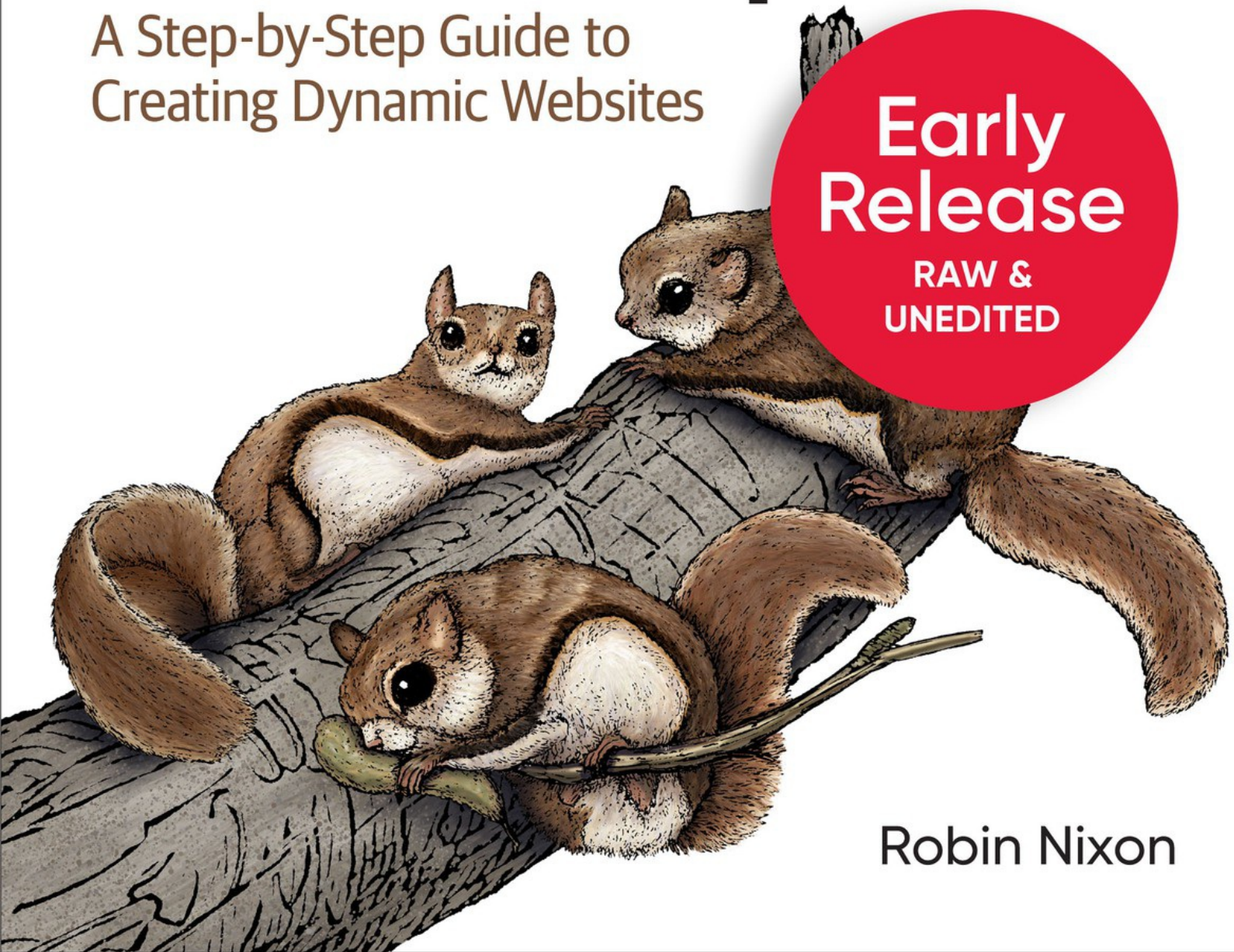
O'REILLY®

Seventh Edition

Learning PHP, MySQL & JavaScript

A Step-by-Step Guide to Creating Dynamic Websites

Early Release
RAW & UNEDITED



Robin Nixon

Learning PHP, MySQL & JavaScript

SEVENTH EDITION

A Step-by-Step Guide to Creating Dynamic Websites

Robin Nixon

Learning PHP, MySQL & JavaScript

by Robin Nixon

Printed in the United States of America.

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

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

Acquisitions Editor: Amanda Quinn

Development Editor: Rita Fernando

Production Editor: Elizabeth Faerm

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

August 2024: Seventh Edition

Revision History for the Early Release

- 2023-12-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098152352* for release details.

this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15229-1

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Introduction to Dynamic Web Content

A NOTE FOR EARLY RELEASE READERS

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

This will be the 1st chapter of the final book. Please note that the GitHub repo is available at *https://github.com/robinnixon/lpmj7*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

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.

THE ADVENT OF WEB 1.0

Web 1.0 was only given its name when the term Web 2.0 was coined. It consisted of static pages connected via hyperlinks and had a small selection of elements like frames and tables, with styling made directly in the HTML (as there was no CSS). Content came from a server's file system, and there was no use made of databases for storing and accessing data. Navigation and layout in Web 1.0 was managed with simple buttons and graphics, while interaction was very limited.

Today we take these simple tools for granted, but back then, the concept was revolutionary. The most connectivity so far experienced by at-home modem users was dialing up and connecting to a bulletin board 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 five million users. It soon became obvious, though, that something was missing. Yes, pages of text and graphics with hyperlinks to take you to other pages was a brilliant concept, but the results didn't reflect the instantaneous potential of computers and the internet to meet the particular needs of each user with dynamically changing content. Using the web was a very dry and plain experience, even if we did now have scrolling text and animated GIFs!

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

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

HTTP and HTML: Berners-Lee's Basics

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

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

A web server can usually handle multiple simultaneous connections, and when not communicating with a client, it spends its time listening for an incoming connection. When one arrives, the server sends back a response to confirm its receipt.

The Request/Response Procedure

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

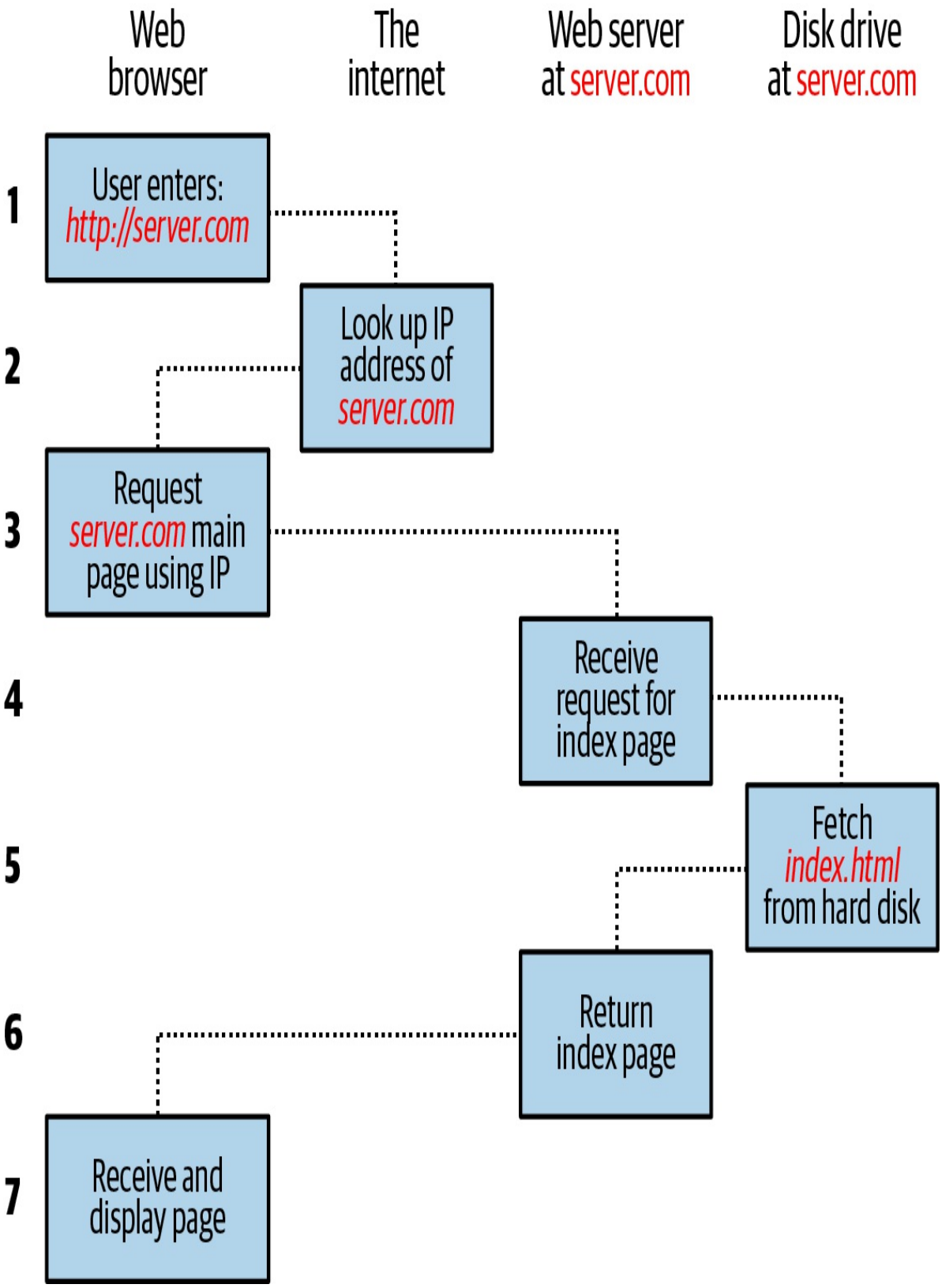


Figure 1-1. The basic client/server request/response sequence

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

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

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

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

For dynamic web pages, the procedure is a little more involved, because it may bring both PHP and MySQL into the mix. For instance, you may click a picture of a raincoat. Then PHP will put together a request using the standard database language, SQL—many of whose commands you will learn in this book—and send the request to the MySQL server. The MySQL server will return information about the raincoat you selected, and the PHP code will wrap it all up in some HTML, which the server will send to your browser (see Figure 1-2).

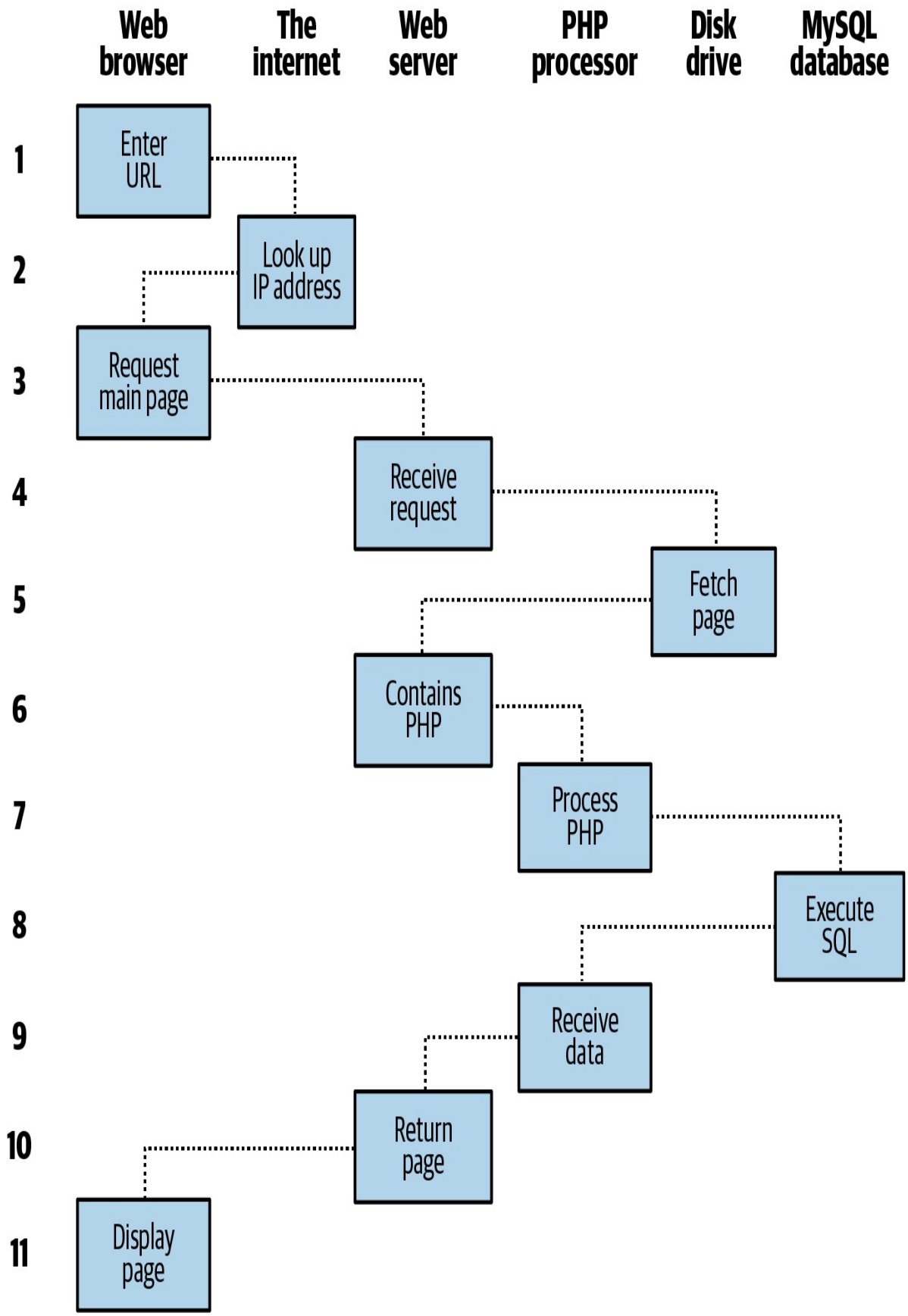

Web browser
The internet
Web server
PHP processor
Disk drive
MySQL database
1
Enter URL
2
Look up IP address
3
Request main page
4
Receive request
5
Fetch page
6
Contains PHP
7
Process PHP
8
Execute SQL
9
Receive data
10
Return page
11
Display page

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

The steps are as follows:

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

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

The HTML pages returned to the browser in each example may well contain JavaScript, which will be interpreted locally by the client, and which could initiate another request—the same way embedded objects such as images would.

The Benefits of PHP, MySQL, JavaScript, CSS, and HTML

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 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 beginnings of Web 2.0.

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

MariaDB: The MySQL Clone

After Oracle purchased Sun Microsystems (the owners of MySQL), the community became wary that MySQL might not remain fully open source, so MariaDB was forked from it to keep it free under the GNU GPL. Development of MariaDB is led by some of the original developers of MySQL, and it retains exceedingly close compatibility with MySQL. Therefore, you may well encounter MariaDB on some servers in place of MySQL—but not to worry, everything in this book works equally well on both MySQL and MariaDB. For all intents and purposes, you can swap one with the other and notice no difference.

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

Using PHP

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

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

Here's the latest news.
```

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

The final output of the two parts looks like this:

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

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

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

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

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

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

Using MySQL

Of course, there's not a lot of point to being able to change HTML output dynamically unless you also have a means to track the information users provide to your website as they use it. In the early days of the web, many sites used "flat" text files to store data such as usernames and passwords. But this approach could cause problems if the file wasn't correctly locked against

corruption from multiple simultaneous accesses. Also, a flat file can get only so big before it becomes unwieldy to manage—not to mention the difficulty of trying to merge files and perform complex searches in any kind of reasonable time.

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

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

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

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

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

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

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

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

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

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

Using JavaScript

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

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

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

This code snippet tells the web browser to interpret everything within the `<script>` tags as JavaScript, which the browser does by writing the text `Today is` to the current document, along with the date, using the JavaScript function `Date`. The result will look something like this:

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

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

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

Asynchronous communication is what allows web pages to begin to resemble standalone programs, because they don’t have to be reloaded in their entirety to display new content. Instead, an asynchronous call can pull in and update a single element on a web page, such as changing your photograph on a social networking site or replacing a button that you click with the answer to a question. This subject is fully covered in Chapter 17.

SUPPLEMENTARY MATERIAL

A range of supplementary material is made available online (along with all the examples from the book) in a GitHub repository, comprising the following extra chapters in PDF format:

1. Introduction to CSS
2. Introduction to jQuery
3. Introduction to jQuery Mobile
4. Introduction to HTML5
5. The HTML5 Canvas
6. HTML5 Audio and Video
7. Other HTML5 Features
8. What's new in PHP 9 and MySQL 9

Using CSS

CSS is the crucial companion to HTML, ensuring that the HTML text and embedded images are laid out consistently and in a manner appropriate for the user's screen. With the emergence of the CSS3 standard in recent years, CSS now offers a level of dynamic interactivity previously supported only by JavaScript. For example, not only can you style any HTML element to change its dimensions, colors, borders, spacing, and so on, but now you can also add animated transitions and transformations to your web pages, using only a few lines of CSS.

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

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

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

As you'll learn in the supplementary chapter in the GitHub repository, Introduction to CSS, 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.

In the main body of the book you'll also learn all the new and more advanced features that come with CSS3, such as borders, shadows, text effects, transitions, transformations, and the tremendous power of the Flexbox and CSS Grid technologies.

And Then There's HTML5

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

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

It has taken a few years for HTML5 to develop, but now we are at a very solid and stable version 5.3 (the working group notes for which you can view here). It's a never-ending cycle of development, though, 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 the supplemental PDF chapter, *Introduction to HTML5*, available in the book's GitHub repository.

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

The Apache Web Server

In addition to PHP, MySQL, JavaScript, CSS, and HTML5, there's a sixth hero in the dynamic web: the web server. In the case of this book, that means the Apache web server. We've discussed a little of what a web server does during the HTTP server/client exchange, but it does much more behind the scenes.

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

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

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

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

Node.js: An Alternative to Apache

In 2009 developer Ryan Dahl was dissatisfied with Apache and its difficulties with handling large numbers of concurrent connections, and came up with a solution he called Node.js, which combined Google's V8 JavaScript engine, an event loop, and a low-level I/O API. Shortly afterwards, a package manager was introduced for the Node.js environment called NPM, which made it easier for programmers to publish and share source code of Node.js packages, simplifying installation, updating, and uninstallation of packages.

In 2011, Microsoft and Joyent implemented a native Windows version of Node.js, but then, after disagreements on how the project should progress, there followed a few years of competing versions (the other implementation being called io.js) until the communities finally opted to reunite and merged the two projects back together in 2015. This new version incorporated the V8 engine ES6 features, and offered a long-term support release cycle.

As of 2023 Node.js has reached version 19 and has become a fully mainstream alternative to the Apache web sever. This new edition of the book would have been remiss to not detail its benefits, and provide enough information to get you up and running with it, if you should so choose. Here are some of the main reasons you might make that choice:

Node.js uses an event-driven, non-blocking I/O model, allowing it to handle a large number of concurrent connections efficiently. This non-blocking nature enables scalable and high-performance applications, making it ideal for building real-time web applications, chat applications, and streaming services, for example.

It allows developers to use JavaScript on both the front-end and back-end, making it a full-stack development environment. This eliminates the need to switch between different programming languages, enabling better code reusability, and streamlining the development process. Yes, that

means you won't have to keep up-to-date with PHP if you make the switch.

Being built on the V8 JavaScript engine, Node,js provides exceptional performance, executing JavaScript code quickly and efficiently, resulting in faster response times and improved overall application performance. Additionally, Node.js has a small memory footprint, making it resource-efficient and suitable for deploying on cloud platforms.

As you will learn, there are many other solid reasons for using Node.js, but just these few are already highly persuasive.

About Open Source

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

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

Bringing It All Together

The real beauty of PHP, MySQL, JavaScript (sometimes aided by React, 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 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).

Google

How you'll sign in

Create a Gmail address for signing in to your Google Account

Username

arthurjohnson @gmail.com

That username is taken. Try another.

Available: aj8245778

Next

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

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

1. The server outputs the HTML to create the web form, which asks for the necessary details, such as username, first name, last name, and email address.
2. At the same time, the server attaches some JavaScript to the HTML to monitor the username input box and check for two things: whether some text has been typed into it, and whether the input has been deselected because the user has clicked another input box.
3. Once the text has been entered and the field deselected, in the background the JavaScript code passes the username that was entered back to a PHP script on the web server and awaits a response.
4. The web server looks up the username and replies back to the JavaScript regarding whether that name has already been taken.
5. The JavaScript then places an indication next to the username input box to show whether the name is available to the user—perhaps a green checkmark or a red cross graphic, along with some text.
6. If the username is not available and the user still submits the form, the JavaScript interrupts the submission and re-emphasizes (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.

Now, as in all this book's chapters, I recommend you see whether you can answer the following questions to check that you have absorbed its contents.

Questions

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

See Chapter 1 Answers in the Appendix for the answers to these questions.

Chapter 2. Setting Up a Development Server

A NOTE FOR EARLY RELEASE READERS

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

This will be the 2nd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/robinnixon/lpmj7*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

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

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

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

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

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

What Is a WAMP, MAMP, or LAMP?

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

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

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

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

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

Installing AMPPS on Windows

There are several available WAMP servers, each offering slightly different configurations. Different editions of this book have recommended different WAMP products according to which seems to offer the best features and appears the most reliable at the time. Currently AMPPS looks like the best option (although you could choose other alternatives if your preferred and still be able to follow through the examples in this book). You can download AMPPS by clicking the download button on the website's home page. (There are also Mac and Linux versions available; see "Installing AMPPS on macOS" and "Installing a LAMP on Linux".)

I recommend that you always download the latest stable release (as I write this, it's 4.3, the installer for which is about 46 MB in size). The various Windows, macOS, and Linux installers are listed on the download page.

Once you've downloaded the installer, run it to bring up the window shown in Figure 2-1. Before arriving at that window, though, if you use an antivirus program or have User Account Control activated on Windows, you may first be shown one or more advisory notices, and will have to click Yes and/or OK to continue with the installation.

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

```
C:\Program Files (x86)\Ampps
```

Figure 2-1. The opening window of the installer

> **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.

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

Once you have decided where to install AMPPS, click Next, decide where shortcuts should be

saved (the default shown is usually just fine), and click Next again to choose which icons you wish to install, as shown in Figure 2-2. On the screen that follows, click the Install button to start the process.

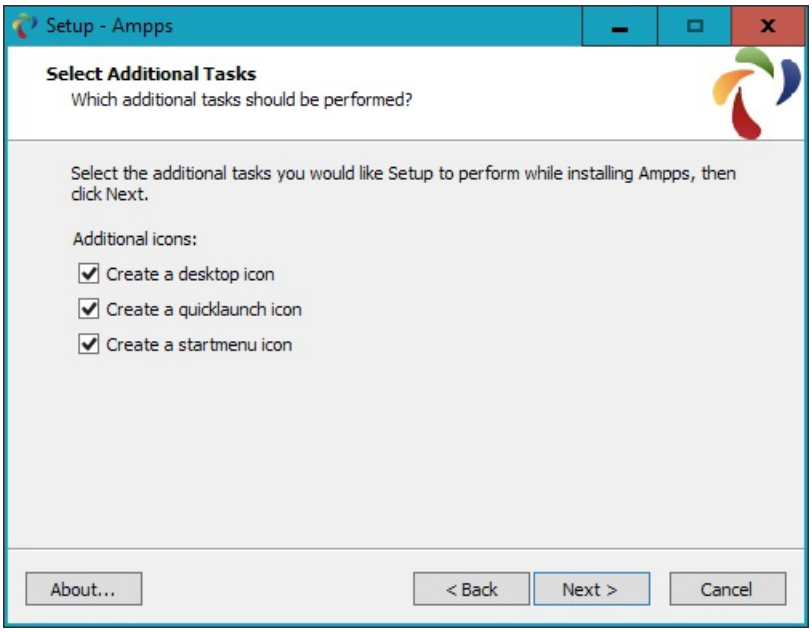


Figure 2-2. Choose which icons to install

Installation will take a few minutes, after which you should see the completion screen in Figure 2-3, and you can click Finish.

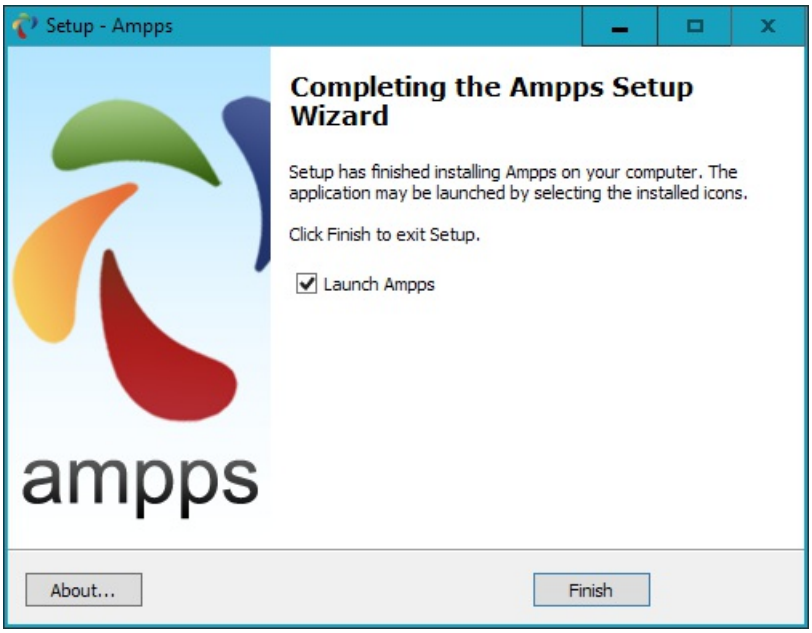


Figure 2-3. AMPPS is now installed

The final thing you must do is install Microsoft Visual C++ Redistributable, if you haven't already. A window will pop up to prompt you, as shown in Figure 2-4. Click Yes to start the

installation or No if you are certain you already have it. Or, you can always proceed anyway, and you will be told whether you don't need to reinstall it.

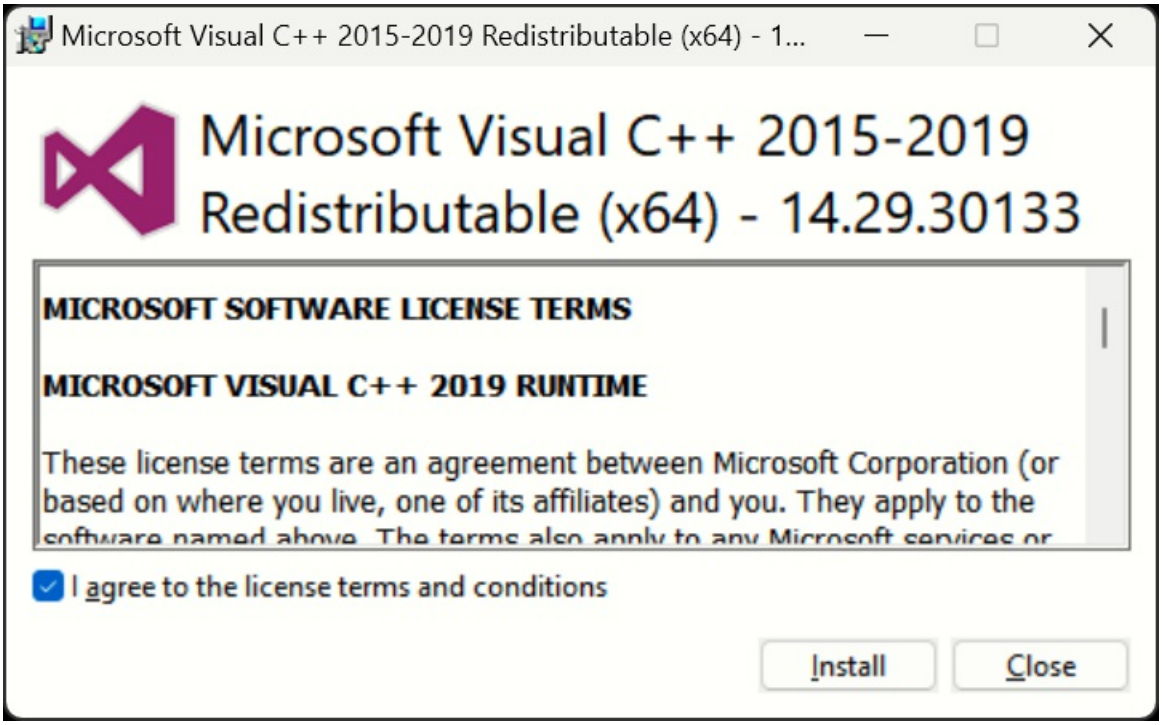


Figure 2-4. Install the Visual C++ redistributable if you don't already have it

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

Once AMPPS is installed, the control window shown in Figure 2-5 should appear at the bottom right of your desktop. You can also call up this window using the AMPPS application shortcut in the Start menu or on the desktop, if you allowed these icons to be created.

Before proceeding, if you have any further questions, I recommend you acquaint yourself with the AMPPS documentation, otherwise you are set to go—there's always a Support link at the bottom of the control window that will take you to the AMPPS website, where you can open up a trouble ticket should you need to.

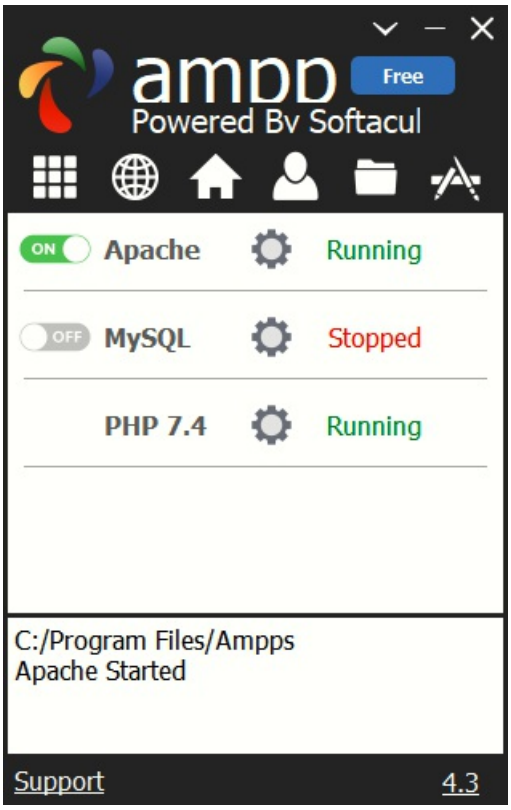


Figure 2-5. The AMPPS control window

You may notice that the default version of PHP in AMPPS is 7.4. If you wish to try other versions for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window and then select Change PHP Version, whereupon a new menu will appear from which you can choose to install a different version.

Testing the Installation

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

```
localhost
127.0.0.1
```

This will call up an introductory screen, where you will have the opportunity to secure AMPPS by giving it a password (see Figure 2-6). It is up to you now as to whether or not to secure the program. If only you will have access to the PC you may choose not to. But if there could be any security implications then you probably should password protect the installation.

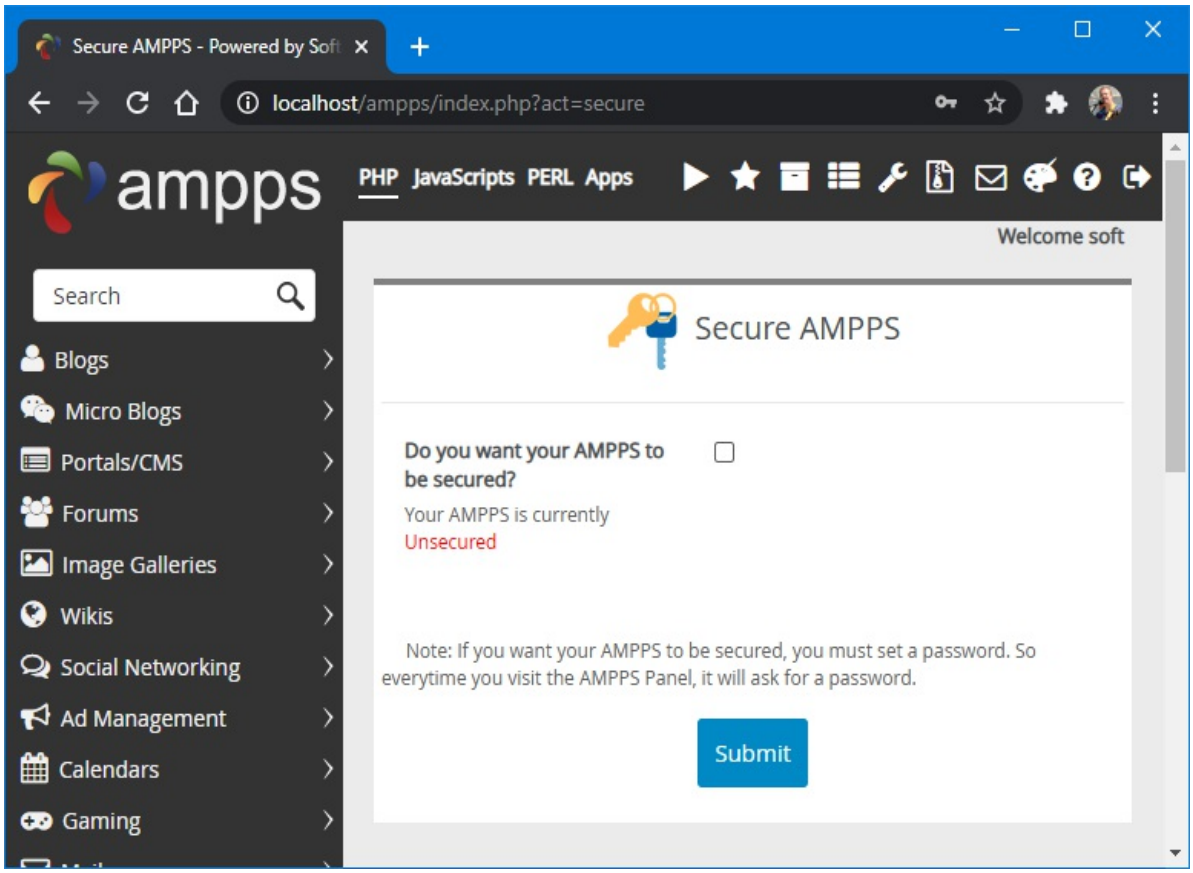


Figure 2-6. The initial security setup screen

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

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

```
localhost
```

This time, rather than seeing the initial screen about setting up security, you should see something similar to Figure 2-7, although the files shown may be different.

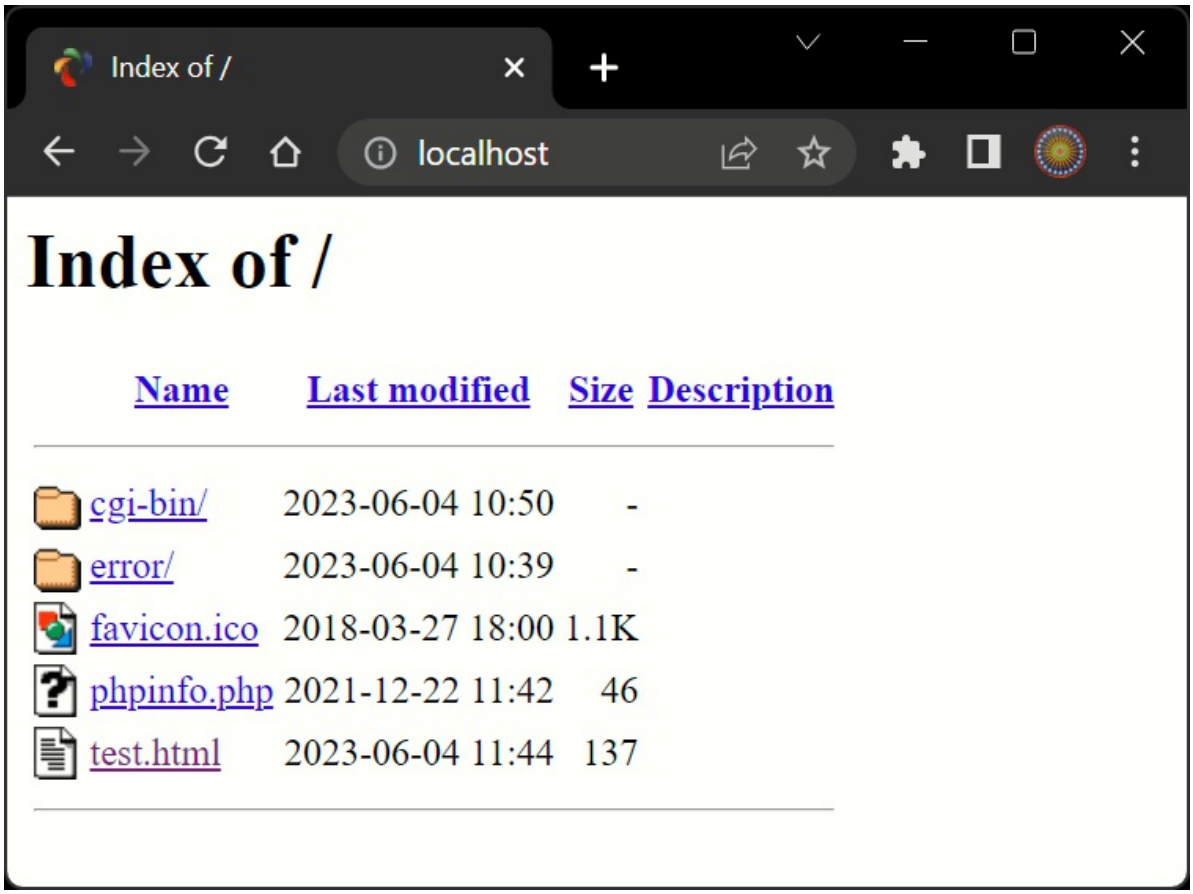


Figure 2-7. Viewing the document root

Accessing the Document Root (Windows)

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

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

```
C:\Program Files\Ampps\www
```

To ensure that you have everything correctly configured, you should now create the obligatory "Hello World" file. So, create a small HTML file along the following lines using Windows Notepad (or *Notepad++* on Windows, or *Atom* on a Mac, or your choice of many others available but not a rich word processor such as Microsoft Word):

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

```
    Hello World!
  </body>
</html>
```

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

You can now call this page up in your browser by entering the following URL in its address bar (see Figure 2-8):

```
localhost/test.html
```

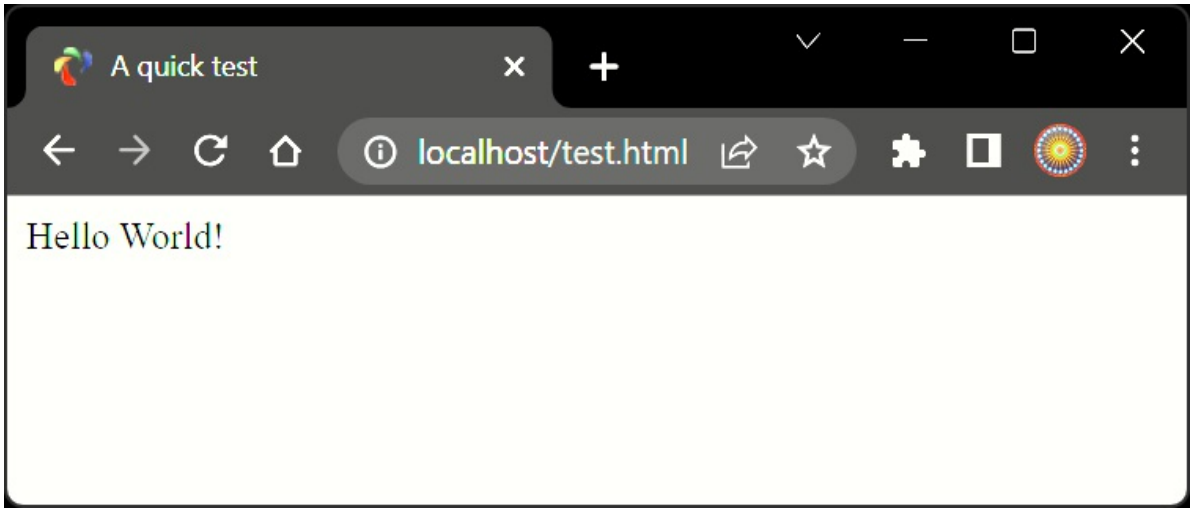


Figure 2-8. Your first web page

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

Alternative WAMPs

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

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

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

- EasyPHP
- XAMPP
- WAMPServer

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

Installing AMPPS on macOS

AMPPS is also available on macOS, and you can download it from the AMPPS website (as I write, the current version is 4.3, and the installer size is around 38 MB).

If your browser doesn't open it automatically once it has downloaded, double-click the *.dmg* file, and then drag the *AMPPS* folder over to your *Applications* folder (see Figure 2-9).

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 macOS password to proceed.

Once AMPPS is up and running, a control window similar to the one shown in Figure 2-5 will appear at the bottom left of your desktop.

> **NOTE**
>
> You may notice that the default version of PHP in AMPPS is 7.4. If you wish to try out a different version for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window, and then select Change PHP Version, whereupon a new menu will appear in which you can choose to install other versions of PHP.

Accessing the Document Root (macOS)

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

```
/Applications/Ampps/www
```

To ensure that you have everything correctly configured, you should now create the obligatory "Hello World" file. So, create a small HTML file along the following lines using the TextEdit program or any other program or text editor but not a rich word processor such as Microsoft Word or Pages:

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

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

You can now call this page up in your browser by entering the following URL in its address bar (to see a similar result to Figure 2-8):

```
localhost/test.html
```

> **NOTE**
>
> Remember that serving a web page from the document root (or a subfolder) is different from loading one into a web browser from your computer's filesystem. The former will ensure access to PHP,

MySQL, and all the features of a web server, while the latter will simply load the file into the browser, which will do its best to display it but will be unable to process any PHP or other server instructions. So, you should generally run examples using the *localhost* preface from your browser's address bar, unless you are certain that the file doesn't rely on web server functionality.

Installing a LAMP on Linux

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

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

```
localhost
```

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

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

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

Working Remotely

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

Accessing MySQL remotely may not be easy either. You should use the secure SSH protocol to log in to 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). I recommend you never use the insecure Telnet protocol to remotely log in to any server.

Logging In

I recommend that, at minimum, Windows users should install a program such as PuTTY for SSH

access (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 in to 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 SFTP or FTPS

To transfer files to and from your web server, you will generally need an FTP, SFTP, or FTPS program. Although FTP is not a secure protocol, software that helps you upload and download files is still often referred to as FTP, but it's FTPS or SFTP that you need to ensure proper security on your web server. If you go searching the web for a good client, you'll find so many that it could take you quite a while to come across one with all the right features for you.

> **DON'T USE FTP**
>
> FTP is insecure and should not be used. There are far safer methods than FTP for transferring files, such as using Git or similar technologies. Also, SSH-based SFTP (Secure File Transfer Protocol) and SCP (Secure Copy Protocol) are gaining traction. Good FTP programs, however, will also support SFTP and FTPS (FTP-SSL). Often the means of file transfer you use will be down to the policies of the company you work for, but for personal use an FTP program such as FileZilla (discussed next) will provide most (if not all) of the functionality and security you require.

My preferred SFTP program is the open source FileZilla, for Windows, Linux, and macOS 10.5 or newer (see Figure 2-10). Full instructions on how to use FileZilla are available on the wiki.

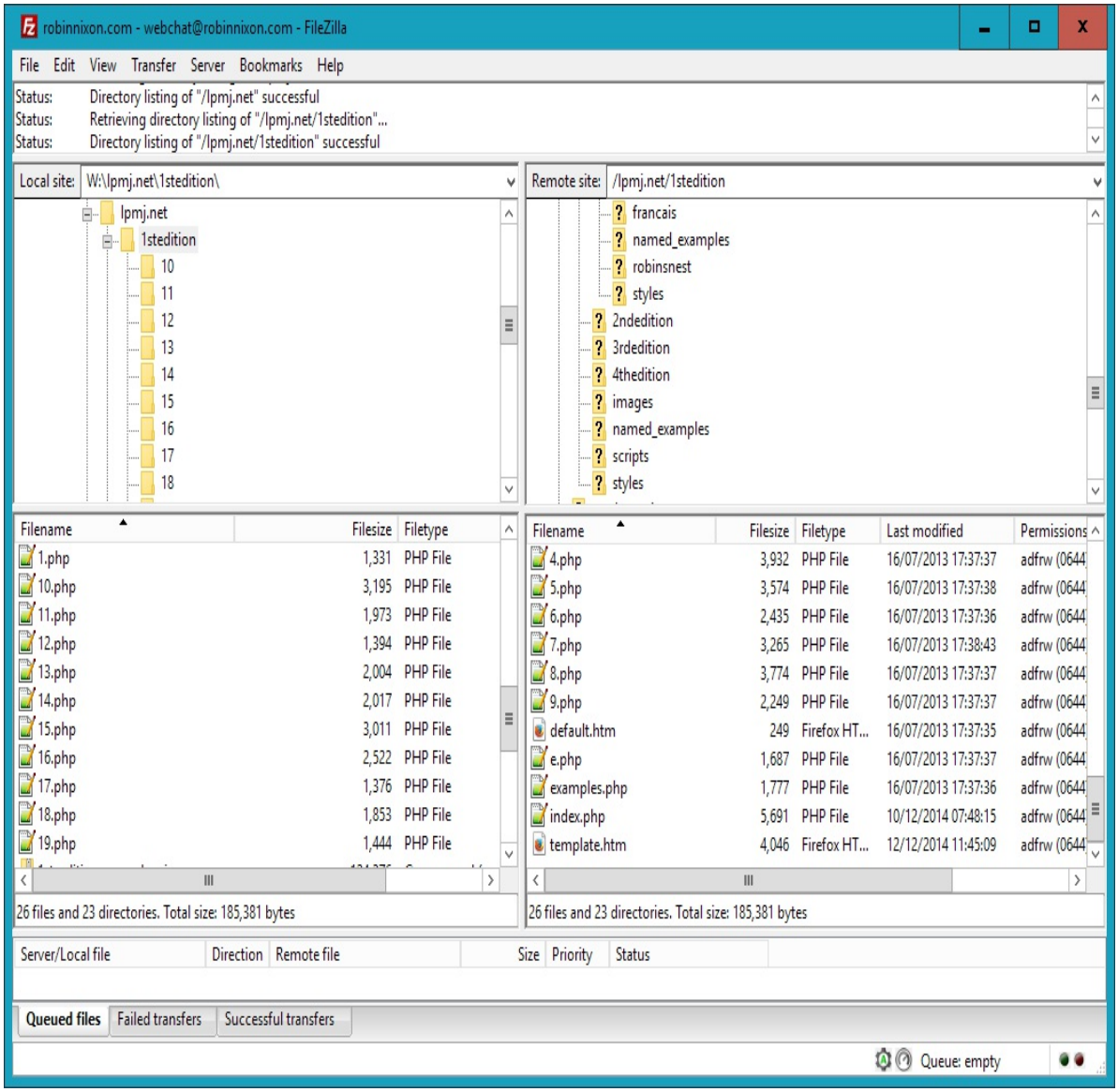


Figure 2-10. FileZilla is a full-featured SFTP program

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

Using a Code Editor

Although a plain-text editor works for editing HTML, PHP, and JavaScript, there have been some tremendous improvements in dedicated code 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 Visual Studio Code (VSC) from Microsoft because it's powerful; runs on all of Windows, Mac, and Linux; and is free (see

Figure 2-11). It is also a comprehensive developing environment and is becoming ever more standard in the industry.


File Edit Selection View Go Run Terminal Help
functions.php - Untitled (Workspace) - Visual Studio Code
EXTENSIONS: MARKETPLACE
@id:teabyii.ayu
Ayu
A simple theme with bright colors and co...
teabyii
Installed
functions.php
C: > Users > robin > OneDrive > Documents > GitHub > lpmj7 > robinsnest > functions.php
<?php // Example 01: functions.php
$host = 'localhost'; // Change as necessary
$data = 'robinsnest'; // Change as necessary
$user = 'robinsnest'; // Change as necessary
$pass = 'password'; // Change as necessary
$chrs = 'utf8mb4';
$attr = "mysql:host=$host;dbname=$data;charset=$chrs";
$opts =
[
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
PDO::ATTR_EMULATE_PREPARES => false,
];
try
{
$pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
throw new PDOException($e->getMessage(), (int)$e->getCode());
}
function createTable($name, $query)
{
queryMysql("CREATE TABLE IF NOT EXISTS $name($query)");
echo "Table '$name' created or already exists.
";
}

Figure 2-11. Program editors (like Visual Studio Code) are superior to plain-text editors

As you can see from Figure 2-11, VSC 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 it will highlight the matching ones so that you can check whether you have too many or too few. In fact, VSC does a lot more in addition, which you will discover and enjoy as you use it. You can download a copy from the Visual Studio website.

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, but I believe you will be hard pushed to find something better than the now industry standard VSC, so you should at least know how to use this product as many job positions will require it.

Having reached the end of this chapter you will now have everything set up and installed ready to commence your journey into mastering the various development technologies in this book, beginning with a solid introduction to PHP in the following chapter. But before you go, take a couple of minutes to answer the questions below to ensure you have remembered the main points.

Questions

1. What is the difference between a WAMP, a MAMP, and a LAMP?
2. What do the IP address 127.0.0.1 and the URL *http://localhost* have in common?
3. What is the purpose of an FTP program?
4. Name the main disadvantage of working on a remote web server.
5. Why is it better to use a program editor instead of a plain-text editor?

See Chapter 2 Answers in the Appendix for the answers to these questions.

Chapter 3. Introduction to PHP

A NOTE FOR EARLY RELEASE READERS

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

This will be the 3rd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/robinnixon/lpmj7*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

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

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. Let's 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 and save it as a PHP document (for example, saving *index.html* as *index.php*), and it will display identically to the original.

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

```
<?php
```

The first thing you may notice is that the tag has not been closed. This is because entire sections of PHP can be placed inside this tag, and they finish only when the closing part is encountered, which looks like this:

```
?>
```

A small PHP “Hello World” program might look like Example 3-1.

Example 3-1. Invoking PHP

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

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

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

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

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

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

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

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

This Book's Examples

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

In addition to listing all the examples by chapter and example number, some of the examples may require explicit filenames, in which case copies of the example(s) are also saved using the filename(s) in the same folder (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, and 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
```

> **SINGLE-LINE # COMMENTS**
>
> As well as using `//` to signify the start of a single-line comment, you can use the # symbol. However, this is less common and, as of PHP version 8, single-line comments starting with `#[` now have a special meaning (being treated as attributes). Consequently I prefer to stick with the `//` style.

When you need to use multiple lines, there's a second type of comment, which looks like Example 3-2.

Example 3-2. A multi-line 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 an 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 (if you have ever come across these), 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;
```

One of the most common causes 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

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

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

Variables

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

String variables

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

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

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):

```
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;
```

Now let's bring all these variables together to form a complete program, as in Example 3-4.

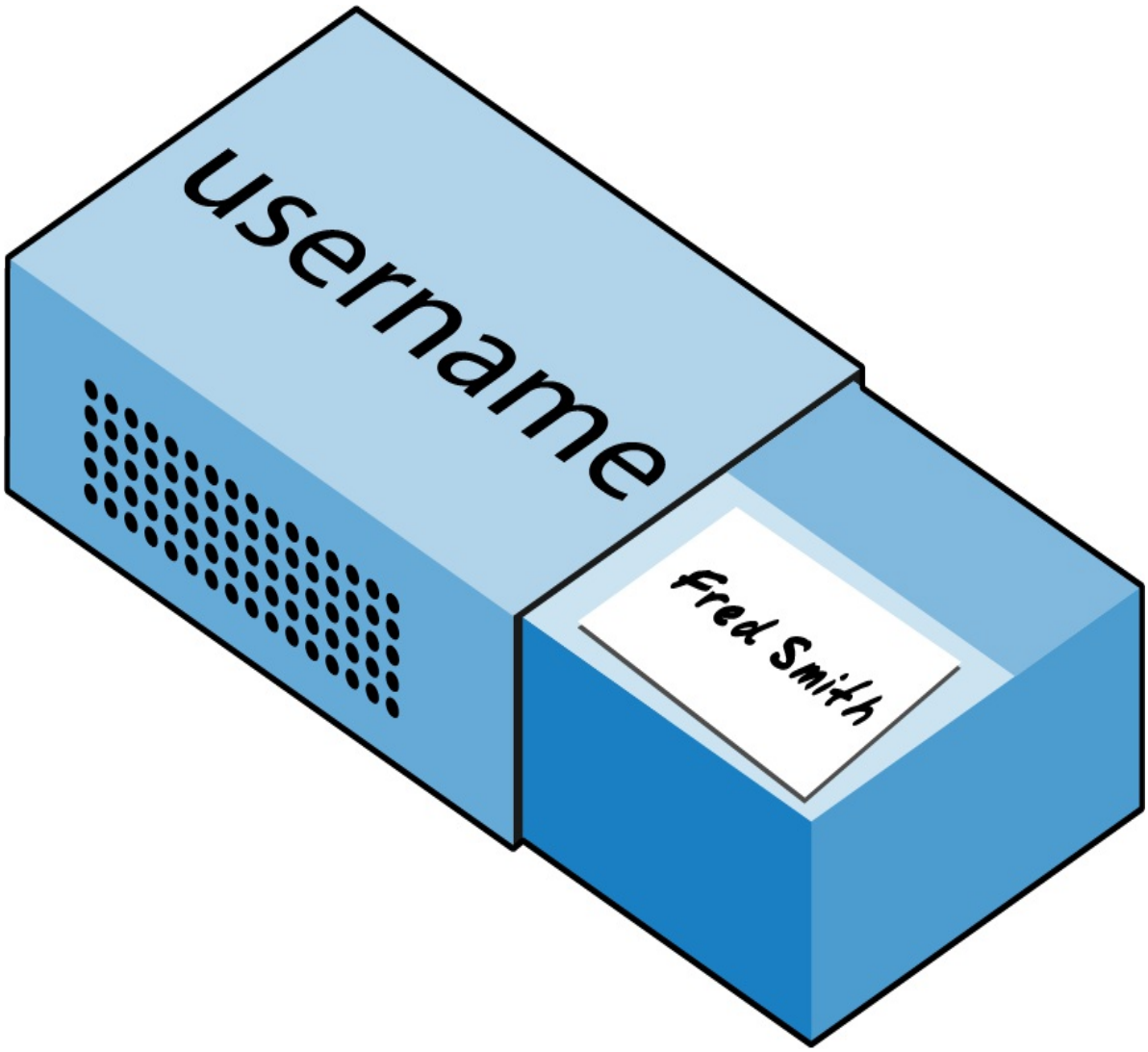


Figure 3-1. You can think of variables as matchboxes containing items

Example 3-4. Your first PHP program

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

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

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

NOTE

> In the unlikely event that during the installation of your web server (as detailed in Chapter 2) you changed the port assigned to the server to anything other than 80, then you must place that port number within the URL in this and all other examples in this book. So, for example, if you changed the port to 8080, the preceding URL would become this:
>
> ```
> http://localhost:8080/test1.php
> ```
>
> I won't mention this again, so just remember to use the port number (if required) when trying examples or writing your own code.

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

Numeric variables

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

```
$count = 17;
```

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

```
$count = 17.5;
```

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

Arrays

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

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

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

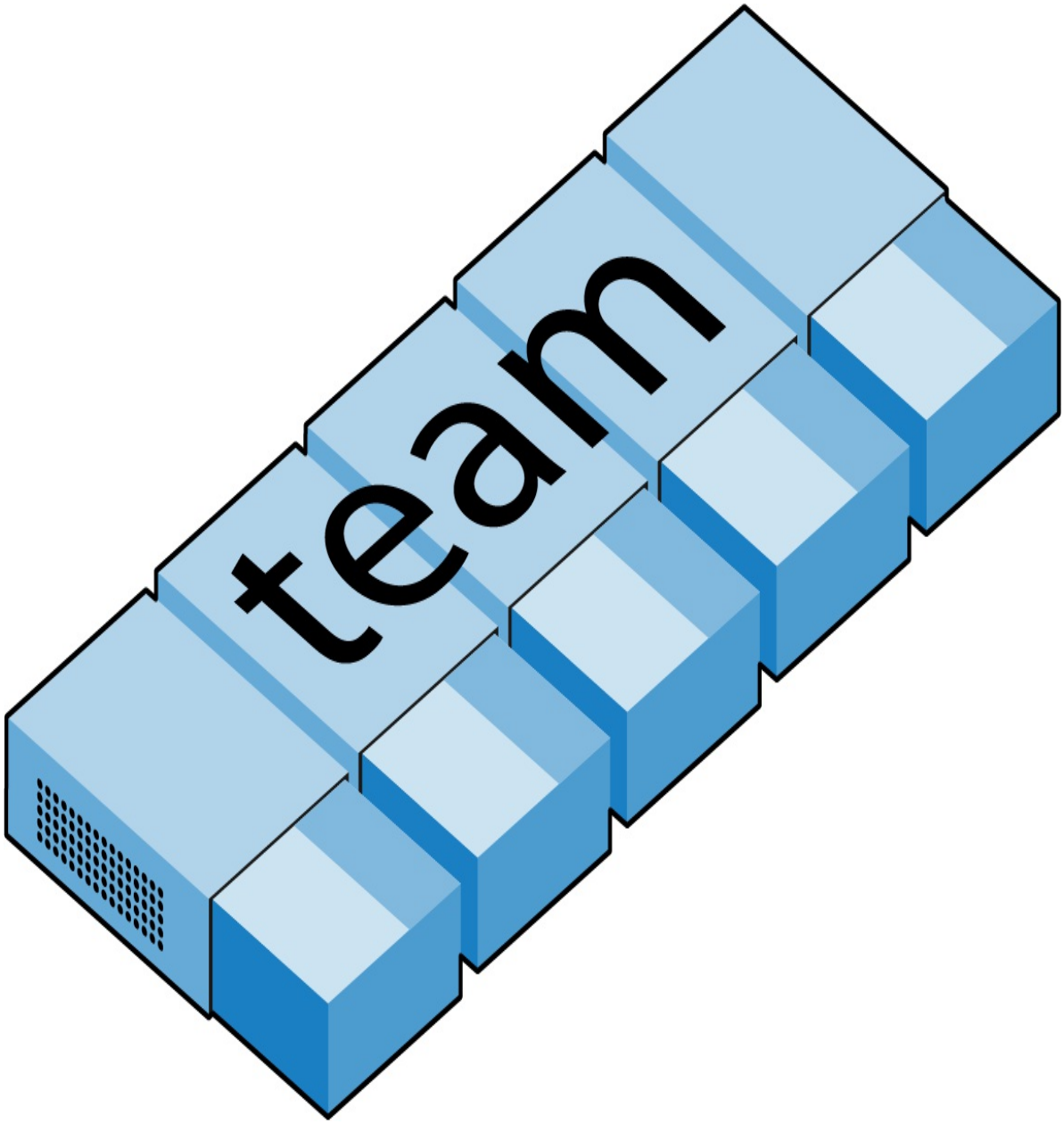


Figure 3-2. An array is like several matchboxes glued together

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

```
array();
```

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

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

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

The reason the previous statement has the number 3, not 4, is that 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 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 one other in a matrix of three rows by three columns using an array named `$oxo` (see Figure 3-3).

Figure 3-3. A multidimensional array simulated with matchboxes

You can now place a piece of paper with either an *x* or an *o* on it in the correct matchbox for each move played. To do this in PHP code, you have to set up an array containing three more arrays, as in Example 3-5, in which the array is set up with a game already in progress.

Example 3-5. Defining a two-dimensional array

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

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

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

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

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

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

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

Variable-naming rules

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

- Variable names, after the dollar sign, must start with a letter of the alphabet or the _ (underscore) character.
- Variable names can contain only the characters `a`–`z`, `A`–`Z`, `0`–`9`, and _ (underscore).
- Variable names may not contain spaces. If a variable name must comprise more than one

word, a good idea is to separate the words with the _ (underscore) character (e.g., `$user_name`).

- Variable names are case-sensitive. The variable `$High_Score` is not the same as the variable `$high_score`.

> **NOTE**
>
> To allow extended ASCII characters that include accents, PHP also supports the bytes from 127 through 255 in variable names. However, be aware that programmers using English keyboards will have difficulty accessing any you use.

Operators

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

```
echo 6 + 2;
```

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

Arithmetic operators

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

Table 3-1. Arithmetic operators

Operator	Description	Example
+	Addition	`$j + 1`
-	Subtraction	`$j - 6`
*	Multiplication	`$j * 11`
/	Division	`$j / 4`
%	Modulus (the remainder after a division is performed)	`$j % 9`
++	Increment	`++$j`

--	Decrement	--$j
**	Exponentiation (or power)	$j**2

Assignment operators

Assignment 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;
```

The /= and *= operators are similar, but for division and multiplication, the .= operator concatenates variables, such that $a .= "." will append a period to the end of $a, and %= assigns the modulus.

Table 3-2. Assignment operators

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

Comparison operators

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

Table 3-3. Comparison operators

Operator	Description	Example
==	Is *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`
<>	Is *not equal* to	`$j <> 23`
===	Is *identical* to	`$j === "987"`
!==	Is *not identical* to	`$j !== "1.2e3"`

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

Logical operators

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

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

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

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

Table 3-4 shows the logical operators.

Table 3-4. Logical operators

Operator	Description	Example
`&&`	*And*	`$j == 3 && $k == 2`
`and`	Low-precedence *and*	`$j == 3 and $k == 2`
`||`	*Or*	`$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 should avoid using them except when they are the only option, as in the following statement, which *must* use the `or` operator (`||` cannot be used to force a second statement to execute if the first fails):

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

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

```
$safe = ($ammonia xor $bleach);
```

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

Variable Assignment

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

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

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

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

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

Variable incrementing and decrementing

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

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

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

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

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

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

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

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

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

String concatenation

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

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

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

```
You have 5 messages.
```

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

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

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

String types

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

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

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

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

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

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

Escaping characters

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

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

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

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

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

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

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

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

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

Multiline Commands

There are times when you need to output quite a lot of text from PHP, and using several `echo` (or `print`) statements would be time-consuming and messy. To overcome this, PHP offers two conveniences. The first is just to put multiple lines between quotes, as in Example 3-6. Variables can also be assigned, as in Example 3-7.

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

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

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

  - $author.";
?>
```

Example 3-7. A multiline string assignment

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

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

  - $author.";
```

```
?>
```

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

Example 3-8. Alternative multiline `echo` statement

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

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

  - $author.
_END;
?>
```

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

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

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

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

Example 3-9. A multiline string variable assignment

```
<?php
  $author = "Scott Adams";

  $out = <<<_END
  Normal people believe that if it ain't broke, don't fix it.
  Engineers believe that if it ain't broke, it doesn't have enough
```

```
    features yet.

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

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

Be careful not to place a semicolon directly after the first occurrence of `_END`, as that would terminate the multi-line block before it had even started and cause a `Parse error` message.

By the way, the `_END` tag is simply one I chose for these examples because it is unlikely to be used anywhere else in PHP code and is therefore unique. You can use any tag you like, such as `_SECTION1` or `_OUTPUT` and so on. Also, to help differentiate tags such as this from variables or functions, the general practice is to preface them with an underscore.

Using a Nowdoc

If you wish to prevent PHP from parsing any variables encountered within a heredoc, you can use a *nowdoc* instead. It works in almost the same way, except that whatever you name you choose for your end tag should be enclosed in single quotes at the start of the nowdoc, as in Example 3-10, where the difference between it and Example 3-9 is shown in bold.

Example 3-10. A Nowdoc multiline 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;
?>
```

In this instance $author will *not* be replaced with the string Scott Adams, and will simply remain displayed as $author.

> **NOTE**
>
> Laying out text over multiple lines is usually just a convenience to make your PHP code easier to read, because once it is displayed in a web page, HTML formatting rules take over and whitespace is suppressed (but `$author` in our example will still be replaced with the variable's value).
>
> So, for example, if you load these multi-line 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 loosely typed language. This means that variables do not have to be declared before they are used and that PHP always converts variables to the type required by their context when they are accessed.

For example, you can create a multiple-digit number and extract the *n*th digit from it simply by assuming it to be a string. In Example 3-11, the numbers `12345` and `67890` are multiplied together, returning a result of `838102050`, which is then placed in the variable `$number`.

Example 3-11. Automatic conversion from a number to a string

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

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

The same goes for turning a string into a number, and so on. In Example 3-12, 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 3-12. 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, but do remember that sometimes automatic conversions do not operate quite as you might expect.

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.

For example, you can use a constant to hold the location of your server root (the folder with the main files of your website). You would define such a constant like this:

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

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

```
$directory = ROOT_LOCATION;
```

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

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

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

Predefined Constants

PHP comes ready-made with dozens of predefined constants that you won't generally use as a beginner. However, there are a few—known as the *magic constants*—that you will find useful. The names of the magic constants always have two underscores at the beginning and two at the end so that you won't accidentally try to name one of your own constants with a name that is already taken. They are detailed in Table 3-5. The concepts referred to in the table will be introduced in future chapters.

Table 3-5. PHP's magic constants

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.
`__FUNCTION__`	The function name. Returns the function name as it was declared (case-sensitive). In PHP 4, its value is always lowercase.
`__CLASS__`	The class name. Returns the class name as it was declared (case-sensitive). In PHP 4, its value is always lowercase.
`__METHOD__`	The class method name. The method name is returned as it was declared (case-sensitive).
`__NAMESPACE__`	The name of the current namespace. This constant is defined at compile time (case-sensitive).

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

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

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

The Difference Between the echo and print Commands

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

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

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

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

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

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

Functions

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

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

To create a function, declare it in the manner shown in Example 3-13.

Example 3-13. 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 2027*. Any number of parameters can be passed between the initial parentheses; we have chosen to accept just one. The curly braces enclose all the code that is executed when you later call the function. Note that the first letter within the `date` function call in this example is a lowercase letter L, not to be confused with the number 1.

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

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

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

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

which passes to `longdate` the current time less the number of seconds since 17 days ago (17

days × 24 hours × 60 minutes × 60 seconds).

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

Variable Scope

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

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

Local variables

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

One set of local variables is the list of arguments to a function. In the previous section, we defined a function that accepted a parameter named `$timestamp`. This is meaningful only in the body of the function; you can't get or set its value outside the function.

For another example of a local variable, take another look at the `longdate` function, which is modified slightly in Example 3-14.

Example 3-14. 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 3-15. Here `$temp` has been created *before* we call the `longdate` function.

Example 3-15. This attempt to access `$temp` in function `longdate` will fail

```
<?php
```

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

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

However, because `$temp` was neither created within the `longdate` function nor passed to it as a parameter, `longdate` cannot access it. Therefore, this code snippet outputs only the date, not the preceding text. In fact, depending on how PHP is configured, it may first display the error message `Notice: Undefined variable: temp`, something you don't want your users to see.

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

Some ways to repair Example 3-15 appear in Examples 3-16 and 3-17.

Example 3-16. 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 3-16 moves the reference to `$temp` out of the function. The reference appears in the same scope where the variable was defined.

Example 3-17. 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 3-17 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 in to your website and want all your code to know whether it is interacting with a logged-in user or a guest. One way to do this is to use the `global` keyword before a variable, such as `$IS_LOGGED_IN`:

```
global $IS_LOGGED_IN;
```

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

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

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

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

Static variables

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

Here's an interesting case. What if you have a local variable inside a function that you don't want any other parts of your code to have access to, but you would also like to keep its value for the next time the function is called? Why? Perhaps because you want a counter to track how many times a function is called. The solution is to declare a *static* variable, as shown in Example 3-18.

Example 3-18. A function using a static variable

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

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

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

If you plan to use static variables, you should note that you cannot assign the result of an expression in their definitions. They can be initialized only with predetermined values (see Example 3-19).

Example 3-19. Allowed and disallowed static variable declarations

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

Superglobal variables

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

These superglobals contain lots of useful information about the currently running program and

its environment (see Table 3-6). They are structured as associative arrays, a topic discussed in Chapter 6.

Table 3-6. PHP's superglobal variables

Superglobal name	Contents
`$GLOBALS`	All variables that are currently defined in the global scope of the script. The variable names are the keys of the array.
`$_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 common example. Among the many nuggets of information supplied by superglobal variables is the URL of the page that referred the user to the current web page. This referring page information can be accessed like this:

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

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

Superglobals and security

A word of caution is in order before you start using superglobal variables, because they are often

used by hackers trying to find exploits to break into your website. What they do is load up `$_POST`, `$_GET`, or other superglobals with malicious code, such as Unix or MySQL commands that can damage or display sensitive data if you naively access them.

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

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

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

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

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

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

Questions

1. What tag is used to invoke PHP to start interpreting program code? And what is the short form of the tag?
2. What are the two types of comment tags?
3. Which character must be placed at the end of every PHP statement?
4. Which symbol is used to preface all PHP variables?
5. What can a variable store?
6. What is the difference between `$variable = 1` and `$variable == 1`?
7. Why do you suppose that an underscore is allowed in variable names (`$current_user`), whereas hyphens are not (`$current-user`)?
8. Are variable names case-sensitive?
9. Can you use spaces in variable names?

10. How do you convert one variable type to another (say, a string to a number)?
11. What is the difference between `++$j` and `$j++`?
12. Are the operators `&&` and `and` interchangeable?
13. How can you create a multiline `echo` or assignment?
14. Can you redefine a constant?
15. How do you escape a quotation mark?
16. What is the difference between the `echo` and `print` commands?
17. What is the purpose of functions?
18. How can you make a variable accessible to all parts of a PHP program?
19. If you generate data within a function, what are a couple of ways to convey the data to the rest of the program?
20. What is the result of combining a string with a number?

See Chapter 3 Answers in the Appendix for the answers to these questions.

Chapter 4. Expressions and Control Flow in PHP

A NOTE FOR EARLY RELEASE READERS

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

This will be the 4th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/robinnixon/lpmj7*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

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

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

Expressions

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

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

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

Which in PHP would be:

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

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

TRUE or FALSE?

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

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

PHP doesn't actually print the predefined constants if you ask it to do so as in Example 4-1. For each line, the example prints out a letter followed by a colon and a predefined constant. PHP arbitrarily assigns a numerical value of `1` to TRUE, so `1` is displayed after `a:` when the example runs. Even more mysteriously, because `b:` evaluates to FALSE, it does not show any value. In PHP the constant FALSE is defined as NULL, another predefined constant that denotes nothing.

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

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

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

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

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

Example 4-2. Four simple Boolean expressions

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

The output from this code is:

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

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

Literals and Variables

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

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

Example 4-3. Literals and variables

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

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

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

```
a: 73
b: Hello
c:
d: Brian
e: 37
```

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

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

Example 4-4. An expression and a statement

```
<?php
  $days_to_new_year = 366 - $day_number; // Expression

  if ($days_to_new_year < 30)
  {
     echo "Not long now till new year";  // Statement
  }
?>
```

Operators

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

Table 4-1. PHP operator types

Operator	Description	Example
Arithmetic	Basic mathematics	`$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 < $b`
Execution	Execute contents of backticks	`` `ls -al` ``
Increment/decrement	Add or subtract 1	`$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`), take a single operand.
- *Binary* operators, which represent the bulk of PHP operators (including addition, subtraction, multiplication, and division), take two operands.
- The one *ternary* operator, which takes the form `expr ? x : y`, requires three operands. It's a terse, single-line `if` statement that returns `x` if `expr` is `TRUE` and `y` if `expr` is `FALSE`.

Operator Precedence

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

Example 4-5. Three equivalent expressions

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

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

Example 4-6. Three expressions that are also equivalent

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

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

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

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

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

Example 4-8. Three expressions showing implied parentheses

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

PHP evaluates the sub-expressions within parentheses first to derive the semi-completed expressions in Example 4-9.

Example 4-9. After evaluating the sub-expressions 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 order in this table is not arbitrary but carefully designed so that the most common and intuitive precedences are the ones you can get without parentheses. For instance, you can separate two comparisons with an `and` or `or` and get what you'd expect.

Associativity

We've been looking at processing expressions from left to right, except where operator precedence is in effect. But some operators require processing from right to left, and this direction of processing is called the operator's *associativity*. For some operators, there is no associativity.

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

Table 4-3. Operator associativity

Operator	Description	Associativity
`< <= >= == != === !== <>`	Comparison	None
`!`	Logical `NOT`	Right
`~`	Bitwise `NOT`	Right
`++ --`	Increment and decrement	Right

`(int)`	Cast to an integer	Right
`(double) (float) (real)`	Cast to a floating-point number	Right
`(string)`	Cast to a string	Right
`(array)`	Cast to an array	Right
`(object)`	Cast to an object	Right
`@`	Inhibit error reporting	Right
`= += -= *= /=`	Assignment	Right
`.= %= &= |= ^= <<= >>=`	Assignment	Right
`+`	Addition and unary plus	Left
`-`	Subtraction and negation	Left
`*`	Multiplication	Left
`/`	Division	Left
`%`	Modulus	Left
`.`	String concatenation	Left
`<< >> & ^ |`	Bitwise	Left
`?:`	Ternary	Left
`|| && and or xor`	Logical	Left
`,`	Separator	Left

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

Example 4-11. A multiple-assignment statement

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

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

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

Relational Operators

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

Equality

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

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

```
<?php
  $month = "March";

  if ($month == "April") echo "A quarter of a year has passed";
?>
```

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 (as in the case of *or* below, which is in lower rather than uppercase).

Example 4-16. The logical operators in use

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 4-5. All possible PHP logical expressions

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

TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Conditionals

Conditionals alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways. Conditionals are central to creating dynamic web pages—the goal of using PHP in the first place—because they make it easy to render different output each time a page is viewed.

I'll present three basic conditionals in this section: the `if` statement, the `switch` statement, and the `?` operator. In addition, looping conditionals (which we'll get to shortly) execute code over and over until a condition is met.

The if Statement

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

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

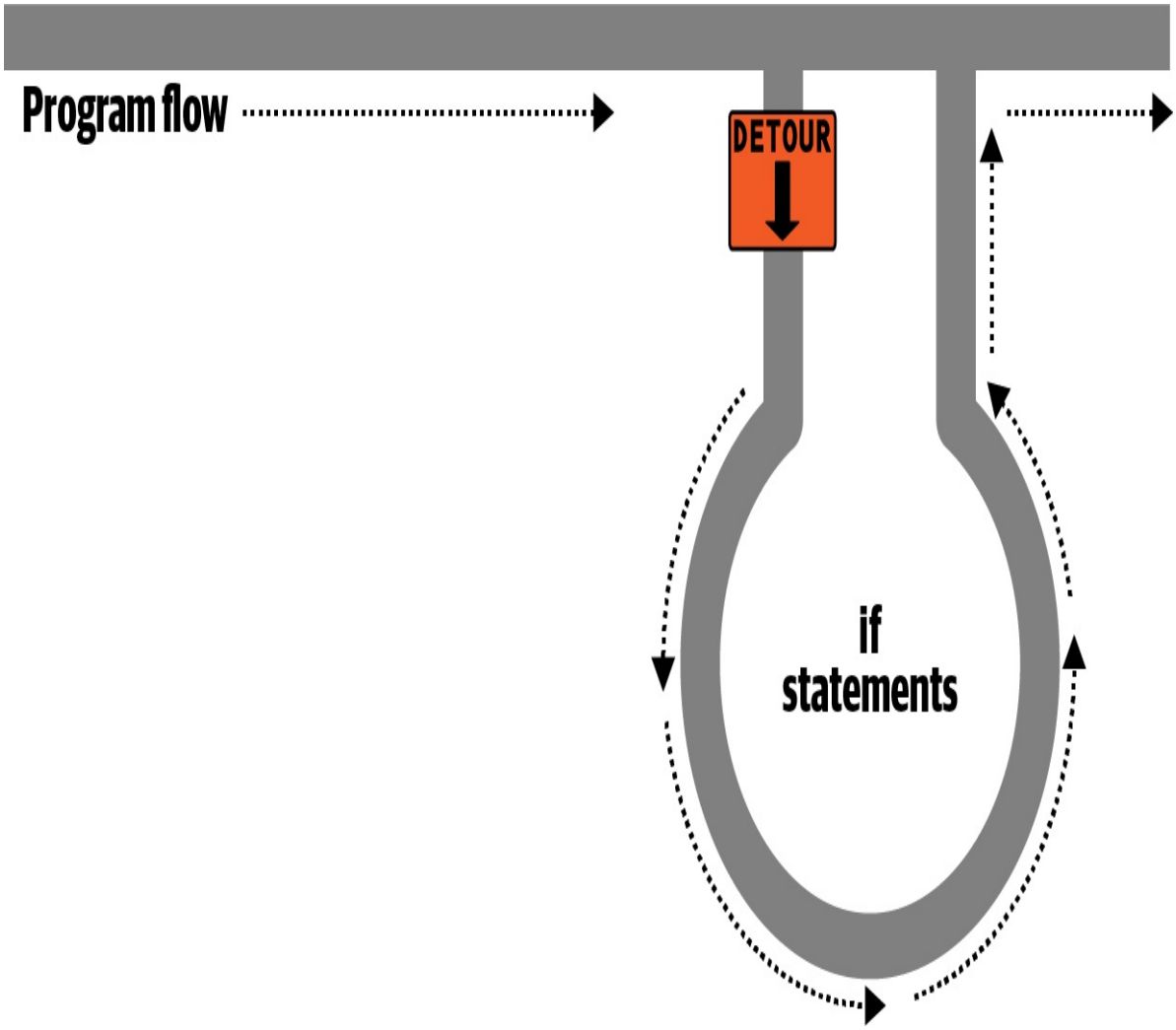


Figure 4-1. Program flow is like a single-lane highway

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

The actions to take when an `if` condition is `TRUE` are generally placed inside curly braces (`{ }`). You can ignore the braces if you have only a single statement to execute, but if you always use curly braces, you'll avoid having to hunt down difficult-to-trace bugs, such as when you add an extra line to a condition and it doesn't get evaluated due to the lack of braces.

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

statements.

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

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

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

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

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

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

In this book, opening curly braces generally start on a new line. Some people like to place the first curly brace to the right of the conditional expression; others start a new line with it. Either of these is fine, because PHP allows you to set out your whitespace characters (spaces, newlines, and tabs) any way you choose. However, you will find your code easier to read and debug if you indent each level of conditionals with a tab.

The else Statement

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

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.

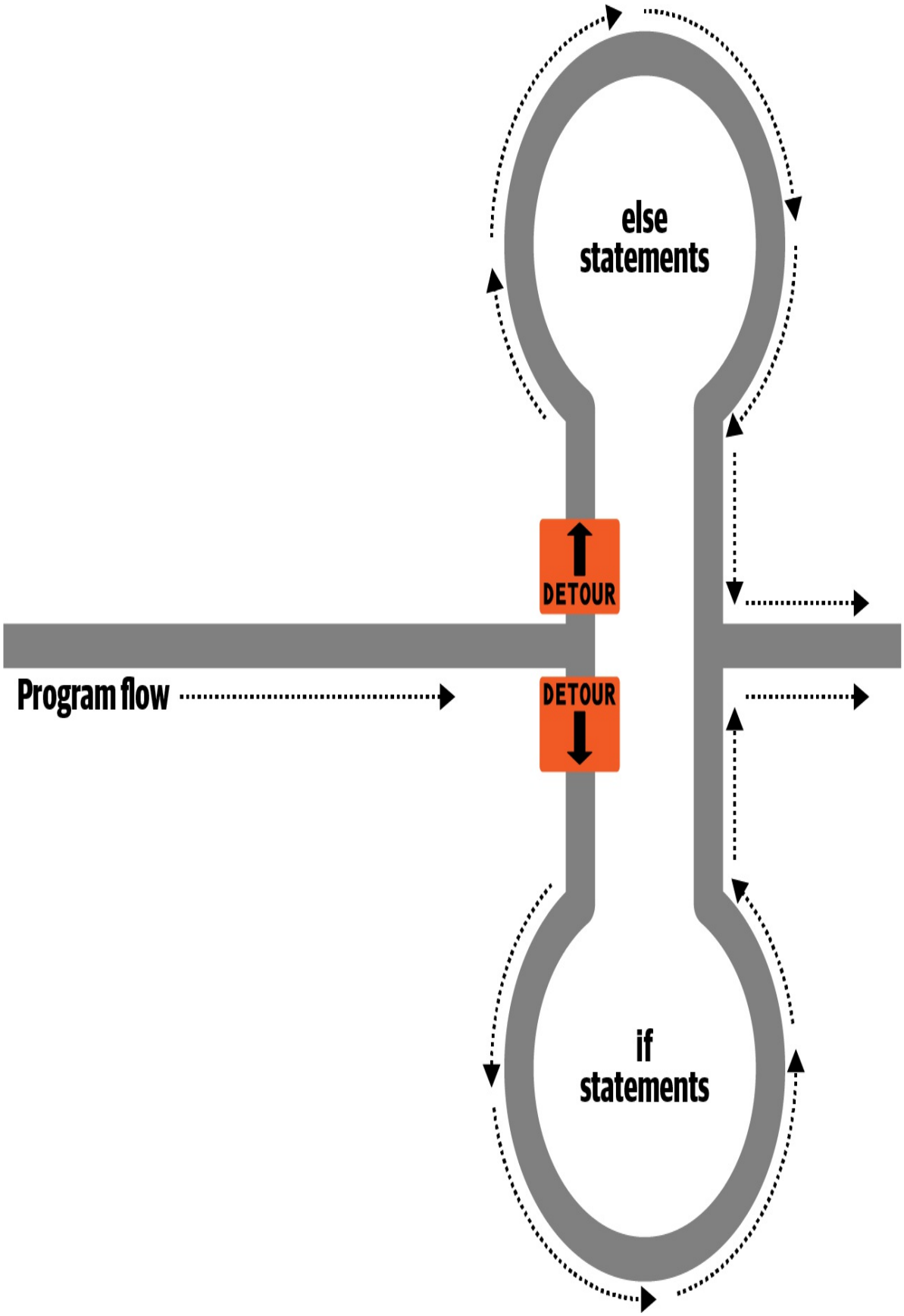

else
statements
DETOUR
Program flow
DETOUR
if
statements

Figure 4-2. 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, if you've ascertained that you have $100 or more in the bank, the `else` statement is executed, placing some of this money into your savings account.

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

The elseif Statement

There are also times when you want a number of different possibilities to occur, based upon a sequence of conditions. You can achieve this using the `elseif` statement. As you might imagine, it is like an `else` statement, except that you place a further conditional expression prior to the conditional code. In Example 4-21, you can see a complete `if...elseif...else` construct.

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

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

```
?>
```

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

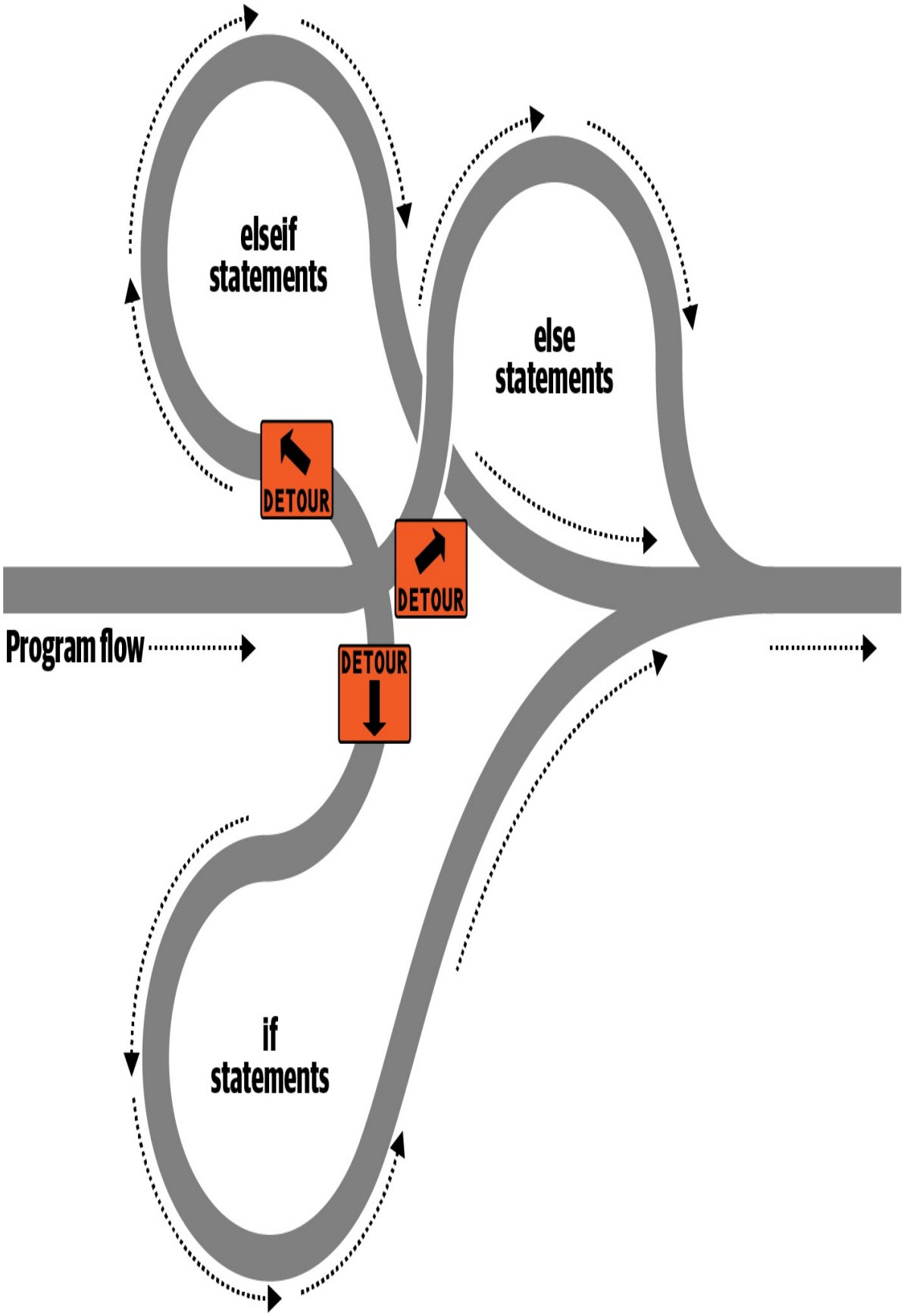

elseif
statements
else
statements
DETOUR
DETOUR
DETOUR
Program flow
if
statements

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`; you also cannot have an `elseif` before an `if` statement.

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

The switch Statement

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

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

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

Example 4-22. A multiline `if...elseif...else` statement

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

If we use a `switch` statement, the code might look like Example 4-23.

Example 4-23. A `switch` statement

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

```
        echo "You selected News";
        break;
    case "Login":
        echo "You selected Login";
        break;
    case "Links":
        echo "You selected Links";
        break;
  }
?>
```

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

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

Breaking out

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

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

Default action

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

Example 4-24. A default statement to add to Example 4-23

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

This replicates the effect of the `else` statement in Example 4-22.

Although a `break` command is not required here because the default is the final sub-statement and program flow will automatically continue to the closing curly brace, should you decide to move 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 ? (or ternary) Operator

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

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

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

Example 4-26. Using the ? operator

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

```
?>
```

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

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

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

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

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

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

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

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

It's a concise way to keep track of the largest value that you've seen as a program progresses. You save the largest value in `$saved` and compare it to `$new` each time you get a new value. Programmers familiar with the `?` operator find it more convenient than `if` statements for such short comparisons. When not used for writing compact code, it is typically used to make some decision inline, such as when you are testing whether a variable is set before passing it to a function.

Looping

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

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

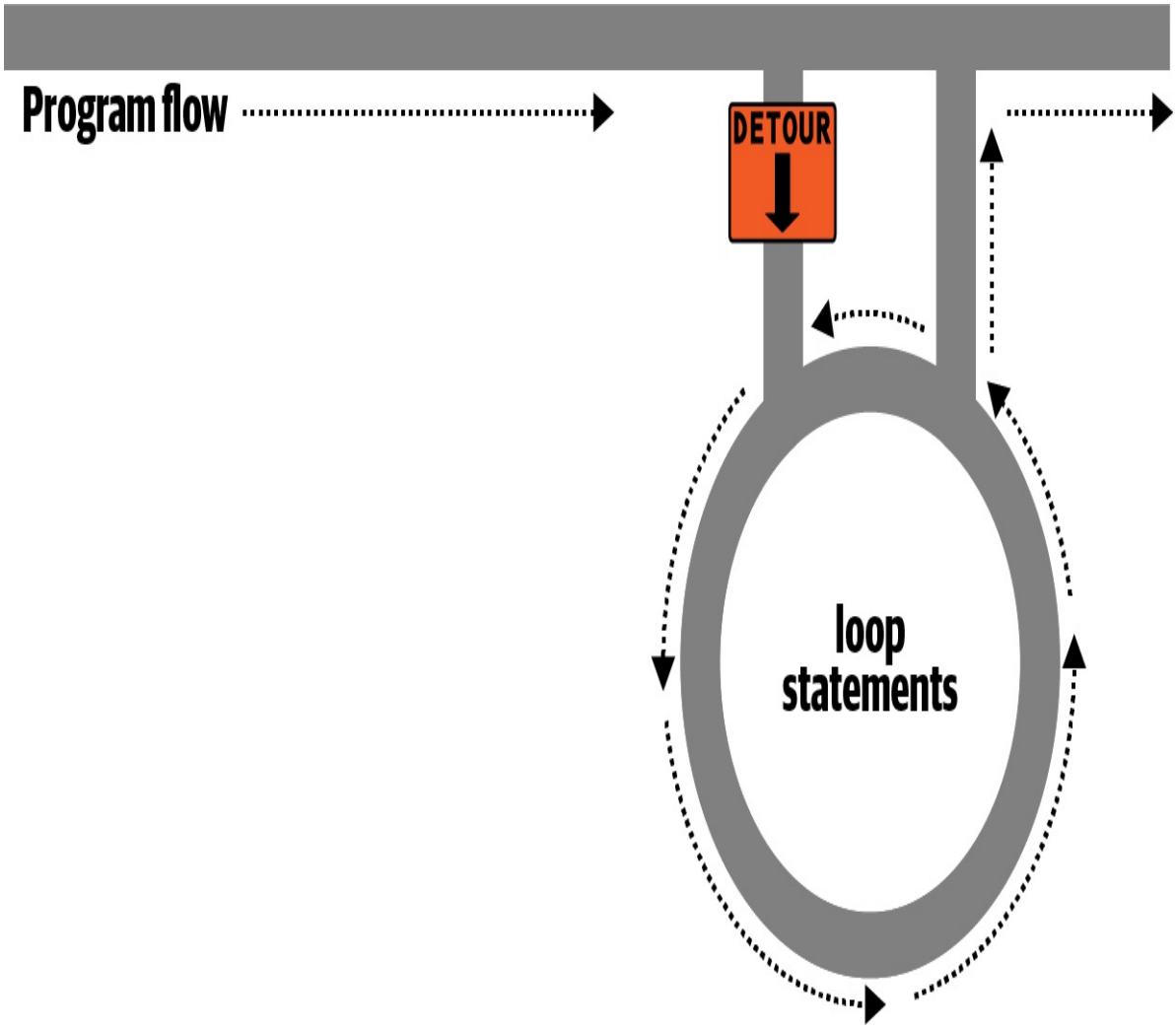


Figure 4-4. Imagining a loop as part of a program highway layout

while Loops

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

Example 4-28. A `while` loop

```
<?php
  $fuel = 10;

  while ($fuel > 1)
  {
    // Keep driving...
```

```
    echo "There's enough fuel";
  }
?>
```

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

> **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. Take a look at Example 4-30.

Example 4-30. A shortened version of Example 4-29

```
<?php
  $count = 0;

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

In this example, it was possible to move the `++$count` statement from the statements inside the `while` loop into the conditional expression of the loop. What now happens is that PHP encounters the variable `$count` at the start of each iteration of the loop and, noticing that it is prefaced with the increment operator, first increments the variable and only then compares it to the value `12`. You can therefore see that `$count` now has to be initialized to `0`, not `1`, because it is incremented as soon as the loop is entered. If you keep the initialization at `1`, only results between 2 and 12 will be output.

do...while Loops

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

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

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

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

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

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

```
<?php
```

```
  $count = 1;

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

for Loops

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

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

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

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

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

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

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

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

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

Example 4-34. The `for` loop from Example 4-33 with added curly braces

```
<?php
  for ($count = 1 ; $count <= 12 ; ++$count)
  {
```

```
    echo "$count times 12 is " . $count * 12;
    echo "<br>";
  }
?>
```

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

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

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

```
$i = 1, $j = 1  // Initialize $i and $j
$i + $j < 10    // Terminating condition
$i++, $j++      // Modify $i and $j at the end of each iteration
```

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

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

Breaking Out of a Loop

Just as you saw how to break out of a `switch` statement, you can also break out of a `for` loop (or any loop) using the same `break` command. This step can be necessary when, for example, one of your statements returns an error and the loop cannot continue executing safely. One case in which this might occur is when writing a file returns an error, possibly because the disk is full (see Example 4-35).

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

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

  for ($j = 0 ; $j < 100 ; ++$j)
```

```
  {
    $written = fwrite($fp, "data");

    if ($written == FALSE) break;
  }

  fclose($fp);
?>
```

This is the most complicated piece of code that you have seen so far, but you're ready for it. We'll look into the file-handling commands in Chapter 7, but for now all you need to know is that the first line opens the file *text.txt* for writing in binary mode and then returns a pointer to the file in the variable `$fp`, which is used later to refer to the open file.

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

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

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

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

using the `NOT` operator, like this:

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

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

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

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

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

```
break 2;
```

The continue Statement

The `continue` statement is a little like a `break` statement, except that it instructs PHP to stop

processing the current iteration of the loop and move right to its next iteration. So, instead of breaking out of the whole loop, PHP exits only the current iteration.

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

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

```
<?php
  $j = 11;

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

    if ($j == 0) continue;

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

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

Implicit and Explicit Casting

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

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

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

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

  echo $c;
?>
```

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

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

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

You can explicitly cast variables and literals to the types shown in Table 4-6.

Table 4-6. PHP's cast types

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

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

PHP Dynamic Linking

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

But although it is possible to build an entire website this way, it's not recommended, because

your source code will grow and grow and start to become unwieldy, as it has to account for every possible action a user could take.

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

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

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

Dynamic Linking in Action

One of the more popular PHP-driven applications on the web today is the Content Management System (CMS) WordPress (see Figure 4-5). 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, all linked dynamically.

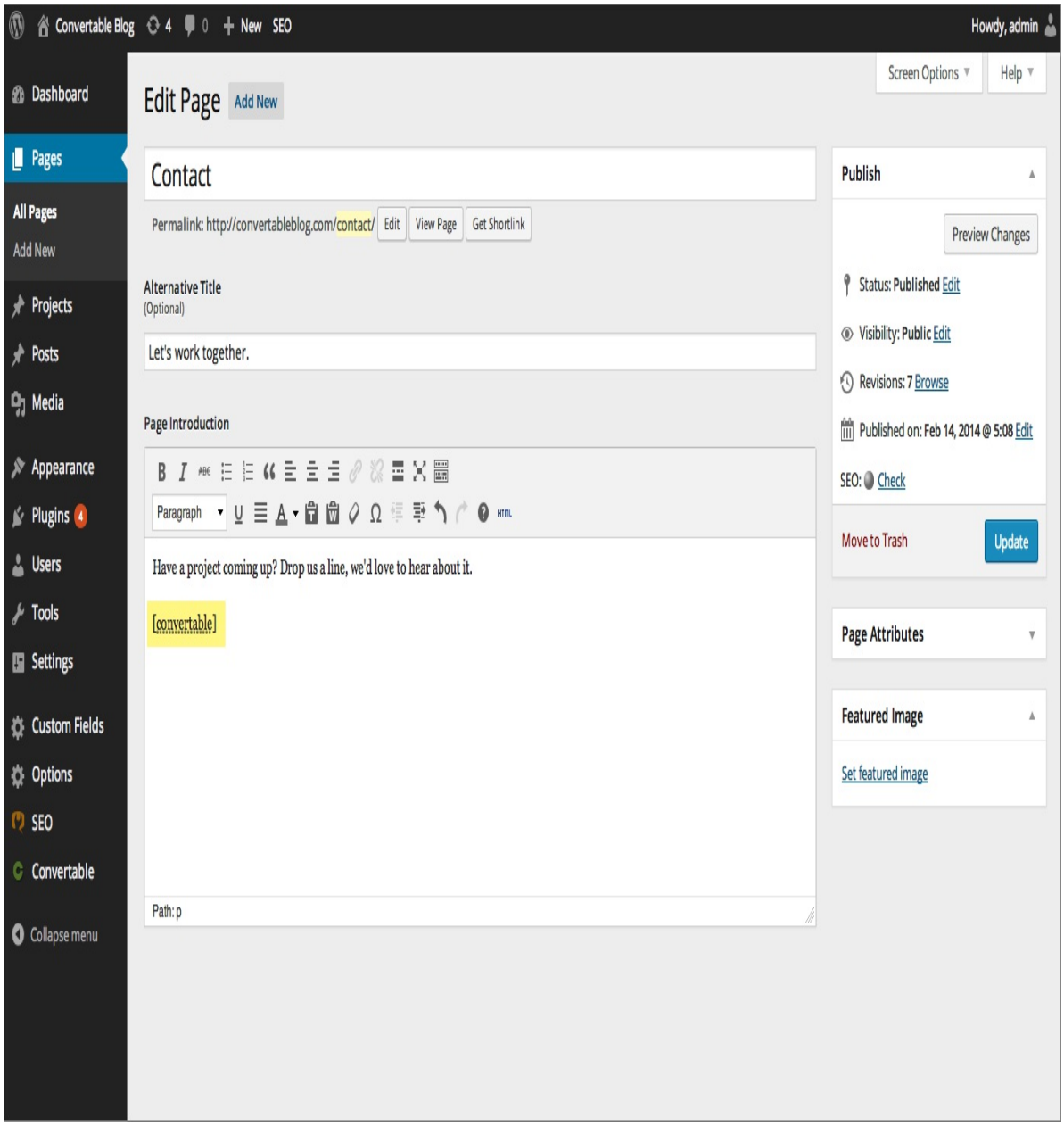


Figure 4-5. The WordPress CMS

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

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

following questions.

Questions

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

See Chapter 4 Answers in the Appendix for the answers to these questions.

Chapter 5. PHP Functions and Objects

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/robinnixon/lpmj7*.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

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

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

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

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

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

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

> **NOTE**
>
> It is now highly unusual (and definitely not recommended) to use any version of PHP lower than 5.4. Therefore, this chapter assumes that this release is the bare minimum version you will be working with, and I would strongly recommend you stick with version 7.4 (the latest release as of the time of writing). Should you need to use a different version for any reason you can install one from the AMPPS control panel, as described in Chapter 2.

PHP Functions

PHP comes with hundreds of ready-made, built-in functions, making it a very rich language. To use a function, call it by name. For example, you can see the `date` function in action here:

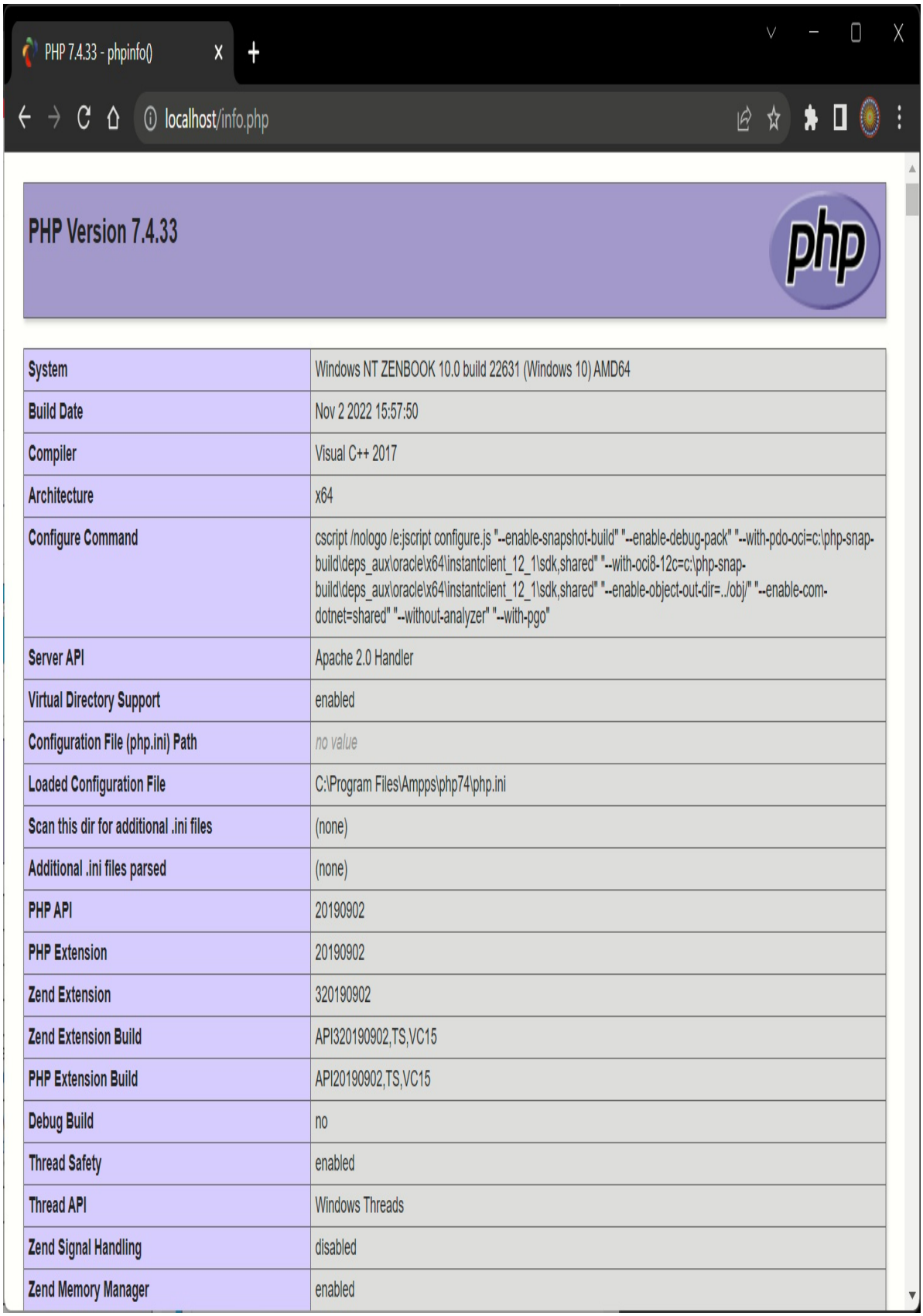
```
echo date("l"); // Displays the day of the week
```

The parentheses tell PHP that you're referring to a function. Otherwise, it thinks you're referring to a constant or variable.

Functions can take any number of arguments, including zero. For example, `phpinfo`, as shown next, displays lots of information about the current installation of PHP and requires no argument:

```
phpinfo();
```

The result of calling this function can be seen in Figure 5-1.



PHP Version 7.4.33

System	Windows NT ZENBOOK 10.0 build 22631 (Windows 10) AMD64
Build Date	Nov 2 2022 15:57:50
Compiler	Visual C++ 2017
Architecture	x64
Configure Command	cscript /nologo /e:jscript configure.js "--enable-snapshot-build" "--enable-debug-pack" "--with-pdo-oci=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--with-oci8-12c=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	*no value*
Loaded Configuration File	C:\Program Files\Ampps\php74\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20190902
PHP Extension	20190902
Zend Extension	320190902
Zend Extension Build	API320190902,TS,VC15
PHP Extension Build	API20190902,TS,VC15
Debug Build	no
Thread Safety	enabled
Thread API	Windows Threads
Zend Signal Handling	disabled
Zend Memory Manager	enabled

Figure 5-1. The output of PHP's built-in `phpinfo` function

> **WARNING**
>
> The `phpinfo` function is extremely useful for obtaining information about your current PHP installation, but that information could also be very useful to potential hackers. Therefore, never leave a call to this function in any web-ready code.

Some of the built-in functions that use one or more arguments appear in Example 5-1.

Example 5-1. Three string functions

```
<?php
  echo strrev(" .dlrow olleH"); // Reverse string
  echo str_repeat("Hip ", 2);   // Repeat string
  echo strtoupper("hooray!");   // String to uppercase
?>
```

This example uses three string functions to output the following text:

```
Hello world. Hip Hip HOORAY!
```

As you can see, the `strrev` function reversed the order of characters in the string, `str_repeat` repeated the string `"Hip "` twice (as required by the second argument), and `strtoupper` converted `"hooray!"` to uppercase.

Defining a Function

The general syntax for a function is as follows:

```
function function_name([parameter [, ...]])
{
  // Statements
}
```

The first line of the syntax indicates the following:

- A definition starts with the word `function`.
- A name follows, which must start with a letter or underscore, followed by any number of letters, numbers, or underscores.
- The parentheses are required.
- One or more parameters, separated by commas, are optional (as indicated by the square

brackets).

Function names are case-insensitive, so all of the following strings can refer to the `print` function: `PRINT`, `Print`, and `PrInT`.

The opening curly brace starts the statements that will execute when you call the function; a matching curly brace must close it. These statements may include one or more `return` statements, which force the function to cease execution and return to the calling code. If a value is attached to the `return` statement, the calling code can retrieve it, as we'll see next.

Returning a Value

Let's take a look at a simple function to convert a person's full name to lowercase and then capitalize the first letter of each part of the name.

We've already seen an example of PHP's built-in `strtoupper` function in Example 5-1. For our current function, we'll use its counterpart, `strtolower`:

```
$lowered = strtolower("aNY # of Letters and Punctuation you WANT");
echo $lowered;
```

The output of this experiment is as follows:

```
any # of letters and punctuation you want
```

We don't want names all lowercase, though; we want the first letter of each part of the sentence capitalized. (We're not going to deal with subtle cases such as Mary-Ann or Jo-En-Lai for this example.) Luckily, PHP also provides a `ucfirst` function that sets the first character of a string to uppercase:

```
$ucfixed = ucfirst("any # of letters and punctuation you want");
echo $ucfixed;
```

The output is as follows:

```
Any # of letters and punctuation you want
```

Now we can do our first bit of program design: to get a word with its initial letter capitalized, we call `strtolower` on the string first and then `ucfirst`. The way to do this is to nest a call to `strtolower` within `ucfirst`. Let's see why, because it's important to understand the order in which code is evaluated.

Say you make a simple call to the `print` function:

```
print(5-8);
```

The expression `5-8` is evaluated first, and the output is `-3`. (As you saw in the previous chapter, PHP converts the result to a string in order to display it.) If the expression contains a function, that function is evaluated first as well:

```
print(abs(5-8));
```

PHP is doing several things in executing that short statement:

1. Evaluate `5-8` to produce `-3`.
2. Use the `abs` function to turn `-3` into `3`.
3. Convert the result to a string and output it using the `print` function.

It all works because PHP evaluates each element from the inside out. The same procedure is in operation when we call the following:

```
ucfirst(strtolower("aNY # of Letters and Punctuation you WANT"))
```

PHP passes our string to `strtolower` and then to `ucfirst`, producing (as we've already seen when we played with the functions separately):

```
Any # of letters and punctuation you want
```

Now let's define a function (shown in Example 5-2) that takes three names and makes each one lowercase, with an initial capital letter.

Example 5-2. Cleaning up a full name

```
<?php
  echo fix_names("WILLIAM", "henry", "gatES");

  function fix_names($n1, $n2, $n3)
  {
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));

    return $n1 . " " . $n2 . " " . $n3;
  }
?>
```

You may well find yourself writing this type of code, because users often leave their Caps Lock key on, accidentally insert capital letters in the wrong places, and even forget capitals altogether. The output from this example is shown here:

```
William Henry Gates
```

Returning an Array

We just saw a function returning a single value. There are also ways of getting multiple values from a function.

The first method is to return them within an array. As you saw in Chapter 3, an array is like a bunch of variables stuck together in a row. Example 5-3 shows how you can use an array to return function values.

Example 5-3. Returning multiple values in an array

```
<?php
  $names = fix_names("WILLIAM", "henry", "gatES");
  echo $names[0] . " " . $names[1] . " " . $names[2];

  function fix_names($n1, $n2, $n3)
  {
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));

    return array($n1, $n2, $n3);
  }
?>
```

This method has the benefit of keeping all three names separate, rather than concatenating them into a single string, so you can refer to any user simply by first or last name without having to extract either name from the returned string.

Passing Arguments by Reference

In PHP versions prior to 5.3, you used to be able to preface a variable with the & symbol at the time of calling a function (for example, `increment(&$myvar);`) to tell the parser to pass a reference to the variable, not the variable's value. This granted a function access to the variable (allowing different values to be written back to it).

> **CAUTION**
>
> Call-time pass-by-reference was deprecated in PHP 5.3 and removed in PHP 5.4. You should therefore not use this feature other than on legacy websites, and even there it is recommended you rewrite code that passes by reference, because it will halt with a fatal error on newer versions of PHP.

However, *within* a function definition, you may continue to access arguments by reference. This concept can be hard to get your head around, so let's go back to the matchbox metaphor from

Imagine that instead of taking a piece of paper out of a matchbox, reading it, copying what's on it onto another piece of paper, putting the original back, and passing the copy to a function (phew!) you could simply attach a piece of thread to the original piece of paper and pass one end of it to the function (see Figure 5-2).

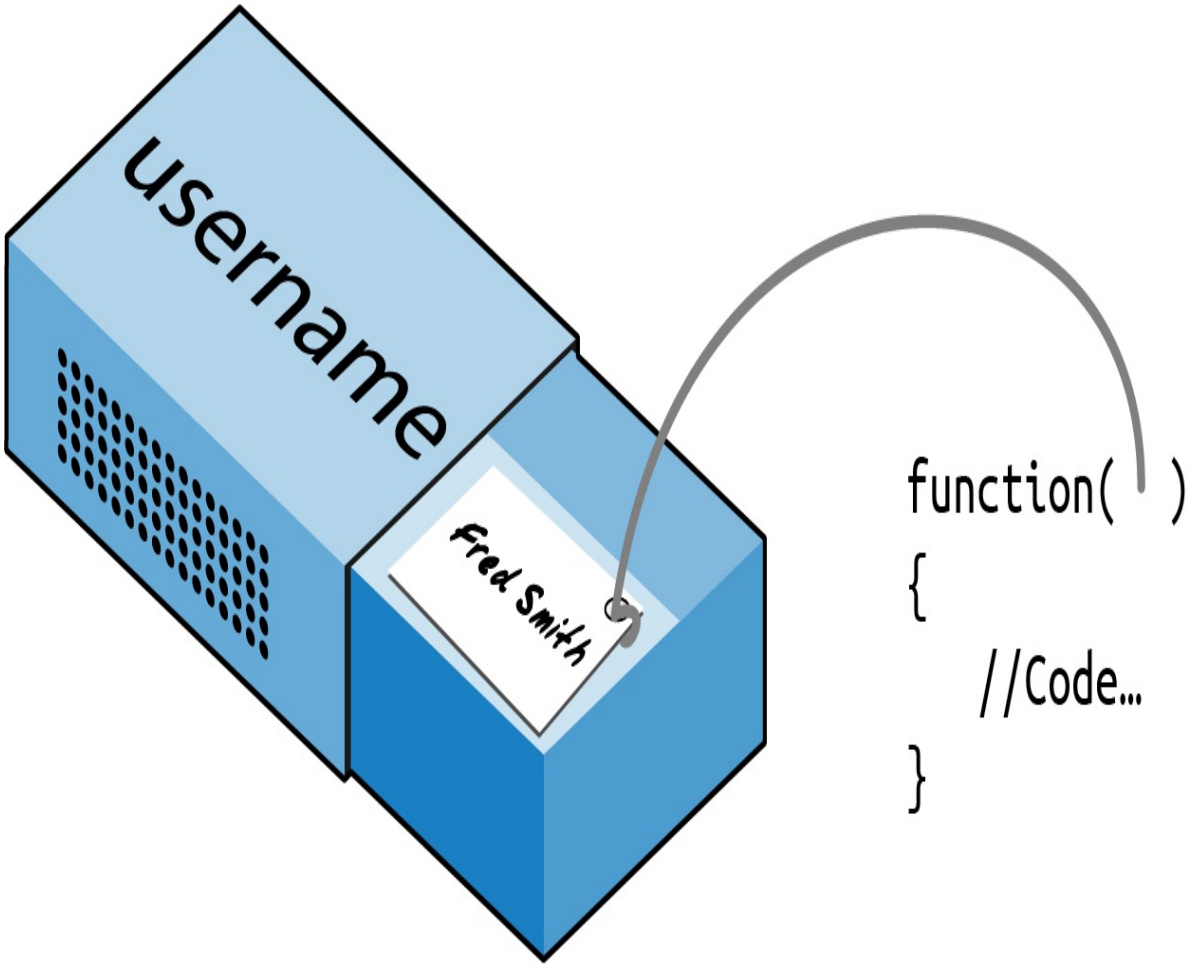


Figure 5-2. Imagining a reference as a thread attached to a variable

Now the function can follow the thread to find the data to be accessed. This prevents all the overhead of creating a copy of the variable just for the function's use. What's more, the function can now modify the variable's value.

This means you can rewrite Example 5-3 to pass references to all the parameters, and then the function can modify these directly (see Example 5-4).

Example 5-4. Passing values to a function by reference

```
<?php
  $a1 = "WILLIAM";
  $a2 = "henry";
  $a3 = "gatES";
```

```
  echo $a1 . " " . $a2 . " " . $a3 . "<br>";
  fix_names($a1, $a2, $a3);
  echo $a1 . " " . $a2 . " " . $a3;

  function fix_names(&$n1, &$n2, &$n3)
  {
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));
  }
?>
```

Rather than passing strings directly to the function, you first assign them to variables and print them out to see their "before" values. Then you call the function as before, but within the function definition you place an & symbol in front of each parameter to be passed by reference.

Now the variables `$n1`, `$n2`, and `$n3` are attached to "threads" that lead to the values of `$a1`, `$a2`, and `$a3`. In other words, there is one group of values, but two sets of variable names are allowed to access them.

Therefore, the function `fix_names` only has to assign new values to `$n1`, `$n2`, and `$n3` to update the values of `$a1`, `$a2`, and `$a3`. The output from this code is:

```
WILLIAM henry gatES
William Henry Gates
```

As you see, both of the `echo` statements use only the values of `$a1`, `$a2`, and `$a3`.

Returning Global Variables

The better way to give a function access to an externally created variable that is not passed as an argument is by declaring it to have global access from within the function. The `global` keyword followed by the variable name gives every part of your code full access to it (see Example 5-5).

Example 5-5. Returning values in global variables

```
<?php
  $a1 = "WILLIAM";
  $a2 = "henry";
  $a3 = "gatES";

  echo $a1 . " " . $a2 . " " . $a3 . "<br>";
  fix_names();
  echo $a1 . " " . $a2 . " " . $a3;

  function fix_names()
  {
    global $a1; $a1 = ucfirst(strtolower($a1));
```

```
    global $a2; $a2 = ucfirst(strtolower($a2));
    global $a3; $a3 = ucfirst(strtolower($a3));
  }
?>
```

Now you don't have to pass parameters to the function, and it doesn't have to accept them. Once declared, these variables retain global access and are available to the rest of your program, including its functions.

Recap of Variable Scope

A quick reminder of what you know from Chapter 3:

- *Local variables* are accessible just from the part of your code where you define them. If they're outside of a function, they can be accessed by all code outside of functions, classes, and so on. If a variable is inside a function, only that function can access the variable, and its value is lost when the function returns.
- *Global variables* are accessible from all parts of your code, whether within or outside of functions.
- *Static variables* are accessible only within the function that declared them but retain their value over multiple calls.

Including and Requiring Files

As you progress in your use of PHP programming, you are likely to start building a library of functions that you think you will need again. You'll also probably start using libraries created by other programmers.

There's no need to copy and paste these functions into your code. You can save them in separate files and use commands to pull them in. There are two commands to perform this action: `include` and `require`.

The include Statement

Using `include`, you can tell PHP to fetch a particular file and load all its contents. It's as if you pasted the included file into the current file at the insertion point. Example 5-6 shows how you would include a file called *library.php*.

Example 5-6. Including a PHP file

```
<?php
  include "library.php";

  // Your code goes here
?>
```

Using include_once

Each time you issue the `include` directive, it includes the requested file again, even if you've already inserted it. For instance, suppose that *library.php* contains a lot of useful functions, so you include it in your file, but you also include another library that includes *library.php*. Through nesting, you've inadvertently included *library.php* twice. This will produce error messages, because you're trying to define the same constant or function multiple times. So, you should use `include_once` instead (see Example 5-7).

Example 5-7. Including a PHP file only once

```
<?php
  include_once "library.php";

  // Your code goes here
?>
```

Then, any further attempts to include the same file (with `include` or `include_once`) will be ignored. To determine whether the requested file has already been executed, the absolute filepath is matched after all relative paths are resolved (to their absolute paths) and the file is found in your `include` path.

> **NOTE**
>
> In general, it's probably best to stick with `include_once` and ignore the basic `include` statement. That way, you will never have the problem of files being included multiple times.

Using require and require_once

A potential problem with `include` and `include_once` is that PHP will only *attempt* to include the requested file. Program execution continues even if the file is not found.

When it is absolutely essential to include a file, `require` it. For the same reasons I gave for using `include_once`, I recommend that you generally stick with `require_once` whenever you need to `require` a file (see Example 5-8).

Example 5-8. Requiring a PHP file only once

```
<?php
  require_once "library.php";

  // Your code goes here
?>
```

PHP Version Compatibility

PHP is in an ongoing process of development, and there are multiple versions. If you need to check whether a particular function is available to your code, you can use the `function_exists` function, which checks all predefined and user-created functions.

Example 5-9 checks for `array_combine`, a function specific to only some versions of PHP.

Example 5-9. Checking for a function's existence

```
<?php
  if (function_exists("array_combine"))
  {
    echo "Function exists";
  }
  else
  {
    echo "Function does not exist - better write our own";
  }
?>
```

Using code such as this, you can take advantage of features in newer versions of PHP and yet still have your code run on earlier versions where the newer features are unavailable, as long as you replicate any features that are missing. Your functions may be slower than the built-in ones, but at least your code will be much more portable.

You can also use the `phpversion` function to determine which version of PHP your code is running on. The returned result will be similar to the following, depending on the version:

```
7.4.33
```

PHP Objects

In much the same way that functions represent a huge increase in programming power over the early days of computing, where sometimes the best program navigation available was a very basic `GOTO` or `GOSUB` statement, *object-oriented programming* (OOP) takes the use of functions in a different direction.

Once you get the hang of condensing reusable bits of code into functions, it's not that great a leap to consider bundling the functions and their data into objects.

Let's take a social networking site that has many parts. One handles all user functions—that is, code to enable new users to sign up and existing users to modify their details. In standard PHP, you might create a few functions to handle this and embed some calls to the MySQL database to keep track of all the users.

To create an object to represent the current user, you could create a class, perhaps called `User`, that would contain all the code required for handling users and all the variables needed for manipulating the data within the class. Then, whenever you need to manipulate a user's data, you could simply create a new object with the `User` class.

You could treat this new object as if it were the actual user. For example, you could pass the object a name, password, and email address; ask it whether such a user already exists; and, if not, have it create a new user with those attributes. You could even have an instant messaging object, or one for managing whether two users are friends.

Terminology

When creating a program to use objects, you need to design a composite of data and code called a *class*. Each new object based on this class is called an *instance* (or *occurrence*) of that class.

The data associated with an object is called its *properties*; the functions it uses are called *methods*. In defining a class, you supply the names of its properties and the code for its methods. See Figure 5-3 for a jukebox metaphor for an object. Think of the CDs that it holds in the carousel as its properties (or, more likely these days, its properties will be an online database of song files); the method of playing them is to press buttons on the front panel. There is also a slot for inserting coins (the method used to activate the object) and a laser disc reader (the method used to retrieve the music, or properties, from the CDs), or software to download and play an online file.

Figure 5-3. A jukebox: a great example of a self-contained object

When you're creating objects, it is best to use *encapsulation*, or writing a class in such a way that only its methods can be used to manipulate its properties. In other words, you deny outside code direct access to its data. The methods you supply are known as the object's *interface*.

This approach makes debugging easy: you have to fix faulty code only within a class. Additionally, when you want to upgrade a program, if you have used proper encapsulation and maintained the same interface, you can simply develop new replacement classes, debug them fully, and then swap them in for the old ones. If they don't work, you can swap the old ones back in to immediately fix the problem before further debugging the new classes.

Once you have created a class, you may find that you need another class that is similar to it but not quite the same. The quick and easy thing to do is to define a new class using *inheritance*. When you do this, your new class has all the properties of the one it has inherited from. The original class is now called the *parent* (or occasionally the *superclass*), and the new one is the *subclass* (or *derived* class).

In our jukebox example, if you invent a new jukebox that can play a video along with the music, you can inherit all the properties and methods from the original jukebox superclass and add some new properties (videos) and new methods (a movie player).

An excellent benefit of this system is that if you improve the speed or any other aspect of the superclass, its subclasses will receive the same benefit. On the other hand, any change made to the parent/superclass could break the subclass.

Declaring a Class

Before you can use an object, you must define a class with the `class` keyword. Class definitions contain the class name (which is case-insensitive), its properties, and its methods. Example 5-10 defines the class `User` with two properties, which are `$name` and `$password` (indicated by the `public` keyword—see "Property and Method Scope"). It also creates a new instance (called `$object`) of this class.

Example 5-10. Declaring a class and examining an object

```
<?php
  $object = new User;
  print_r($object);

  class User
  {
    public $name, $password;

    function save_user()
    {
      echo "Save User code goes here";
    }
  }
?>
```

Here I have also used an invaluable function called `print_r`. It asks PHP to display information about a variable in human-readable form. (The `_r` stands for *human-readable*.) In the case of the new object `$object`, it displays the following:

```
User Object
(
  [name]     =>
  [password] =>
)
```

However, a browser compresses all the whitespace, so the output in a browser is slightly harder to read:

```
User Object ( [name] => [password] => )
```

In any case, the output says that `$object` is a user-defined object that has the properties `name` and `password`.

Creating an Object

To create an object with a specified class, use the `new` keyword, like this: `$object = new Class`. Here are a couple of ways in which we could do this:

```
$object = new User;
$temp   = new User('name', 'password');
```

On the first line, we simply assign an object to the `User` class. In the second, we pass arguments to the call.

A class may require or prohibit arguments; it may also allow arguments without explicitly requiring them.

Accessing Objects

Let's add a few lines to Example 5-10 and check the results. Example 5-11 extends the previous code by setting object properties and calling a method.

Example 5-11. Creating and interacting with an object

```
<?php
  $object = new User;
  print_r($object); echo "<br>";

  $object->name     = "Joe";
  $object->password = "mypass";
  print_r($object); echo "<br>";

  $object->save_user();

  class User
  {
    public $name, $password;

    function save_user()
    {
      echo "Save User code goes here";
    }
  }
?>
```

As you can see, the syntax for accessing an object's property is *`$object->property`*. Likewise, you call a method like this: *`$object->method()`*.

You should note that the example `property` and `method` do not have `$` signs in front of them.

If you were to preface them with $ signs, the code would not work, as it would try to reference the value inside a variable. For example, the expression `$object->$property` would attempt to look up the value assigned to a variable named `$property` (let's say that value is the string `brown`) and then attempt to reference the property `$object->brown`. If `$property` is undefined, an attempt to reference `$object->NULL` would occur and cause an error.

When looked at using a browser's View Source facility, the output from Example 5-11 is as follows:

```
User Object
(
  [name]     =>
  [password] =>
)
User Object
(
  [name]     => Joe
  [password] => mypass
)
Save User code goes here
```

Again, `print_r` shows its utility by providing the contents of `$object` before and after property assignment. From now on, I'll omit `print_r` statements, but if you are working along with this book on your development server, you can put some in to see exactly what is happening.

You can also see that the code in the method `save_user` was executed via the call to that method. It printed the string reminding us to create some code.

> **NOTE**
>
> You can place functions and class definitions anywhere in your code, before or after statements that use them. Generally, though, it is considered good practice to place them toward the end of a file.

Cloning Objects

Once you have created an object, it is passed by reference when you pass it as a parameter. In the matchbox metaphor, this is like keeping several threads attached to an object stored in a matchbox so that you can follow any attached thread to access it.

In other words, making object assignments does not copy objects in their entirety. You'll see how this works in Example 5-12, where we define a very simple `User` class with no methods and only the property `name`.

Example 5-12. Copying an object

```
<?php
  $object1       = new User();
  $object1->name = "Alice";
  $object2       = $object1;
  $object2->name = "Amy";

  echo "object1 name = " . $object1->name . "<br>";
  echo "object2 name = " . $object2->name;

  class User
  {
    public $name;
  }
?>
```

Here, we first create the object `$object1` and assign the value `Alice` to the `name` property. Then we create `$object2`, assigning it the value of `$object1`, and assign the value `Amy` just to the `name` property of `$object2`—or so we might think. But this code outputs the following:

```
object1 name = Amy
object2 name = Amy
```

What has happened? Both `$object1` and `$object2` refer to the *same* object, so changing the `name` property of `$object2` to `Amy` also sets that property for `$object1`.

To avoid this confusion, you can use the `clone` operator, which creates a new instance of the class and copies the property values from the original instance to the new instance. Example 5-13 illustrates this usage.

Example 5-13. Cloning an object

```
<?php
  $object1       = new User();
  $object1->name = "Alice";
  $object2       = clone $object1;
  $object2->name = "Amy";

  echo "object1 name = " . $object1->name . "<br>";
  echo "object2 name = " . $object2->name;

  class User
  {
    public $name;
  }
?>
```

Voilà! The output from this code is what we initially wanted:

```
object1 name = Alice
object2 name = Amy
```

Constructors

When creating a new object, you can pass a list of arguments to the class being called. These are passed to a special method within the class, called the *constructor*, which initializes various properties.

To do this you use the function name `__construct` (that is, `construct` preceded by two underscore characters), as in Example 5-14.

Example 5-14. Creating a constructor method

```
<?php
  class User
  {
    function __construct($param1, $param2)
    {
      // Constructor statements go here
    }
  }
?>
```

Destructors

You also have the ability to create *destructor* methods. This ability is useful when code has made the last reference to an object or when a script reaches the end. Example 5-15 shows how to create a destructor method. The destructor can do clean-up such as releasing a connection to a database or some other resource that you reserved within the class. Because you reserved the resource within the class, you have to release it here, or it will stick around indefinitely. Many system-wide problems are caused by programs reserving resources and forgetting to release them.

Example 5-15. Creating a destructor method

```
<?php
  class User
  {
    function __destruct()
    {
      // Destructor code goes here
    }
  }
?>
```

Writing Methods

As you have seen, declaring a method is similar to declaring a function, but there are a few differences. For example, method names beginning with a double underscore (__) are reserved (for example, for `__construct` and `__destruct`), and you should not create any of this form.

You also have access to a special variable called `$this`, which can be used to access the current object's properties. To see how it works, take a look at Example 5-16, which contains a different method from the `User` class definition called `get_password`.

Example 5-16. Using the variable `$this` in a method

```
<?php
  class User
  {
    public $name, $password;

    function get_password()
    {
      return $this->password;
    }
  }
?>
```

`get_password` uses the `$this` variable to access the current object and then return the value of that object's `password` property. Note how the preceding `$` of the property `$password` is omitted when we use the `->` operator. Leaving the `$` in place is a typical error you may run into, particularly when you first use this feature.

Here's how you would use the class defined in Example 5-16:

```
$object           = new User;
$object->password = "secret";

echo $object->get_password();
```

This code prints the password `secret`.

Declaring Properties

It is not necessary to explicitly declare properties within classes, as they can be implicitly defined when first used. To illustrate this, in Example 5-17 the class `User` has no properties and no methods but is legal code.

Example 5-17. Defining a property implicitly

```
<?php
  $object1       = new User();
  $object1->name = "Alice";

  echo $object1->name;

  class User {}
?>
```

This code correctly outputs the string `Alice` without a problem, because PHP implicitly declares the property `$object1->name` for you. But this kind of programming can lead to bugs that are infuriatingly difficult to discover, because `name` was declared from outside the class.

To help yourself and anyone else who will maintain your code, I advise that you get into the habit of always declaring your properties explicitly within classes. You'll be glad you did.

Also, when you declare a property within a class, you may assign a default value to it. The value you use must be a constant and not the result of a function or expression. Example 5-18 shows a few valid and invalid assignments.

Example 5-18. Valid and invalid property declarations

```
<?php
  class Test
  {
    public $name  = "Paul Smith"; // Valid
    public $age   = 42;           // Valid
    public $time  = time();       // Invalid - calls a function
    public $score = $level * 2;   // Invalid - uses an expression
  }
?>
```

Declaring Constants

In the same way that you can create a global constant with the `define` function, you can define constants inside classes. The generally accepted practice is to use uppercase letters to make them stand out, as in Example 5-19.

Example 5-19. Defining constants within a class

```
<?php
  Translate::lookup();

  class Translate
  {
    const ENGLISH = 0;
    const SPANISH = 1;
    const FRENCH  = 2;
    const GERMAN  = 3;
    // ...

    static function lookup()
    {
      echo self::SPANISH;
    }
  }
?>
```

You can reference constants directly, using the `self` keyword and double colon operator. Note that this code calls the class directly, using the double colon operator at line 1, without creating an instance of it first. As you would expect, the value printed when you run this code is `1`.

Remember that once you define a constant, you can't change it.

Property and Method Scope

PHP provides three keywords for controlling the scope of properties and methods (*members*):

`public`

: Public members can be referenced anywhere, including by other classes and instances of the object. This is the default when variables are declared with the `var` or `public` keywords, or when a variable is implicitly declared the first time it is used. The keywords `var` and `public` are interchangeable because, although deprecated, `var` is retained for compatibility with previous versions of PHP. Methods are assumed to be `public` by default.

`protected`

: These members can be referenced only by the object's class methods and those of any subclasses.

`private`

: These members can be referenced only by methods within the same class—not by subclasses.

Here's how to decide which you need to use:

- Use `public` when outside code *should* access this member and extending classes *should* also inherit it.
- Use `protected` when outside code *should not* access this member but extending classes *should* inherit it.
- Use `private` when outside code *should not* access this member and extending classes also *should not* inherit it.

Example 5-20 illustrates the use of these keywords.

Example 5-20. Changing property and method scope

```
<?php
  class Example
  {
    var $name   = "Michael"; // Same as public but deprecated
    public $age = 23;        // Public property
    protected $usercount;    // Protected property

    private function admin() // Private method
```

```
    {
      // Admin code goes here
    }
  }
?>
```

Static Methods

You can define a method as `static`, which means that it is called on a class, not on an object. A static method has no access to any object properties and is created and accessed as in Example 5-21.

Example 5-21. Creating and accessing a static method

```
<?php
  User::pwd_string();

  class User
  {
    static function pwd_string()
    {
      echo "Please enter your password";
    }
  }
?>
```

Note how we call the class itself, along with the static method, using a double colon (also known as the *scope resolution* operator), not `->`. Static functions are useful for performing actions relating to the class itself but not to specific instances of the class. You can see another example of a static method in Example 5-19.

> **NOTE**
>
> If you try to access `$this->property`, or other object properties from within a static function, you will receive an error message.

Static Properties

Most data and methods apply to instances of a class. For example, in a `User` class, you will want to do such things as set a particular user's password or check when the user has been registered. These facts and operations apply separately to each user and therefore use instance-specific properties and methods.

But occasionally you'll want to maintain data about a whole class. For instance, to report how many users are registered, you will store a variable that applies to the whole `User` class. PHP provides static properties and methods for such data.

As shown briefly in Example 5-21, declaring members of a class `static` makes them accessible without an instantiation of the class. A property declared `static` cannot be directly accessed within an instance of a class, but a static method can.

Example 5-22 defines a class called `Test` with a static property and a public method.

Example 5-22. Defining a class with a static property

```
<?php
  $temp = new Test();

  echo "Test A: " . Test::$static_property . "<br>";
  echo "Test B: " . $temp->get_sp()        . "<br>";
  echo "Test C: " . $temp->static_property . "<br>";

  class Test
  {
    static $static_property = "I'm static";

    function get_sp()
    {
       return self::$static_property;
    }
  }
?>
```

When you run this code, it returns the following output:

```
Test A: I'm static
Test B: I'm static
Notice: Undefined property: Test::$static_property
Test C:
```

This example shows that the property `$static_property` could be directly referenced from the class itself via the double colon operator in Test A. Also, Test B could obtain its value by calling the `get_sp` method of the object `$temp`, created from class `Test`. But Test C failed, because the static property `$static_property` was not accessible to the object `$temp`.

Note how the method `get_sp` accesses `$static_property` using the keyword `self`. This is how a static property or constant can be directly accessed within a class.

Inheritance

Once you have written a class, you can derive subclasses from it. This can save lots of painstaking code rewriting: you can take a class similar to the one you need to write, extend it to a subclass, and just modify the parts that are different. You achieve this using the `extends` keyword.

In Example 5-23, the class `Subscriber` is declared a subclass of `User` by means of the

`extends` keyword.

Example 5-23. Inheriting and extending a class

```
<?php
  $object           = new Subscriber;
  $object->name     = "Fred";
  $object->password = "pword";
  $object->phone    = "012 345 6789";
  $object->email    = "fred@bloggs.com";
  $object->display();

  class User
  {
    public $name, $password;

    function save_user()
    {
      echo "Save User code goes here";
    }
  }

  class Subscriber extends User
  {
    public $phone, $email;

    function display()
    {
      echo "Name:  " . $this->name     . "<br>";
      echo "Pass:  " . $this->password . "<br>";
      echo "Phone: " . $this->phone    . "<br>";
      echo "Email: " . $this->email;
    }
  }
?>
```

The original `User` class has two properties, `$name` and `$password`, and a method to save the current user to the database. `Subscriber` extends this class by adding an additional two properties, `$phone` and `$email`, and includes a method of displaying the properties of the current object using the variable `$this`, which refers to the current values of the object being accessed. The output from this code is as follows:

```
Name:  Fred
Pass:  pword
Phone: 012 345 6789
Email: fred@bloggs.com
```

The parent keyword

If you write a method in a subclass with the same name as one in its parent class, its statements will override those of the parent class. Sometimes this is not the behavior you want, and you

need to access the parent's method. To do this, you can use the `parent` operator, as in Example 5-24.

Example 5-24. Overriding a method and using the `parent` operator

```
<?php
  $object = new Son;
  $object->test();
  $object->test2();

  class Dad
  {
    function test()
    {
      echo "[Class Dad] I am your Father<br>";
    }
  }

  class Son extends Dad
  {
    function test()
    {
      echo "[Class Son] I am Luke<br>";
    }

    function test2()
    {
      parent::test();
    }
  }
?>
```

This code creates a class called `Dad` and a subclass called `Son` that inherits its properties and methods and then overrides the method `test`. Therefore, when line 2 calls the method `test`, the new method is executed. The only way to execute the overridden `test` method in the `Dad` class is to use the `parent` operator, as shown in function `test2` of class `Son`. The code outputs the following:

```
[Class Son] I am Luke
[Class Dad] I am your Father
```

If you wish to ensure that your code calls a method from the current class, you can use the `self` keyword, like this:

```
self::method();
```

Subclass constructors

When you extend a class and declare your own constructor, you should be aware that PHP will

not automatically call the constructor method of the parent class. If you want to be certain that all initialization code is executed, subclasses should always call the parent constructors, as in Example 5-25.

Example 5-25. Calling the parent class constructor

```
<?php
  $object = new Tiger();

  echo "Tigers have...<br>";
  echo "Fur: " . $object->fur . "<br>";
  echo "Stripes: " . $object->stripes;

  class Wildcat
  {
    public $fur; // Wildcats have fur

    function __construct()
    {
      $this->fur = "TRUE";
    }
  }

  class Tiger extends Wildcat
  {
    public $stripes; // Tigers have stripes

    function __construct()
    {
      parent::__construct(); // Call parent constructor first
      $this->stripes = "TRUE";
    }
  }
?>
```

This example takes advantage of inheritance in the typical manner. The `Wildcat` class has created the property `$fur`, which we'd like to reuse, so we create the `Tiger` class to inherit `$fur` and additionally create another property, `$stripes`. To verify that both constructors have been called, the program outputs the following:

```
Tigers have...
Fur: TRUE
Stripes: TRUE
```

Final methods

When you wish to prevent a subclass from overriding a superclass method, you can use the `final` keyword. Example 5-26 shows how.

Example 5-26. Creating a `final` method

```
<?php
  class User
  {
    final function copyright()
    {
      echo "This class was written by Joe Smith";
    }
  }
?>
```

Once you have digested the contents of this chapter, you should have a strong feel for what PHP can do for you. You should be able to use functions with ease and, if you wish, write object-oriented code. In Chapter 6, we'll finish off our initial exploration of PHP by looking at the workings of PHP arrays, but first test your understanding of this chapter with the questions below.

Questions

1. What is the main benefit of using a function?
2. How many values can a function return?
3. What is the difference between accessing a variable by name and by reference?
4. What is the meaning of *scope* in PHP?
5. How can you incorporate one PHP file within another?
6. How is an object different from a function?
7. How do you create a new object in PHP?
8. What syntax would you use to create a subclass from an existing one?
9. How can you cause an object to be initialized when you create it?
10. Why is it a good idea to explicitly declare properties within a class?

See Chapter 5 Answers in the Appendix for the answers to these questions.

About the Author

Robin Nixon is a broadcaster and author with over 40 years of experience with writing software, developing websites and apps, and managing teams of developers. He also has an extensive history of writing about computers and technology, with a portfolio of over 500 published magazine articles and over 40 books, many of which have been translated into other languages.

Robin started his computing career in the Cheshire homes for disabled people, where he was responsible for setting up computer rooms in a number of residential homes, and for evaluating and tailoring hardware and software so that disabled people could use the new technology - sometimes by means of only a single switch operated by mouth or finger. Robin's first computer was a Tandy TRS 80 Model 1 with a massive 4KB of RAM!

Robin eventually went on to work for some of the UK's top-selling IT magazine publishers, where he held several roles including editorial, promotions, and cover disc editing.

With the dawn of the Internet in the 1990s, Robin helped spearhead many new web developments such as the first Internet radio station, one of the first webmail services, the first fully interactive chat service (before the development of Ajax), and the first widespread use of pop-up window technology.