unit selectable with a mouse click, by using the `<label>` and `</label>` tags.
- 7.

To convert HTML into a format that can be displayed but will not be interpreted as HTML by a browser, use the PHP `htmlentities` function.

8.

You can help users complete fields with data they may have submitted elsewhere by using the `autocomplete` attribute, which prompts the user with possible values.

9.

To ensure that a form is not submitted with missing data, you can apply the `required` attribute to essential inputs.

# Chapter 13 Answers

1. Cookies should be transferred before a web page's HTML because they are sent as part of the headers.
2. To store a cookie in a web browser, use the `setcookie` function.
3. To destroy a cookie, reissue it with `setcookie`, but set its expiration date in the past.
4. Using HTTP authentication, the username and password are stored in `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`.
5. The `password_hash` function is a powerful security measure because it is a one-way function that converts a string to a long hexadecimal string of numbers that cannot be converted back, and is therefore very hard to crack as long as strong passwords are required from users (for example, at least 8 characters in length, including randomly placed numbers and punctuation marks).
6. When a string is salted, extra characters known only by the programmer are added to it before `hash` conversion (this should be left up to PHP to handle for you). This ensures that users with the same password will not have the same hash, and prevents the use of precomputed hash tables.
- 7.

A PHP session is a group of variables unique to the current user, passed along with successive requests so that the variables remain available as the user visits different pages.

8.

To initiate a PHP session, use the `session_start` function.

9.

Session hijacking is where a hacker somehow discovers an existing session ID and attempts to take it over.

10.

Session fixation is when an attacker attempts to force a user to log in using the wrong session ID, thus compromising the connection's security.

## Chapter 14 Answers

1.

To enclose JavaScript code, you use `<script>` and `</script>` tags.

2.

By default, JavaScript code will output to the part of the document in which it resides. If it's in the head, it will output to the head; if in the body, it outputs to the body.

3.

You can include JavaScript code from other files in your documents by either copying and pasting them or, more commonly, including them as part of a `<script src='filename.js'>` tag.

4.

The equivalent of the `echo` and `print` commands used in PHP is the `JavaScript document.write` function (or method).

5.

To create a comment in JavaScript, preface it with `//` for a single-line comment or surround it with `/*` and `*/` for a multiline comment.

6.

The JavaScript string concatenation operator is the `+` symbol.

7.

Within a JavaScript function, you can define a variable that has local scope by preceding it with the `var` keyword upon first assignment.

8.

To display the URL assigned to the link with an **id** of **thislink** in all main browsers, you can use the two following commands:

```
document.write(document.getElementById('thislink').href)  
document.write(thislink.href)
```

9.

The commands to change to the previous page in the browser's history array are:

```
history.back()  
history.go(-1)
```

10.

To replace the current document with the main page at the *oreilly.com* website, you could use the following command:

```
document.location.href = 'http://oreilly.com'
```

# Chapter 15 Answers

1. The most noticeable difference between Boolean values in PHP and JavaScript is that PHP recognizes the keywords TRUE, true, FALSE, and false, whereas only true and false are supported in JavaScript. Additionally, in PHP, TRUE has a value of 1, and FALSE is NULL; in JavaScript they are represented by true and false, which can be returned as string values.
2. Unlike PHP, no character is used (such as \$) to define a JavaScript variable name. JavaScript variable names can start with and contain uppercase and lowercase letters as well as underscores; names can also include digits, but not as the first character.
3. The difference between unary, binary, and ternary operators is the number of operands each requires (one, two, and three, respectively).
4. The best way to force your own operator precedence is to surround the parts of an expression to be evaluated first with parentheses.
5. You use the identity operator when you wish to bypass JavaScript's automatic operand type changing.
6. The simplest forms of expressions are literals (such as numbers and strings) and variables, which simply evaluate to themselves.
- 7.

The three conditional statement types are `if`, `switch`, and the `?:` operator.

8.

Most conditional expressions in `if` and `while` statements are literals or Booleans and therefore trigger execution when they evaluate to `true`. Numeric expressions trigger execution when they evaluate to a nonzero value. String expressions trigger execution when they evaluate to a nonempty string. A `NULL` value is evaluated as `false` and therefore does not trigger execution.

9.

Loops using `for` statements are more powerful than `while` loops because they support two additional parameters to control loop handling.

10.

The `with` statement takes an object as its parameter. Using it, you specify an object once; then, for each statement within the `with` block, that object is assumed.

# Chapter 16 Answers

1. JavaScript functions and variable name are case-sensitive. The variables `Count`, `count`, and `COUNT` are all different.
2. To write a function that accepts and processes an unlimited number of parameters, access parameters through the `arguments` array, which is a member of all functions.
3. One way to return multiple values from a function is to place them all inside an array and return the array.
4. When defining a class, use the `this` keyword to refer to the current object.
5. The methods of a class do not have to be defined within the class definition. If a method is defined outside the constructor, the method name must be assigned to the `this` object within the class definition.
6. New objects are created via the `new` keyword.
7. You can make a property or method available to all objects in a class without replicating it within the object by using the `prototype` keyword to create a single instance, which is then passed by reference to all the objects in the class.

8.

To create a multidimensional array, place subarrays inside the main array.

9.

The syntax you would use to create an associative array is *key* : *value*, within curly braces, as in the following:

```
assocarray =  
{  
    "forename" : "Paul",  
    "surname"   : "McCartney",  
    "group"     : "The Beatles"  
}
```

10.

A statement to sort an array of numbers into descending numerical order would look like this:

```
numbers.sort(function(a, b){ return b - a })
```

# Chapter 17 Answers

1.

You can send a form for validation prior to submitting it by adding the JavaScript `onsubmit` attribute to the `<form>` tag. Make sure that your function returns `true` if the form is to be submitted, and `false` otherwise.

2.

To match a string against a regular expression in JavaScript, use the `test` method.

3.

Regular expressions to match characters not in a word could be any of `/[^w]/`, `/[\W]/`, `/^w/`, `/\W/` `/[^a-zA-Z0-9_]/`, and so on.

4.

A regular expression to match either of the words *fox* or *fix* could be `/f[oí]x/`.

5.

A regular expression to match any single word followed by any nonword character could be `/\w+\W/g`.

6.

A JavaScript function using regular expressions to test whether the word *fox* exists in the string `The quick brown fox` could be as follows:

```
document.write(/fox/.test("The quick brown fox"))
```

7.

A PHP function using a regular expression to replace all occurrences of the word *the* in **The cow jumps over the moon** with the word *my* could be as follows:

```
$s=ereg_replace("/the/i", "my", "The cow jumps over the moon");
```

8.

The HTML attribute used to precomplete form fields with a value is **value**, which is placed within an **<input>** tag and takes the form **value="value"**.

## Chapter 18 Answers

1.

It's necessary to write a function for creating new XMLHttpRequest objects because Microsoft browsers use two different methods of creating them, while all other major browsers use a third. By writing a function to test the browser in use, you can ensure that your code will work on all major browsers.

2.

The purpose of the `try...catch` construct is to set an error trap for the code inside the `try` statement. If the code causes an error, the `catch` section will be executed instead of a general error being issued.

3.

An XMLHttpRequest object has six properties and six methods (see Tables “XML HTTP Prequest - Object Apostrophes PROP” and “XML HTTP Prequest - Object Apostrophes METH”).

4.

You can tell that an asynchronous call has completed when the `readyState` property of an object has a value of 4.

5.

When an asynchronous call successfully completes, the object's `status` property will have a value of 200.

6.

The `responseText` property of an XMLHttpRequest object contains the value returned by a successful asynchronous call.

7.

The `responseXML` property of an `XMLHttpRequest` object contains a DOM tree created from the XML returned by a successful asynchronous call.

8.

To specify a callback function to handle asynchronous responses, assign the function name to the `XMLHttpRequest` object's `onreadystatechange` property. You can also use an unnamed, inline function.

9.

To initiate an asynchronous request, an `XMLHttpRequest` object's `send` method is called.

10.

The main differences between asynchronous GET and POST requests are that GET requests append the data to the URL rather than passing it as a parameter of the `send` method, while POST requests pass the data as a parameter of the `send` method and require the correct form headers to be sent first.

# Chapter 19 Answers

1.

To import one stylesheet into another, you use the `@import` directive, like this:

```
@import url('styles.css');
```

2.

To import a stylesheet into a document, you can use the HTML `<link>` tag:

```
<link rel='stylesheet' href='styles.css'>
```

3.

To directly embed a style into an element, use the `style` attribute, like this:

```
<div style='color:blue;'>
```

4.

The difference between a CSS ID and a CSS class is that an ID is applied to only a single element, whereas a class can be applied to many elements.

5.

In a CSS declaration, ID names are prefixed with a # character (e.g., `#myid`) and class names with a . character (e.g., `.myclass`).

6.

In CSS, the semicolon (;) is used as a separator between declarations.

7.

To add a comment to a stylesheet, you enclose it between /\* and \*/ opening and closing comment markers.

8.

In CSS, you can match any element using the \* universal selector.

9.

To select a group of different elements and/or element types in CSS, you place a comma between each element, ID, or class.

10.

Given a pair of CSS declarations with equal precedence, to make one have greater precedence over the other, you append the !important declaration to it, like this:

```
p { color:#ff0000 !important; }
```

# Chapter 20 Answers

1.

The CSS3 operators ^=, \$=, and \*= match the start, end, or any portion of a string, respectively.

2.

The property you use to specify the size of a background image is **background-size**, like this:

```
background-size:800px 600px;
```

3.

You can specify the radius of a border using the **border-radius** property:

```
border-radius:20px;
```

4.

To flow text over multiple columns, use the **column-count**, **column-gap**, and **column-rule** properties (or their browser-specific variants), like this:

```
column-count:3;  
column-gap :1em;  
column-rule :1px solid black;
```

5.

The four functions with which you can specify CSS colors are `hsl`, `hsla`, `rgb`, and `rgba`. For example:

```
color:rgba(0%,60%,40%,0.4);
```

6.

To create a gray shadow under some text, offset diagonally to the bottom right by 5 pixels, with a blurring of 3 pixels, use this declaration:

```
text-shadow:5px 5px 3px #888;
```

7.

You can indicate that text is truncated with an ellipsis by using this declaration:

```
text-overflow:ellipsis;
```

8.

To include a Google web font such as Lobster in a web page, first select it from <http://fonts.google.com>, then copy the provided `<link>` tag into the `<head>` of your HTML document. It will look something like this:

```
<link href='http://fonts.googleapis.com/css?family=Lobster'>
```

```
rel='stylesheet'>
```

You can then refer to the font in a CSS declaration such as this:

```
h1 { font-family:'Lobster', arial, serif; }
```

9.

The CSS declaration you would use to rotate an object by 90 degrees is:

```
transform:rotate(90deg);
```

10.

To set up a transition on an object so that when any of its properties are changed, the change will transition immediately in a linear fashion over the course of half a second, use this declaration:

```
transition:all .5s linear;
```

# Chapter 21 Answers

1.

The **O** function returns an object by its ID, the **S** function returns the **style** property of an object, and the **C** function returns an array of all objects that access a given class.

2.

You can modify a CSS attribute of an object using the **setAttribute** function, like this:

```
myobject.setAttribute('font-size', '16pt')
```

You can also (usually) modify an attribute directly (using slightly modified property names where required), like this:

```
myobject.fontSize = '16pt'
```

3.

The properties that provide the width and height available in a browser window are **window.innerHeight** and **window.innerWidth**.

4.

To make something happen when the mouse passes over and out of an object, attach to the **onmouseover** and **onmouseout** events.

5.

To create a new element, use code such as this:

```
elem = document.createElement('span')
```

To add the new element to the DOM, use code such as this:

```
document.body.appendChild(elem)
```

6.

To make an element invisible, set its **visibility** property to **hidden** (set it to **visible** to restore it again). To collapse an element's dimensions to zero, set its **display** property to **none** (setting this property to **block** is one way to restore it to its original dimensions).

7.

To set up a single event at a future time, call the **setTimeout** function, passing it the code or function name to execute and the time delay in milliseconds.

8.

To set up repeating events at regular intervals, use the **setInterval** function, passing it the code or function name to execute and the time delay between repeats in milliseconds.

9.

To release an element from its location in a web page to enable it to be moved around, set its **position** property to **relative**, **absolute**, or **fixed**. To restore it to its original place, set the property to **static**.

10.

To achieve an animation rate of 50 frames per second, you should set a delay between interrupts of 20 milliseconds. To calculate this value, divide 1,000 milliseconds by the desired frame rate.

## Chapter 22 Answers

1.

The symbol commonly used as the factory method for creating jQuery objects is `$`. Alternatively, you can use the method name `jQuery`.

2.

To link to minified release 3.2.1 of jQuery from the Google CDN, you could use HTML such as this:

```
<script src='http://ajax.googleapis.com/ajax/libs/jquery/
3.2.1/jquery.min.js'></script>
```

3.

The jQuery `$` factory method accepts CSS selectors in order to build a jQuery object of matching elements.

4.

To get a CSS property value, use the `css` method, supplying just a property name. To set the property's value, supply a property name and a value to the method.

5.

To attach a method to the element `elem`'s click event to make it slowly hide, you could use code such as the following:

```
$('#elem').click(function() { $(this).hide('slow') } )
```

6.

In order to be able to animate an element, you must assign a value of **fixed**, **relative**, or **absolute** to its **position** property.

7.

You can run methods at once (or sequentially if animations) by chaining them together with periods, like this:

```
$('#elem').css('color', 'blue').css('background',  
'yellow').slideUp('slow')
```

8.

To retrieve an element node object from a jQuery selection object, you can index it with square brackets, like this:

```
$('#elem')[0]
```

or use the **get** method, like this:

```
$('#elem').get(0)
```

9.

To display the sibling element immediately preceding one with the ID of **news** in bold, you could use this statement:

```
$('#news').prev().css('font-weight', 'bold')
```

10.

You can make a jQuery asynchronous GET request using the `$.get` method, like this:

```
$.get('http://server.com/ajax.php?do=this', function(data) {  
    alert('The server said: ' + data) } )
```

## Chapter 23 Answers

1.

Using a CDN to deliver files means you do not have to rely on your own (or your client's) bandwidth, which can save money. Additionally, you can speed up the user's experience because, once a browser has downloaded a file, the same version can be reloaded locally from the cache. A downside is that your web page or web app may not run locally if the user's browser is not connected to the internet at the time.

2.

To define a page of content to jQuery Mobile, you should enclose it within a `<div>` element with a `data-role` attribute of `page`.

3.

The three main parts of a jQuery page are its header, content, and footer. To denote these, you place them within `<div>` elements that respectively have their `data-role` attributes assigned the values `header`, `content`, and `footer`. These three elements must be children of the parent `<div>` discussed in Question 2.

4.

To put multiple jQuery Mobile pages within a single HTML document you can include multiple parent `<div>` elements with a `data-role` attribute of `page`, each containing the three child `<div>`s discussed in Question 3. To link between these pages, you should assign each of these elements a unique `id` (such as `id="news"`), which can then be referenced from anywhere within the HTML document using an anchor such as `<a href="#news">`.

5.

To prevent a web page from being loaded asynchronously you can either give the anchor's or form's `data-ajax` property a value of `false`,

give its `rel` attribute a value of `external`, or supply a value to its `target` attribute.

6.

To set the page transition of an anchor to flip, give its `data-transition` attribute a value of `flip` (or you can use any other of the supported values for the other available transition effects; for example, `data-transition="pop"`).

7.

You can load a page so that it displays as a dialog by giving its `data-rel` attribute a value of `dialog`.

8.

To make an anchor link display as a button, give its `data-role` attribute a value of `button`.

9.

To make a jQuery Mobile element display inline, you give its `data-inline` attribute a value of `true`.

10.

To add an icon to a button, supply the name of a known jQuery Mobile icon as a value to its `data-icon` attribute; for example, `data-icon="gear"`.

## Chapter 24 Answers

1. You can incorporate the React scripts in your web page either by downloading the files and serving them from your own web server, or by using a content delivery network such as [unpkg.com](https://unpkg.com). You then load the scripts from `script` tags in your HTML document.
2. To incorporate XML into your React JavaScript you first need to load the Babel extension, either locally or from a Content Delivery Network (CDN), using a `script` tag.
3. JSX JavaScript `<script>` sections of code require `type="text/babel"` in order to work.
4. You can extend React to your code either as a class using `class Name extends React.Component` or simply by returning code to be rendered by a function's `return` statement. In both cases `ReactDOM.render` must be called to initiate the rendering. Beware that using a function is much less powerful than a class and should only be used for presentational stateless rendering, although it may be faster than using a class.
5. In React pure code doesn't change its inputs, whereas code that does change inputs is considered impure.
6. React keeps track of state with the `props` object and its attributes.
- 7.

To embed an expression within JSX code you place it within curly braces, like this `Hello {props.name}`.

8.

Once a class has been constructed you can only change the state of a value using the `setState` function.

9.

To enable referring to `props` using the `this` keyword within a constructor, you must first call the `super` method, passing it `props`, like this: `super(props)`.

10.

You can create a conditional statement in JSX using the `&&` operator after the expression. For an `IF...THEN...ELSE` statement you can use the ternary `? :` operator.

## Chapter 25 Answers

1. The new HTML5 element for drawing graphics in a web page is the canvas element, created with the <canvas> tag.
2. You need to use JavaScript to access many of the new HTML5 features, such as the canvas and geolocation.
3. To incorporate audio or video in a web page, use the <audio> or <video> tags.
4. In HTML5, local storage offers far greater access to local user space than cookies, which are limited in the amount of data they can hold.
5. In HTML5, you can set up web workers to carry out background tasks for you. These workers are simply sections of JavaScript code.

## Chapter 26 Answers

1.

To create a canvas element in HTML, use the `<canvas>` tag and specify an ID that JavaScript can use to access it, like this:

```
<canvas id='mycanvas'>
```

2.

To give JavaScript access to a canvas element, ensure the element has been given an ID such as `mycanvas`, and then use the `document.getElementById` function (or the `O` function from the `OSC.js` file supplied with the examples archive on the companion website) to return an object to the element. Finally, call `getContext` on the object to retrieve a 2D context to the canvas, like this:

```
canvas = document.getElementById('mycanvas')
context = canvas.getContext('2d')
```

3.

To start a canvas path, issue the `beginPath` method on the context. After creating a path, you close it by issuing `closePath` on the context, like this:

```
context.beginPath()
```

```
// Path creation commands go here  
context.closePath()
```

4.

You can extract the data from a canvas using the `toDataURL` method, which can then be assigned to the `src` property of an image object, like this:

```
image.src = canvas.toDataURL()
```

5.

To create a gradient fill (either radial or linear) with more than two colors, specify all the colors required as stop colors assigned to a gradient object you have already created, and assign them each a starting point as a percent value of the complete gradient (between 0 and 1), like this:

```
gradient.addColorStop(0,      'green')  
gradient.addColorStop(0.3,    'red')  
gradient.addColorStop(0.79,   'orange')  
gradient.addColorStop(1,      'brown')
```

6.

To adjust the width of drawn lines, assign a value to the `lineWidth` property of the context, like this:

```
context.lineWidth = 5
```

7.

To ensure that future drawing takes place only within a certain area, you can create a path and then call the `clip` method.

8.

A complex curve with two imaginary attractors is called a Bézier curve. To create one, call the `bezierCurveTo` method, supplying two pairs of *x* and *y* coordinates for the attractors, followed by another pair for the end point of the curve. A curve is then created from the current drawing location to the destination.

9.

The `getImageData` method returns an array containing the pixel data for the specified area, with the elements consecutively containing the red, green, blue, and alpha pixel values, so four items of data are returned per pixel.

10.

The `transform` method takes six arguments (or parameters), which are, in order, horizontal scale, horizontal skew, vertical skew, vertical scale, horizontal translate, and vertical translate. Therefore, the arguments that apply to scaling are the first and fourth in the list.

## Chapter 27 Answers

1. To insert audio and video into an HTML5 document, use the `<audio>` and `<video>` tags.
2. To guarantee maximum audio playability of compressed, lossy audio on all platforms, you should choose between (or offer both of) the AAC and MP3 audio formats.
3. To play and pause HTML5 media playback, you can call the `play` and `pause` methods of an `<audio>` or `<video>` element.
4. FLAC is a compressed (but not as much as MP3 or AAC) audio format but without any loss of data. It will save you storage space and bandwidth when providing lossless audio content.
5. To guarantee maximum video playability of video on all platforms, you should choose between (or offer both of) the MP4 and WEBM video formats.

## Chapter 28 Answers

1.

To request geolocation data from a web browser, you call the following method, passing the names of two functions you have written for handling access to or denial of the data:

```
navigator.geolocation.getCurrentPosition(granted, denied)
```

2.

To determine whether a browser supports local storage, test the **typeof** property of the **localStorage** object, like this:

```
if (typeof localStorage == 'undefined')  
    // Local storage is not available}
```

3.

To erase all local storage data for the current domain, you can call the **localStorage.clear** method.

4.

The easiest way for a web worker to communicate with a main program is by using the **postMessage** method to send information. The program attaches to the web worker's **onmessage** event to retrieve it.

5.

To stop a web worker from running, issue a call to the **terminate** method of the **worker** object, like this: **worker.terminate()**.

6.

You can prevent the default action of disallowing dragging and dropping for the events that handle these operations by issuing a call to the event object's `preventDefault` method in your `ondragover` and `ondrop` event handlers.

7.

To make cross-document messaging more secure, you should always supply a domain identifier when posting messages, like this:

```
postMessage(message, 'http://mydomain.com')
```

And check for that identifier when receiving them, like this:

```
if (event.origin) != 'http://mydomain.com') // Disallow
```

You can also encrypt or obscure communications to discourage injection or eavesdropping.

## **Chapter 29 Answers**

1.

The answer to this question is entirely up to you. If you have enjoyed this book please tell your friends and leave a review at an online bookstore. If you have any questions, comments, suggestions or addenda, please visit the book's page at [oreilly.com](http://oreilly.com) and leave them there, and thanks for reading!

# Index

---

## Symbols

! (Not) logical operator

in PHP, [Logical operators](#), [Operator Precedence](#), [Logical operators](#)

!= (inequality) operator

in PHP, [Comparison operators](#), [Operator Precedence](#)

!== (not identical) operator

in PHP, [Comparison operators](#), [Operator Precedence](#)

" (quotation marks, double)

in PHP heredocs, [Multiple-Line Commands](#)

in PHP multi-line commands, [Multiple-Line Commands](#)

in PHP strings, [String types](#)

\$ (dollar) symbol

in PHP, [The \\$ symbol](#), [Constants](#), [Accessing Objects](#)

`$GLOBALS` variable, [Superglobal variables](#)

`$this` variable (PHP), [Writing Methods](#)

% (modulus) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

%= (modulus and assignment) operator

in PHP, [Assignment operators](#)

& (ampersand), prefacing PHP variables, [Passing Arguments by Reference](#)

& (And) bitwise operator

in PHP, [Associativity](#)

&& (And) logical operator

in PHP, [Logical operators](#), [Operator Precedence](#), [Logical operators](#)

&= (bitwise AND and assignment) operator

in PHP, [Associativity](#)

' (quotation marks, single)

in PHP heredocs, [Multiple-Line Commands](#)

in PHP strings, [String types](#)

() (parentheses)

casting operators in PHP, [Implicit and Explicit Casting](#)

in functions in PHP, [PHP Functions](#)

operator precedence and, [Operator Precedence](#)

\* (asterisk)

in MySQL, [SELECT](#)

\* (multiplication) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

\*\* (exponentiation) operator, in PHP, [Arithmetic operators](#)

\*/ character, [Using Comments](#)

\*= (multiplication and assignment) operator

in PHP, [Assignment operators](#)

`+` (addition) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

`++` (increment) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

`+=` (addition and assignment) operator

in PHP, [Assignment operators](#)

`,` (comma)

in PHP, [for Loops](#)

`-` (minus sign)

unary operator in PHP, [Associativity](#)

`-` (subtraction) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#), [Associativity](#)

`--` (decrement) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

`-=` (subtraction and assignment) operator

in PHP, [Assignment operators](#)

`->` operator, [Writing Methods](#)

`.=` (concatenation assignment) operator (PHP), [Assignment operators](#), [String concatenation](#), [Operator Precedence](#)

`/` (division) operator

in PHP, [Assignment operators](#), [Operator Precedence](#)

`/` (forward slash)

`/*` and `*/` in multiline comments, [Using Comments](#)

/\* character, [Using Comments](#)

/= (division and assignment) operator  
in PHP, [Arithmetic operators](#), [Operator Precedence](#)

: (colon) character  
in PHP, [TRUE or FALSE?](#)

:: (scope resolution) operator, [Static Methods](#)

; (semicolons)  
in MySQL, [The semicolon](#)  
in PHP, [Semicolons](#), [for Loops](#)

< (less than) operator  
in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)

<< (bitwise left shift) operator  
in PHP, [Operator Precedence](#)

<<< (heredoc) operator (PHP), [Multiple-Line Commands](#)

<<= (bitwise left shift and assignment) operator  
in PHP, [Operator Precedence](#)

<= (less than or equal to) operator  
in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)

<> (not equal) operator (PHP), [Comparison operators](#), [Operator Precedence](#)

<?php?> tag, [Incorporating PHP Within HTML](#)

= (assignment) operator

in PHP, [Assignment operators](#)

not confusing with == operator, [Comparison operators](#), [Equality](#)

== (equality) operator

in PHP, [Comparison operators](#), [Operator Precedence](#), [Equality](#)

not confusing with = operator, [Comparison operators](#)

===(identity) operator

in PHP, [Comparison operators](#), [Operator Precedence](#)

=> operator (PHP), assigning value to array index, [Assignment Using the array Keyword](#)

> (greater than) operator

in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)

>= (greater than or equal to) operator

in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)

>> (bitwise right shift) operator

in PHP, [Operator Precedence](#)

>>= (bitwise right shift and assignment) operator

in PHP, [Operator Precedence](#)

? operator (PHP), [The ? Operator](#)

?:(ternary) operator

in PHP, [Operators](#), [Operator Precedence](#), [The ? Operator](#)

@ (error control) operator (PHP), [Associativity](#)

[] (square brackets)

accessing array elements, [Multidimensional Arrays](#)

in PHP function definitions, [Defining a Function](#)

\ (backslash)

in PHP strings, [Escaping characters](#)

\c (cancel in MySQL), [Canceling a command](#)

\n (newline) character, [Escaping characters](#)

\r (carriage return) character, [Escaping characters](#)

\t (tab) character, [Escaping characters](#), [Multidimensional Arrays](#)

^ (bitwise xor) operator

in PHP, [Operator Precedence](#)

^= (bitwise XOR and assignment) operator

in PHP, [Operator Precedence](#)

\_\_ (double underscore), [Predefined Constants](#), [Writing Methods](#)

{ } (curly braces)

in for loops in PHP, [for Loops](#)

in functions in PHP, [Defining a Function](#)

in if statements in PHP, [The if Statement](#)

in while statements in PHP, [while Loops](#)

| (Or) bitwise operator

in PHP, [Operator Precedence](#)

|= (bitwise OR and assignment) operator

in PHP, [Operator Precedence](#)

|| (Or) logical operator

in PHP, [Logical operators](#), [Operator Precedence](#), [Logical operators](#)

## A

acknowledgments, [Acknowledgments](#)

addition (+) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

advisory locks, [Locking Files for Multiple Accesses](#)

ALTER command (MySQL), [The AUTO\\_INCREMENT attribute](#)-[Creating a FULLTEXT index](#)

adding new columns, [Adding a new column](#)

changing column data types, [Changing the data type of a column](#)

overview of, [The AUTO\\_INCREMENT attribute](#)

removing columns, [Removing a column](#)

renaming columns, [Renaming a column](#)

renaming tables, [Renaming a table](#)

AMPPS server

alternatives to, [Alternative WAMPs](#)

documentation, [Installing AMPPS on Windows](#)

evolution of, [Alternative WAMPs](#)

macOS installation, [Installing AMPPS on macOS](#)-[Accessing the Document Root \(macOS\)](#)

MySQL access, Windows users-Linux users

PHP version selection, [Installing AMPPS on Windows](#), [Installing AMPPS on macOS](#), [PHP Functions and Objects](#)

Windows installation, [Installing AMPPS on Windows](#)-Accessing the Document Root (Windows)

And (&&) logical operator

in PHP, [Logical operators](#), [Operator Precedence](#), [Logical operators](#)

and (low-precedence) logical operator (PHP), [Logical operators](#), [Operator Precedence](#)

Apache web server

benefits of, [The Apache Web Server](#)

in WAMPs, MAMPs, and LAMPs, [What Is a WAMP, MAMP, or LAMP?](#)

arguments, in PHP, [PHP Functions](#), [Passing Arguments by Reference](#)

arithmetic operators

in PHP, [Arithmetic operators](#), [Operators](#), [Operator Precedence](#)

array keyword (PHP), [Assignment Using the array Keyword](#)

arrays (PHP)

array functions, [Using Array Functions](#)-end

assignment using array keyword, [Assignment Using the array Keyword](#)

associative, [Associative Arrays](#)

basics of, [Arrays](#)

benefits of, [PHP Arrays](#)

foreach...as loops, [The foreach...as Loop](#)  
multidimensional, [Multidimensional Arrays](#)  
numerically indexed, [Numerically Indexed Arrays](#)  
returning values from functions in, [Returning an Array](#)  
two-dimensional, [Two-dimensional arrays](#)

array\_combine function (PHP), [PHP Version Compatibility](#)  
AS keyword (MySQL), [Using AS](#)  
assignment  
multiline string assignment in PHP, [Multiple-Line Commands](#)  
multiple-assignment statements, [Associativity](#)  
using array keyword in PHP, [Assignment Using the array Keyword](#)  
assignment operators  
in PHP, [Assignment operators](#), [Operators](#), [Operator Precedence](#),  
[Equality](#)

associative arrays  
in PHP, [Associative Arrays](#)  
associativity, [Associativity](#)  
asterisk (\*)  
in MySQL, [SELECT](#)

asynchronous communication  
benefits of, [Using JavaScript](#)  
role of modern technology in, [Bringing It All Together](#)

Atom feeds, [Date Constants](#)

AUTO\_INCREMENT data type (MySQL), [The AUTO\\_INCREMENT attribute](#)

## B

backslash (\)

in PHP strings, [Escaping characters](#)

BEGIN statement (MySQL), [Using BEGIN](#)

Berners-Lee, Tim, [Introduction to Dynamic Web Content](#)

BINARY data type (MySQL), [The BINARY data type](#)

binary operators, [Operators](#)

bitwise left shift (<<) operator

in PHP, [Operator Precedence](#)

bitwise left shift and assignment (<<=) operator

in PHP, [Operator Precedence](#)

bitwise operators

in PHP, [Operator Precedence](#)

bitwise OR and assignment (|=) operator

in PHP, [Operator Precedence](#)

bitwise right shift (>>) operator

in PHP, [Operator Precedence](#)

bitwise xor (^) operator

in PHP, [Operator Precedence](#)

bitwise XOR and assignment ( $\wedge=$ ) operator

in PHP, [Operator Precedence](#)

BLOB data types (MySQL), [The BLOB data types](#)

Boolean expressions

in PHP, [TRUE or FALSE?](#)

Boolean values, [Expressions](#), [Relational Operators](#)

break command

in PHP looping, [Breaking Out of a Loop](#)

in PHP switch statements, [Breaking out](#)

## C

C++ Redistributable Visual Studio, [Installing AMPPS on Windows](#)

call-time pass-by-reference, [Passing Arguments by Reference](#)

carriage return (\r) character, [Escaping characters](#)

case command

in PHP, [The switch Statement](#)

casting

implicit/explicit in PHP, [Implicit and Explicit Casting](#)

operators in PHP, [Associativity](#)

CHANGE keyword (MySQL), [Renaming a column](#)

CHAR data type (MySQL), [Data Types](#)

character sets, [Data Types](#)

checkdate function (PHP), [Using checkdate](#)

class keyword (PHP), Declaring a Class

\_\_CLASS\_\_ constant, Predefined Constants

classes

in PHP, Terminology-Declaring a Class

clients

request/response procedure, The Request/Response Procedure-The Request/Response Procedure

role in internet communications, HTTP and HTML: Berners-Lee's Basics

clone operator, Cloning Objects

code examples, obtaining and using, Using PHP, This Book's Examples, String variables

colon (:) character

in PHP, TRUE or FALSE?

colors

in HTML, Using printf, Using sprintf

columns

defined, MySQL Basics, MySQL Basics

working with in MySQL, Changing the data type of a column-Removing a column

comma (,)

in PHP, for Loops

commands

in MySQL, The semicolon-Creating a table

multiple-line (PHP), [Multiple-Line Commands](#)-[Multiple-Line Commands](#)

comments

in PHP, [Using Comments](#)

COMMIT command (MySQL), [Using COMMIT](#)

Common Gateway Interface (CGI), [The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5](#)

compact function (PHP), [compact](#)

companion website, [This Book's Examples](#)

comparison operators

in PHP, [Comparison operators](#), [Operators](#), [Operator Precedence](#), [Comparison operators](#)

concatenation, [String concatenation](#)

conditionals (PHP)

? operator, [The ? Operator](#)

else statements, [The else Statement](#)

elseif statements, [The elseif Statement](#)

if statements, [The if Statement](#)

switch statements, [The switch Statement](#)-[Alternative syntax](#)

constants (PHP), [Constants](#)-[Predefined Constants](#), [TRUE or FALSE?](#), [Declaring Constants](#), [Date Constants](#)

constructors

in PHP, [Constructors](#), [Subclass constructors](#)

content types, [Using \\$\\_FILES](#)

continue statements

in PHP, [The continue Statement](#)

control flow (see flow control)

[\\$\\_COOKIE](#) variable, [Superglobal variables](#)

cookies

date constants in PHP, [Date Constants](#)

copy function (PHP), [Copying Files](#)

count function (PHP), [count](#)

COUNT parameter (MySQL), [SELECT COUNT](#)

CREATE INDEX command (MySQL), [Using CREATE INDEX](#)

CSS (Cascading Style Sheets)

benefits of, [The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5-Using CSS](#)

CSV format, dumping data in, [Dumping Data in CSV Format](#)

curly braces ({ })

in for loops in PHP, [for Loops](#)

in functions in PHP, [Defining a Function](#)

in if statements in PHP, [The if Statement](#)

in while statements in PHP, [while Loops](#)

currency conversion, [Precision Setting](#)

D

data types (MySQL)

AUTO\_INCREMENT data type, [The AUTO\\_INCREMENT attribute](#)

BINARY, [The BINARY data type](#)

BLOB, [The BLOB data types](#)

CHAR, [Data Types](#)

DATE and TIME, [DATE and TIME types](#)

numeric, [Numeric data types](#)

TEXT data types, [The TEXT data types](#)

VARCHAR, [Data Types](#)

databases (see also MySQL; database queries)

anonymity and, [Databases and Anonymity](#)

backup and restore for, [Backing Up and Restoring-Planning Your Backups](#)

defined, [MySQL Basics](#)

design considerations, [Database Design](#)

normalization process, [Normalization-When Not to Use Normalization](#)

primary keys in, [Primary Keys: The Keys to Relational Databases](#)

relationships in, [Relationships-Databases and Anonymity](#)

terminology surrounding, [Summary of Database Terms](#)

transactions in, [Transactions-Using EXPLAIN](#)

DATE and TIME data types (MySQL), [DATE and TIME types](#)

date and time functions (PHP)

checkdate function, [Using checkdate](#)

date constants, [Date Constants](#)

date function, [Date and Time Functions](#)

date function format specifiers, [Date and Time Functions](#)

determining current timestamp, [Date and Time Functions](#)

`DateTime` class (PHP), [Date and Time Functions](#)

debugging

PHP magic constants, [Predefined Constants](#)

decrement (--) operator

in PHP, [Arithmetic operators, Operator Precedence](#)

define function (PHP), [Constants, Declaring Constants](#)

`DELETE` command (MySQL), [DELETE](#)

derived classes, [Terminology](#)

`DESC` keyword (MySQL), [ORDER BY](#)

`DESCRIBE` command (MySQL), [Creating a table, Adding a new column, Creating an Index](#)

destructors (PHP), [Destructors](#)

development server setup

AMPPS installation on macOS, [Installing AMPPS on macOS-Accessing the Document Root \(macOS\)](#)

AMPPS installation on Windows, [Installing AMPPS on Windows-Accessing the Document Root \(Windows\)](#)

bundled programming packages, [What Is a WAMP, MAMP, or LAMP?](#)

IDEs (integrated development environments), [Using an IDE](#)

program editors, [Using a Program Editor](#)

recommended browsers, [Setting Up a Development Server](#)

remote access, [Working Remotely-Using FTP, MySQL on a remote server](#)

role of development servers, [Setting Up a Development Server](#)

die function (PHP), [Creating a File](#)

\_\_DIR\_\_ constant, [Predefined Constants](#)

DISTINCT parameter (MySQL), [SELECT DISTINCT](#)

division (/) operator

in PHP, [Arithmetic operators, Operator Precedence](#)

do...while loops

in PHP, [do...while Loops](#)

document root

accessing AMPPS on macOS, [Accessing the Document Root \(macOS\)](#)

accessing AMPPS on Windows, [Accessing the Document Root \(Windows\)](#)

dollar symbol (\$)

in PHP, [The \\$ symbol, Constants, Accessing Objects](#)

DROP keyword (MySQL), [Removing a column](#)

dynamic linking (PHP), [PHP Dynamic Linking-Dynamic Linking in Action](#)

dynamic web design

Apache web server and, [The Apache Web Server](#)

asynchronous communication, [Bringing It All Together](#)

basic web operations, [HTTP and HTML: Berners-Lee's Basics](#)

benefits of modern technologies, [The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5-Using CSS](#)

early history, [Introduction to Dynamic Web Content](#)

HTML5 and, [And Then There's HTML5](#)

open source technologies and, [About Open Source](#)

request/response procedure, [The Request/Response Procedure-The Request/Response Procedure](#)

responsive design, [Handling Mobile Devices](#)

## E

each function (PHP), [The foreach...as Loop](#)

echo statements (PHP)

in arrays, [Multidimensional Arrays](#)

vs. print command, [The Difference Between the echo and print Commands](#)

using operators, [Variable incrementing and decrementing](#)

editors, [Accessing the Document Root \(Windows\)](#), [Accessing the Document Root \(macOS\)](#), [Using a Program Editor](#)

Editra, [Using a Program Editor](#)

else statements

in PHP, [The else Statement](#)

elseif statements (PHP), [The elseif Statement](#)

encapsulation, [Terminology](#)

end function (PHP), [end](#)

\_END...\_END tags (PHP), [Multiple-Line Commands](#)

endswitch command (PHP), [Alternative syntax](#)

`$_ENV` variable, [Superglobal variables](#)

equality (==) operator

in PHP, [Comparison operators](#), [Operator Precedence](#), [Equality](#)

error control (@) operator (PHP), [Associativity](#)

error handling

parse error messages in PHP, [Semicolons](#)

escape characters

in PHP, [Escaping characters](#)

escapeshellcmd function (PHP), [System Calls](#)

exclusive or (xor) logical operator (PHP), [Logical operators](#), [Operator Precedence](#)

exec system call (PHP), [System Calls-System Calls](#)

execution operators (PHP), [Operators](#)

EXPLAIN command (MySQL), [Using EXPLAIN](#)

explicit casting (PHP), [Implicit and Explicit Casting](#)

explode function (PHP), [explode](#)

exponentiation (\*\* operator, in PHP, [Arithmetic operators](#)  
expressions

in PHP, [Expressions-Literals and Variables](#)  
extends keyword (PHP), [Inheritance](#)  
extract function (PHP), [extract](#)

## F

fgets function (PHP), [Reading from Files](#)

file handling (PHP)

copying files, [Copying Files](#)

deleting files, [Deleting a File](#)

file creation, [Creating a File](#)

file pointers and file handles, [Updating Files](#)

file\_exists function, [Checking Whether a File Exists](#)

fopen modes, [Creating a File](#)

form data validation, [Validation](#)

including/requiring files, [Including and Requiring Files](#)

locking files for multiple accesses, [Locking Files for Multiple  
Accesses](#)

moving files, [Moving a File](#)

naming rules, [File Handling](#)

reading files in total, [Reading an Entire File](#)

reading from files, [Reading from Files](#)

sequence of, [Creating a File](#)

updating files, [Updating Files](#)

uploading files, [Uploading Files-Validation](#)

`__FILE__` constant, [Predefined Constants](#)

`$_FILES` array (PHP), [Using `\$\_FILES`](#)

`$_FILES` variable, [Superglobal variables](#)

`file_exists` function (PHP), [Checking Whether a File Exists](#)

`file_get_contents` (PHP), [Reading an Entire File](#)

`final` keyword (PHP), [Final methods](#)

`flock` function (PHP), [Locking Files for Multiple Accesses](#)

flow control (PHP)

conditionals, [Conditionals-The ? Operator](#)

dynamic linking in action, [Dynamic Linking in Action](#)

dynamic linking in PHP, [PHP Dynamic Linking](#)

expressions, [Expressions-Literals and Variables](#)

implicit and explicit casting, [Implicit and Explicit Casting](#)

looping, [Looping-The continue Statement](#)

operators, [Operators-Logical operators](#)

`fopen` function (PHP), [Creating a File](#)

for loops

in PHP, [for Loops](#)

`foreach...as` loops (PHP), [The foreach...as Loop](#)

form data validation, [Validation](#) (see also file handling)

forward slash (/)

/\* and \*/ in multiline comments in PHP, [Using Comments](#)

fread function (PHP), [Reading from Files](#)

fseek function (PHP), [Updating Files](#)

FTP (File Transfer Protocol), [Using FTP](#)

FULLTEXT indexes (MySQL), [Creating a FULLTEXT index](#)

FUNCTION constant, [Predefined Constants](#)

functions (MySQL)

benefits of, [MySQL Functions](#)

functions (PHP)

advantages of, [PHP Functions and Objects](#)

arguments accepted by, [PHP Functions](#)

array functions, [Using Array Functions](#)-end

basics of, [Functions](#)

defining, [Defining a Function](#)

including and requiring files, [Including and Requiring Files](#)

naming rules, [Defining a Function](#)

nesting for execution order, [Returning a Value](#)

passing arguments by reference, [Passing Arguments by Reference](#)

purpose of, [PHP Functions and Objects](#)

returning arrays from, [Returning an Array](#)

returning global variables, [Returning Global Variables](#)  
returning values from, [Returning a Value](#)  
using (calling), [PHP Functions](#)  
variable scope, [Recap of Variable Scope](#)  
version compatibility and, [PHP Version Compatibility](#)  
function\_exists function (PHP), [PHP Version Compatibility](#)

## G

\$\_GET variable, [Superglobal variables](#)  
get\_password method (PHP), [Writing Methods](#)  
global variables  
    in PHP, [Returning Global Variables](#)  
    in PHP), [Global variables](#)  
goto fail bug, [The if Statement](#)  
GRANT command (MySQL), [Creating users](#)  
greater than (>) operator  
    in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)  
greater than or equal to (>=) operator  
    in PHP, [Comparison operators](#), [Operator Precedence](#), [Comparison operators](#)  
GROUP BY keyword (MySQL), [GROUP BY](#)

## H

heredoc (<<<) operator (PHP), [Multiple-Line Commands](#)

HTML (Hypertext Markup Language)

basics of, [HTTP and HTML: Berners-Lee's Basics](#)

converting characters to, [Superglobals and security](#)

incorporating with PHP, [Incorporating PHP Within HTML](#)

origins of, [Introduction to Dynamic Web Content](#)

setting colors with printf, [Using printf](#)

HTML5

development of, [And Then There's HTML5](#)

XHTML syntax and, [And Then There's HTML5, XHTML or HTML5?](#)

htmlentities function (PHP), [Superglobals and security](#)

htmlspecialchars function (PHP), [System Calls](#)

HTTP (Hypertext Transfer Protocol)

basics of, [HTTP and HTML: Berners-Lee's Basics](#)

origins of, [Introduction to Dynamic Web Content](#)

human-readable data, [Declaring a Class](#)

I

identity (==) operator

in PHP, [Comparison operators, Operator Precedence](#)

IDEs (integrated development environments), [Using an IDE, Introduction to PHP](#)

if statements (PHP)

flow control using, [The if Statement](#)

using operators, [Variable incrementing and decrementing](#)

if...else statement (PHP), [The else Statement](#)

if...elseif...else construct (PHP), [The elseif Statement](#)

implicit casting (PHP), [Implicit and Explicit Casting](#)

include statement (PHP), [The include Statement](#)

include\_once statement (PHP), [Using include\\_once](#)

increment (++) operator

in PHP, [Arithmetic operators, Operator Precedence](#)

indexes (MySQL)

creating, [Creating an Index-Creating a FULLTEXT index](#)

joining tables together, [Joining Tables Together-Using AS](#)

purpose of, [Indexes](#)

querying databases, [Querying a MySQL Database-GROUP BY](#)

using logical operators, [Using Logical Operators](#)

inequality (!=) operator

in PHP, [Comparison operators, Operator Precedence](#)

inheritance, [Terminology, Inheritance-Final methods](#)

InnoDB, [Creating a table](#)

INSERT command (MySQL), [Adding data to a table](#)

instances, [Terminology](#)

interfaces, [Terminology](#)

internet media types, [Using \\$\\_FILES](#)

`is_array` function (PHP), [is\\_array](#)

## J

JavaScript

benefits of, [The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5-Using CSS](#)

`JOIN...ON` construct (MySQL), [JOIN...ON](#)

justification, right or left, [String Padding](#)

## K

key/value pairs

associative arrays in PHP, [The foreach...as Loop](#)

keys (MySQL), [Primary Keys: The Keys to Relational Databases](#)

## L

LAMP (Linux, Apache, MySQL, and PHP), [What Is a WAMP, MAMP, or LAMP?, Installing a LAMP on Linux](#)

less than (<) operator

in PHP, [Comparison operators, Operator Precedence, Comparison operators](#)

less than or equal to (<=) operator

in PHP, [Comparison operators, Operator Precedence, Comparison operators](#)

LIMIT keyword (MySQL), [LIMIT](#)

[\\_\\_LINE\\_\\_](#) constant, [Predefined Constants](#)

links

dynamic linking in PHP, [PHP Dynamic Linking-Dynamic Linking in Action](#)

list function (PHP), [The foreach...as Loop](#)

literals

in PHP, [Literals and Variables](#), [Implicit and Explicit Casting](#)

local variables (PHP), [Local variables](#), [Recap of Variable Scope](#)

logical operators

in MySQL, [Using Logical Operators](#)

in PHP, [Logical operators](#), [Operators](#), [Operator Precedence](#), [Logical operators](#)

looping (PHP)

basics of, [Looping](#)

breaking out of loops, [Breaking Out of a Loop](#)

continue statements, [The continue Statement](#)

do...while loops, [do...while Loops](#)

for loops, [for Loops](#)

foreach...as loops, [The foreach...as Loop](#)

while loops, [while Loops](#)

M

magic constants, [Predefined Constants](#)

MAMP (Mac, Apache, MySQL, and PHP), [What Is a WAMP, MAMP, or LAMP?](#)

many-to-many relationship, [Many-to-Many](#)

MariaDB, [MariaDB: The MySQL Clone](#)

`MATCH...AGAINST` construct (MySQL), [MATCH...AGAINST](#)

`_METHOD_` constant, [Predefined Constants](#)

methods

in PHP, [Terminology](#), [Writing Methods](#), [Property and Method Scope](#)

MIME (Multipurpose Internet Mail Extension), [Using \\$\\_FILES](#)

minus sign (-)

unary operator in PHP, [Associativity](#)

`mktime` function (PHP), [Date and Time Functions](#)

`MODIFY` keyword (MySQL), [Changing the data type of a column](#)

modulus (%) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

modulus and assignment (%=) operator

in PHP, [Assignment operators](#)

multidimensional arrays

in PHP, [Multidimensional Arrays](#)

multiple-line comments, [Using Comments](#)

multiplication (\*) operator

in PHP, [Arithmetic operators](#), [Operator Precedence](#)

# MySQL

anonymity and, [Databases and Anonymity](#)

backup and restore in, [Backing Up and Restoring-Planning Your Backups](#)

basics of, [MySQL Basics](#)

benefits of, [Preface](#), [The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5-Using CSS](#), [Introduction to MySQL](#)

built-in functions, [MySQL Functions](#)

cancelling commands in, [Canceling a command](#)

command prompts, [The semicolon](#)

command-line access by OS type, [Accessing MySQL via the Command Line-MySQL on a remote server](#)

command-line interface use, [Using the Command-Line Interface](#)

common commands, [MySQL Commands](#)

data types, [Data Types-The AUTO\\_INCREMENT attribute](#)

database creation, [Creating a database](#)

database design considerations, [Database Design](#)

database terms, [Summary of Database Terms](#)

default storage engine for, [Creating a table](#), [Transaction Storage Engines](#)

EXPLAIN command, [Using EXPLAIN](#)

multiple-line commands, [The semicolon](#)

normalization process, [Normalization-When Not to Use Normalization](#)

phpMyAdmin access, [Accessing MySQL via phpMyAdmin](#)  
primary keys in, [Primary Keys: The Keys to Relational Databases](#)  
relationships in, [Relationships-Databases and Anonymity](#)  
remote access, [Working Remotely, MySQL on a remote server](#)  
stopwords in, [Creating a FULLTEXT index](#)  
table creation, [Creating a table](#)  
tables, adding data to, [Adding data to a table](#)  
tables, backup and restore, [Using mysqldump-Planning Your Backups](#)  
tables, deleting, [Deleting a table](#)  
tables, joining, [Joining Tables Together-Using AS](#)  
tables, renaming, [Renaming a table](#)  
transactions in, [Transactions-Using EXPLAIN](#)  
user creation, [Creating users](#)  
working with columns, [Changing the data type of a column-Removing a column](#)  
working with indexes, [Indexes-Using Logical Operators](#)  
mysqldump command, [Using mysqldump-Planning Your Backups](#)

## N

names and naming  
constants in PHP, [Constants](#)  
files in PHP, [File Handling](#)  
functions in PHP, [Defining a Function](#)

SQL commands, MySQL Commands

tables, MySQL Commands

variables in PHP, Variable-naming rules

—NAMESPACE—, Predefined Constants

NATURAL JOIN keyword (MySQL), NATURAL JOIN

newline (\n) character, Escaping characters

normalization process

appropriate use of, When Not to Use Normalization

First Normal Form, First Normal Form

goal of, Normalization

schemas for, Normalization

Second Normal Form, Second Normal Form-Second Normal Form

Third Normal Form, Third Normal Form

Not (!) logical operator

in PHP, Logical operators, Operator Precedence, Logical operators

not equal (<>) operator (PHP), Comparison operators, Operator Precedence

not identical (!==) operator

in PHP, Comparison operators, Operator Precedence

NULL value, TRUE or FALSE?

numeric arrays

in PHP, Numerically Indexed Arrays

numeric data types (MySQL), Numeric data types

numeric variables

in PHP, [Numeric variables](#)

## O

object-oriented programming (OOP), [PHP Objects](#)

objects (PHP)

accessing, [Accessing Objects](#)

basics of, [PHP Objects](#)

cloning, [Cloning Objects](#)

constructors, [Constructors, Subclass constructors](#)

creating, [Creating an Object](#)

declaring classes, [Declaring a Class](#)

declaring constants, [Declaring Constants](#)

declaring properties, [Declaring Properties](#)

destructors, [Destructors](#)

inheritance, [Inheritance-Final methods](#)

property and method scope, [Property and Method Scope](#)

purpose of, [PHP Functions and Objects](#)

static methods, [Static Methods](#)

static properties, [Static Properties](#)

terminology surrounding, [Terminology](#)

writing methods, [Writing Methods](#)

occurrences, [Terminology](#)

one-to-many relationship, [One-to-Many](#)

one-to-one relationship, [One-to-One](#)

open source technologies, [About Open Source](#)

operands, [Operators](#)

operators (PHP)

arithmetic, [Arithmetic operators](#)

assignment, [Assignment operators](#)

associativity of, [Associativity](#)

basics of, [Operators](#)

comparison, [Comparison operators](#)

logical, [Logical operators](#)

overview of types, [Operators](#)

precedence of, [Operator Precedence](#)-[Operator Precedence](#)

relational operators, [Relational Operators](#)-[Logical operators](#)

or (low precedence) logical operator (PHP), [Logical operators](#), [Operator Precedence](#)

Or () bitwise operator

in PHP, [Operator Precedence](#)

Or (||) logical operator

in PHP, [Logical operators](#), [Operator Precedence](#), [Logical operators](#)

ORDER BY keyword (MySQL), [ORDER BY](#)

P

parent keyword (PHP), [The parent keyword](#)  
parentheses ()  
casting operators in PHP, [Implicit and Explicit Casting](#)  
in functions in PHP, [PHP Functions](#)  
operator precedence and, [Operator Precedence](#)  
parse error messages (PHP), [Semicolons](#)  
pass-by-reference feature, [Passing Arguments by Reference](#)  
password property, [Writing Methods](#)  
PHP programming language  
arrays in, [Arrays, PHP Arrays-end](#)  
basic syntax, [Basic Syntax](#)  
benefits of, [Preface, The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5-Using CSS](#)  
code examples, obtaining and using, [This Book's Examples](#)  
comments, [Using Comments](#)  
constants in, [Constants](#)  
constants, predefined, [Predefined Constants, TRUE or FALSE?](#)  
date and time functions, [Date and Time Functions-Using checkdate](#)  
echo vs. print commands, [The Difference Between the echo and print Commands](#)  
file handling, [File Handling-Validation](#)  
flow control in, [Expressions and Control Flow in PHP-Dynamic Linking in Action](#)

functions in, [Functions](#), [PHP Functions and Objects](#)-[PHP Version Compatibility](#)

IDEs (integrated development environments), [Using an IDE](#), [Introduction to PHP](#)

incorporating with HTML, [Incorporating PHP Within HTML](#)

multiple-line commands, [Multiple-Line Commands](#)-[Multiple-Line Commands](#)

objects in, [PHP Objects](#)-[Final methods](#)

operators in, [Operators](#)-[Logical operators](#)

printf function, [Using printf](#)-[Using sprintf](#)

system calls, [System Calls](#)-[System Calls](#)

variable assignment, [Variable Assignment](#)-[Escaping characters](#)

variable typing, [Variable Typing](#)

variables in, [Variables](#)-[Variable-naming rules](#)

version compatibility, [PHP Version Compatibility](#)

version selection, [PHP Functions and Objects](#)

XHTML vs. HTML5, [XHTML or HTML5?](#)

<?php?> tag, [Incorporating PHP Within HTML](#)

phpDesigner IDE, [Using an IDE](#)

phpinfo function, [PHP Functions](#)

phpMyAdmin, accessing MySQL via, [Accessing MySQL via phpMyAdmin](#)

phpversion function, [PHP Version Compatibility](#)

plain text editors, [Using a Program Editor](#)

port assignment, [String variables](#)

`$_POST` variable, [Superglobal variables](#)

precedence, [Operator Precedence](#)

primary keys, [Primary keys](#), [Primary Keys: The Keys to Relational Databases](#)

print command (PHP), [The Difference Between the echo and print Commands](#), [Returning a Value](#)

`printf` function (PHP)

conversion specifier components, [Precision Setting](#)

precision setting, [Precision Setting](#)-[Precision Setting](#)

vs. `print` and `echo` functions, [Using printf](#)

`printf` conversion specifiers, [Using printf](#)

`sprintf` function, [Using sprintf](#)

string conversion specifier components, [String Padding](#)

string padding, [String Padding](#)-[String Padding](#)

`print_r` function (PHP), [Declaring a Class](#)

`private` keyword (PHP), [Property and Method Scope](#)

program editors, [Using a Program Editor](#)

properties (PHP)

declaring, [Declaring Properties](#)

defined, [Terminology](#)

scope of, [Property and Method Scope](#)

`protected` keyword (PHP), [Property and Method Scope](#)

public keyword (PHP), [Property and Method Scope](#)

## Q

quotation marks, double ("")

in PHP heredocs, [Multiple-Line Commands](#)

in PHP multi-line commands, [Multiple-Line Commands](#)

in PHP strings, [String types](#)

quotation marks, single ()

in PHP heredocs, [Multiple-Line Commands](#)

in PHP strings, [String types](#)

## R

records, defined, [MySQL Basics](#) (see also MySQL)

relational operators (PHP)

comparison operators, [Comparison operators](#)

equality, [Relational Operators](#)

logical operators, [Logical operators](#)

relationships, in databases

many-to-many, [Many-to-Many](#)

one-to-many, [One-to-Many](#)

one-to-one relationship, [One-to-One](#)

rename function (PHP), [Moving a File](#)

`$_REQUEST` variable, [Superglobal variables](#)

request/response procedure, [The Request/Response Procedure-The Request/Response Procedure](#)

require statement (PHP), [Using require and require\\_once](#)

require\_once statement (PHP), [Using require and require\\_once](#)

reset function (PHP), [reset](#)

return statements

in PHP, [Defining a Function](#)

ROLLBACK command (MySQL), [Using ROLLBACK](#)

rows, defined, [MySQL Basics](#) (see also MySQL)

RSS feeds, [Date Constants](#)

## S

sanitization, [Superglobals and security](#)

scope

property/method scope in PHP, [Property and Method Scope](#)

variable scope in PHP, [Variable Scope-Superglobals and security, Recap of Variable Scope](#)

scope resolution (::) operator, [Static Methods](#)

security issues

bundled programming packages, [What Is a WAMP, MAMP, or LAMP?](#)

curly braces ({ }), [The if Statement](#)

goto fail bug, [The if Statement](#)

phpinfo function, [PHP Functions](#)

superglobal variables, [Superglobals and security](#)

Y2K38 bug, [Date and Time Functions](#)

SELECT command (MySQL), [SELECT](#)

SELECT COUNT command (MySQL), [SELECT COUNT](#)

SELECT DISTINCT command (MySQL), [SELECT DISTINCT](#)

self keyword (PHP), [Declaring Constants](#)

semicolons (;

in MySQL, [The semicolon](#)

in PHP, [Semicolons, for Loops](#)

`$_SERVER` variable, [Superglobal variables](#)

servers

development server setup, [Setting Up a Development Server-Using an IDE](#)

production server security, [What Is a WAMP, MAMP, or LAMP?](#)

request/response procedure, [The Request/Response Procedure-The Request/Response Procedure](#)

role in internet communications, [HTTP and HTML: Berners-Lee's Basics](#)

`$_SESSION` variable, [Superglobal variables](#)

SHOW command (MySQL), [Windows users](#)

shuffle function (PHP), [shuffle](#)

signed numbers, [Numeric data types](#)

sort function (PHP), [sort](#)

special characters, [Escaping characters](#)

sprintf function (PHP), [Using sprintf](#)

SQL (Structured Query Language), [MySQL Basics](#)

square brackets ([])

- accessing array elements, [Multidimensional Arrays](#)
- in PHP function definitions, [Defining a Function](#)

START TRANSACTION statement (MySQL), [Using BEGIN](#)

statements, in PHP, [Literals and Variables](#), [The if Statement-Alternative syntax](#)

static methods

- in PHP, [Static Methods](#)

static properties

- in PHP, [Static Properties](#)

static variables (PHP), [Static variables](#), [Recap of Variable Scope](#)

stopwords, [Creating a FULLTEXT index](#)

string variables (PHP), [Variables](#)

strings (PHP)

- concatenation of, [String concatenation](#)
- escaping characters in, [Escaping characters](#)

multiple-line commands, [Multiple-Line Commands-Multiple-Line Commands](#)

special characters in, [Escaping characters](#)

string padding, [String Padding-String Padding](#)

types of, [String types](#)

`strrev` function (PHP), [PHP Functions](#)

`strtolower` function (PHP), [Returning a Value](#)

`strtoupper` function (PHP), [PHP Functions](#)

subclasses, [Terminology](#)

subtraction (-) operator

    in PHP, [Arithmetic operators](#), [Operator Precedence](#), [Associativity](#)

superclasses, [Terminology](#)

superglobal variables (PHP), [Superglobal variables](#)

switch statements

    in PHP, [The switch Statement](#)-[Alternative syntax](#)

syntax (PHP), [Basic Syntax](#)

system calls (PHP), [System Calls](#)-[System Calls](#)

## T

tab (\t) character, [Escaping characters](#), [Multidimensional Arrays](#)

tables (see also MySQL)

    adding data to, [Adding data to a table](#)

    backup and restore, [Using mysqldump](#)

    defined, [MySQL Basics](#)

    deleting, [Deleting a table](#)

    joining, [Joining Tables Together](#)-[Using AS](#)

    naming rules, [MySQL Commands](#)

renaming, [Renaming a table](#)

ternary (?:) operator

in PHP, [Operators](#), [Operator Precedence](#), [The ? Operator](#)

TEXT data types (MySQL), [The TEXT data types](#)

TIME and DATE data types (MySQL), [DATE and TIME types](#)

time function (PHP), [Date and Time Functions](#)

timestamps, determining current, [Date and Time Functions](#)

TRUE/FALSE values, [TRUE or FALSE?](#), [Relational Operators](#)

two-dimensional arrays, [Two-dimensional arrays](#)

typographical conventions, [Conventions Used in This Book](#)

## U

ucfirst function (PHP), [Returning a Value](#)

unary operators, [Operators](#)

underscore, double (\_), [Predefined Constants](#), [Writing Methods](#)

unlink function (PHP), [Deleting a File](#)

unsigned numbers, [Numeric data types](#)

UPDATE...SET construct (MySQL), [UPDATE...SET](#)

uploading files, [Uploading Files-Validation](#)

## V

validation

form data validation using PHP, [Validation](#)

VALUES keyword (MySQL), [Adding data to a table](#)

VARCHAR data type (MySQL), [Data Types](#)

variable substitution, [String types](#)

variables (PHP)

assignment, [Variable Assignment-Escaping characters](#)

explicit casting of, [Implicit and Explicit Casting](#)

flow control and, [Literals and Variables](#)

global, [Global variables](#), [Returning Global Variables](#)

incrementing and decrementing, [Variable incrementing and decrementing](#)

local, [Local variables](#), [Recap of Variable Scope](#)

multiple-line commands, [Multiple-Line Commands](#)

naming rules, [Variable-naming rules](#)

numeric, [Numeric variables](#)

scope of, [Variable Scope](#), [Recap of Variable Scope](#)

static, [Static variables](#), [Recap of Variable Scope](#)

string, [Variables](#)

superglobal, [Superglobal variables](#)

typing, [Variable Typing](#)

Visual Studio, [Installing AMPPS on Windows](#)

W

WAMP (Windows, Apache, MySQL, and PHP), [What Is a WAMP, MAMP, or LAMP?](#), [Alternative WAMPs](#)

WHERE keyword (MySQL), [WHERE](#)

while loops

in PHP, [while Loops](#)

whitespace

avoiding, [Incorporating PHP Within HTML](#)

preserving, [Multiple-Line Commands](#)

WordPress, [Dynamic Linking in Action](#)

World Wide Web Consortium, [Date Constants](#)

## X

XHTML syntax, [And Then There's HTML5, XHTML or HTML5?](#)

xor (exclusive or) logical operator (PHP), [Logical operators](#), [Operator Precedence](#)

## Y

Y2K38 bug, [Date and Time Functions](#)

YEAR data type, [Data Types](#)

## Z

ZEROFILL qualifier, [Numeric data types](#)

1. Learning PHP, MySQL & JavaScript

2. Learning PHP, MySQL & JavaScript

3. Preface

a. Audience

b. Assumptions This Book Makes

c. Organization of This Book

d. Supporting Books

e. Conventions Used in This Book

f. Using Code Examples

g. O'Reilly Online Learning

h. How to Contact Us

i. Acknowledgments

4. 1. Introduction to Dynamic Web Content

a. HTTP and HTML: Berners-Lee's Basics

b. The Request/Response Procedure

c. The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5

i. MariaDB: The MySQL Clone

ii. Using PHP

iii. Using MySQL

iv. Using JavaScript

v. Using CSS

- d. And Then There's HTML5
- e. The Apache Web Server
- f. Handling Mobile Devices
- g. About Open Source
- h. Bringing It All Together
- i. Questions

## 5. 2. Setting Up a Development Server

- a. What Is a WAMP, MAMP, or LAMP?
- b. Installing AMPPS on Windows
  - i. Testing the Installation
  - ii. Accessing the Document Root (Windows)
  - iii. Alternative WAMPs
- c. Installing AMPPS on macOS
  - i. Accessing the Document Root (macOS)
- d. Installing a LAMP on Linux
- e. Working Remotely
  - i. Logging In
  - ii. Using FTP
- f. Using a Program Editor
- g. Using an IDE
- h. Questions

## 6. 3. Introduction to PHP

- a. Incorporating PHP Within HTML
- b. This Book's Examples
- c. The Structure of PHP
  - i. Using Comments
  - ii. Basic Syntax
  - iii. Variables
  - iv. Operators
  - v. Variable Assignment
  - vi. Multiple-Line Commands
  - vii. Variable Typing
  - viii. Constants
  - ix. Predefined Constants
  - x. The Difference Between the echo and print Commands
  - xi. Functions
  - xii. Variable Scope
- d. Questions

## 7. 4. Expressions and Control Flow in PHP

- a. Expressions
  - i. TRUE or FALSE?
  - ii. Literals and Variables
- b. Operators
  - i. Operator Precedence

- ii. **Associativity**
- iii. **Relational Operators**

- c. **Conditionals**

- i. **The if Statement**
- ii. **The else Statement**
- iii. **The elseif Statement**
- iv. **The switch Statement**
- v. **The ? Operator**

- d. **Looping**

- i. **while Loops**
- ii. **do...while Loops**
- iii. **for Loops**
- iv. **Breaking Out of a Loop**
- v. **The continue Statement**

- e. **Implicit and Explicit Casting**
- f. **PHP Dynamic Linking**
- g. **Dynamic Linking in Action**
- h. **Questions**

## 8. 5. PHP Functions and Objects

- a. **PHP Functions**
  - i. **Defining a Function**
  - ii. **Returning a Value**

- iii. Returning an Array
  - iv. Passing Arguments by Reference
  - v. Returning Global Variables
  - vi. Recap of Variable Scope
- b. Including and Requiring Files
    - i. The include Statement
    - ii. Using include\_once
    - iii. Using require and require\_once
  - c. PHP Version Compatibility
  - d. PHP Objects
    - i. Terminology
    - ii. Declaring a Class
    - iii. Creating an Object
    - iv. Accessing Objects
    - v. Cloning Objects
    - vi. Constructors
    - vii. Destructors
    - viii. Writing Methods
    - ix. Declaring Properties
    - x. Declaring Constants
    - xi. Property and Method Scope
    - xii. Static Methods

xiii. **Static Properties**

xiv. **Inheritance**

e. **Questions**

## 9. 6. PHP Arrays

a. **Basic Access**

i. **Numerically Indexed Arrays**

ii. **Associative Arrays**

iii. **Assignment Using the array Keyword**

b. **The foreach...as Loop**

c. **Multidimensional Arrays**

d. **Using Array Functions**

i. **is\_array**

ii. **count**

iii. **sort**

iv. **shuffle**

v. **explode**

vi. **extract**

vii. **compact**

viii. **reset**

ix. **end**

e. **Questions**

## 10. 7. Practical PHP

- a. Using printf
  - i. Precision Setting
  - ii. String Padding
  - iii. Using sprintf
- b. Date and Time Functions
  - i. Date Constants
  - ii. Using checkdate
- c. File Handling
  - i. Checking Whether a File Exists
  - ii. Creating a File
  - iii. Reading from Files
  - iv. Copying Files
  - v. Moving a File
  - vi. Deleting a File
  - vii. Updating Files
  - viii. Locking Files for Multiple Accesses
  - ix. Reading an Entire File
  - x. Uploading Files
- d. System Calls
- e. XHTML or HTML5?
- f. Questions

## 11. 8. Introduction to MySQL

- a. MySQL Basics
- b. Summary of Database Terms
- c. Accessing MySQL via the Command Line
  - i. Starting the Command-Line Interface
  - ii. Using the Command-Line Interface
  - iii. MySQL Commands
  - iv. Data Types
- d. Indexes
  - i. Creating an Index
  - ii. Querying a MySQL Database
  - iii. Joining Tables Together
  - iv. Using Logical Operators
- e. MySQL Functions
- f. Accessing MySQL via phpMyAdmin
- g. Questions

## 12. 9. Mastering MySQL

- a. Database Design
  - i. Primary Keys: The Keys to Relational Databases
- b. Normalization
  - i. First Normal Form
  - ii. Second Normal Form
  - iii. Third Normal Form

iv. When Not to Use Normalization

c. Relationships

- i. One-to-One
- ii. One-to-Many
- iii. Many-to-Many
- iv. Databases and Anonymity

d. Transactions

- i. Transaction Storage Engines
- ii. Using BEGIN
- iii. Using COMMIT
- iv. Using ROLLBACK

e. Using EXPLAIN

f. Backing Up and Restoring

- i. Using mysqldump
- ii. Creating a Backup File
- iii. Restoring from a Backup File
- iv. Dumping Data in CSV Format
- v. Planning Your Backups

g. Questions

13. 10. What's new in PHP 8 and MySQL 8

a. PHP 8

- i. Named Parameters

- ii. Attributes
  - iii. Constructor Properties
  - iv. Just-In-Time Compilation
  - v. Union Types
  - vi. Null-safe Operator
  - vii. Match Expressions
  - viii. New Functions
- b. MySQL 8
- i. Updates to SQL
  - ii. JavaScript Object Notation (JSON)
  - iii. Geography Support
  - iv. Reliability
  - v. Speed
  - vi. Management
  - vii. Security
  - viii. Performance
- c. Questions

14. A. Solutions to the Chapter Questions
- a. Chapter 1 Answers
  - b. Chapter 2 Answers
  - c. Chapter 3 Answers
  - d. Chapter 4 Answers

- e. Chapter 5 Answers
- f. Chapter 6 Answers
- g. Chapter 7 Answers
- h. Chapter 8 Answers
- i. Chapter 9 Answers
- j. Chapter 10 Answers
- k. Chapter 11 Answers
- l. Chapter 12 Answers
- m. Chapter 13 Answers
- n. Chapter 14 Answers
- o. Chapter 15 Answers
- p. Chapter 16 Answers
- q. Chapter 17 Answers
- r. Chapter 18 Answers
- s. Chapter 19 Answers
- t. Chapter 20 Answers
- u. Chapter 21 Answers
- v. Chapter 22 Answers
- w. Chapter 23 Answers
- x. Chapter 24 Answers
- y. Chapter 25 Answers
- z. Chapter 26 Answers
- aa. Chapter 27 Answers

ab. [Chapter 28 Answers](#)

ac. [Chapter 29 Answers](#)

## 15. Index