Learning PHP, MySQL, & JavaScript

With jQuery, CSS, & HTML5

Learning PHP, MySQL, & JavaScript

With jQuery, CSS, & HTML5

Fifth Edition

Robin Nixon

1. Learning PHP, MySQL, & JavaScript
2. Preface
    1. Audience
    2. Assumptions This Book Makes
    3. Organization of This Book
    4. Supporting Books
    5. Conventions Used in This Book
    6. Using Code Examples
    7. Safari® Books Online
    8. How to Contact Us
    9. Acknowledgments
3. 1. Introduction to Dynamic Web Content
    1. HTTP and HTML: Berners-Lee's Basics
    2. The Request/Response Procedure
    3. The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5
        1. Using PHP
        2. Using MySQL
        3. Using JavaScript
        4. Using CSS
    4. And Then There's HTML5
    5. The Apache Web Server
    6. Handling mobile devices
    7. About Open Source
    8. Bringing It All Together
    9. Questions
4. 2. Setting Up a Development Server
    1. What Is a WAMP, MAMP, or LAMP?
    2. Installing Ampps on Windows
        1. Testing the Installation
    3. Installing Ampps on Mac OS X
    4. Installing a LAMP on Linux
    5. Working Remotely
        1. Logging In
        2. Using FTP
    6. Using a Program Editor
    7. Using an IDE
    8. Questions
5. 3. Introduction to PHP
    1. Incorporating PHP Within HTML
    2. This Book's Examples

3. The Structure of PHP
    1. Using Comments
    2. Basic Syntax
    3. Variables
    4. Operators
    5. Variable Assignment
    6. Multiple-Line Commands
    7. Variable Typing
    8. Constants
    9. Predefined Constants
    10. The Difference Between the echo and print Commands
    11. Functions
    12. Variable Scope
4. Questions

6. 4. Expressions and Control Flow in PHP
    1. Expressions
        1. TRUE or FALSE?
        2. Literals and Variables
    2. Operators
        1. Operator Precedence
        2. Associativity
        3. Relational Operators
    3. Conditionals
        1. The if Statement
        2. The else Statement
        3. The elseif Statement
        4. The switch Statement
        5. The ? Operator
    4. Looping
        1. while Loops
        2. do...while Loops
        3. for Loops
        4. Breaking Out of a Loop
        5. The continue Statement
    5. Implicit and Explicit Casting
    6. PHP Dynamic Linking
    7. Dynamic Linking in Action
    8. Questions
7. About the Author

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, jQuery, 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 the Web 2.0 technology known as Ajax will obtain a thorough grounding in all of these core technologies: PHP, MySQL, JavaScript, CSS, and HTML5, and learn the basics of the jQuery and jQuery Mobile libraries 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. Following that, you will 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 and in-browser validation and error handling, plus a comprehensive primer on using the popular jQuery 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 jQuery libraries can make your development job a great deal easdier, and then moving 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 could 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. To learn more about any of these titles, simply enter the ISBN shown next to it into the search box at *http://oreilly.com* or at any good online book seller's website.

- *Dynamic HTML: The Definitive Reference* (9780596527402) by Danny Goodman
- *PHP in a Nutshell* (9780596100674) by Paul Hudson
- *MySQL in a Nutshell* (9780596514334) by Russell Dyer
- *JavaScript: The Definitive Guide* (9780596805524) by David Flanagan
- *CSS: The Definitive Guide* (9780596527334) by Eric A. Myer
- *HTML5: The Missing Manual* (9781449363260) 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.

`Constant width`
: Indicates command-line options, variables and other code elements, HTML tags, macros, and the contents of files.

`Constant width bold`
: Shows program output or highlighted 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

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 CD-ROM 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. There is a companion website to this book at *http://lpmj.net*, where you can download all the examples from this book in a single zip file.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning PHP, MySQL & JavaScript, 5th Edition* by Robin Nixon (O'Reilly). Copyright 2018 Robin Nixon,

."

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*.

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 *http://bit.ly/lpmjch_4e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, 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 once again thank my editor, Andy Oram, and everyone who worked so hard on this book, including

??? ??? for his comprehensive technical review, ??? ??? for overseeing production, ??? ??? for copy editing, ??? ??? for proofreading, Robert Romano and Rebecca Demarest for their illustrations, ??? ??? for interior design, ??? ??? for creating the index, Karen Montgomery for the original sugar glider front cover design, ??? ??? for the latest book cover, 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, tools that we now take 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 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 take place 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 these up to clients directly from this cache, rather than fetching them all the way from the source server.

A web server can usually handle multiple simultaneous connections and—when not communicating with a client—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).

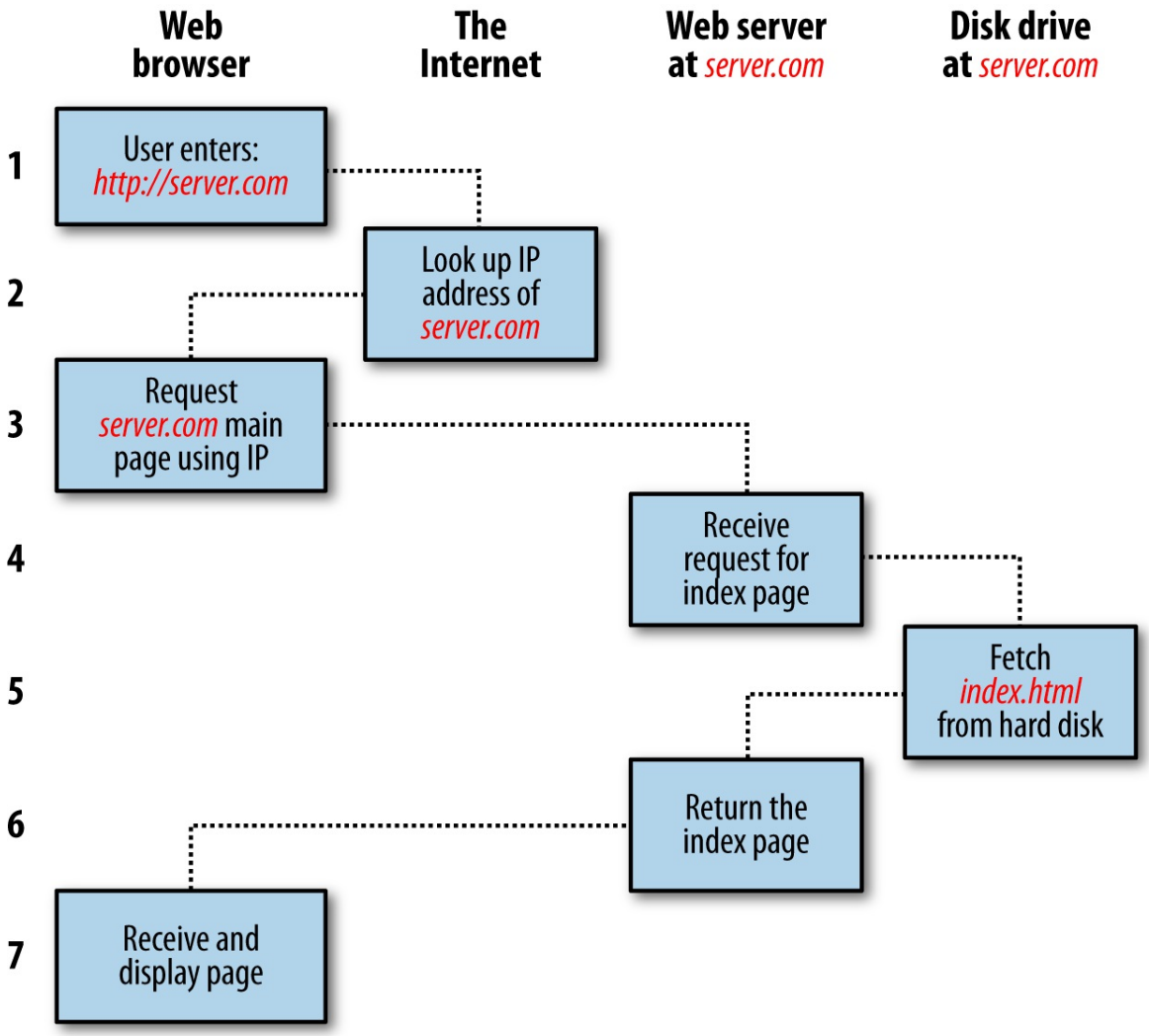

Figure 1-1. The basic client/server request/response sequence

Each step in the request and response sequence is 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 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 page is retrieved by the server and returned to the browser.
7. Your browser displays the web page.

For an average web page, this process 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 looked 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 its 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. The 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).

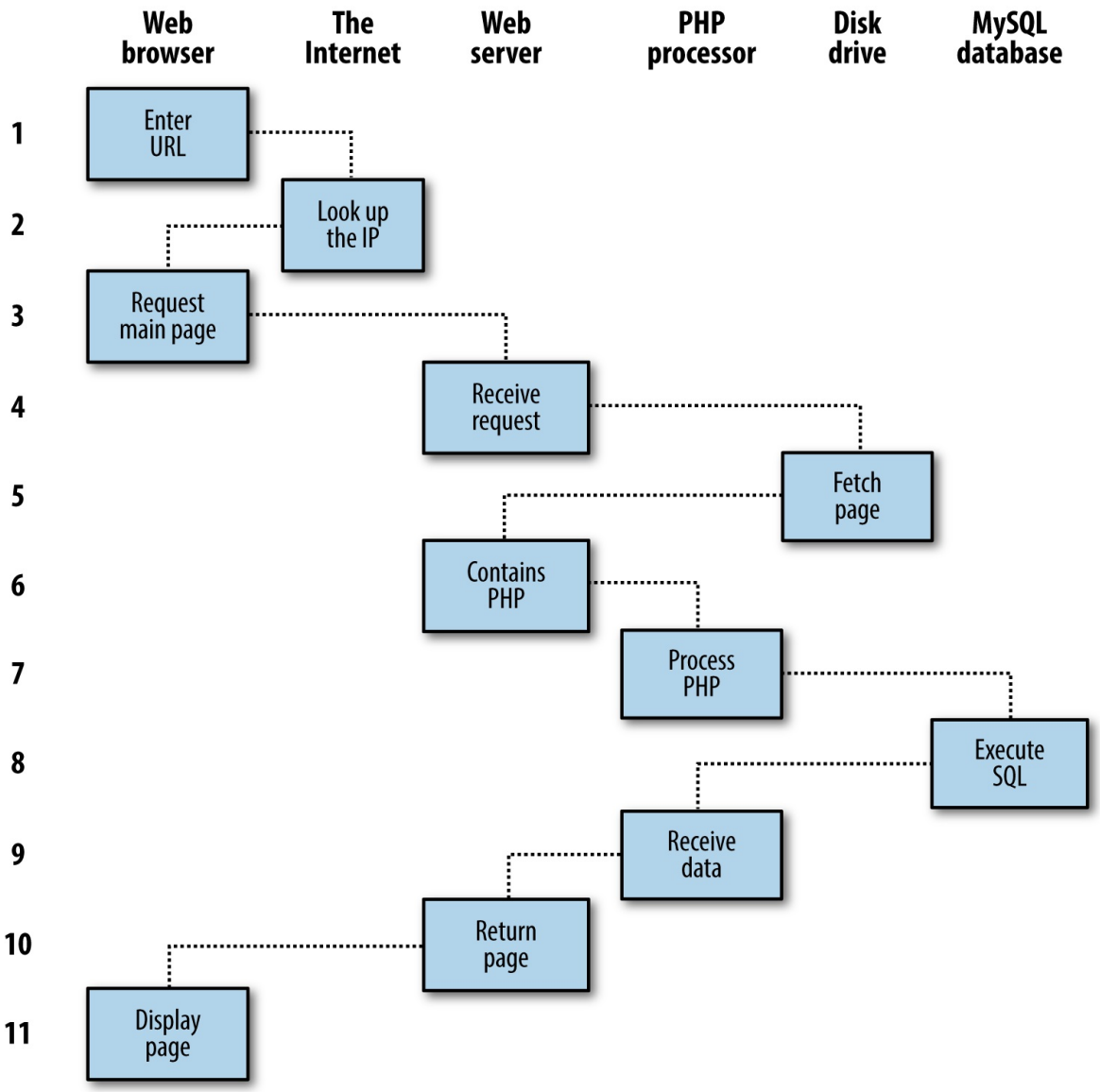

Figure 1-2. A dynamic client/server request/response sequence

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.

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 when there is now a wide range of sophisticated frameworks of JavaScript functions you can now call on to really speed up web development, such as the well-known jQuery, which is now probably the most common way programmers access asynchronous communication features.

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 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 `INSERT` command here shows how simple it can be to add new data to a database. `INSERT` is an example of Structured Query Language (SQL), a language designed in the early 1970s and 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 create them out of multiple PHP calls to MySQL.

Using JavaScript

The oldest of the three core technologies 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 optimise your code for different browsers. However there remain millions of legacy browsers that will be in use 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 then does by writing the text `Today is` to the current document, along with the date, by using the JavaScript function `Date`. The result will look something like this:

```
Today is Sun Jan 01 2017 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 the process in which web pages begin to resemble standalone programs, because they don’t have to be reloaded in their entirety. 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 [Link to Come].

Then, in [Link to Come], 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, but jQuery is by far the most popular. Due to continuous maintenance, it is extremely reliable, and a major tool in the utility kit of many seasoned web developers.

Using CSS

CSS is the crucial companion to HTML, ensuring that the HTML text and embeddd 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 [Link to Come], 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. It was called *HTML5* and it began development 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). But it’s a never-ending cycle of development, and more functionality is sure to be built into it over time. 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 [Link to Come].

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 element. 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 varying 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 is now 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 optimised for a range of different computing devices—one less thing for you to worry about.

To learn how to make full use of its power, the final chapter of this book creates a simple social networking example website, using jQuery Mobile to make it fully responsive and display well on anything from a small mobile phone screen to a tablet, or a desktop computer.

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 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 jQuery 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).

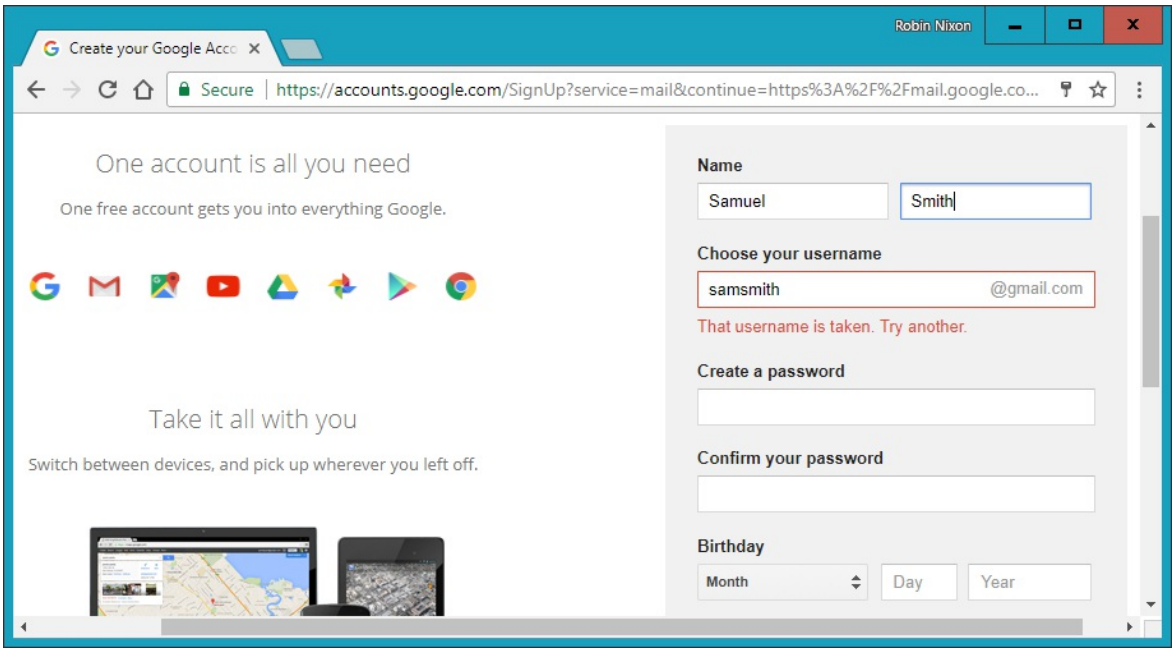

Figure 1-3. Gmail uses asynchronous communication to check the availability of usernames

The steps involved in this asynchronous process would 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: (a) whether some text has been typed into it, and (b) 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 one 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 Mobile so important for developing modern websites and web apps?

Chapter 2. Setting Up a Development Server

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 Internet Explorer, 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 Apple iOS and Google Android phones and tablets.

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 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 in the quickest time with a minimum 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.

But 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, but out 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 at *ampps.com*, as shown in Figure 2-1.

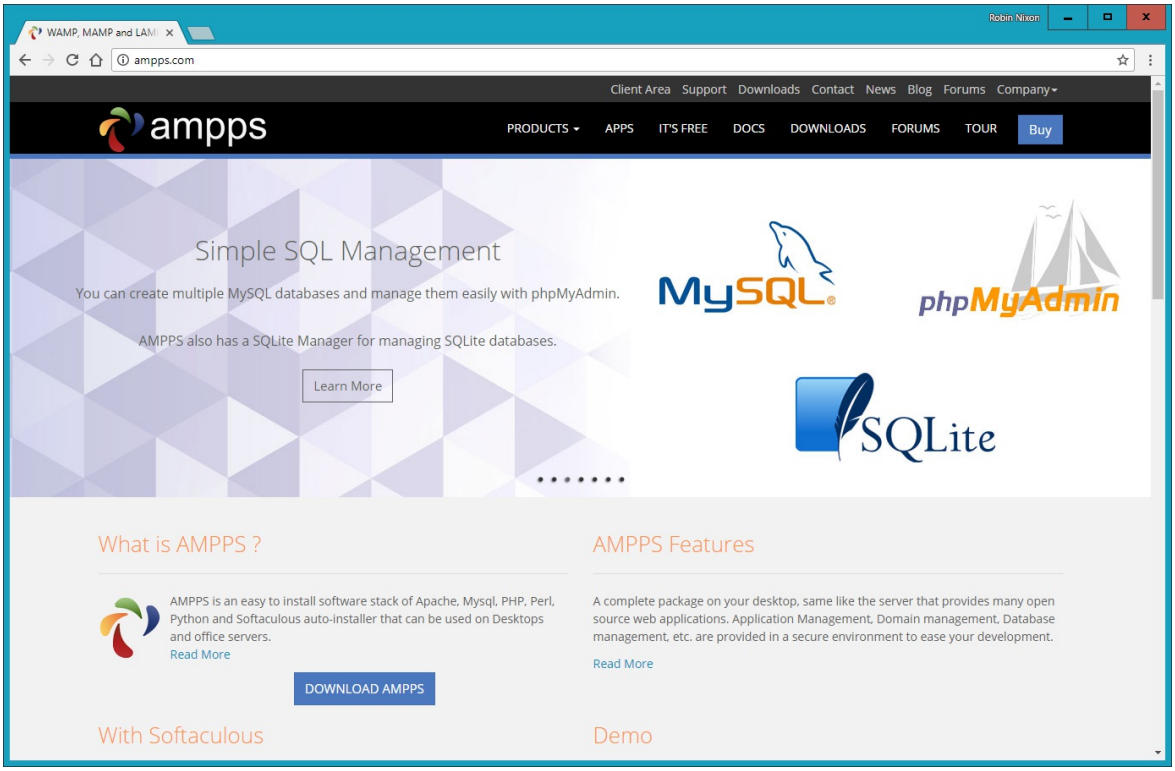

Figure 2-1. The Ampps website

I recommend that you always download the latest stable release (as I write this, it's 3.7, which is a little over 200MB in size). The various Windows, OS X, and Linux installers are listed on the download page.

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 Figure 2-2. Before arriving at that window, though, if you use an anti-virus 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 installation.

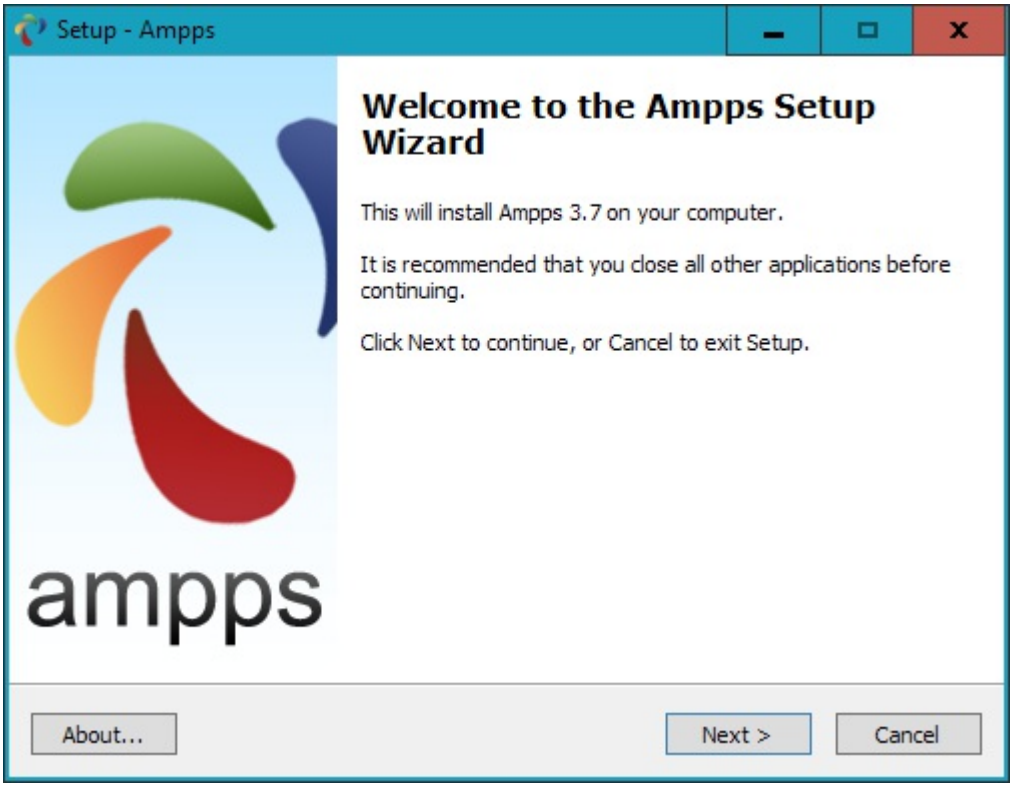

Figure 2-2. The opening window of the installer

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, which 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
```

Once you have decided where to install Ampps, click Next, choose a Start menu folder name, and click Next. You can 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 off.

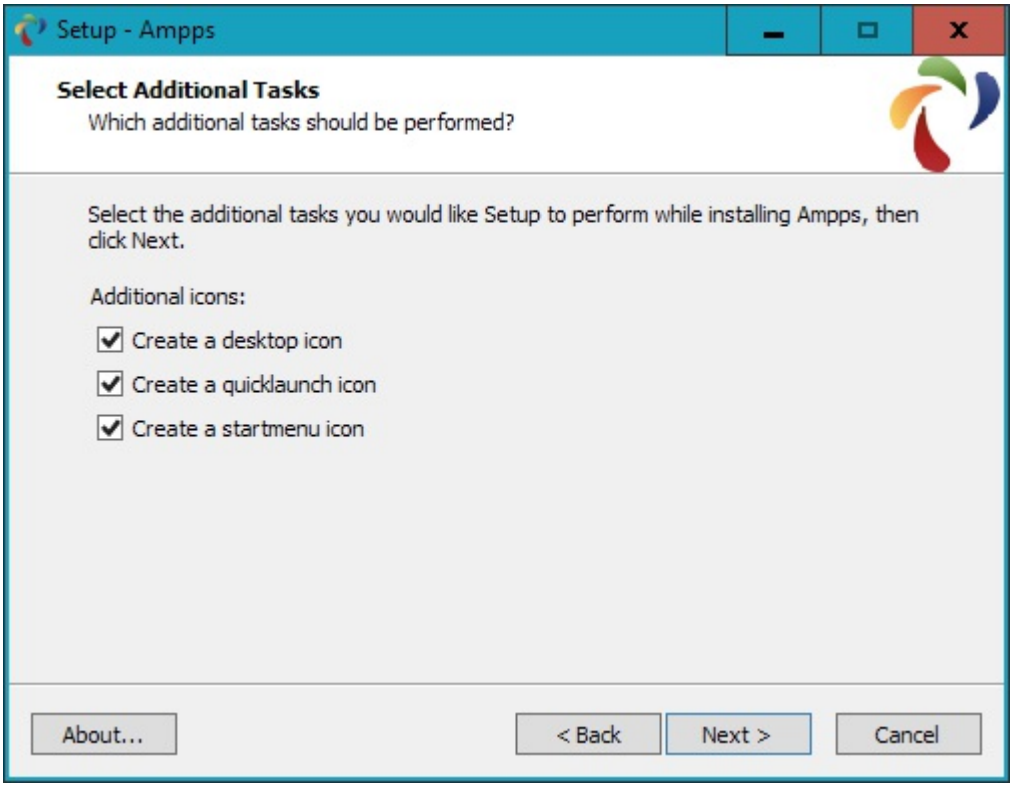

Figure 2-3. Choose which icons to install

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

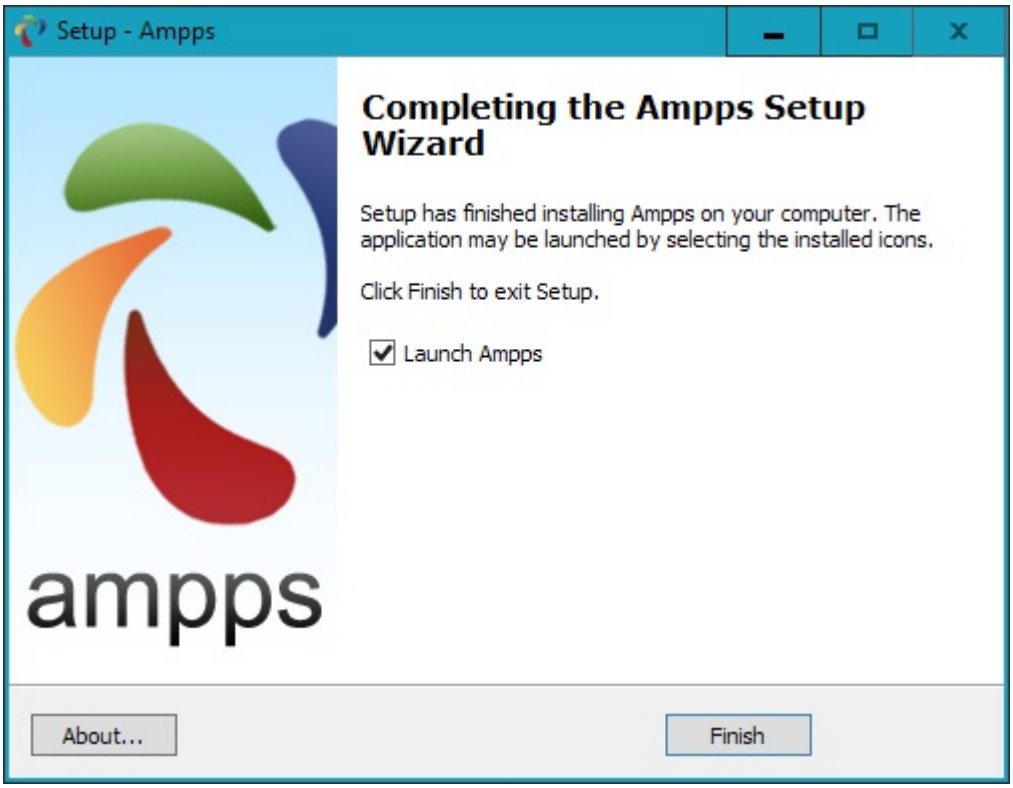

Figure 2-4. Ampps is now installed

The final thing you must do is install C++ Redistributable Visual Studio, 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 in Figure 2-5. Click Yes to start the installation or No if you are certain you already have it.

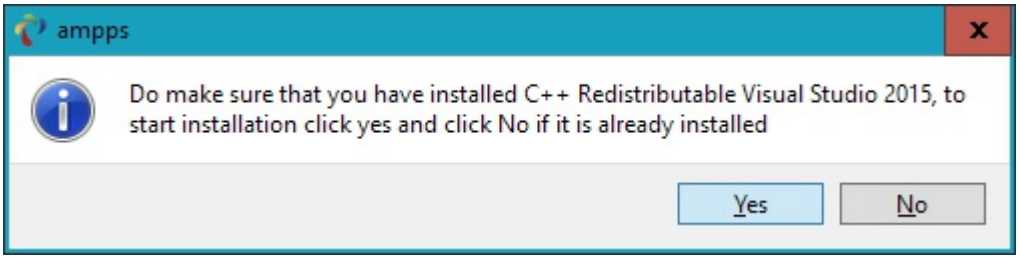

Figure 2-5. Install C++ Redistributable Visual Studio if you don't already have it

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 Figure 2-6 should appear at the bottom-right of your desktop. This window can also be called up using the Ampps application in the Start menu, or on the desktop if you allowed these icons to be created.

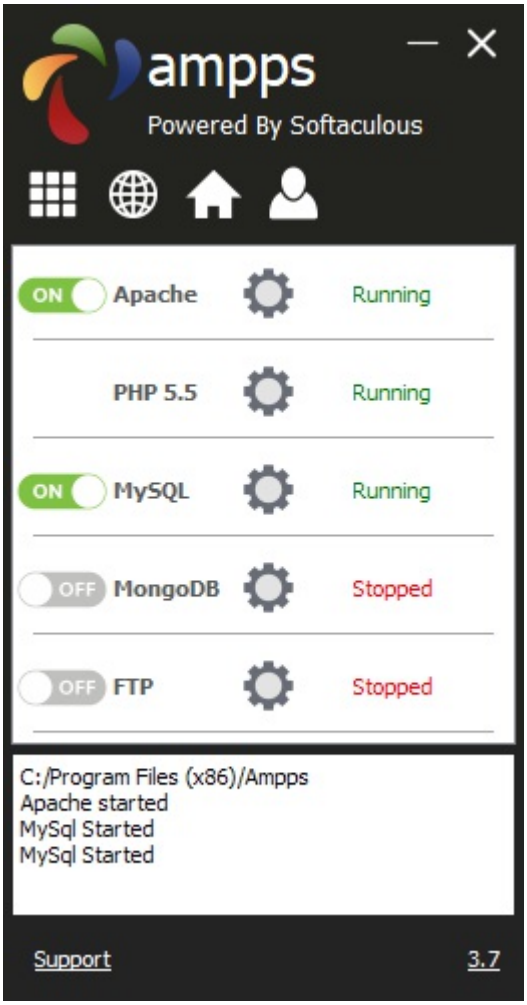

Figure 2-6. The Ampps control window

Before proceeding, I recommend you acquaint yourself with the Ampps documentation available at *ampps.com/wiki*. Once you have digested this, and should you still have an issue, there's a useful 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.

Other versions of PHP

You may notice that the default version of PHP in Ampps is 5.5. In other sections of this book I detail some of the more important changes in PHP 7. So if you wish to try them out for yourself, 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 between a number of versions from 5.3 up to 7.1.

Testing the Installation

The first thing to do at this point is verify that everything is working correctly. To do this, you are going to try to display the default web page, which will have been saved in the server's document root folder (see Figure 2-7). Enter either of the following two URLs into the address bar of your browser:

```
localhost
127.0.0.1
```

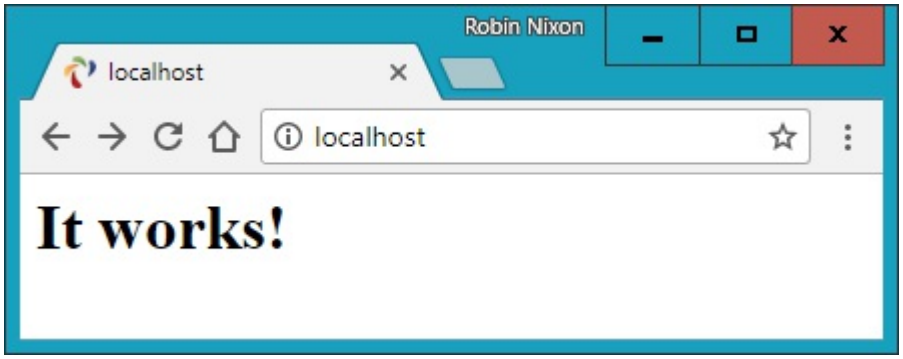

Figure 2-7. The default home page

The word *localhost* is used in URLs to specify the local computer, which will also respond to the IP address of 127.0.0.1, so you can use either method of calling up the document root of your web server.

Accessing the document root

The *document root* is the directory that contains the main web documents for a domain. This directory 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 document root:

```
C:\Program Files (x86)\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 previously discussed,

using the filename *test.html*. If you are using Notepad, make sure that 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 one of the following URLs in its address bar (see Figure 2-8):

```
http://localhost/test.html
http://127.0.0.1/test.html
```

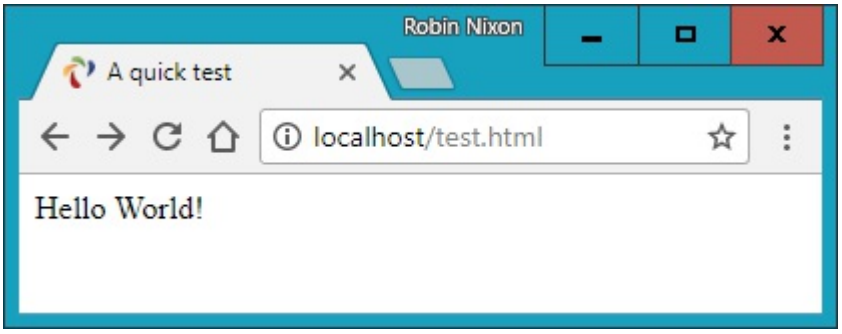

Figure 2-8. Your first web page

Correctly loading web pages into a browser

Remember that serving a web page from the document root (or a sub-folder) is different from loading one into a web browser from your computer's file system. The former will ensure access to PHP, MySQL and all the features of a web server, while the latter will simply load a 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 a file doesn't rely on web server functionality.

Alternative WAMPs

When software is updated, it sometimes works differently from what you'd expected, 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: *easyphp.org*
- XAMPP: *apachefriends.org*
- WAMPServer: *wampserver.com/en*
- Glossword WAMP: *glossword.biz/glosswordwamp*

Installing Ampps on Mac OS X

Ampps is also available on OS X, and you can download it from *http://ampps.com*, as shown previously in Figure 2-1 (it's length will likely be around 512Mb).

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 2-9)

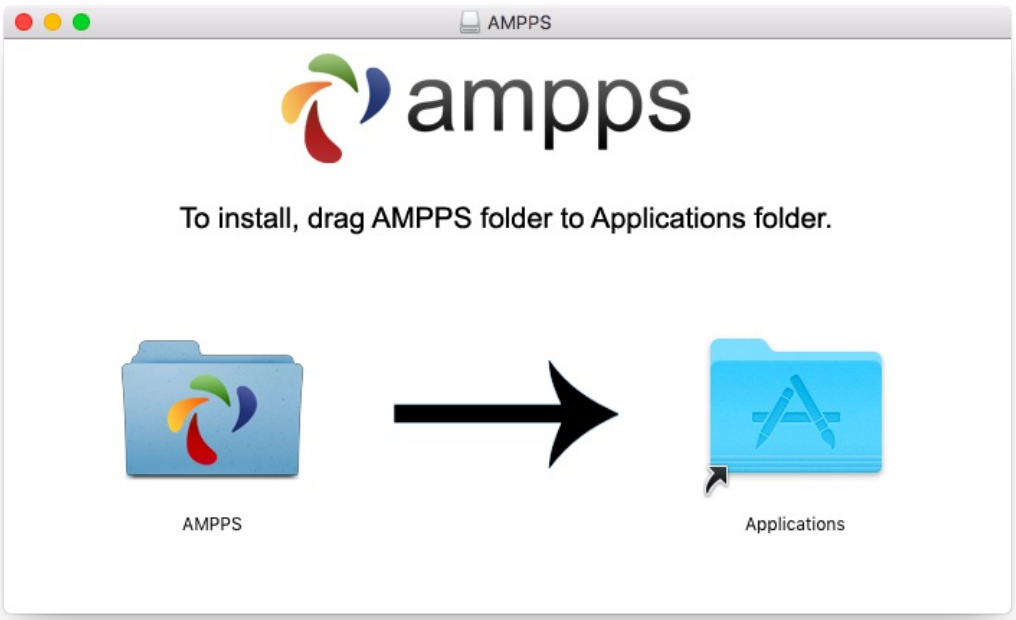

Figure 2-9. Drag the AMPPS folder to Applications

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 OS X password to proceed.

Once Ampps is up and running, a control window similar to the one shown in Figure 2-6 will appear at the bottom left of your desktop.

Other versions of PHP

You may notice that the default version of PHP in Ampps is 5.5. In other sections of this book I detail some of the more important changes in PHP 7. So if you wish to try them out for yourself, 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 between a number of versions from 5.3 up to 7.1.

Accessing the document root

By default Ampps will use the following location as 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 the *TextEdit* program (from your *Applications* folder):

```
<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 previously discussed, using the filename *test.html*.

You can now call this page up in your browser by entering one of the following URLs in its address bar (to see a similar result to [Link to Come]):

```
http://localhost/test.html
http://127.0.0.1/test.html
```

Correctly loading web pages into a browser

Remember that serving a web page from the document root (or a sub-folder) is different from loading one into a web browser from your computer's file system. The former will ensure access to PHP, MySQL and all the features of a web server, while the latter will simply load a 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 a 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.

However, 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:

```
http://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 at *ampps.com*.

Installation is similar to the sequence shown in the Mac OS X section, and if you need further assistance on using the software, please refer to the documentation at *ampps.com/wiki*.

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 may 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, available at *http://putty.org*, 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, available from *filezilla-project.org*, for Windows, Linux, and Mac OS X 10.5 or newer (see [Link to Come]). Full instructions on how to use FileZilla are at *wiki.filezilla-project.org*.

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

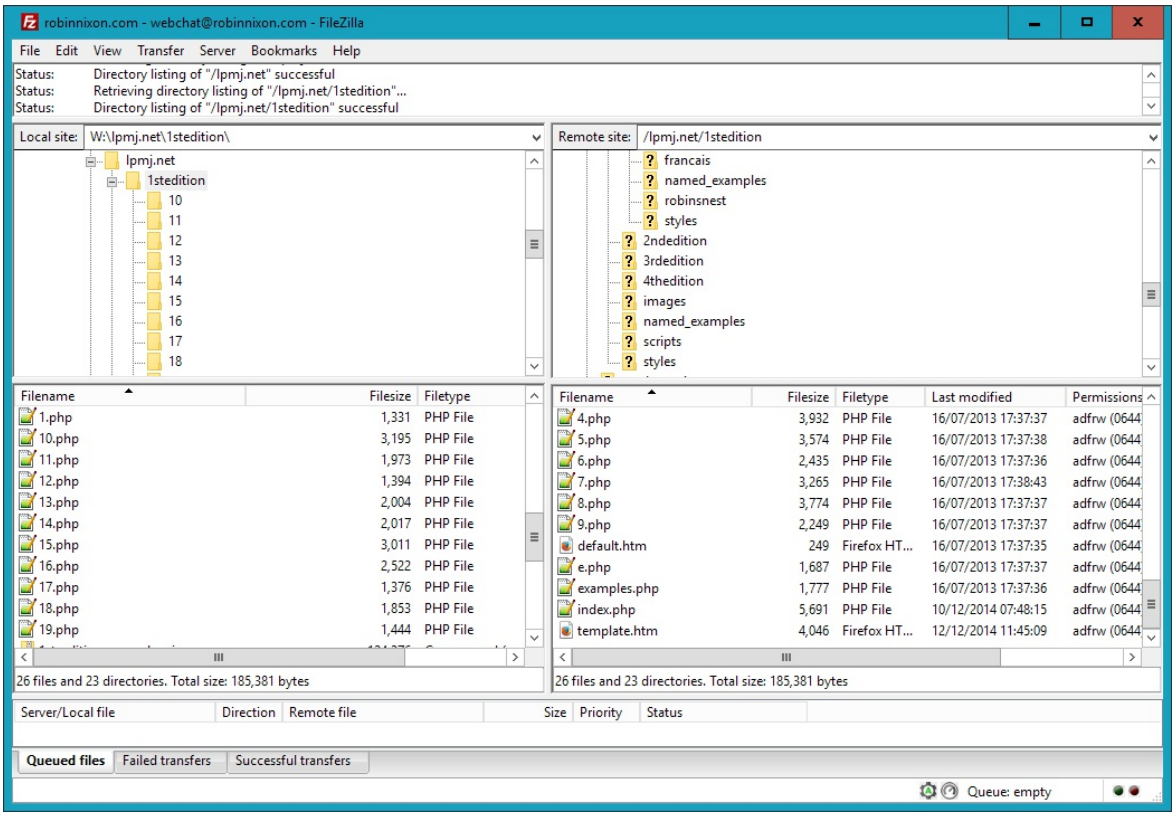

Figure 2-10. FileZilla is a fully featured FTP program

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 2-11), because it's free and available on Mac, Windows, and Linux/Unix, and it suits the way I program. You can download a copy by visiting *http://editra.org* and selecting the Download link toward the top left of the page, where you can also find the documentation for it. 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 Debugging environment, see "Using an IDE".

Anyway, as you can see from Figure 2-11, 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 pair 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.

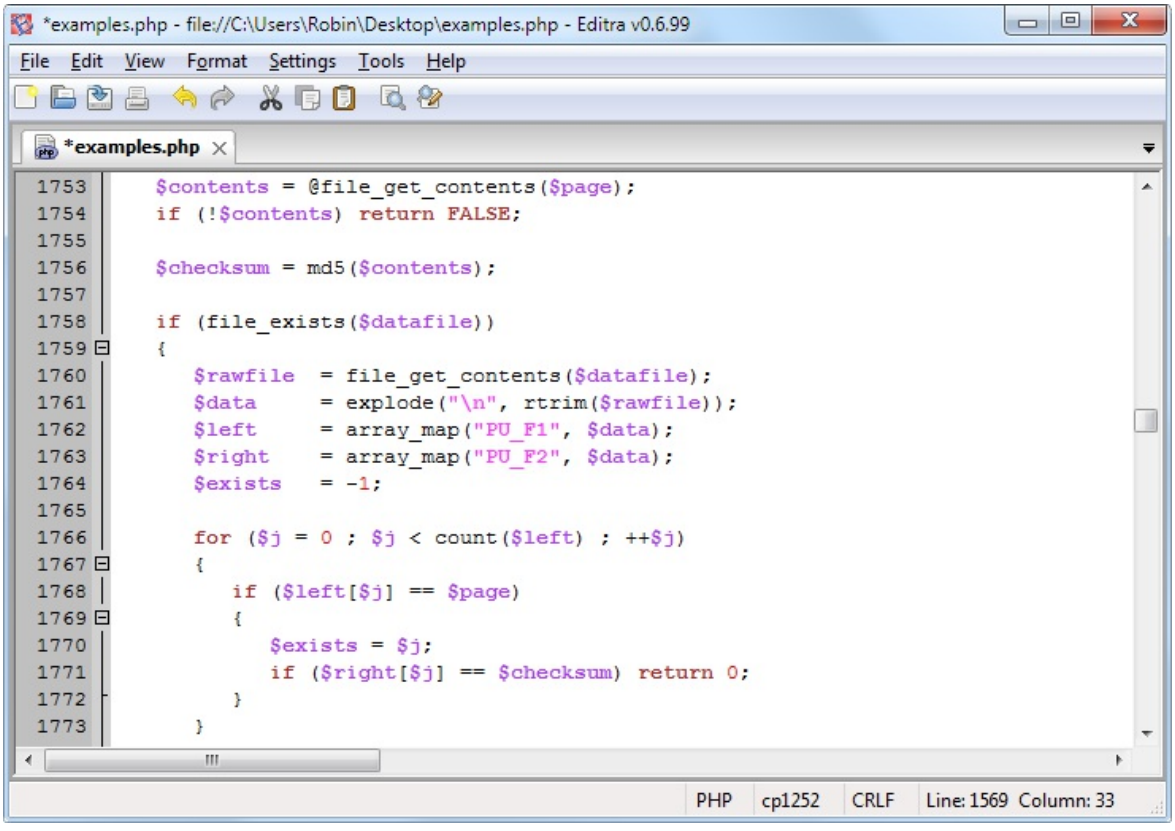

Figure 2-11. Program editors are superior to plain-text editors

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* (IDEs), which offer many additional features such as in-editor debugging and program testing, as well as function descriptions and much more.

Figure 2-12 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.

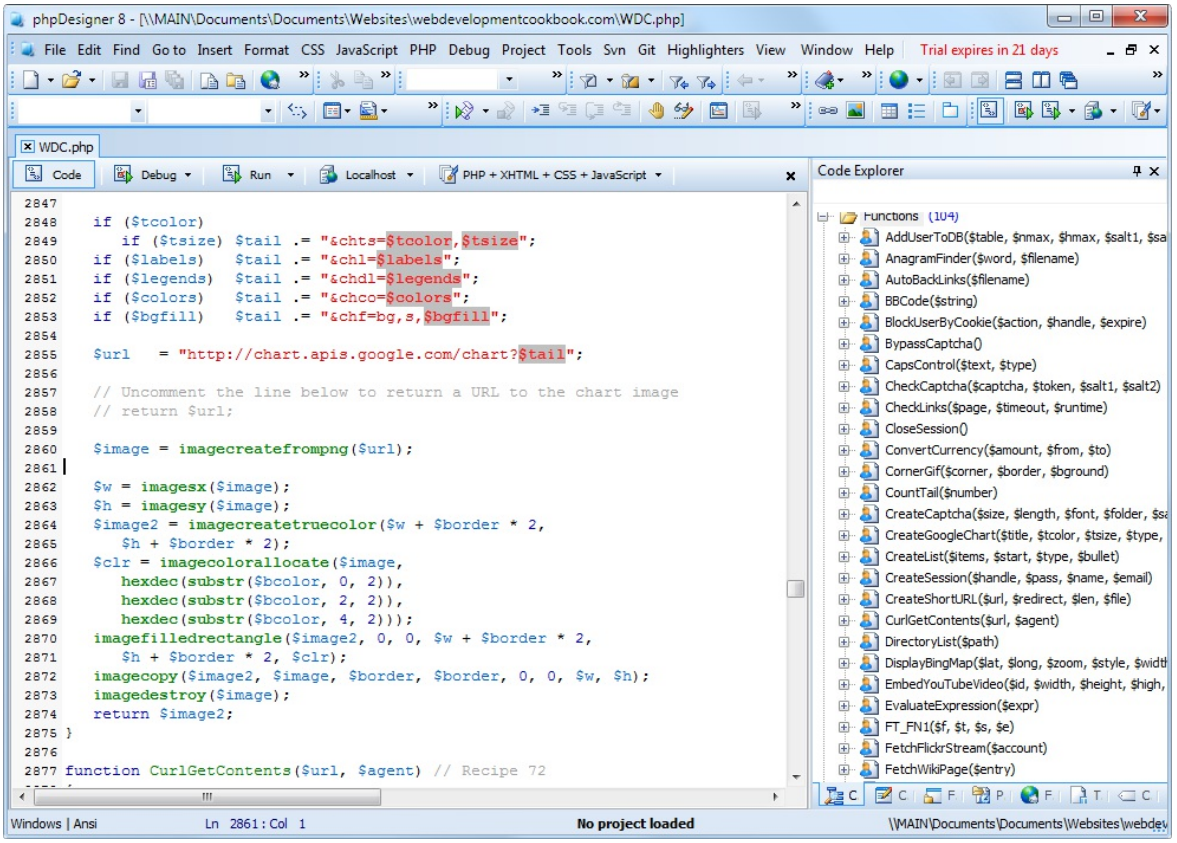

Figure 2-12. When you're using an IDE such as phpDesigner, PHP development becomes much quicker and easier

When developing with an IDE, you can set breakpoints and then run all (or portions) of your code, which will then 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, most of which are commercial, but there are some free ones too. Table 2-1 lists some of the most popular PHP IDEs, along with their download URLs.

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.

Table 2-1. A selection of PHP IDEs

IDE	Download URL	Cost	Win	Mac	Lin
Eclipse PDT	*http://eclipse.org/pdt/downloads/*	Free	✓	✓	✓
Komodo IDE	*http://activestate.com/Products/komodo_ide*	$295	✓	✓	✓
NetBeans	*http://www.netbeans.org*	Free	✓	✓	✓
phpDesigner	*http://mpsoftware.dk*	$39	✓		
PHPEclipse	*http://phpeclipse.de*	Free	✓	✓	✓
PhpED	*http://nusphere.com*	$99	✓		✓
PHPEdit	*http://www.phpedit.com*	$117	✓		

You should take the time to install a program editor or IDE you are comfortable with, and 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?

Chapter 3. Introduction to PHP

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 [Link to Come].

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 less feature-rich editors.

Many of these development environments 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.

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 such as an *index.html* file and save it 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 say 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 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 onto the website at *http://lpmj.net*, which you can download to your computer by clicking the 5th Edition Examples link in the heading section (see Figure 3-1).

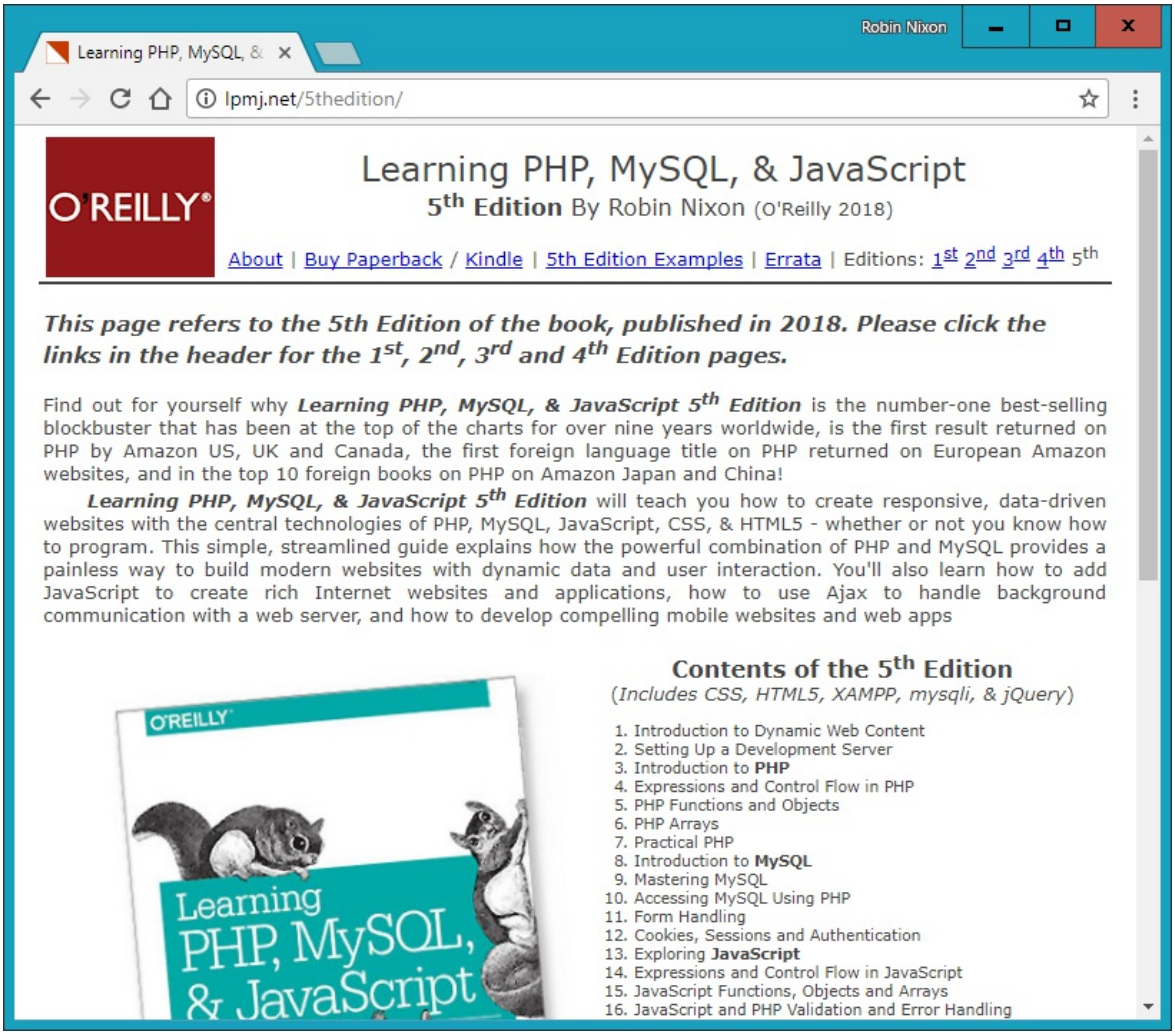

Figure 3-1. Viewing examples from this book at *http://lpmj.net*

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 our 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 3-2). Well, that's the same process as assigning a string value to a variable, like this:

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

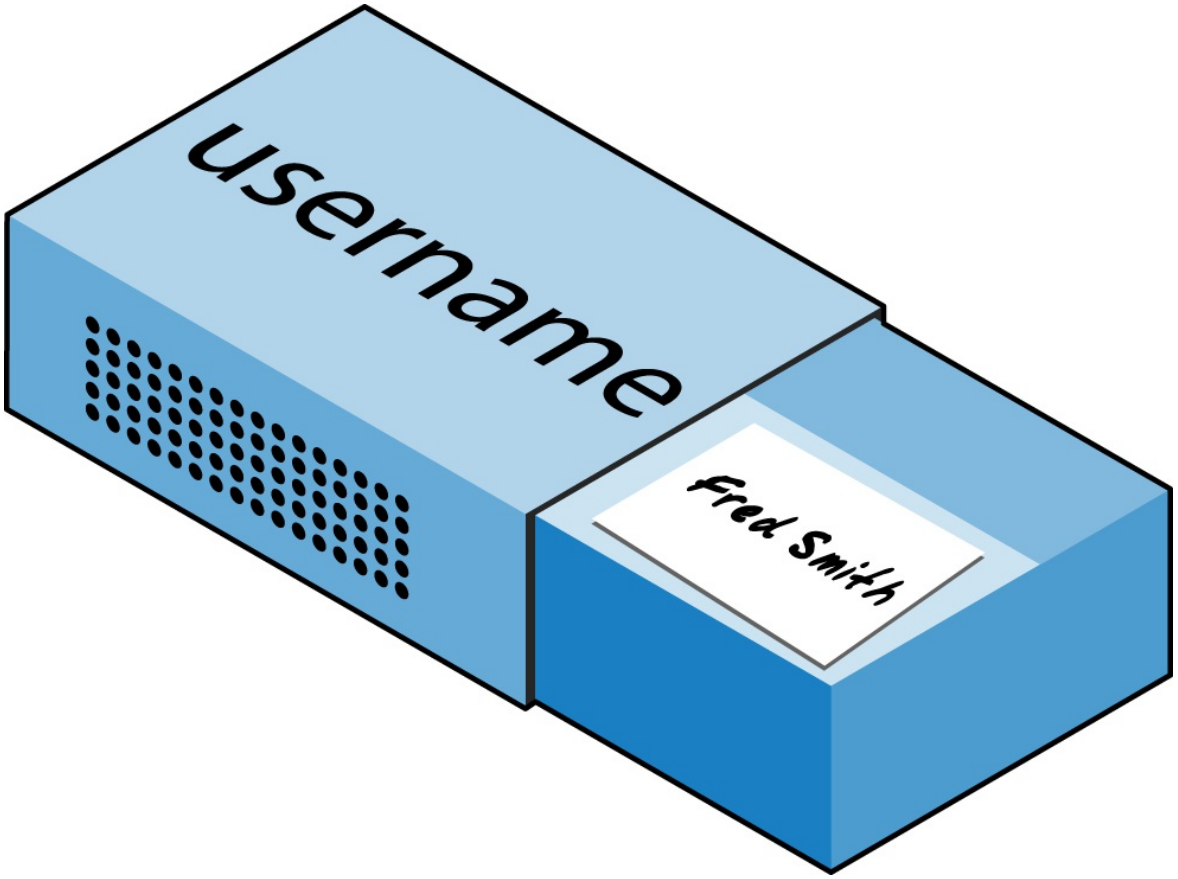

Figure 3-2. You can think of variables as matchboxes containing items

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 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 becomes 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 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 read 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 whole top of the matchbox assembly we would write the word *team* (see Figure 3-3). The equivalent of this in PHP would be the following:

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

This syntax is more complicated than the ones I've explained 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
```

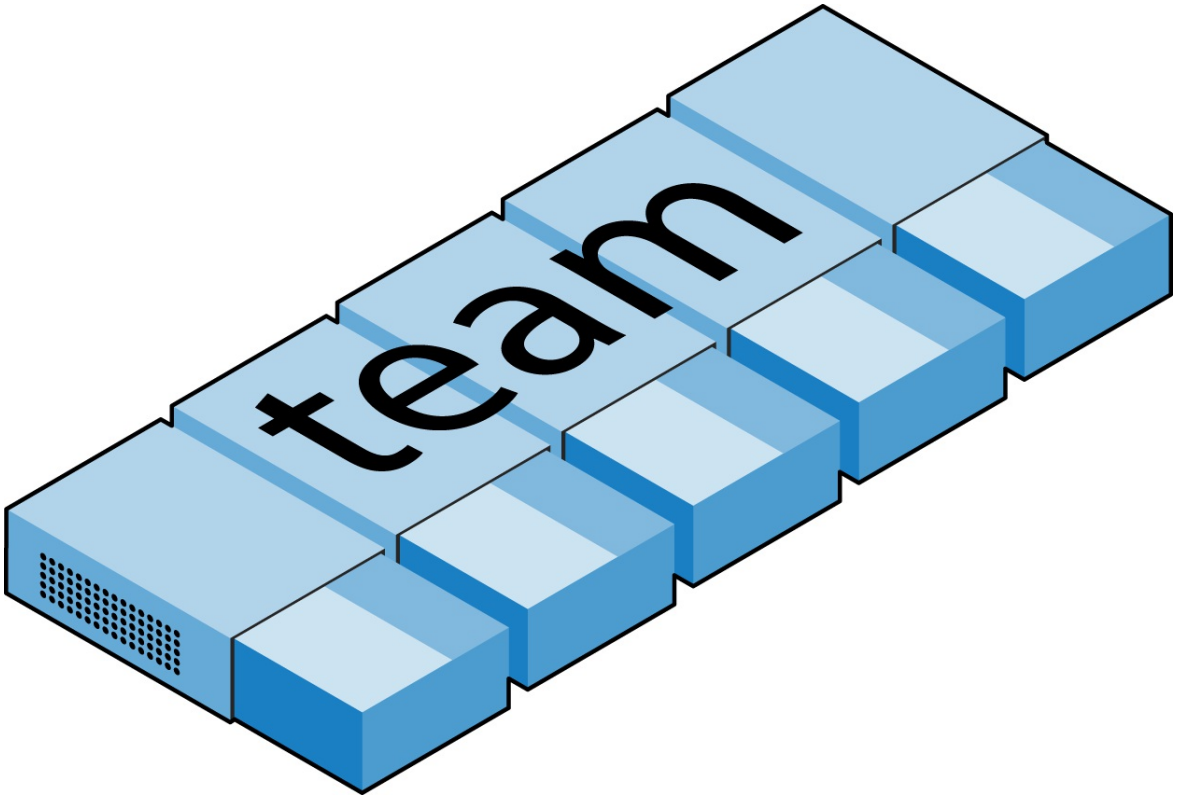

Figure 3-3. An array is like several matchboxes glued together

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 3×3 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 3-4).



Figure 3-4. A multidimensional array simulated with matchboxes

You can now place a piece of paper with either an *x* or an *o* 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 column or 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. It 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 [Link to Come].

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 must comprise more than one word, a good idea is to separate 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 mathematics, such as plus, minus, multiply, and divide. But several other types of operators exist: the string, comparison, and logical. 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 (plus, minus, times, 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	`$j + 1`
`-`	Subtraction	`$j - 6`
`*`	Multiplication	`$j * 11`
`/`	Division	`$j / 4`
`%`	Modulus (the remainder after a division is performed)	`$j % 9`
`++`	Increment	`++$j`
`--`	Decrement	`--$j`
`!`	Not	`!$j`

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).

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.

Table 3-3. Comparison operators

Operator	Description	Example
`==`	Is *equal* to	`$j == 4`
`!=`	Is *not equal* to	`$j != 21`
`>`	Is *greater than*	`$j > 3`
`<`	Is *less than*	`$j < 100`
`>=`	Is *greater than or equal* to	`$j >= 15`
`<=`	Is *less than or equal* to	`$j <= 8`

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 timing):

```
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
`&&`	*And*	`$j == 3 && $k == 2`
`and`		`$j == 3 and $k == 2`

	Low-precedence *and*	
`||`	*Or*	`$j < 5 || $j > 10`
`or`	Low-precedence *or*	`$j < 5 or $j > 10`
`!`	*Not*	`! ($j == $k)`
`xor`	*Exclusive or*	`$j xor $k`

Note that `&&` is usually interchangeable with `and`; the same is true for `||` and `or`. However, because `and` and `or` have a lower precedence you really should avoid ever using them, except when they are the only option, as in the following statement, which *must* use the `or` operator:

```
$html = file_get_contents($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 the 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

```
$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 test whether it has the value `10`; if it does, output its value. But you can also require PHP to increment (or, 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 word 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 the single quotation mark (apostrophe), 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 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;` tag *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 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 slash (`\`).

Example 3-9 shows how to use the same syntax to assign multiples 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. But 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` is still 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 the following snippet of code, the numbers 12345 and 67890 are multiplied together, returning a result of 838102050, which is then placed in the variable `$number`, as shown in Example 3-10.

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 (remembering 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 `$` (as with regular variables), and that you can define them only using the `define` function.

It is generally considered a good practice to use only uppercase 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

Magic constant	Description
`__LINE__`	The current line number of the file.
`__FILE__`	The full path and filename of the file. If used inside an `include`, the name of the included file is returned. Some operating systems allow aliases for directories, called *symbolic links*; in `__FILE__` these are always changed to the actual directories.
`__DIR__`	The directory of the file. If used inside an `include`, the directory of the included file is returned. This is equivalent to *`dirname`*`(__FILE__)`. This directory name does not have a trailing slash unless it is the root directory. (Added in PHP 5.3.0.)
`__FUNCTION__`	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.
`__CLASS__`	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.
`__METHOD__`	The class method name. (Added in PHP 5.0.0.) The method name is returned as it was declared (case-sensitive).
`__NAMESPACE__`	The name of the current namespace (case-sensitive). This constant is defined at compile time. (Added in PHP 5.3.0.)

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 also 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, unless you use a function. And if you decide to change the data 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 the previous example is a lower case 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 days × 24 hours × 60 minutes × 60 seconds).

Functions can also accept multiple parameters and return multiple results, using techniques that I'll develop 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, it will first display the error message `Notice: Undefined variable: temp`.

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 Example 3-15 and Example 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 its 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 the 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 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;        // Disallowed (will produce a Parse error)
  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 [Link to Come].

Table 3-6. PHP's superglobal variables

Superglobal name	Contents
`$GLOBALS`	All variables that are currently defined in the global scope of the script. The variable names are the keys of the array.
`$_SERVER`	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.
`$_GET`	Variables passed to the current script via the HTTP Get method.
`$_POST`	Variables passed to the current script via the HTTP Post method.
`$_FILES`	Items uploaded to the current script via the HTTP Post method.
`$_COOKIE`	Variables passed to the current script via HTTP cookies.
`$_SESSION`	Session variables available to the current script.
`$_REQUEST`	Contents of information passed from the browser; by default, `$_GET`, `$_POST`, and `$_COOKIE`.
`$_ENV`	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 bit of information that many sites employ. 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 naïvely 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 `<` and `>` so that they are rendered harmless, as are all quotes and backslashes, and so on.

Therefore, here is 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 background in using PHP. In Chapter 4, we'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?

Chapter 4. Expressions and Control Flow in PHP

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:

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

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 also 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—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 in an example like 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: []
```

Note

Now that we are fully into the age of HTML5, and XHTML is no longer being planned to supersede HTML, you do not need to use the self-closing <br /> form of the <br> tag, or any void elements (ones without closing tags), because the / is now optional. Therefore, I have chosen to use the simpler style in this book. However, you must still use the <br /> form of HTML syntax when using XHTML.

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 or FALSE look like internally. In fact, even those names rarely appear in code.

Literals and Variables

These are 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 with an initial 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 a *statement*. 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 that range from arithmetic, string, and logical operators to assignment, comparison, and more (see Table 4-1).

Table 4-1. PHP operator types

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

Each operator takes a different number of operands:

- *Unary* operators, such as incrementing (`$a++`) or negation (`!$a`), which take a single operand.
- *Binary* operators, which represent the bulk of PHP operators, including addition, subtraction, multiplication, and division.
- One *ternary* operator, which takes the form `? x : y`. It's a terse, single-line `if` statement that returns `x` if `m` is true and `y` if `m` 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, so let's look at a few in 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, 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* precedencies 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.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
`++ --`	Increment/decrement
`!`	Logical
`* / %`	Arithmetic
`+ - .`	Arithmetic and string
`<< >>`	Bitwise
`< <= > >= <>`	Comparison
`== != === !==`	Comparison
`&`	Bitwise (and references)
`^`	Bitwise
`|`	Bitwise
`&&`	Logical
`||`	Logical
`? :`	Ternary
`= += -= *= /= .= %= &= != ^= <<= >>=`	Assignment
`and`	Logical
`xor`	Logical
`or`	Logical

The choices in this table are not just arbitrary., but are carefully chosen 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 becomes important in cases in which you do not explicitly force precedence, so you need to be aware of the default actions of operators, as detailed in Table 4-3, which lists operators and their associativity.

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 beginner 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 encountered a few times in this chapter, the equality operator is == (two equal signs). It is important not to confuse it with the = (single equal 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 dynamic web pages —the goal of using PHP in the first place—because they make it easy to create 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 where it started 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).

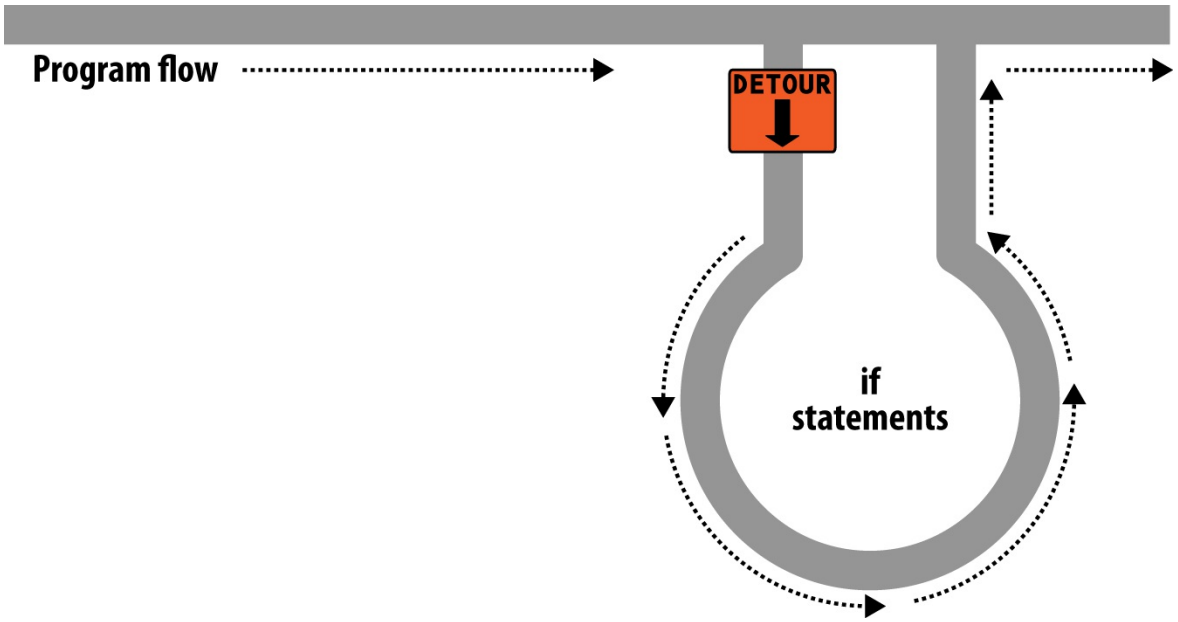

Figure 4-1. Program flow is like a single-lane highway

The contents of the `if` condition can be any valid PHP expression, including equality, comparison, tests for `0` and `NULL`, and even the values returned by 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, `{ }`. However, 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 lack of braces.

A Cautionary Tale

A notorious bug known as the “goto fail” bug haunted Apple’s SSL (Secure Socket Layer) code in its 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, allowing a malicious attacker to get a secure certificate to be accepted when it should be rejected. So if in doubt, place braces around your `if` statements.

(Note that for brevity and clarity, 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 dollars (or whatever your currency is). If so, you pay yourself 1,000 dollars and then add it to the balance. (If only making money were that simple!)

If the bank balance is 100 dollars 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.

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.

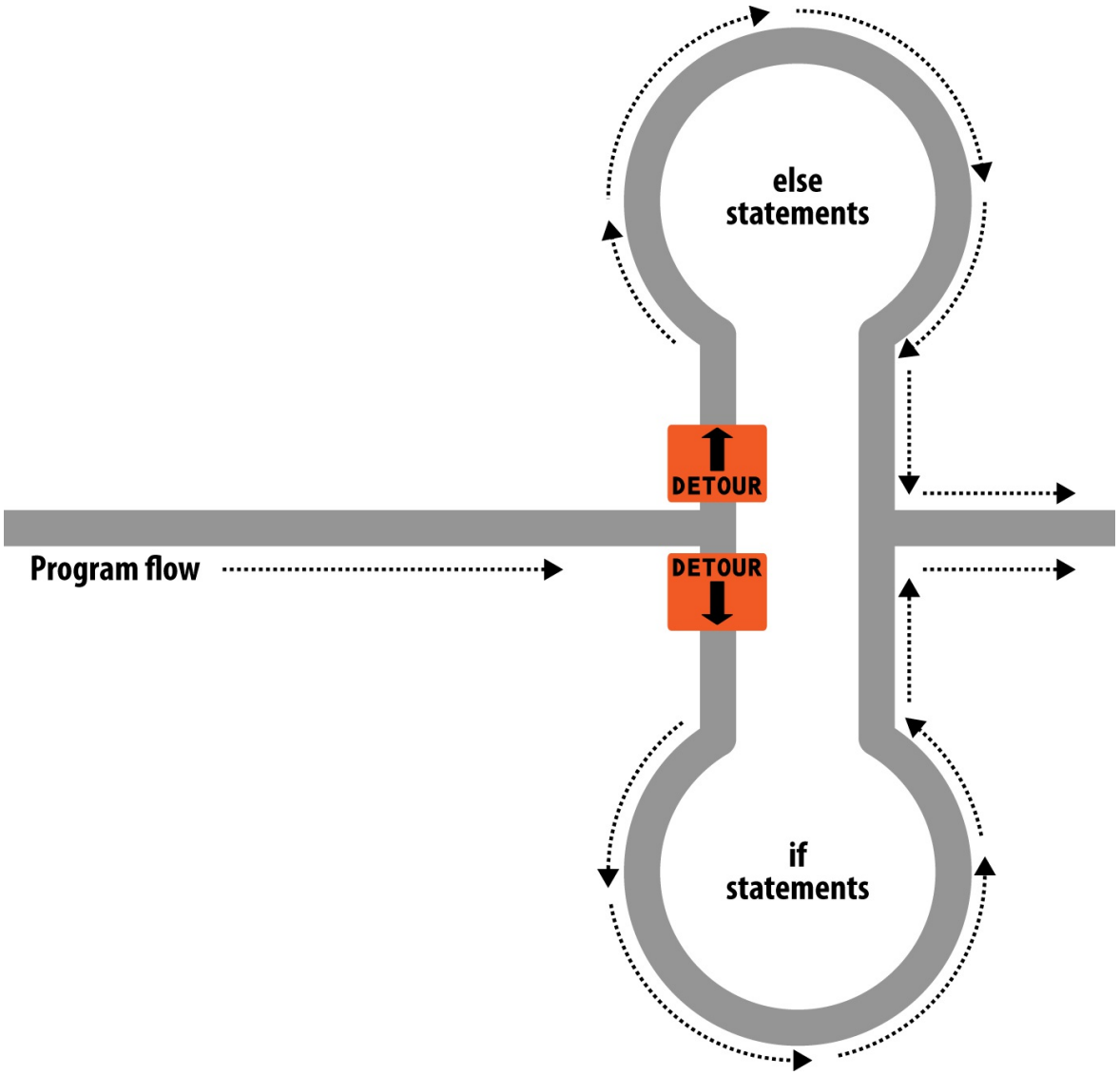

Figure 4-2. The highway now has an if detour and an else detour

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, now that you've ascertained that you have $100 or more in the bank, the `else` statement is executed, by which you place 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).

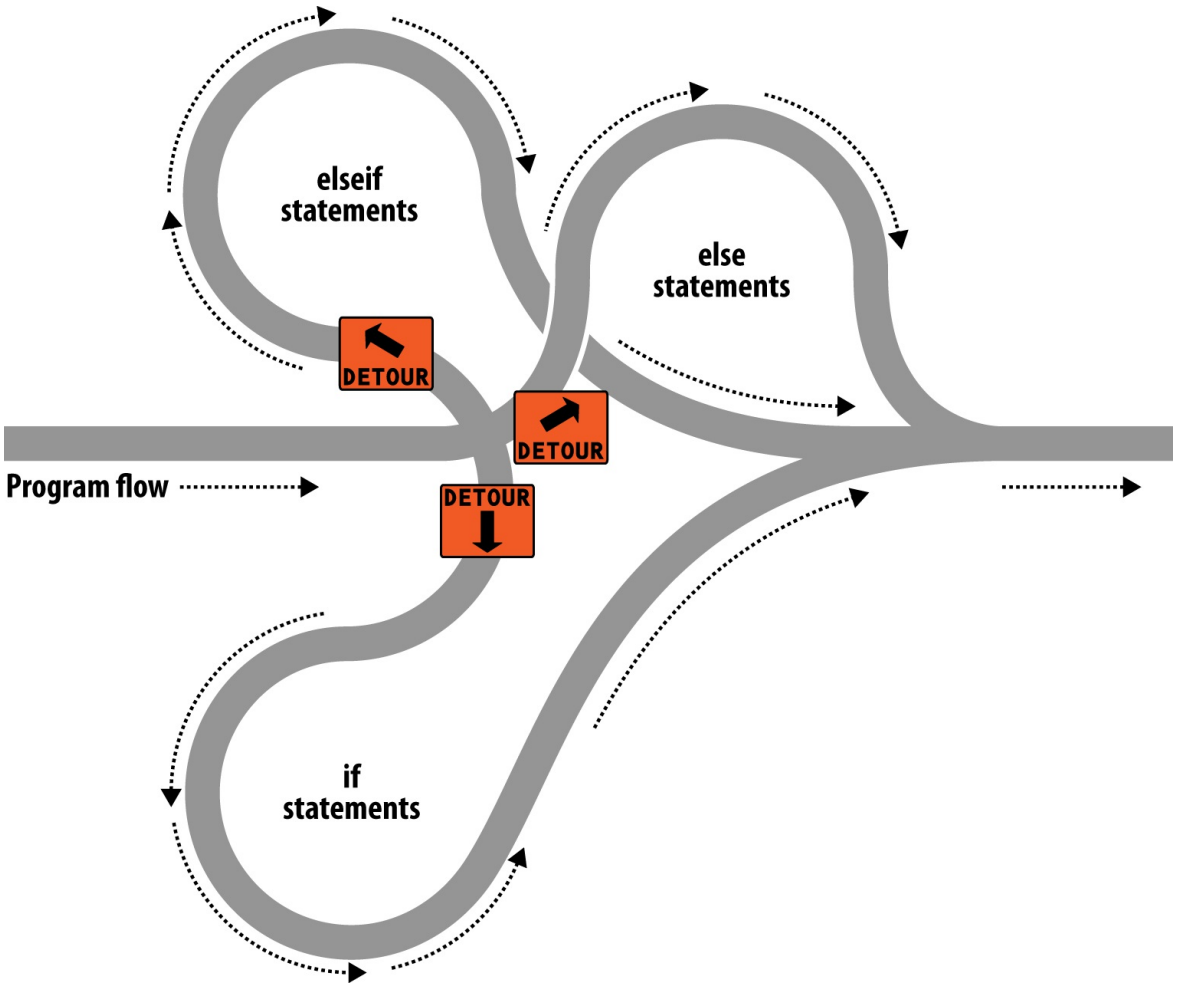

Figure 4-3. The highway with if, elseif, and else detours

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 each 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...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 break out of 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—can be exited only under the right program conditions.

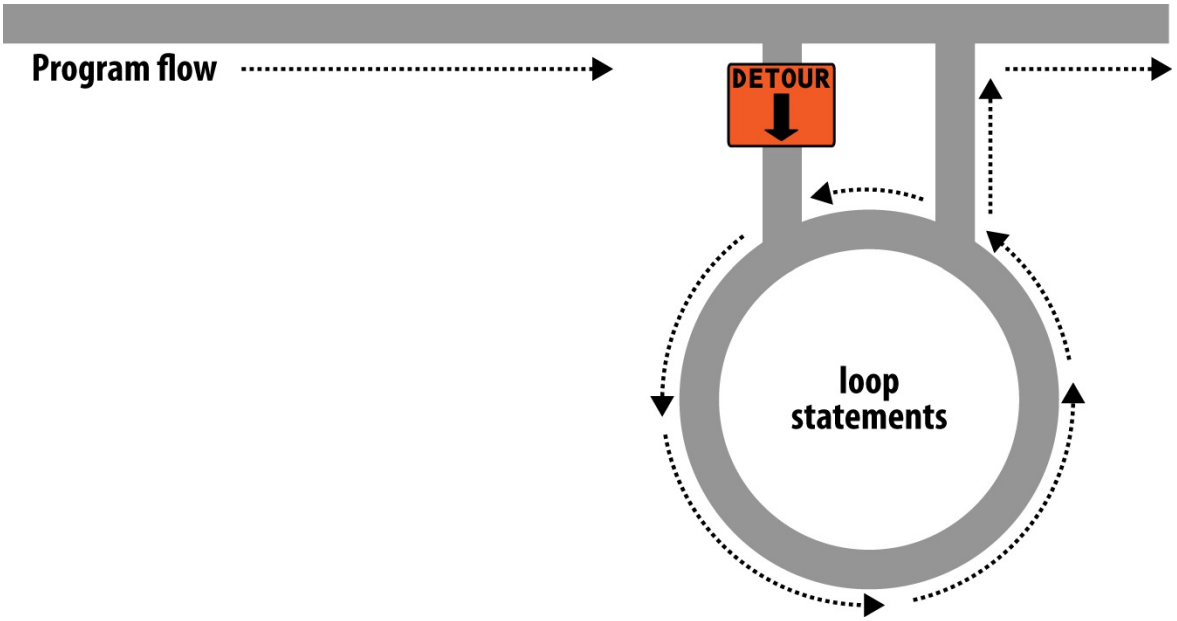

Figure 4-4. Imagining a loop as part of a program highway layout

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 `++$count` statement from the statements inside the `while` loop and into the conditional expression of the loop. What now happens is that PHP encounters the variable `$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 times table for 12

```
<?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 the loop's `echo` statement being executed before you 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 times table for 12 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. 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, the line

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

can be simplified 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 = 10;

  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 statement `continue` 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.

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.

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

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 well 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.

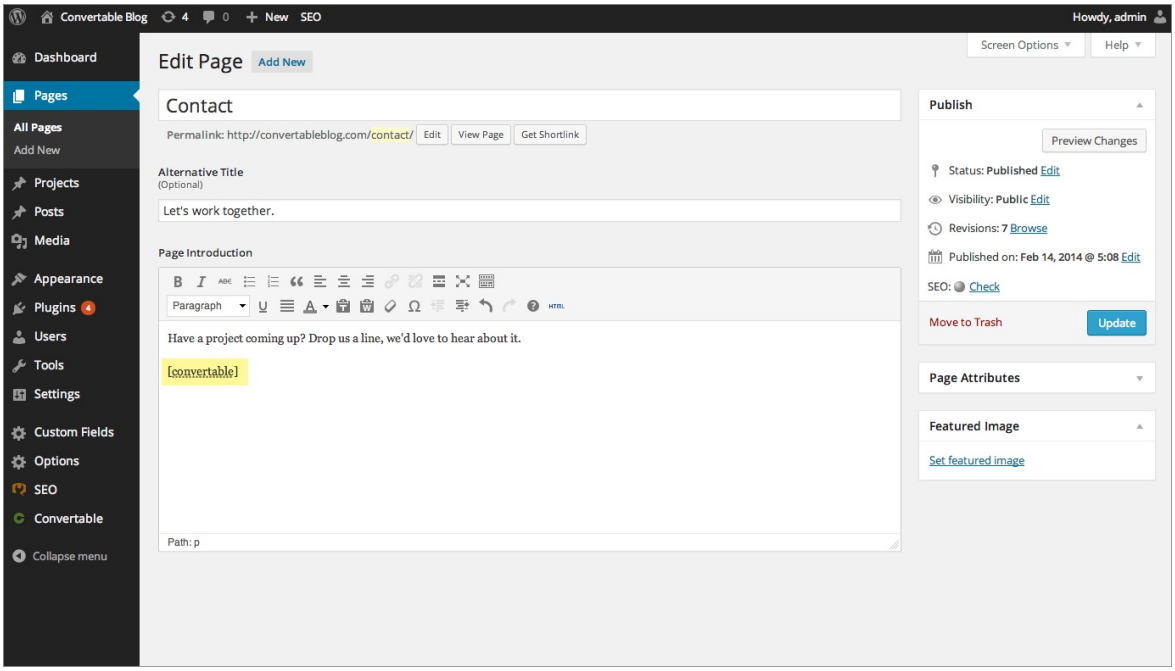

Figure 4-5. The WordPress blogging platform's dashboard

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 she needs, 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 on 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?

About the Author

Robin Nixon has over 30 years of experience with writing software and developing websites and apps. He also has an extensive history of writing about computers and technology, with a portfolio of over 500 published magazine articles and over 30 books, many of which have been translated into other languages. He is also a prolific Internet video course instructor

As well as IT, his interests include psychology and motivation (which he also writes about), artificial intelligence research, many types of music (both playing and listening to), playing and creating board games, studying philosophy and culture, and enjoying good food and drink.

Robin lives on the southeast coast of England (where he writes full time) with his five children and wife, Julie (a trained nurse and university lecturer). Between them they also foster three disabled children. You can keep up with Robin's (sporadic) posts via his website at *robinnixon.com*.