Getting Good with

PHP

Andrew Burgess
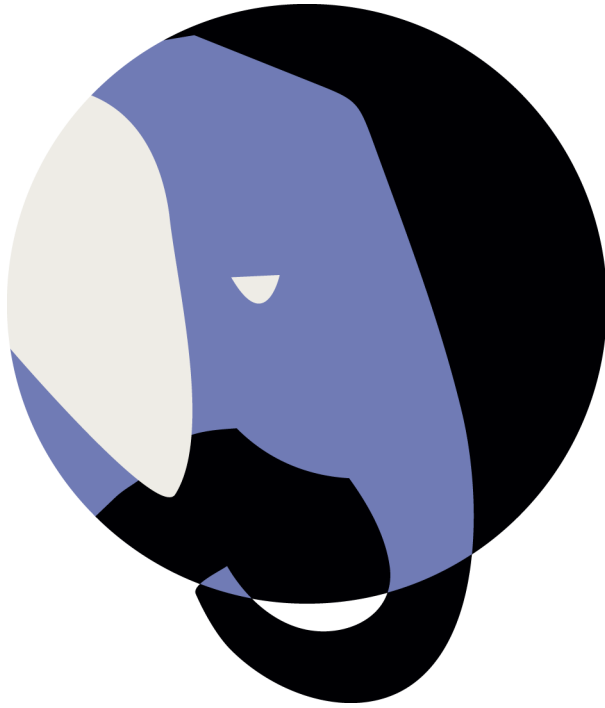
ROCKABLE ✳

Acknowledgement

There might be only one name on the cover, but no one writes a book alone. I am eternally grateful to the following parties:

- **God**, for orchestrating not just the making of this book, but my whole life, really.

- **Dad** and **Mom** — as well as my **siblings** and **grandparents** — for their support and encouragement throughout the entire process. They were, and always have been, willing springboards for ideas and invaluable advisors. There's no exaggeration in saying that without them, the book wouldn't exist.

- The whole **Envato** crew and contractors, but especially **Jeffrey**, for taking me on as a writer when I'd never done any tech- or tutorial-writing before and encouraging me (whether he knows it or not) to continually become a better writer and developer; **Naysan**, for suggesting I write this book, and for organizing the legalities and logistics so that I could focus on the writing; **Peter**, for being a meticulous editor, and reigning me in when I tried to do crazy things with the English language; and the **book design and layout folks**, who made all this tough-on-the-brain teaching so easy on the eyes.

- And finally, you, **the reader**, without whom the efforts of the aforementioned parties would be a complete waste.

Contents

Chapter 3   39

1

Chapter 1

It's more than fair to say that PHP is one of the mainstays of the Internet. It's been around for over a decade and a half, and in that time it's become the default first foray into the world of server-side coding for many. If you're attempting to make that move now, I hope this book will prove a worthy guide.

So, let's go! Please keep your hands in the book or on your keyboard at all times; eating and drinking is permitted, but no flash photography.

What is PHP?

Before we actually get started, I want to make sure you know what you're getting into. After all, it'd be a crying shame for you to read two-thirds of the book before realizing that PHP isn't what you wanted to learn.

So, what is PHP? First off, the name PHP stands for "PHP: Hypertext Preprocessor." Ignoring the mind-bending recursive part (https://en.wikipedia.org/wiki/Recursive_acronym), this means that PHP is primarily used for preprocessing hypertext. You'll often intermix PHP with HTML; the HTML isn't processed until it gets to the browser, but the PHP is executed on the server, and its output (typically HTML or some other text) replaces the PHP code.

This tells us two things: firstly, PHP is a *server-side language*. None of your PHP ever hits the browser — it's processed on the server. The other thing that might not be entirely obvious if you've just worked with HTML and CSS previously is that PHP is a *programming language*. It's not like HTML and CSS at all: when you're writing PHP, you're writing real code that will perform some task, usually based on some input or variable conditions. Of course, this could be just outputting some text or HTML, but often it's more.

Who is this Book For?

There's no way that a single book could meet every single PHP programmer wannabe where they are and help them learn the ropes. And this book doesn't need to do that, since there are plenty of other books, websites, and tutorials that are top-notch. Here's who I imagine the audience of this book to be: it's the designer who wants to learn PHP so that he or she can use some of the great PHP-based content management systems in their client work. It's the front-end developer who's good with HTML, CSS, jQuery, and maybe some raw JavaScript, and wants to start building more dynamic websites from scratch. If you're someone who understands the front-end of the web pretty well, but you wouldn't really call yourself a "programmer," then this book will, I hope, be helpful to you.

So, yes, I'm aiming for beginners, but I'm also aiming for short: this book is meant to be read in a weekend (okay, maybe a long weekend). This means that there's plenty of PHP goodness that I just don't have room to address. To make amends for this, I've included two appendices. Appendix A is a list of topics that we didn't discuss: it's a good list of things to check out. Appendix B is a list of resources to check out: blogs, books, and more.

Why Learn PHP?

Just in case you're still on the fence about learning PHP at all, let's take a minute to talk about what you can do with it. The problem here is that asking what can be done with PHP is like asking what can be done with a paintbrush. My little sister can mess around with one and do something pretty creative. But give one to Van Gogh or Picasso, and, well, that's a completely different story.

It's the same with PHP. After reading this book, you should be able to do some basic, yet really handy things that will improve your websites. However, don't forget that there are very popular libraries

and frameworks that use PHP. Some of the biggest websites you've ever visited are coded in PHP; ever heard of Facebook?

So, what will you be able to do with PHP? Check this out:

- You'll be able to change values on your site based on user input or other values (e.g. change the greeting based on the time of day.)

- You'll be able to use the information that a user enters into a form, maybe by giving them appropriate content based on that info (think search results) or by storing that information within a database.

- You'll be able to let your users upload files to your server.

- You'll be able to build pages "on the fly" by combining templates with content from a database, all right as the viewer requests that specific page.

If any of these things sound enticing, good! And if they don't, maybe they've made you think of something else that you've wanted to do with your websites. Whatever your aspirations may be, there's a pretty good chance you can achieve them with PHP.

One thing to note: PHP is a regular programming language, and as such, it's capable of more than just adding some punch to your website. You could use it to write scripts and programs that have nothing to do with the web and servers. This isn't overly common, but it can be done. However, the plan here is to stick to PHP in the context of the web, deal?

How Do You Install PHP?

Still with me? Good. So, you've decided that you really do want to learn PHP? Well, then, we'd better get it installed. Since PHP is a server-side language, and the language doesn't execute in your browser, you need to install the PHP interpreter on your local machine if you want to develop in PHP. While it's a pretty similar

process on both Windows and Mac OS X, I'll walk you through both. In both cases, there are great packages that bring all the necessary pieces to the game and make it incredibly easy to start playing.

Installing PHP on Windows

On a Windows computer, the best way to get PHP onto your system is by installing WAMP; besides PHP, this package has Apache2 for a web server and MySQL for databases. You probably aren't familiar with these technologies but don't worry; we'll explore them later.

We'll begin by heading over to the WAMP Homepage (http://wampserver. com/en).

ROCK* TIP

You've probably figured out what WAMP stands for, right: **W***indows,* **A***pache2,* **M***ySQL and* **P***HP.*

!

Click *Start Experimenting WAMPServer.* This will bring you to the downloads section of the page. Choose the correct download, depending on whether you're running a 32- or 64-bit rig. A form will

pop up, but you don't have to fill it out, just click the link "*download it directly.*" Once it's downloaded, run that puppy.

It's a pretty normal installation process; the only parts that might trip you up are the last few steps. The installer will ask you to choose your default browser; just browse to the right `.exe` file and hit *Open*. Then, it will ask you to set some PHP mail parameters; just leave the defaults. After you're finished, WAMP Server should launch automatically (if you haven't unchecked that box). Henceforth, you'll find a *Start WAMP Server* item in your programs menu. Once you choose that, you'll see an icon in your task bar:

If you click on that icon, you'll get a menu that looks like this:
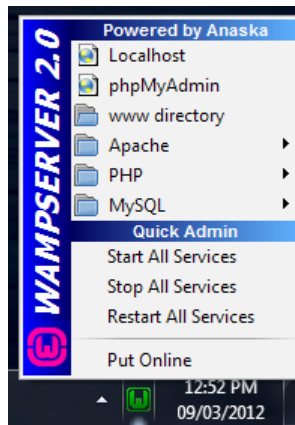
See that *Put Online* option? Click that. In a second or so, your servers will be online. Then, click the *Localhost* option, at the top of that menu. This will open your browser to the WAMP start page.

To use WAMP, you'll have to put your PHP code in the right place. That right place is the `www` folder, which you'll find at `C:\Program Files\WAMP\www` (you've got a shortcut to the `www` folder in the
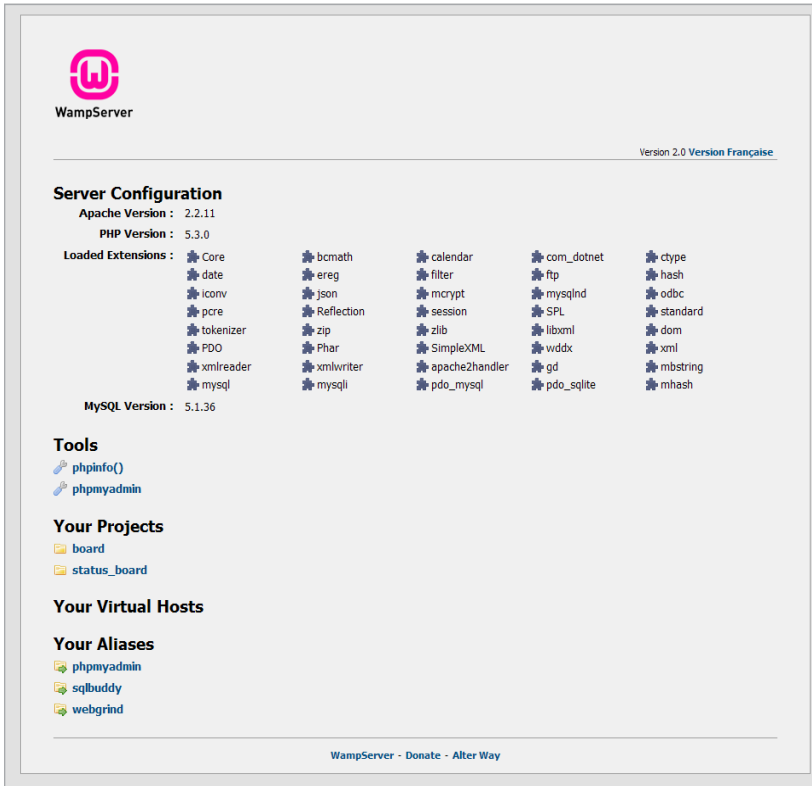
The WAMP start page.

WAMP task bar menu). Any folders that you make in that directory will show up as projects on the WAMP start page. Or, you can just send your browser to `http://localhost/YOUR_FOLDER_NAME` to see your work.

Installing PHP on Mac OS X

If you're running Mac OS X, using MAMP is the best way to get up and running with PHP. Load the MAMP homepage (http://www.mamp.info/) and click the *Download Now* button under the grey MAMP logo (you won't need MAMP Pro today).

Once the rather large zip file has finished downloading, open it up and run the `MAMP.pkg`. Walk through that installer; there shouldn't be any surprises. Once you're done, launch the MAMP app found at `/Applications/MAMP/MAMP.app` within your applications folder. You'll get a window that looks like this:

ROCK★
TIP

*Yes, that's right: MAMP stands for **M**ac, **A**pache2, **M**ySQL and **P**HP.*
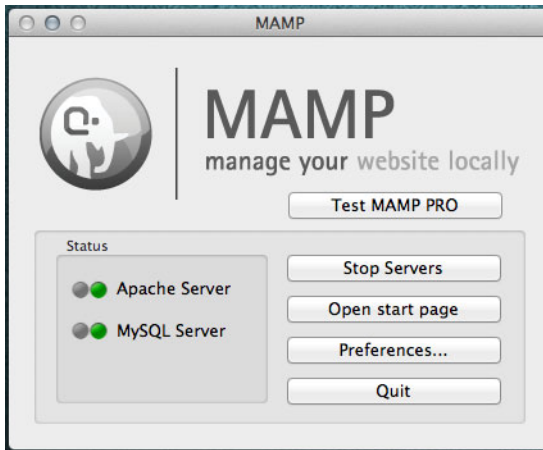
The servers will start up immediately, and a "Welcome to MAMP" page should open. That's it! You've installed MAMP and have a working version of PHP on your Mac.

You might want to check out the MAMP start page; you can bring it up in your default browser by clicking the *Open start page* button.

This start page has some information about your PHP installation and some of the other components installed with MAMP. Later, in Chapter 4, we'll come back here briefly when we look at databases.

To actually use your MAMP installation, you'll have to put your projects in the `htdocs` folder. This resides at `/Applications/MAMP/htdocs/`. A folder here named `my_project` could then be viewed at `http://localhost:8888/my_project`.

There's one more thing to do: MAMP has the option to display PHP errors turned off by default. Let's turn that on. Open up the file `/Applications/MAMP/bin/php/php5.3.6/conf/php.ini` and find the line that says this:

```
display_errors = Off
```

Change that to this:

```
display_errors = On
```

If you have MAMP running, you'll have to stop the servers and restart them. Now, you'll see any PHP error messages that come up from typos, misspelled PHP statements, and other things along the way. We'll discuss PHP errors at the top of Chapter 5.

PHP on Your Server

Obviously, having PHP on your local system only helps you in development. Once you are ready to deploy your website, you'll need PHP on your server. The good news here is that most of the web hosting services you'll find already have PHP installed for you. For the most part, you'll just have to FTP your freshly made PHP files up to that server and they'll work just fine. Be aware that some complications could arise; we'll tackle them right at the end of the book.

The Example Files

You'll notice many diverse and sundry code examples herein. You can get your hands on those examples: they're in the package you downloaded when you bought this book.

Once you've got the examples folder, you'll have to put them in a folder in the `www` (for Windows) or `htdocs` (for Mac) folder. Then, load up `localhost/examples/index.php`. From there, you'll find links to add the examples, with the headings for the examples in the book matching those link names. Note that if a code snippet doesn't have a heading like the following example, it's not important enough to have an example page:

Example 1-1

```
echo "The code will go here";
```

Note that some code snippets are broken into multiple chunks; those ones are have a letter in the title, like so:

Example 1-2-a

```
echo "part 1";
```

Example 1-2-b

```
echo "part 2";
```

This is only done for the more complex snippets, which I will explain piece by piece. Finally, some examples are actually multiple files, so they have a full word instead of just a letter.

Example 1-3 file_one

```
echo "file one";
```

Example 1-3 file_two

```
echo "file two";
```

Summary

In this chapter we've looked at what PHP is and why you should be excited about learning it. We've also discussed installing it on your machine. Now, don't you dare move on to Chapter 2 until you're ready to start learning PHP.

2

Chapter 2

So, you're ready to start writing some real code, are you? No reason to waste another moment; let's begin!

PHP Files

We'll start with the most basic of things: obviously, PHP is text, and it will therefore be stored in regular text files. These files usually have a `.php` file extension; while there are other extensions that PHP uses, that's the most common, and the only one that you'll use for a long time.

You'll probably recall that I mentioned in Chapter 1 that we could add bits of PHP to an HTML file. That's the truth; in fact, you can even add bits of PHP to CSS or JavaScript files. Part of the trick here is to give the file that `.php` extension; that's how the PHP interpreter knows which file to process before sending it to the browser. But, how does it know what lines of code to parse?

That's where the PHP opening and closing tags come in. You just wrap your lines of PHP in <`?php` and `?`>. The interpreter will process only those parts of the text as PHP, replacing them with the output of your code. Here's a quick example. Open up your text editor of choice and put in this text:

Example 2-1

```
<h1> Hello PHP! </h1>
<?php
  echo "<p>I'm getting good with PHP.</p>";
?>
```

Don't worry too much about what exactly this does; for now, just know that the **echo** command will print out whatever text we pass to it. Now, remember how we said that all of your PHP projects will be folders in that **www / htdocs** folder that

WAMP/MAMP created? Well, create a folder named `php_book` in there, and save this text file as `index.php`. Then, load up `http://localhost:8888/php_book/` in your browser (if you're on a PC with WAMP, you don't need the port number: just `localhost/php_book` should do it). You should see something like this:

Hello PHP!

I'm getting good with PHP.

(Of course, you could just load up *example 2-1* from the example files.)

As you can see, the HTML portion of our file loaded normally. Then, the PHP portion is interpreted and the correct output is, well, put out. If you view the page source, you'll see that it's plain HTML, as if the PHP was never there. Note that the line-breaks aren't essential; we could have just as easily put `<?php echo "<p>I'm getting good with PHP.</p>"; ?>` on a single line.

So, that's how you can mix together some HTML and PHP. Just note that from now on, I'll not show the PHP tags in our code examples unless we're mixing it with HTML.

Variables

Now that you have some idea of how you'd work with a PHP file, let's actually start learning the language. We're going to start with variables. Now, I'm not expecting you to know much about programming, so we're starting from the very beginning. You can think of a variable as a storage place for a value. You aren't familiar with values yet, but for now, think of them as pieces of data, like some text or a number. In our little example above, the text we output into our page was a value. Let's use that text as an example.

So, a variable is a storage place; it's something you can come back to, to get the same value again. Here's an example:

```
$message = "<p>I'm getting good with PHP.</p>";
```

We have the name of the variable first: it begins with a dollar sign and can be followed by any letters, numbers, or underscores (as long as the first character after the dollar sign isn't a number). That's followed by an equal sign, and then the value we want to store in the variable. We end the line with a semicolon.

> **ROCK***
> **TIP**
>
> *Most lines of PHP end with a semicolon. I say "most" because it isn't actually lines that end with a semicolon, but statements. That's harder to explain at your level, so for now, just know that if the line doesn't end with a curly brace, it should probably end with a semicolon.*

Now, anywhere you'd like to use the text of the message, you can just use the variable `$message`. For example:

Example 2-2

```
$message = "<p>I'm getting good with PHP.</p>";
echo $message;
```

If variables aren't making sense to you yet, don't worry. As we go on, you'll see more about how they are used and where they are useful. But now, let's talk about the values that get stored in those variables.

Values

The real name for values is *types*. As in, different types of data (sometimes, they're called *datatypes*). While you can make your own datatypes in PHP, there are a bunch of basic ones built in, and we're going to get acquainted with them now.

Strings

You remember the text that we used in our previous examples? Well, that was a string. A string is any characters between two quotes. For example,

```
"this is a string"
```

You'll notice that that string is delimited by double quotes. You can also delimit strings with single quotes:

```
'this is also a string'
```

Now, let's say we wanted to do something radical, like put a single quote inside a string that's delimited with single quotes, or a double quote inside a string delimited with double quotes. Pretty radical, eh? A little thinking might help you come up with the right answer: just use the other type of quote around the string content:

```
"That's how you do it. 'Gotcha,' you reply."
```

That will work… some of the time. But, there will be times you can't switch the type of quote on the outside (more on that in a second). When that's the case, you'll have to escape the quote inside of the string. To escape a character means to preface it with a backslash. This is done in pretty much every programming language, and it lets the interpreter know that the character following the backslash shouldn't be processed the way it normally would.

```
'That\'ll do it.'
```

Normally, the above string would end when it comes to that second single quote. But that backslash keeps that from happening.

Now, you're probably curious about the times that you can't just switch the wrapping quotes to the other type. The secret here is that double and single quoted strings were not created equal. Double quoted strings have special magic powers; they can do interpolation.

Interpolation isn't actually all that magical, but it is powerful. Here's the idea: when using double quotes around a string, you can put variables right inside the string and the values of those variables will be put inside the string. See here:

Example 2-3

```
$name = "Sherlock";
echo "Hello, $name.";
```

If you try to run this code, you should see the output "Hello, Sherlock." That's string interpolation. If we had wrapped that string in single quotes, we'd have gotten "Hello, $name" instead. Note that it doesn't matter whether the string being interpolated (that is, the value "in" the variable) used double or single quotes.

Numbers

In many programming languages, numbers can be pretty confusing for beginners. This is because what we humans think of as "just numbers" are actually extremely complicated for computers. They divide them into categories depending on whether they are whole numbers, decimals, positive numbers, negative numbers, or numbers within certain ranges. It can be pretty tough to keep track of.

Thankfully, PHP makes it fairly simple for the beginner. There are only two types of numbers: integers (whole numbers) and floating point numbers (also called floats: they're fractional numbers, with digits after the decimal point). But, you don't even have to worry about those types, because PHP will convert back and forth when necessary. So, when you want to use a number, use it just like you normally would. Of course, you don't separate numbers every three digits with commas or space or anything (like 1,234,567 or 8 901); the only non-number characters you use is the decimal point, a minus sign (for negative numbers), or an **E** (or **e**) for scientific notation.

```
1001.234
```

```
-10
```

```
1.234e5
```

The last one is just a shorter way to write 123400. And that's all you need to know to get started with numbers.

Booleans

There isn't an easier type than the Boolean. This is because it only has two values: `true`, or `false`. Those words are *keywords* in PHP, which means you can't use them for anything other than a Boolean value. Oh, and they're not case-sensitive either, so `TRUE` and `FALSE`. One thing to know, however, is that every value in PHP evaluates to either true or false. More on this in the section below on logical operators.

Null

This one you might find confusing at first, but just store this away for later. This is the data type that represents nothing: the valueless value. It's simple `null`. That's a keyword too, and it's also case-insensitive. One place `null` pops up is when you create a variable, but don't give it a value: it defaults to the value `NULL`.

```
$will_be_null;
```

Array

Here's the last data type that we'll look at for now. It's the most complex type that we've looked at yet, and that's partly because it can be made up of the other variable types that we've looked at. See, in its most basic form, an array is just a list. Let's see how this works:

```
$an_array = array("HTML", "CSS", "JavaScript", "PHP");
```

Here we see an array of strings, four strings to be exact. Notice how we set up the array: we use the word `array`, followed by an open parenthesis. Then, we have the items in our list; each one is followed by a comma, except the final one. Then, we close our parentheses. What you don't know yet is that the `array()` part of this is actually a function call; just keep this on a side burner until we discuss functions in a couple of minutes.

So, that's one way to create an array. But, let's thicken the plot here. Many programming languages have two types of arrays: *numeric arrays* and *associative arrays*. See, there are two things to remember about arrays: they're ordered and they're indexed. *Ordered* means that the order of the items is important: so in our example above, `HTML` will always be the first item in that array, and `PHP` will always be the last. *Indexed* means that each item in the array has a number or string that allows us to get to that value in the array. The difference here is that numeric arrays use numbers to retrieve values, and associative arrays use strings.

By default, arrays use numeric indices; also by default, the indices start at 0, **not** 1. So, based on that array up there, `$an_array[0]` holds the value `HTML`, and `$an_array[2]` is `JavaScript`. That's the notation for getting an item out of the array. After the variable name, square brackets encase the index number. You could almost think of the bracketed number as a "sub-variable."

Oh, I mentioned associative arrays, didn't I? Well, associative arrays use strings as their indices. Of course, you have to define a string for each value you put in the array:

```
$person = array(
  "name" => "Sherlock Holmes",
  "birthdate" => "January 6, 1854",
  "married" => false,
  "interests" => array("reading", "chemistry", "crime",
    "violin")
);
```

Instead of just a single value between commas, we have a bit more. It follows the pattern `key => value`. Those are important terms, too: when you use a string as the index, it's called the key. You'll find the key-value pair to be something you'll run into often in programming languages. Between the key and the value is a =>, known as a `T_DOUBLE_ARROW` (what? No, I think that's a great name *rolls eyes*). And, you can get array items here just like with the numerically indexed arrays: so `$person["name"]` will be `"Sherlock Holmes"`.

In the case of both numeric and associative arrays, you can use that square-bracket syntax not just to retrieve values, but to assign values as well:

```
$an_array[4] = "SQL";
$person["best friend"] = "John Watson";
```

I'll note one more thing about arrays before we leave them for a while. You might have picked this up already, but PHP arrays are not confined to one data type per array. You can mix and match, just like we did with `$person`: we have two string values, a Boolean value, and another array. Mixing data types within arrays is perfectly legal and legitimate.

Those aren't the only data types that PHP has, but they're the basic ones, and they're all you need to know for now. Let's change directions and look at something a bit different next.

Comments

Every modern programming language that I've ever heard of lets the programmer write notes within the code for later reference (or for others who might be reading the code). That's called commenting. There are several ways to define a comment in PHP. For single line comments, begin the line with a number sign and write your commend after that. Alternatively, you can start the line with two back-slashes, like you might do in JavaScript.

```
# this is a comment
```

```
// this is also a comment
```

For a multi-line comment, begin with a slash and then an asterisk. To close the comment, reverse that. See here:

```
/* this is
a multiline
comment */
```

In both cases, you don't have to worry about starting the comment at the beginning of the line; you can actually have some code, and then start the comment after that.

```
$name = "Sherlock"; # Holmes, of course. Who else?
```

Of course, these are bad examples of comments. Good comments document a feature, remind you to make a change, or explain some quirky code. Use them sparingly, but use them well.

Operators

Now, let's move on to operators. You couldn't have a programming language without operators; they're the glue that brings all of your variables and values together into one cohesive living thing. As you might guess, operators help us operate on values. Sound abstract? Don't worry; it's very straightforward.

Arithmetic Operators

If you're old enough to have been through grade 1 or 2 (hey, I welcome all ages), this will be old hat to you. The arithmetic operators perform arithmetic on your numbers. It couldn't be simpler. Check this out.

Example 2-4

```
$num = 10;
$num = $num + 10; # Addition
$num = $num - 5; # Subtraction
$num = $num / 2; # Division
$num = $num * 0.2; # Multiplication
echo $num; # outputs 1.5
```

Above are four most common arithmetic operators, which I expect you'll be very familiar with. As you can see, we can take any two numbers and use the operators to work on them. In this case, we're redefining the value of the variable $num every time; there's no rule against changing the value of a variable.

There's one more arithmetic operator: modulus (%). It's a little trickier; it returns the remainder of the first number when it is divided by the second number. So,

```
5 % 3; # 2
```

The modulus operator is a great way to find out if a number is odd or even. If $num % 2 returns 0, it's even; if it returns 1, it's odd.

The String Operator

There's only one operator for strings, and that's the concatenation operator. The symbol is . (a period or full-stop). When you want to concatenate more than one string, here's how:

> **ROCK***
> **TIP**
>
> *You'd better get familiar with the term "returning." If some piece of code* **returns** *a value, that value is the result of that execution, the value you get back. So, 2 + 2 "returns" 4.*
>
> **!**

```
"first string " . "second string";
```

This will give you (or "return") **"first string second string"**. This is handy in edge cases where you can't exactly use string interpolation. Oh, and if you work with other languages like

JavaScript, be sure not to accidentally use + to concatenate strings; that doesn't work in PHP.

Assignment Operators

You're already familiar with the first assignment operator; that's = (equal sign). This operator simply assigns the value on the right to the variable (or array position) on the left.

But wait, there's more. The arithmetic operators and the string operator have combined assignment versions. Remember our example for the arithmetic operators? Here it is again with the combined assignment operators instead:

Example 2-5

```
$num = 10;
$num += 10; # Addition
$num -= 5; # Subtraction
$num /= 2; # Division
$num *= 0.2; # Multiplication
echo $num; # outputs 1.5
```

Notice what we do: **$num** = **$num + 10** becomes **$num += 10**. These operators are useful when you want to perform an operation on a variable and reassign the new value to that variable. They just reduce the amount of typing you have to do.

Incrementing / Decrementing Operators

Okay, these are the last arithmetic operators. By now, this line is easy for you:

```
$num = $num + 1;
```

We're just adding **1** to the value of **$num**. But with what you just learned, you can improve that line:

```
$num += 1;
```

Pretty smooth, eh? Well, excuse me while I knock your socks off, because there's an even shorter way to do that:

```
$num++;
```

That's right; the incrementing operator (++) adds 1 to a number, updating the actual value of that variable. There's the decrementing operator too: **$num--**.

But there's something to be aware of; as you might have guessed, every operator returns a value. So, let's say **$num = 5**. Here's the curious part: **$num++** sets **$num** to 6; however, it returns 5. This is called the post-increment operator, because it performs the incrementing after it returns. However, there's the pre-increment version that does the incrementing first, and then returns. It's as simple as putting the double plus signs in front of the variable. When we do **++$num**, it returns 6, and sets **$num** to 6. Of course, all this goes for the decrement operator as well.

Comparison Operators

Next up is a set of operators that are useful whenever you want to compare two values. All the comparison operators return a Boolean: either **true** or **false**. If after reading this section you're confused about where these would be useful, just wait. They're mainly used with control structures, which we'll look at in the next chapter; once you understand those, comparison operators will make a lot more sense.

First, we have two different operators for checking if two values are equal. (Why two different operators? You'll see.) There are double-equals (==) and triple-equals (===). Let's begin with an example:

```
1 == 1; # true
"can" == "cannot"; # false
10 === 20; # false
"something" === "something"; # true
```

As you can see, we can compare pretty much any type of value. But what's the difference between double-equals and triple-equals? Well, double-equals (also called the equal operator) tries to convert both values to the same type before comparing. Because of that, `1 == "1"` is true, even though the "1" on the right — because it's enclosed in quotation marks — is technically a string. The opposite value — which is not enclosed in quotation marks — is a number; PHP realized that we can convert that string to a number and still have it make sense (which would not be possible with a string like **"one"**). Triple-equals (known as the identity operator) doesn't try to interpret and convert values into matching data types: `1 === "1"` will always be false. Which one you want to use depends on the situation, but you'll probably be looking for triple-equals most often.

There's also the reverse of the equal and identical operators: the not equal and not identical operators. They work exactly as the others do, but in reverse:

```
10 != "10"; # false, because it converts the string to a
  number
10 !== "10"; # true
```

However, there's more than just equality to test. How about greater-than and less-than operators?

```
5 > 10; # false
5 < 10; # true
```

Two left: greater-than-or-equal-to and less-than-or-equal-to.

```
5 >= 10; # true
4 <= 4; # true
```

The difference with these is that if the operands are equal, it returns true. As in the last example, 4 is not less than 4, but it is equal to 4, making the statement true.

Logical Operators

There's another type of operator to discuss here, and that's the logical operators. This might be somewhat confusing at first, but you should know them. They're most useful with conditional statements, so once you learn about those in the next chapter, this part should become much more clear. Before, you meet 'em, remember two things: first, as we've seen, many pieces of PHP code return a value. These pieces are called expressions. Second, every value can be interpreted as being either true or false. Most values are true: the false ones are these:

- `""` (empty string)

- `0`

- `false` (of course)

- `null`

- `array()` (an empty array)

Now, let's learn three logical operators. The logical operators always return a Boolean value (you might know that some of JavaScript's logical operators don't work this way).

The first one is pretty simple: it's the *Not* operator, and it simply returns the opposite Boolean value of whatever value you use it on. An example clarifies:

```
$name = "Sherlock";
! $name # false
$married = false;
!$married; # true
```

As you can see, you just preface the value with an exclamation point (or "bang"), and it reverses the value's Boolean value. What's this useful for? When you learn about conditional statement's next chapter, you'll find out that you might want to do something "if a value is true" or "if a value is not true." The *Not* operator is useful

in simplifying the wording of your code by reserving the Boolean value.

Then, there's the *And* operator. You use two values with the *And* operator, one before it and one after it. The *And* operator is useful in conditional statements when you want to make sure that two things are true: It will only return `true` if both operands (the values you use with it) are true. For example:

```
$name = "Sherlock";
$detective = true;
$married = false;
$name === "Sherlock" && $detective # true
$married && $name === "Sherlock" # false
```

Pretty simple, isn't it? If **both** statements are true, it returns `true`, otherwise it returns `false`. Note also that we can use our other comparison operators as one of the operands: that side of the operator just equates to whatever Boolean value would come out of the comparison.

The last logical operator, the *Or* operator, works very much in the same way as the *And* operator: however, it returns `true` if only one of the operands is true. So:

```
$name = "Sherlock";
$criminal = false;
$married = false;
$name === "Sherlock" || $criminal # true
$criminal || $married # false
```

There's an important thing to note about the *And* and *Or* operators: they don't work any harder than they have to. This means that if the first operand decides the whole case, the second one isn't evaluated. For example, if the first operand in a use of the *And* operator is false, then there's no sense in evaluating the second one, because the *And* operator requires both to be true. Similarly, if the first operand in a use of the *Or* operator is true, there's no need to

go on to the second one. It's important to remember this if you're using functions as operands. You'll learn in the next section that functions return values similarly to operators: keep in mind that if the function call is the second operand, it won't be called unless it needs to be.

Conditional Operator

The last operator we'll look at is the conditional operator. It's also called the *ternary* operator, because it's the only operator that takes three operands. Here's how it works.

```
conditional ? if_true : if_false;
```

We start with a conditional statement, followed by a question mark. Then, we have another expression that is executed if the condition is true. After a colon, there's a second expression that executes only if the condition is false. This is useful if you want to set a variable based on a condition. See here:

```
$message = $logged_in ? "Welcome, $username" : "Please
    log in.";
```

Note that sometimes people will wrap the conditional statement in parentheses, just to keep their code cleaner.

Functions

We're finally at the last section of this chapter. Let's discuss functions. Sounds ominous, I know, but you'll find otherwise, I hope.

First, of course, you'll want to know what a PHP function is. And so I introduce to you my *Two Paradigms of Functions*, which I find useful when explaining functions. First, think of a function as simply a bunch of lines of code, wrapped into a single entity. Often, you'll want to perform the same action or task several times. Do you want to write the lines of code that perform that action again and again? No. So, you write them once and put them in a

function. Then, you can call that function whenever you want to do that thing.

The *Second Paradigm of Functions* is that a function is like a black box: you'll put values into it and get a value out of it, but you can't really control what happens inside. Of course, you can control what's inside the function when you write the function, but I'm talking about when you use it. You'll hand values to the function (or sometimes, hand it nothing), and get a value back.

So, what's the syntax for writing one of these functions?

```
function say_hi ($name) {
  return "Hello, $name";
}
```

We start with the keyword `function`, followed by the name we want to give our function. Functions follow the same naming rules as variables: use letters, numbers, and underscores. After that, we have a set of parentheses. What's between those parentheses is important; that's the entrance for any values that you want to pass into the function. These are called parameters: in the example above, we have one parameter, `$name`. Notice that it's defined just like a variable. We can then use those values inside the function. *A terminology tip:* using a value or variable as a function parameter is sometimes termed as *passing* the value as a parameter.

After the parentheses, there's a set of curly braces. Between those braces are the lines of code that make up the body of the function. Notice we're using the word `return`; whatever value comes after that keyword is what is returned from the function, the output of the black box.

Where are functions useful? They'll be useful when you need to perform the same action repeatedly. Depending on the way you're organizing your code, functions are often used to break huge chunks of code into smaller, more manageable more **reusable**

pieces that each do a single job. You'll see functions in use as we go on.

Code Style

Let's close out this chapter with a few comments on style. Of course, we could write a book on PHP code style, but I only want to mention two things. First, we should discuss the naming of variables and functions. If you're familiar with JavaScript, you've probably named your variables and functions with camelCase: `messagesSent` and `sayHello`. However, most PHP developers use underscores: `messages_sent` and `say_hello`. Of course, there are no real rules on how you name your variables and functions. Just be aware that underscores are, in general, the way it's done in PHP.

The other thing involves the placement of the curly braces in functions (this goes for control structure too, which we'll discuss in the next chapter). In the example above, I put the opening curly brace on the same line I started the function on.

```
function fn () {
  return "fn";
}
```

Some PHP developers put that opening curly brace on the next line, like so:

```
function fn ()
{
  return "fn";
}
```

I don't think either one is clearly used more. Just be aware that, for the most part, the whitespace between these characters and statements doesn't affect how they work.

Summary

It's been a long chapter, eh? I've thrown a lot at you in these few pages, so don't feel bad if you need to re-read this chapter. Make sure you have a good handle on the stuff we've covered here, because, sister, it ain't getting any easier.

3

Chapter 3

By this time, you should be getting pretty comfy with PHP; so, now it's time to layer on the complexity. Not only are we going to be learning some of the most important parts of the language, we're going to be using examples that will begin to get you familiar with seeing more than one line of PHP at a time.

Control Structures

Apart from the basic language concepts that we've already covered, control structures really are the bread and butter of any language. Here's why: it's a rare case indeed when you're only required to perform a single, simple action using PHP. More often, you'll either want to follow a carefully-designed process over and over (maybe to a list of items or values), or you'll want to do something different depending on another value (maybe some user input). It's in these cases that you turn to control structures. Let's meet 'em.

`if` and `else` (and `elseif`)

The most basic control structure is the if-statement. It allows us to find out if a certain condition is true before we execute some code. For example, let's say I have only one friend (yes, on the whole planet); we'll call him Watson. I could use this code to determine what to say when I meet a person on the street:

Example 3-1

```
$person = array("name" => "Watson");
if ($person["name"] === "Watson") {
  echo "Hello, Watson!";
}
```

If the person I meet has the name of "Watson", I will respond with "Hello, Watson!" If you run the above code, you should get the

right output. The syntax goes like this: the `if` keyword comes first, then a set of parentheses. Inside the parentheses, we have the condition; this is where the conditional operators we talked about last chapter come in to play. In this case, we're using the equality operator to check to see if `$person["name"]` is equal to the string "Watson". The final part is the curly braces. It's kind of like a function in the sense that all of the code within those braces executes, as long as the preceding control structure returns "true."

Hey, but what happens if I meet someone who isn't my old pal Watson? I don't want to just ignore them; in fact, since I've only got one friend, I probably want to try to meet some folks. This is where the `else` keyword comes in handy:

Example 3-2

```php
$person = array("name" => "Mycroft");

if ($person["name"] === "Watson") {
  echo "Hello, Watson!";
} else {
  echo "Hi, I'm Andrew! What's your name?";
}
```

This time, I'm meeting Mycroft, not Watson. So the condition in our if-statement won't be true; it will be false. Because it's false, the code in those curly braces won't execute. This time, however, we've added an `else` statement. The else-statement executes only if the condition is false, only if the if-statement code block doesn't execute. Run that code and you'll get "Hi, I'm Andrew! What's your name?"

Now, let's say I get to know Mycroft; I've just doubled my friends! But we all have some friends that, you know, you talk to a bit differently. Well, Mycroft is one of those friends. Ponder this one:

Example 3-3

```
$person = array("name" => "Mycroft");

if ($person["name"] === "Watson") {
  echo "Hello, Watson!";
} elseif ($person["name"] === "Mycroft") {
  echo "Hey, Mycroft! How are ya?";
} else {
  echo "Hi, I'm Andrew! What's your name?";
}
```

Now, I've got a custom greeting for each of my friends. Here's what we've done: we've added an `elseif` statement between our `if` statement and our `else` statement. We can have as many of these as we want; after the keyword `elseif` (oh, that can also be two words: `else if`), we have another conditional statement in parentheses, just like with the `if` statement. And then, of course, there's the code block, which gets executed if the condition passes.

That's the `if` statement and his buddies for you. There is one more thing to remember as we go on with these control; there's no reason you can't put an `if` statement inside of an `if` statement, or place any of the other control structures within **other** control structures. Let's look at some more control structures that we can combine into nested, dynamic programming statements.

`for` / `foreach`

Well, I've been getting out more, recently, and my list of friends has grown:

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");
```

Let's say I want to list all four of my friends. This is where our next control structure comes in handy: `for` loops. The name makes sense: you loop over the same code again and again, once for each item in a list. Let's look at an example:

Example 3-4

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");

echo "<ul>";

for ($i = 0; $i < 4; $i++) {
  echo "<li> $friends[$i] </li>";
}
echo "</ul>";
```

You should be able to cope with most of this code, but… woah. Woah is right. That **for** loop is quite a bit more complex than the **if** statements. You should understand the **for** keyword, and the block enclosed in curly braces: the code in there is what is executed for each item in our list. But what exactly is going on in those parentheses?

Well, to start, notice that there are three expressions in the parentheses; the first two have to have a semicolon at the end, so they're separated from the one ahead of them. Here's how these three expressions work: the first one is evaluated (or executed, run) before the looping begins. The second one is evaluated before each loop begins; if the value it returns is true (or evaluates to true) the loop is run. The third expression is run after each looping.

What's up with all these expressions? Well, their "standard" usage is what I've shown above: the first expression creates any variables we need for the loop. We'll need an iteration variable, to keep track of the index of item we're looping on. As you can see, we create a variable $i which starts with the value 0. Since arrays are zero-based, that's the index of the first item in the array. Before the loop executes, we evaluate the second expression: in this case $i < 4 is true, so the loop executes and we output a list item with the name of a friend: in this case, that's Watson, since he's at index 0. Finally, the last expression is executed, since the loop has finished its first execution: we increment $i by 1, so it now equals 1. Then, the second expression is evaluated to see if we should repeat the loop: $i < 4 is still true, so we loop again. This continues until the

second expression is false. In this example, that will be when `$i` equals 4.

That's your basic `for` loop. However, the `for` loop is an incredibly flexible construct; in fact, all those expressions are optional. You can check out the official documentation (http://php.net/manual/en/control-structures.for.php) for examples of how to use the for-loop in more complex ways.

Oh, one more thing; normally, you wouldn't hard-code the number in the second expression, as we did up there. You would usually use the `count` function which returns the number equal to the total items within the array. Let me demonstrate with the first part of a `for` loop.

```
for ($i = 0; $i < count($friends); $i++)
```

That's better, but let's take it one step further. That way, we have to call `count($friends)` before every loop. We should save the value in a variable and use that for our comparison:

```
for ($i = 0, $length = count($friends); $i < $length; $i++)
```

As you can see, we can make multiple variable declarations in one expression by separating them with a comma. We used `count` to determine the total number of items in the array, and then we set up a variable to keep track of how many times we have run the loop. Once we've run the loop for every single item in the array, the conditions return false, and the loop stops.

But, what about looping over associative arrays? You know, the ones with strings instead of numbers as indices. Since those don't have number indices, it doesn't make sense to use a normal for-loop. For these, there is another looping construct that works better: the foreach-loop. See here:

Example 3-5

```
$person = array(
   "name" => "Sherlock Holmes",
   "job"  => "Private Detective",
   "birthdate" => "January 6, 1854"
);
echo "<ul>";

foreach($person as $value) {
   echo "<li>$value</li>";
}
echo "</ul>";
```

Here's a similar snippet to the one we had with the for-loop. Instead of the **for** keyword, we use the **foreach** keyword. And, instead of the three statements inside the parentheses, we have something different: the array variable, the keyword **as**, and a new variable, which will be the value of an item in the array, one for each loop. Running that code should give you a list of the values of the array:

- Sherlock Holmes
- Private Detective
- January 6, 1854

However, what about the keys (indices) of that array? That could be important info, right? Well, the foreach-loop allows us to get at that too. Try replacing the loop above with this one:

Example 3-6

```
foreach($person as $key => $value) {
   echo "<li>His $key is $value.</li>";
}
```

We use the same syntax as you do inside an associative array: now, we have two variables to use inside the loop: **$key** and **$value**.

There's only one more thing to mention about `foreach` loops. That's this: you can use them for regular numeric arrays as well, both with and without the keys.

return/break/continue

Next, we've got a few keywords that shake things up a lot. First is `return`, which we've already met back when discussing functions. To formalize this introduction, I'll tell you that the `return` statement ends whatever function it is part of; there's no point in putting any code after `return`, because it won't be run. Whatever value you pass to the `return` statement is the value that will be returned to wherever the function was called. For example,

Example 3-7

```
function make_greeting ($name) {
  return "Hello, $name";
}

$greeting = make_greeting("Watson");
echo $greeting;
```

The above code will output "Hello, Watson", because that's what was returned from the function.

The next two movers-and-shakers are pretty similar; and they're both used inside looping structures. First, let's meet `continue`. This keyword skips the rest of the loop that it's a part of, and it starts the next iteration. For example, if I want to print out a list of my friends' names, I could do the following:

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");

foreach($friends as $friend) {
  echo "$friend, ";
}
```

That's second nature to you now, right? But let's say Mycroft and I have had a little, oh, a "falling out," shall we call it? Well, here's what I can do:

Example 3-8

```
foreach($friends as $friend) {
  if ($friend === "Mycroft") {
    continue;
  }
  echo "$friend, ";
}
```

Here, if **$friend** equal "Mycroft", then we **continue**. This means we skip the rest of this loop and start the next iteration. This will output: "Watson, Sherlock, Lestrade,": that's **continue** at work.

Then there's **break**. Sounds more destructive, and I guess you could say it is. That's because **break** acts just like **continue**, but ends the **entire** loop, not just one cycle. For example, let's change **continue** in our above example to **break**:

Example 3-9

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");

foreach($friends as $friend) {
  if ($friend == "Mycroft") {
    break;
  }
  echo "$friend, ";
}
```

Now, the output is just "Watson,"; **break** completely ends the **foreach** loop.

As mentioned above, both **continue** and **break** work with all looping constructs.

switch

Remember back to my list of friends? And how I had a different greeting for Watson than for Mycroft? Well, as you know, I've gotten a few more friends since then, and I have a custom greeting for each of them. So, what's the best way to give them each their own custom greeting?

With what we've learned so far, that'd be the if-statement, right?

Example 3-10

```
$friend = "Watson";
if ($friend === "Watson") {
  echo "Hello, Watson.";
} elseif ($friend === "Mycroft") {
  echo "'sup, Mycroft?";
} elseif ($friend === "Sherlock") {
  echo "Good day, Sherlock!";
} elseif ($friend === "Lestrade") {
  echo "How are you, Lestrade?";
} else {
echo "Hi, I'm Andrew! What's your name?";
}
```

That works fine... but don't you think it's a bit wordy? Notice two things about this code: first, there's a lot of **elseif** statements. Second, all the conditional statements (within the parentheses) are pretty similar: they all compare the variable to another string. There's actually a special conditional construct for situations like this, and it's called the **switch** statement. Check this out:

Example 3-11

```
$friend = "Watson";

switch ($friend) {
  case "Watson":
    echo "Hello, Watson.";
```

```
      break;
    case "Mycroft":
      echo "'sup, Mycroft?";
      break;
    case "Sherlock":
      echo "Good day, Sherlock!";
      break;
    case "Lestrade":
      echo "How are you, Lestrade?";
      break;
    default:
      echo "Hi, I'm Andrew! What's your name?";
  }
```

Checked it out? Now let's discuss it: We start with the `switch` keyword, and the value in question within the parentheses. Then, inside the curly braces, we have a bunch of cases. Each case works like this: the `case` keyword, the value to compare the first value to, then a colon. After that, we have the lines that will run if the comparison matches. You'll notice that we end each code block with `break`. This is because the `switch` statement is somewhat deceiving by default. You might think that only the code "under" a given case is executed when the case matches. However, that's not the case (pun unintended). When a case is matched, the code from there **to the end** of the `switch` statement is executed. In our above example, if we removed the `break` statements, all the greetings would be printed, because `$friend` matches the first case. Go on, try it. By including a `break` for each `case`, then we can keep this from happening.

One more point of interest: notice the last case: `default`. This case will run no matter what (assuming the `switch` statement isn't broken before it hits `default`). So, if no other case matches, the code under `default`, at the very least, will be executed.

Final thought: the `switch` statement works with any value, not just strings. Of course, it only works when comparing equality (it's the equivalent of the equality operator, ==).

require / include / require_once / include_once

Next up are a posse of players that are of paramount importance. Without these guys, writing PHP would be pretty messy. That's because these guys allow us to separate our PHP into multiple files, keeping it strictly organized.

Try this: let's put our list of friends in a separate file, `friends.php`:

`friends.php`

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");
$friend = array_rand($friends);
```

The second line just puts one of my friends in the variable `$friend`. That `array_rand` function just returns to us a random item from the array we pass it.

Now, in our `index.php` file, we'll do this:

Example 3-12

```
include "friends.php";

echo $friend; # or, you can put the switch statement from
  above here.
```

See that `include` statement? For our purposes, you can think of that as simply pulling the code from that file it mentions into the file that has the `include` statement. Run that example and it's just as though the code from the `friends.php` file was put in the example file itself: any variables and functions accessible in `friends.php` are now accessible from the example file.

This is what all four of these control structures do, and you use them in the same way, but they each have their own twist (kind of like my friends). Here's the list:

- `include`

- `require`

- `include_once`

- `require_once`

The main difference lies in what happens when the file can't be found. If `include` can't find the file, it will output a warning, but go on executing the code after the `include` statement. If `require` can't find the file, it will issue a fatal error and not execute any more code. Don't worry, we'll talk about warnings and errors in the last chapter.

What's up with `include_once` and `require_once`? These work just like their normal counterparts, except for one thing: if the file in question has already been `include`d or `require`d in that file, they won't bring it in again. So, if you execute the following example, you'll get three names: a friend from my list, Moriarty, and another friend from my list:

Example 3-13

```
include("friends.php");
echo "$friend <br />";
$friend = "Moriarty";
echo "$friend <br />";
include("friends.php");
echo $friend;
```

However, what do you think this will do?

Example 3-14

```
include_once("friends.php");
echo "$friend <br />";
$friend = "Moriarty";
echo "$friend <br />";
include_once("friends.php");
echo $friend;
```

If you run this one, you'll get Moriarty twice; this is because the second `include_once` will know that `friends.php` has already been included, and won't include it again, so the `$friend` variable isn't re-set again.

I should mention that you aren't restricted to pulling in files from the same directory. These structures actually check a few places when they're trying to locate the file, but for now, you only need to know that you can use regular file paths to show where the file is:

- `utilities/ui_messages.php`
- `../controllers/home.php`

Oh, by the way, you may see any of these used with parentheses when you're looking at code out in the wild:

- `include("friends.php");`
- `require("config.php");`
- `include_once("models/user.php");`
- `require_once("config/database.php");`

Because these aren't functions (they're language constructs), they don't require the parentheses, at least not for this simple, straight-forward use. But you might see them included, so just know that it's all good.

Final Thought on Control Structures

I've got two final interesting points about control structures. First, there's an alternative syntax that you might run into when looking at other programmers' control structures:

Example 3-15

```
$friend = "Watson";

if ($friend === "Watson"):
  echo "Hello, Watson.";
elseif ($friend === "Mycroft"):
  echo "'sup, Mycroft?";
else:
  echo "Hi!";
endif;
```

Notice that instead of curly braces delimiting the blocks, we have a colon at the start, and `endif` at the end. For the in-between blocks, they have no ending mark. This alternative syntax can be used for more than `if` statements: there's also `endfor`, `endforeach`, and `endswitch` to replace the ending curly brace of the other control structures. All of them use a colon instead of the opening curly brace.

The other interesting point is that you can use HTML right inside control structures. Remember how we can have snippets of HTML and PHP interspersed in one file? Get a load of this:

Example 3-16

```
<?php
$friend = "Mycroft";

if ($friend === "Watson"): ?>
  <p>Hello, Watson.</p>
<?php elseif ($friend === "Mycroft"): ?>
```

```
    <p>Hey, Mycroft! How are you?</p>
  <?php else: ?>
    <p>Hi!</p>
  <?php endif; ?>
```

Here, we start with PHP, switch to HTML, and continue going back and forth. Pretty nifty, eh, how we can go back and forth right within an `if` statement. Of course, this works for the other control structures, too.

Usually, you'll see these two things together: the alternate syntax and the interspersed HTML and PHP. I guess this is because it might be confusing to have `<?php } ?>` in your code, especially if you've got several nested control structures mixed in with HTML, where indenting can get a tad messy.

PHP Internal Functions

We've already talked about creating our own functions. But PHP has many internal, or built-in, functions: these are the ones that "come with" PHP. We've already seen two of 'em: `count` and `array_rand`. But there are many more: depending on how you slice it, there are hundreds, thousands, or maybe hundreds of thousands of functions that PHP offers. Of course, we can't look at them all in this book, and many of them you may never use. But, you can't write much PHP without knowing a bunch of these functions, so let's check out enough of them to give you a solid handle on basic PHP.

String Functions

Let's start with string functions; you'll be using strings a lot, so these are good to know.

Outputting Strings

While we haven't really talked about outputting strings yet, we've certainly done a lot of that using **echo**, remember? **echo** isn't a

real function; it's a language construct, but we use it similarly to a function. It's pretty simple: whatever string you pass to `echo` will be outputted. You can actually pass multiple strings to it and they will be concatenated and outputted. There's also `print` (another language construct), which is like `echo`, but only takes a single string.

While this next one isn't just a string function, I'll put this in here: there's another outputting function named `print_r`: it stands for "print readable," and it's useful for outputting things like arrays and objects (which we aren't discussing in this book, but are like a combination of related variables and functions). Load up this example and check out what's printed out:

Example 3-17

```
$person = array("name" => "Sherlock Holmes", "job" =>
  "detective");
echo "<pre>";
print_r($person);
echo "</pre>";
```

Output

```
Array(
   [name] => Sherlock Holmes
   [job] => detective
)
```

To get the full benefit of `print_r`, you need to output `<pre>` HTML tags around it. This way, the whitespace will be kept.

Many of the example files for the functions below use `print_r` to show the result of the function; however, I haven't put the `print_r` line in the examples, just to keep things clean.

Breaking Up and Getting Together

If you want to break up a string, there's no better function to use than `explode`. Pass it a string and a character to divide on, like so:

Example 3-18

```
$str = "This is a string.";
$arr = explode(" ", $str); # returns array("This", "is",
  "a" "string");
```

We want to split that string on its spaces, so, we pass a string with a single space as the delimiting parameter. The **explode** function returns a new array with the parts of the fractured string. Of course, that delimiter can be more than a single character, so **explode("is a", $str)** in the above context returns an array with two items: "**This** " and " **string**" (note the following and preceeding space in each of those items, respectively).

Then there's the reverse: **impode** (also known by its alias **join**). Pass it a "glue" parameter and an array: the array items are pasted into a string, with the glue between every two items:

Example 3-19

```
$arr = array("123", "456", "7890");
$phone_number = implode("-", $arr); # "123-456-7890";
```

Couldn't be simpler.

A Case of Changed Case

Ever want to change the case of a letter, or a word? This friendly fellowship of functions is at your service; all these next examples return true:

- **lcfirst** changes the first letter of the string to lowercase: **lcfirst("Andrew") === "andrew"**

- **ucfirst** changes the first letter of the string to uppercase: **ucfirst("andrew") === "Andrew"**

- **ucwords** changes the first letter of every word in the string to uppercase: **ucwords("getting good with PHP") === "Getting Good With PHP"**

- **strtolower** changes the whole string to lowercase:
  `strtolower("HEY THERE, WATSON") === "hey there, watson"`

- **strtoupper** changes the whole string to uppercase:
  `strtoupper("hey there, watson") === "HEY THERE, WATSON"`

Keeping Thing Trimmed

There are three super-simple functions that you'll find immensely useful when processing user input. You'll find that users are wasteful creatures, always leaving extra whitespace around the string values they submit. Pretty simple:

Example 3-20

```
trim("        string with much padding        ");
  # returns "string with much padding"
```

There's also the **ltrim** and **rtrim**, with only trim whitespace from the left or right side of the string, respectively. As a handy bonus, all three of these functions can take a second parameter, a string of characters that can be trimmed:

Example 3-21

```
trim("_#_#_#_#_content_#_#_#_#_", "#_"); # returns "content"
```

Replacements

What programming language is complete without a way to replace sections of strings? Well, PHP has it: **str_replace**. Give this function three parameters: a string to search for, a string to replace it with, and a string to do the searching and replacing on. As you might expect, **str_replace** returns a modified string.

Example 3-22

```
$str = "The original string";
echo str_replace("original", "modified", $str); # outputs
    "The modified string"
```

There's also **str_ireplace** (notice the **i** in there). It works identically, except that it's case-insensitive.

How Long?

You've already met the **strlen** function, remember? It simply returns the number of characters in the string:

Example 3-23

```
echo strlen("Getting Good with PHP"); # returns 21
```

Needle in a Haystack

There will be times when you want to find a string within a string, and those times are the times when you will want to use the **strpos** function. It takes two parameters: the *haystack*, or the string in which to search, and the *needle*, what substring to search for. If the search term is found, you'll get a number back, that number being the position within the original string where the substring begins.

Example 3-24

```
echo strpos("They call this string a haystack", "t");
    # returns 10
```

If you want to start the string searching at a specific point, you can pass that number as the third parameter. So, if we want to skip the first "t", we can do this:

Example 3-25

```
echo strpos("They call this string a haystack", "t", 11);
  # returns 16
```

Et Cetera

PHP is a huge language, as you might have guessed. There is a plethora of functions for manipulating strings, and the point of this book isn't to provide you with a comprehensive resource. It's meant to give you enough to feel confident doing the basics. For more string function goodness, check out the string function documentation (http://ca2.php.net/manual/en/ref.strings.php).

Array Functions

Next, for your viewing pleasure, we have a wonderful array of array functions. Once again, we're hardly going to scratch the surface. I've hand-picked this mini-selection of array functions that I think will be most useful to you as you get started. Let's go!

Pushin' and Poppin'

Adding and removing items is one the most common things you'll do with your arrays. Here are four handy methods that help with that:

`array_push` is the quickest way to add one or more items to the end of the array:

Example 3-26

```
$friends = array("Watson", "Mycroft");
array_push($friends, "Sherlock");
```

This function (and most array functions) modifies the array you pass it, instead of returning a new array. This means that `$friends` now has three items: "Watson", "Mycroft", and "Sherlock". You can

add as many items as you want at a time simply by passing more parameters.

`array_pop` is the opposite of `array_push`: it pulls the last item off of the array and returns it.

Example 3-27

```
$friends = array("Watson", "Mycroft");
echo array_pop($friends); # Mycroft
```

If you run that, the name "Mycroft" will be echoed, and `$friends` will only contain one item.

Then, there's `array_unshift`: this function adds an item (or several) to the beginning of the array.

Example 3-28

```
$friends = array("Watson", "Mycroft");
array_unshift($friends, "Sherlock", "Lestrade");
```

Now, `$friends` will be equal to `array("Sherlock", "Lestrade", "Watson", "Mycroft")`: the new items have been put onto the front of the original array. Of course, you can use `array_unshift` to prepend a single item too.

Finally, `array_shift`: if you're following the pattern, you might guess that this pulls the first item off the front of the array and returns it. You'd be right:

Example 3-29

```
$friends = array("Watson", "Mycroft");
echo array_shift($friends);
```

This outputs "Watson" and leaves Mycroft in the array alone.

Mappin' and Walkin'

Very often, you'll want to do something to every item in an array; we've seen how **for** and **foreach** loops are good for that. But, let's say you want to do something to every item in the array and collect the results within a new array. Sure, you can do that with a loop, but the **array_map** function makes it easier. You simply hand it the name of a function, and the array you want it to operate on:

Example 3-30

```
$friends = array("Watson", "Mycroft", "Sherlock");
$F = array_map("strtoupper", $friends);
echo "<pre>";
print_r($F);
echo "</pre>";
```

Here's the output:

```
Array(
    [0] => WATSON
    [1] => MYCROFT
    [2] => SHERLOCK
)
```

As you can see, the **strtoupper** function was called on each of the items in the original array. Note two things: first, a new array is returned and the original array is undamaged. Second, we pass the name of the function as a string (as the first parameter); of course, you can give it the names of your own functions, not just the built-in ones:

Example 3-31

```
function alternate_case($str) {
  $str = str_split($str);
  foreach($str as $k => $v) {
    $str[$k] = ($k % 2 == 0) ? strtoupper($v) : strtolower($v);
  }
```

```
    return join("", $str);
  }

$friends = array("Watson", "Mycroft", "Sherlock");
$F = array_map("alternate_case", $friends);

echo "<pre>";
print_r($F);
echo "</pre>";
```

Here's a little homework for you: run the above code and see the output. Then, come back and figure out what exactly is going on in that function. You've learned about all the parts that I've used here: you just need to figure out how they work together! (Feel free to catch me on Twitter if you're having trouble.)

If you like **array_map**, but don't need to collect values into an array, try **array_walk**. Pretty much the same, but only true or false is returned, based on if the function was successfully run or not. Don't miss the fact the function name is passed as the second parameter, unlike **array_map**.

Example 3-32

```
function greet ($name) {
  echo "Hello, $name! <br />";
}
$friends = array("Watson", "Mycroft", "Sherlock");
array_walk($friends, "greet");
```

Searching High and Low

What about searching within arrays? That's something you'll want to do, right? Well, there are two methods that you can use. The one that's probably more useful is **array_search**. Pass it a value to search for and an array to search within, and it will return the key for the appropriate item in the array.

Example 3-33

```
$friends = array("Watson", "Mycroft", "Sherlock");
echo array_search("Sherlock", $friends); # 2
```

That'll output the number 2; the index of the searched-for item in the array.

If you just want to find out if something is within an array, but not actually do anything about it, there's the **in_array** function. Same parameters and all, it just returns **true** or **false**, depending on whether the item exists or not:

Example 3-34

```
$friends = array("Watson", "Mycroft", "Sherlock");
var_dump(in_array("Lestrade", $friends));
```

Try that: you should get **bool(false)**. The **var_dump** function is similar to **print_r**; however, it can output Booleans and **print_r** can't.

Slicin' and Dicin'

Sometimes, you'll have an array that you want to perform some madness on. For example, how about slicing out part of that array? The **array_slice** function can help with that. It takes an array as the first parameter. The second parameter is the offset: that's the index that the slice will start at. If it's a negative number, it will be counted from the end of the array. But beware: the first item in the array has an index of 0, but the last item isn't -0, it's -1.

The third parameter is simply the length of the slice. Run this to get the idea:

Example 3-35

```
$friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");
echo "<pre>";
print_r(array_slice($friends, 1,1)); # "Mycroft"
```

```
    print_r(array_slice($friends, -1, 2)); # "Lestrade"
    print_r(array_slice($friends, 0, 3)); # "Watson",
       "Mycroft", "Sherlock"
    echo "</pre>";
```

After each line, you can see what items are in the outputted arrays. Notice the second one in particular: I gave it a length of 2, but it only has one item. That's because we started at the last item, and there aren't any items after it, so we only get the one.

Then there's **array_splice**; parameter-wise, it's pretty similar to **array_slice**. You hand it an array, a starting point, and a length. But this time, there's a fourth parameter: either a single value or an array of values that will replace the values that are being sliced out.

Example 3-36

```
    $friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");
    array_splice($friends, 0, 3, "Mrs. Hudson");
```

It's important to realize that **array_splice** returns an array of the values that are sliced from the parameter array, just like **array_slice**. The changes are actually made in the original array. Run this to see what I mean:

Example 3-37

```
    $friends = array("Watson", "Mycroft", "Sherlock", "Lestrade");
    echo "<pre>";
    print_r(array_splice($friends, -1, 1, array("Gregory",
       "Irene")));
    print_r($friends);
    print_r(array_splice($friends, 0, 3, "Mrs. Hudson"));
    print_r($friends);
    echo "</pre>";
```

Sorting Things Out

If you want to sort the items in an array, that's possible too. Start with the `asort` function: You just pass it an array, and it sorts the array by value, alphabetically. Of course, the indices are kept in with their appropriate values:

Example 3-38

```
$person = array("name" => "Sherlock", "job" =>
  "Detective", "age" => "Unknown");
asort($person);
print_r($person);
```

This outputs the following:

```
Array ( [job] => Detective [name] => Sherlock [age] =>
  Unknown )
```

This also works for arrays with numeric indices: just like with the associative arrays, though, the indices are kept with their values.

Example 3-39

```
$a = array("z", "x", "r", "a");
asort($a);
print_r($a); # Array ( [3] => a [2] => r [1] => x [0] => z )
```

There's also `arsort`, which sorts reverse-alphabetically.

Counting Your Chickens (After They Hatch)

We've already met the `count` function, but let's formalize this intro-duction. This function simply takes an array as a parameter and returns the total number of items there are in the array. No more to say.

Example 3-40

```
$arr = array(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
echo count($arr); # 17
```

Summing it all Up

Last one: if you ever want to add up the numbers in an array, you know who to call: `array_sum`:

Example 3-41

```
$arr = array(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
echo array_sum($arr); # 136
```

As you can see, it's pretty straightforward. End of story, I guess.

Date and Time Functions

There's a good chance you'll have to work with dates or times at one time or another. So, it's a good idea to know a few functions for working with them.

parse_date

Let say you're getting a date from a site user, and they're giving it to you as a string. You could request that they input the date in a specific format, but there will always be that one person who ignores the rules. This is where the `parse_date` function will come in handy. It takes a date string and returns to you an associative array with details of the date.

Example 3-42

```
print_r( date_parse("2012-03-07 13:46") );
```

Here's part of the output (I've omitted a few keys/values relating to the timezone and any error that occurred (none in this case)).

```
Array(
  [year] => 2012
  [month] => 3
  [day] => 7
  [hour] => 13
  [minute] => 46
)
```

time

If you're doing any work at all with dates and times, you'll soon have to get familiar with *Unix timestamps*. A Unix timestamp is simply a number: it's the number of seconds that have passed since January 1, 1970. To get the current Unix timestamp, you can use the `time` function.

Example 3-43

```
echo time(); # for me, 1331147363
```

You can read more about Unix timestamps on Wikipedia (https://en.wikipedia.org/wiki/Unix_time).

strftime

Obviously, Unix timestamps are not very user-friendly. For practical date formats, we turn to the `strftime` function. Many programming languages have this function, and they all work the same way, for the most part. You pass this function a string with tokens that tell it how to format the date. You can pass a timestamp as the second parameter: if you don't, it will use the current time as the default.

Example 3-44

```
echo strftime("%B %d, %Y"); # March 7, 2012
```

As you can see, we're using three tokens in the format string: **%B** stands for the month name, **%d** stands for the day of the month,

and `%Y` stands for the year. There are a heap of other tokens that you can use: to learn them, check out the handy little website http://strfti.me/, which gives you a quick reference and a textbox to test your format strings in.

Math Functions

You can't be a programmer without doing at least a little math, so here are a handful of math helper functions that you'll probably find invaluable:

max/min

You can probably guess what the `max` and `min` functions do: you hand them a bunch of number parameters, and they'll return the highest or lowest number, respectively. You can also pass them an array of numbers instead of individual parameters:

Example 3-45

```
echo max(12, 9, 4,16, 2, 3.14); # 16
```

```
echo min(array( 7, 1, 11, 5, 9 )); # 1
```

mt_rand

Need a random number? `mt_rand` is your function. If you want a number within a specific range, pass the min and max values as parameters. Otherwise, `mt_rand` will return a number somewhere between 0 and the highest value that your computer hardware can support.

Example 3-46

```
echo mt_rand();
```

```
echo mt_rand(0, 10);
```

Why `mt_rand`, and not just `rand`? Well, there is a `rand` function, but `mt_rand` uses the Mersenne Twister algorithm to generate a random number faster than `rand`.

`round/ceil/floor`

Need to round numbers? There are three functions available for that. First, there's `round`, which, when handed a float (remember, that's a number with digits after the decimal point), will return the integer (whole number) closest to that one, using the rounding rules you learned back in school.

Example 3-47 (a)

```
echo round(5.5); # 6
```

```
echo round(5.4); # 5
```

If you want to keep a certain number of digits after the decimal point, pass that number as the second parameter:

Example 3-47 (b)

```
echo round(3.14159, 2); # 3.14
```

The functions `ceil` and `floor` are simpler: they only round up or down respectively, regardless of what's after the decimal point:

Example 3-47 (c)

```
echo ceil(5.4); # 6
```

```
echo floor(5.5); # 5
```

JSON Functions

JSON is a great way to pass data around, especially if you're doing AJAX from the browser. If you're familiar with JavaScript, you'll know that JavaScript objects (which become JSON) are actually very similar to PHP's associative arrays (or PHP objects).

To convert an associative array to JSON, you pass it to the **json_encode** function:

Example 3-48

```
$person = array("name" => "Sherlock Holmes", "job" =>
   "detective");
print_r( json_encode($person)); # {"name":"Sherlock
   Holmes","job":"detective"}
```

To convert JSON to a PHP object, you use the function **json_ decode**, and pass the JSON string as the first parameter. But we're kind of ignoring objects in this book, so you can pass **true** as a second parameter to get an associative array back instead.

Example 3-49

```
$arr = json_decode('{"name":"Sherlock Holmes","job":       ▶
   "detective"}', true);
print_r($arr); # Array() [name] => Sherlock Holmes [job]
   => detective )
```

File Functions

Reading and writing files is something that you'll surely need to do, so let's finish by covering that.

fopen

Before doing anything with a file, you'll need to open it. It's very simple: there's an **fopen** function that opens a file. The first parameter is the path to the file. The second parameter is the mode you want to open it in. Here are a couple of the modes:

- **r** opens to read.

- **r+** opens to read and write, with the pointer at the beginning of the file.

- **w** opens to write; clears all content from the file, or creates the file if it doesn't exist.

- **w+** opens to read and write; clears all content from the file or creates the file if it doesn't exist.

- **a** opens to write, with the pointer at the end of the file.

- **a+** opens to read and write, with the pointer at the end. It creates the file if it doesn't exist.

What's all this pointer business? Well, the pointer points to the place in the file where the next string will be written. For example, if you open a new file for writing, the pointer will be at the beginning of that file. After you write some text to the file, the pointer will be at the end of that text, so that next time you write text to the file, the text you wrote the first time will not be overwritten.

So, we use the **fopen** function to open a file; that function returns to us a handle for the file. We'll use that in some of the other functions.

The following file functions will help you manipulate files in certain ways. Let's say we want to read from a file.

Reading a File

There are three functions for reading a file. Your choice of which one you use depends on how much of the file you want read. First, there's **get_file_contents**: this one reads the entire file into a string for you. So, let's say I have this text file

quote.txt

```
You will not apply my precept," he said, shaking
his head. "How often have I said to you that when
you have eliminated the impossible, whatever
remains, however improbable, must be the truth?
```

If we want to read that all into a single string, we just do this:

Example 3-50

```
$quote = file_get_contents("./quote.txt");
echo $quote;
```

This function is actually the odd one out of the three, because it doesn't require you to use **fopen** first: you just pass it the path to the file, and you're done.

Then, there's **fgets**. This function takes a file handle as a parameter, and returns one line of the file. Of course, the pointer is moved forward, so that the next file you call that function with that file handle, you'll get the next line.

Example 3-51

```
$file = fopen("./quote.txt", "r");
$line1 = fgets($file);
$line2 = fgets($file);
echo $line1;
echo $line2;
```

Run that example, and you'll see the first two lines of the file echoed out.

Finally, there's **fread**. This one takes two parameters: the file handle and a number. That number is the number of bytes that the function will read out. If you put a number that's larger than the number of bytes in the file, **fread** will read the whole file.

Example 3-52

```
$file = fopen("./quote.txt", "r");
echo fread($file, 29);
```

This example should print out the phrase "You will not apply my precept."

So, those functions will help when reading a file. What about writing one?

Writing a File

Handily, PHP has three functions for writing to a file, all of which are equivalents to the three functions for reading the file. Let's look at them in the same order, shall we?

First, there's `file_put_contents`: just like `file_get_contents`, we don't need to use `fopen`. Just pass the function two strings: the first is the file path, the second is the string that's to be written to the file.

Example 3-53

```
$str = "this is the content that will be written to the
  file.";
file_put_contents("new_file.txt", $str);
echo file_get_contents("new_file.txt");
```

If you run this example, you'll see the contents of the `$str` outputted to the browser.

That leaves `fputs` and `fwrite`. These two functions are actually the same function: `fputs` is just an alias for `fwrite`. So, how do we use `fwrite`? It's very simple: two parameters. The first is the file handle, the second is the string to write:

Example 3-54

```
$file = fopen("quote2.txt", "w+");
fwrite($file, "So shines a good deed in a weary world.");
echo file_get_contents("quote2.txt");
```

Now, you should see the quote printed to the browser. And now, you can write files!

fclose

One more to go, and that's `fclose`. This takes a file handle as a parameter and closes the handle. With the basic file work we've

done so far, it actually isn't necessary, but it's a good idea to get into the habit of using it. It will save you from any unfortunate file errors down the line.

There's no point in having an example here: every example we have above that uses `fopen` to create a file handler should use `fclose` to close it.

The Oddities

You may have noticed that there's a rather significant range of differences among the names of functions we've looked at. For example, some names separate their "words" with underscores (`str_replace`) while others don't (`strtolower`). Some use full words (`array_splice`) while some use abbreviations (`arsort`). This is just something you'll have to get used to; the many functions and extensions that are part of PHP are pretty diverse and inconsistent, but don't let that get you down. As you work with PHP, you'll simply memorize the functions that you use the most.

That brings me to a final word about functions. Don't — not for one second — think that you need to memorize every PHP function. There's absolutely no shame in using the PHP documentation. And let me tell you, that's some of the best documentation I have ever seen; it's so complete and comprehensive. Lastly, there are so many functions that I haven't even mentioned in this chapter, so I recommend two things. First, take the time to look around the PHP function list (http://php.net/docs.php), just to get the feel for what's available. Second, when you need to do something that seems like it might have a built-in function, just Google it. You'd be surprised how often there is an internal function for what you want to do.

Summary

In this chapter, we've looked at some of the control structures that PHP gives us to make our code more complex. We've also

seen a handful of functions that you'll use pretty often in your programming.

Next up, we're going to move to working with PHP on the web!

4

Chapter 4

Up until this point, the PHP that we've discussed isn't specifically for the web. But, now we're going to turn a corner and talk about the specific ways that PHP is used on the web. But first, we'll take an important rabbit-trail through the topic of scope (which isn't just for the web).

Scope

Scope is basically the context in which you can use a given variable. For example, you know quite well that if we create a variable, we can then use it from within that file.

```
$friend = "Watson";
echo "Hello $friend!";
```

However, what about from within a function that exists in the same file? Try this:

Example 4-1

```
$friend = "Watson";
function greet () {
  echo "Hello $friend!";
}

greet();
```

If you run this, you'll get a notice telling you that the variable `$friend` was undefined. What? Well, PHP is a bit awkward in that variables from outside a function aren't available within a function by default. If we want to use the `$friend` variable, we have to explicitly say that we know it's coming from the outside:

Example 4-2

```php
$friend = "Watson";

function greet () {
  global $friend;
  echo "Hello $friend!";
}

greet();
```

That'll work for you; you just have to use the `global` keyword to let PHP know that the variable is from outside the function. This works the other way as well: use `global` on variables you're going to define within a function to make them available from outside that function.

Another thing about scope: remember how we looked at including and requiring files? Well, any variables within the included/required files are available just as if they were made in the first that required them.

And that's all you need to know for now about scope!

Superglobals

Superglobals can sound pretty intimidating, I'd say. But, they're actually pretty simple. Basically, they are a group of variables that are available in all scopes: as you know, this means that superglobals are available inside and outside of functions. They're all associative arrays, and most are there to offer you special information. We'll meet them all quickly now, and then we'll do a deep dive into a few important ones.

So, here they are:

- **$GLOBALS**: Any global variables you make are also items in this associative array. Instead of declaring them with `global`, as we did above, you could just access them through the **$GLOBALS** array; so, using our above example, we could

have just used `$GLOBALS["friend"]` instead of prefacing the function with `global $friend`.

- `$_SERVER`: This one includes a bunch of information about the server that's running your instance of PHP, and also any important info about the execution environment.

- `$_GET`: This variable holds any HTTP GET request variables; more on this later.

- `$_POST`: And this one holds any HTTP POST request variables; again, stay tuned.

- `$_FILES`: This is where any HTTP file uploads are stored.

- `$_REQUEST`: Here's your info about the HTTP Request that requested this file.

- `$_SESSION`: This array is where you'll store variables that you'll need across multiple pages; more coming.

- `$_ENV`: This is where they keep the environmental information.

- `$_COOKIE`: HTTP cookie information; yes, more on this one too.

Now that you've met them all, we'll look at a few important ones that you'll use often.
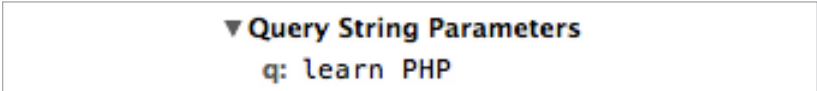
But, let's first step back a moment and look at the big picture. One of the main reasons that you want to learn PHP is so that you can build more dynamic websites. In many cases, this dynamic-ness will come from responding to user input. The main way this user input comes is via variables in the HTTP requests. If you aren't familiar with HTTP requests, that's okay. I'll show you how the data goes from the browser to the server (where your PHP gets it).

$_GET

An HTTP GET request (often just called a GET request) is what is sent from your browser to the server when you type a URL or click on a link. In cases like these, you probably aren't sending any data to the server with that request, but you certainly can. Here's a good example of a GET request in action: DuckDuckGo is a search engine, similar to Google[1]. When I perform a search for "learn PHP" here's the URL that I'm sent to:

```
https://duckduckgo.com/?q=learn+PHP
```

The important part here is everything after the question mark (`?`): that portion of the URL is called the *query string*, and it holds the data. The data is in key-value form, with the key coming first, then an equal sign, and then the value (which has been URL-encoded, of course). If I open the Chrome web inspector, I can inspect this HTTP GET request:

▼ **Query String Parameters**
    q: learn PHP

We can see the one query string parameter, in this case. If there were multiple parameters, they would be separated in the URL by ampersands, like this: **https://duckduckgo.com/?q=learn+php& other=something_else**.

Now, how are these query strings created? Well, you could write them yourself, and sometimes that's appropriate. But most often, you'll use an HTML form, which will do it for you. Let's look at an example:

Example 4-3

```php
<?php

if( isset($_GET["name"]) ): ?>
  <p>Hello, <?php echo $_GET["name"] ?>!</p>
```

---

[1] I'm not using Google for this example because their query strings are several hundred characters long.

```
<?php else: ?>
  <form action="./" method="GET">
    <p> Please write your name below:</p>
    <p> <input type="text" name="name" /></p>
    <p> <button type="submit"> Submit </button></p>
  </form>
<?php endif; ?>
```
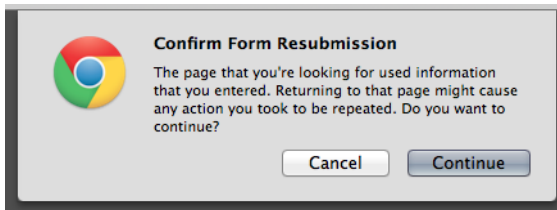
We're doing a few interesting things here. First, we're mixing HTML
and PHP. Second, we're using the alternative syntax for if-state-
ments. Third, we're altering the content of the page based on the
whether or not the variable `name` was sent via a GET request. We're
using the `isset` function, which returns `true` if the variable you
pass into the function exists and is not `NULL`. So, if `$_GET["name"]`
has been set, we know its value comes from the GET request that
requested this page. Therefore, we can output a message. If it has
not been sent, we output a form that allows the user to send that
data. Notice the `<form>` tag: you must set the `action` and `method`
attributes. The `action` attribute tells the form what URL to send
the request to; in this case, we're just requesting the same page,
although we could send the request to a different page without any
problem. The `method` attribute is the HTTP method that should
be used in this request. We're using GET here (we'll use POST
later). This tells the browser how to send the data. Since it's a GET
request, it will put the data into a query string and append it to the
end of the URL. So, if I open `http://localhost:8888/ggwp4/` in
my browser, I'll get the form.

Please write your name below:

[   ]

Submit

I then fill in my name and click *Submit*. The request is sent, and the
URL changes to `http://localhost:8888/ggwp4/?name=`

`Andrew+Burgess`. Notice the attached query string. Now, the `name` parameter is set, and so I see the message.

> Hello, Andrew Burgess!

Note that there's nothing special going on behind the scenes with the query string. You could just type the URL `http://localhost:8888/ggwp4/?name=Sherlock+Holmes` into your browser and you'll get the message. (You know, of course, that these URLs would work just the same with `index.php` in there, after `ggwp4/`.)

It's really that simple to make a GET request with parameters and use them with PHP. What's this good for? Well, since you can see the parameters right in the URL, you don't want to use it for anything private, like logging a user in or changing data on the server. But, any task where you want to take some not-so-private data from the user, GET is great. Here are some usage examples, coupled with example query strings:

- signing them up for an email newsletter: `name=Andrew&email=andrew@example.com`

- asking their location so you can provide the appropriate content: `zip=12345`

- taking search results: `query=learning+PHP`

- showing a specific product: `prodid=123` or `product=pc-12`

You aren't familiar with the POST request yet, but know that the data it passes to the server is not passed as part of the URL, but more discretely in the HTTP headers instead (I'll explain more later). Of course, this is why it's better for sensitive data. However, there's a benefit to having the data in the URL: it makes them bookmark-able. More on this in the POST section.

It's common for beginners to get all excited about the idea of having dynamic user input, and take the idea too far. For example, there's technically nothing stopping you from having a single

index.php file on your website, and using an if/else or switch statement to show different content depending on the query string. You might have these pages:

- example.com/index.php as your home page

- example.com/index.php?page=about as your about page

- example.com/index.php?page=portfolio as your portfolio page

Should you do this? I wouldn't do it for two reasons: 1) I prefer nice clean URLs, and 2) I think having the markup for an entire website in one file would get rather messy. Sure, as you're learning, I'd give it a try. The wonderful thing about it is that you don't have to repeat your header and footer code:

```
<!doctype html>
<html>
<head>
  <title> Example Site </title>
</head>
<body>
  <h1> Example Site Header </h1>
  <ul>
    <li><a href="./">Home</a></li>
    <li><a href="./?page=about">About</a></li>
    <li><a href="./?page=portfolio">Portfolio</a></li>
  </ul>
  <?php if (isset($_GET["page"])) {
    $name = $_GET["page"]; ?>
    <?php if ($name == "about") { ?>
      <p> All about whatever </p>
    <?php } elseif ($name == "portfolio") { ?>
      <p> My greatest works. </p>
    <?php }
  } else { ?>
    <p> Normal Homepage Material </p>
```

```
    <?php } ?>
  </body>
  </html>
```

You should know enough by now to figure that out. Play around with it, but I don't recommend the one-single-PHP-page approach for real sites. Notice that I used the standard syntax for if-statements here: try writing that kind of thing once and you'll be convinced that the alternate syntax is the only way to go for interspersed PHP and HTML **:)**.

$_POST

A lot of what you've just learned about using GET requests is very applicable to POST requests. In fact, to make our previous example work with POST instead of GET, there's only three changes to make. That's changing every instance of "GET" to "POST": it's found twice in the superglobal name, and a third time in the `<form>`'s `method` attribute.

Example 4-4

```
  <?php if( isset($_POST["name"]) ): ?>
    <p>Hello, <?php echo $_POST["name"] ?>!</p>
  <?php else: ?>
    <form action="./" method="POST">
      <p> Please write your name below:</p>
      <p> <input type="text" name="name" /></p>
      <p> <button type="submit"> Submit </button></p>
    </form>
  <?php endif; ?>
```

Now, try out the example. You'll notice that when we click the *Submit* button, the page refreshes, but the URL doesn't change. As you know, this is because we're POSTing to the same file. If I open up the network panel on Chrome's web inspector, and look at that POST request, I can see that the data shows under the "form data" section:

▼ **Form Data**   view URL encoded
name: Andrew Burgess

Now, here's a twist: try hitting the refresh button on your browser. You'll probably get a message asking you to verify that you want to resubmit the data that you already submitted.

Confirm Form Resubmission
The page that you're looking for used information that you entered. Returning to that page might cause any action you took to be repeated. Do you want to continue?

[Cancel]   [**Continue**]

Why the prompt? Well, let's just say that if the first POST request did something like submit an order to an online store, refreshing the page would send that data again and make a second order. That's rarely what you want: so, browsers will make you double-check that refresh.

So, as we've already said, POST requests are better for sending secure data, because the data is only stored within the headers. However, this means our users can't bookmark that page. For example, let's say we have a search box on our website, when the user types in a search query and hits *Search*, we pass their query to our PHP via a POST request to `/search.php`. In the PHP, we'll probably use that query to pull the right data out of a database, format it into HTML results, and output that to the user. However, if our user then bookmarks that page and comes back to it later, we'll have a problem: when clicking a bookmark, the URL is requested via a GET request; and anyway, we don't know what data was sent to that page (because it was in the HTTP headers, which aren't saved with the bookmark), so even if we could do a POST request, we wouldn't know what data to send. So, that `search.php` page is in a predicament: it didn't receive the appropriate POST variables, so it can't do its job. This is why a well written PHP script will use `isset` to see if the GET or POST variables that it needs are set; if

they aren't, it can provide a sensible fallback. In our search example, it would be a much better idea to send that data as a GET request. There's really no need to hide the user's search query, and it makes the page bookmark-able.

There is one more concept to clarify about GET and POST requests: by default, it doesn't make a difference to the PHP file which method was used to get the file. As the programmer, you must choose a method based on the sensitivity of the data, bookmarking ability, and other factors. Once you choose a method, you can then choose what to do if POST or GET variables are set.

Finally, where does AJAX fit into all this? If you've ever made an AJAX request from JavaScript, you'll know that you must specify whether you want to make a GET request or a POST request. On the server side, it really doesn't make that much of a difference whether the request was from a browser or via AJAX. The difference is that whatever would be displayed in the browser will instead be the response data of the AJAX request. This is one place where those JSON functions we talked about would be useful.

Persistence

One thing about the web is that there's no persistence. By that, I mean that if you request a page from a server, and then 10 seconds later request another page, that server doesn't know, by default, that the same person made both of those requests. Passing values from page to page via GET and POST help with this a little bit, but let's step it up a notch. There are two other ways to create a persistent browsing experience: *cookies* and *sessions*.

Cookies

With PHP, we can set cookies. Cookies are just little bits of information that we can store in the browser. Then, at future visits, we can retrieve and utilize that information.

Let's build a small example. Let's say we have a web store that sells office supplies (well, in our example, just pens). Let's start with the pen product page:

Example 4-5 (`e4-5-pen.php`)

```php
<?php
setcookie("product", "pens", time() + 60*60*24*30);
setcookie("time", time(), time() + 60*60*24*30);
?>
```

```
<p> Some info about pens. </p>
<p><a href="./">&larr; Back</a></p>
```

We're setting two cookies on this page: the `setcookie` function takes a number of parameters. The first parameter is the name of the cookie, and the second is the value that we want to store. In this case, we're storing a product name and the current time. The third parameter is when we want the cookie to expire: we're setting it here by getting the current time (in seconds, remember) and adding the number of seconds in one month (just a little multiplication). Of course, after all that, we have some information about the pens we're selling. Let me note one thing about cookies here: The information that sets the cookies in the browser is sent as an HTTP header. Therefore, if you plan to set a cookie, you'll have to do it before you start to output anything. Cookies come first: no HTML or using `echo`, `print`, etc. until the cookies are set.

So, how do we use these cookies? Well, that will be on the homepage. Here's the scenario we're creating: a visitor will visit our site and look at the pens. Then, some time later, they will return. When they hit the homepage, they'll get a message saying that we are now offering some new pens (assuming we are). So:

Example 4-5 (`e4-5-index.php`)

```php
<?php
if ( isset($_COOKIE["category"]) && isset($_COOKIE[   ▶
   "time"]) ) {
```

```php
    if (new_item($_COOKIE["category"], $_COOKIE["time"]) ) {
      $new_items = true;
    }
  }
  function new_item ($category, $time) {
    if (latest_product_time($category) > $time) {
      return true;
    }
  }

  function latest_product_time($cat) {
    return time() - 60;
  }
  ?>

  <a href="e4-5-pens.php"> Pens </a>
  <?php if (isset($new_items)) {
    echo "<p> Hey! Since you were last here, we got some new
      {$_COOKIE["category"]}.</p>";
    echo "<p> <a href='e4-5-pens.php'> Check out the
      {$_COOKIE["category"]}</a></p>";
  } ?>
```

Yes, that's a lot of code, so take a deep breath before you get dizzy. At the top, we start by checking to see if the cookies that we set on the pen page were actually set. This is where the **$_COOKIES** superglobal array comes in: any cookies that you set via **setcookie** will become an item in the **$_COOKIES** array; this is how we read cookies.

If we don't find the cookies, that means that the visitor has either never been to that page, or it has been over 30 days since they last visited (in which case, we won't give them any special message). If we do find those cookies, then we'll pass them both to a function called **new_item**. This function just checks to see if we've received new items in that category since the visitor was last here. As you'll see, we call another function: **latest_product_time**. This function takes a category and returns the time that we added

the latest product in that category to the website. In the real world, this would probably connect to a database and get the info that way. However, we're just returning `time()` – `60`, which is one minute before now. This way, if you go back to the home page immediately, you won't get the message… but wait 60 seconds and refresh.

Notice that we don't output the message in our initial if-statement: we just set `$new_items` to true. Then, at the bottom of our markup, we check to see if that variable is set. If so, we output the message. When the visitor goes to see the new pens, we will send them new cookies; one will be the same, but the `time` cookies will be updated.

So, how do you get rid of a cookie? Eat it! Actually, you just reset it and set the expiry date to a date in the past. It's pretty easy; just subtract anything from `time()`:

```
setcookie("mycookie", "whatever", time() - 60);
```

So, now you've got a pretty good idea of how cookies work… but where are they useful? Well, anywhere you want to remember some information about your users. For example, if your site allows any customizations like making the font larger or smaller, or changing the background color or image, that info will be stored in a cookie. Many site analytics scripts use cookies for recording data; several content management systems (such as WordPress) will use cookies for things like the name and email address you use when making comments. Also, any website where you log in and stay logged in for a while (like, several days), stores that login information in the form of cookies.

You should be able to dig up the list of cookies that all the sites you've visited have stored in your browser (Google a bit if you need help). Don't worry if you find their names and values cryptic: they're probably encrypted for security. Keep in mind that users can clear cookies from their browser at any time, so you should never depend on them for critical information.

Sessions

Perhaps cookies are a little too persistent for what you have planned. Let's say that you want be able to set values that you can use across several pages (and not pass them back and forth), but you don't want them to be remembered beyond the user's visit. For example, you know how many websites have a "remember me" checkbox when you're signing in? If you check that box, they'll store the login info needed to keep you from having to log in on every page as cookies. However, if you don't check the box, they want to store the information temporarily; just for your visit. After you close the browser, they want that information to be lost. This is where sessions come in handy: they store whatever info you give them only for that browsing session: once you close the browser, your "session" is over and all session information is gone.

So, how do we create and use sessions? Well, it's pretty different from cookies. On any page where you plan to use session variables, you start by calling:

```
session_start();
```

I should note that even if you just want to read (and not write) session variables on a given page, you still have to call that function. After that, you can just set and get items in the superglobal array `$_SESSION`. Session variables will persist until the user closes their browser. Here's a simple example:

Example 4-6

```php
<?php
session_start();
if (isset($_GET["name"])) {
  $_SESSION["name"] = $_GET["name"];
?>
<p>Thanks! <a href='e4-6.php'>Go here for your message
  </a>.</p>
```

```php
<?php } else if (isset($_SESSION["name"])) {
  echo "Hi {$_SESSION["name"]}!";
} else { ?>
  <form action="e4-6.php" method="GET">
  <p>Name: <input type="text" name="name" /></p>
  <input type="submit" value="submit"/>
  </form>
<?php } ?>
```

Of course, we begin by starting the session. We then check for a GET request parameter `name`; if that exists, we'll set the session variable `name` to the same value, and ask the user to go to `session.php` (without the GET query string) so they can see the message. There's where the next part of the if-statement comes in. If there's a session variable `name`, we'll use that to output a message. Otherwise, we'll output a form, which will allow the visitor to put in their name and submit the GET request.

ROCK*
TIP

You'll notice I'm once again mixing PHP and HTML. I'm trying to get you comfortable with seeing and reading that, because it's pretty common, and it's something you'll be doing often.

!

Load up that example and put in your name. Once you get the message, refresh the page a few times to make sure you get the message each time; you could even browse to a different website. If you're using a tabbed browser (and who doesn't these days?), try closing the tab and going to a page in a new tab (browsers typically share one "session" across all tabs). You should always get the message. Now, quit the browser completely (don't just close the window), reopen it, and go to our example page. Now you should get the form instead of the message, indicating that you've started a brand new session.

You can remove a session variable by using the `unset` function (this will un-set any variable, not just session variables):

```
unset($_SESSION["name"]);
```

You can get rid of all the current session variables and the session info (which we aren't going into) by calling `session_destroy`.

Now, how about a bit of homework? (It's not required, but you might find it interesting.) I want you to try something: Load up our session example, and get to the point where it is showing you the message. Then, manually add a query string to the URL so that the current value of the session variable `name` will be overwritten with something new.

Databases

Along the same lines of persistence, we have databases. Of course, you don't use a database to store quite the same data that you store in a session or within cookies: databases are quite a different dragon to slay. Let's talk about this.

Just in case you haven't had anything to do with a database before, let's take a minute to discuss what exactly a database is. Sure, you know it's a way to store data, but that doesn't tell you much. Several types of databases have become popular, so you might see names like "noSQL" when reading about databases. We can't cover everything here, so what we'll be talking about are reliable old relational databases.

You might find it easier to get your mind around relational databases if you think about how they're similar to your regular old spreadsheet. Here is some basic-but-important terminology.

- **The database** is the single entity where all the data you're working with is stored. It's a lot like a spreadsheet file in how it organizes information.

ROCK★ TIP

Why are they called relational? According to Wikipedia (https:// en.wikipedia.org/wiki/ Relational_database), it's because they conform to a relation model theory.

- **A table** is akin to a sheet in a spreadsheet, and like any other table of data you've ever seen, it has rows and columns. A database can have many tables, which can be related in different ways. A table houses many records.

- **A record** is a group of related values; it usually has a value that corresponds to every field in the table. This is analogous to a single row in a spreadsheet.

- **A field** in a database is the column in a spreadsheet. It is the "title" for a single piece of information in a record.

- **A data value** is like a single cell: it's the intersection of a record and a field.

Based on this knowledge, let's create a faux database; check this out:

ID	First Name	Last Name	Occupation
1	Sherlock	Holmes	Private Detective
2	John	Watson	Doctor
3	Irene	Adler	Singer
4	G.	Lestrade	Inspector

Here's a database with several tables. The table we're looking at right now has three fields: ID, First Name, Last Name, and Occupation. It has four records: one for each person. Every cell in this table is a data value.

But wait, there's more (as they say). There are pretty strict rules for creating a database. First, you must define what *data type* every field should hold; more on data types later. Also, every table must have a field that will be unique for every record: this is called a *primary key*. That's what the leftmost ID field in the above table is.

Alright: I think you have enough database background to start using one. Here's the micro-project that we'll work on together to

give you a taste of working with databases: we'll see if we can build an HTML table that shows us data from a table in our database. We'll make this match the example table I showed you above.

The first step is to actually build our database. This means we need database software; thankfully, when you installed WAMP or MAMP at the beginning of this book, you also installed and configured MySQL Server. MySQL is a database management system, and while there are other systems available, there's no doubt that MySQL is the top banana, at least as far as the number of users goes. MySQL uses SQL (Structured Query Language) to build databases and work with the data inside them. But we've installed WAMP / MAMP, we've installed phpMyAdmin, which is a decent user interface for building databases, so we'll be using that to build our database.

So, we'll start by opening the start page of either WAMP or MAMP. Find the phpMyAdmin link, click that, and you should see something like this:
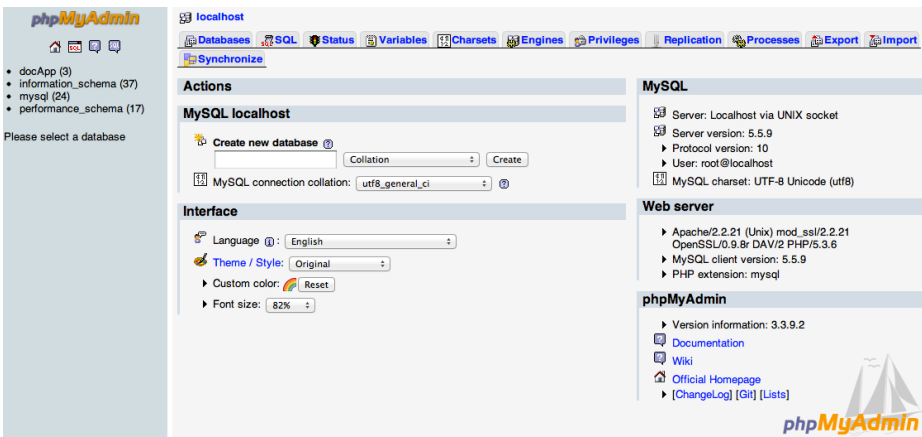
There's a lot here that we won't be discussing at all. Right now, just type "php_book" in the textbox under *Create new database*; then click *Create*:

Yes, this is a poor database name, but it will do for now.

Great! You've created your first database. There are three interesting bits of information on the next screen:

First, notice the left sidebar: it has the text "php_book (0)"; this tells us that the database **php_book** currently has 0 tables. Second, see that green bar near the top that said, "Database php_book has been created"? Right under that, you'll see the SQL code that you would have written to do what you just did via the UI. You'll see this a lot in phpMyAdmin, and it's a good idea to pay attention to it. Even though we won't be learning SQL in this book, find a few good tutorials on line to learn the basic commands and syntax, and then do things in phpMyAdmin and inspect the SQL it shows you afterwards.

Third, at the bottom, we can create a table in our **php_book** database. We'll give it the name `characters` (as in, the characters in a story, not in a string), and type **4** in the *Number of fields* textbox. Hit *Go*.

Now you'll see a table with empty text boxes and drop-down menus. This is where we defined the properties of the four fields that we said our table should have. Have a look at what I've filled in, and then we'll discuss:

Field	Type ⑦	Length/Values[1]	Default[2]	Collation	Attributes	Null	Index	A_I	Comments
id	INT ⬍		None ⬍	⬍	⬍	☐	PRIMARY ⬍	☑	
firstName	VARCHAR ⬍	100	None ⬍	⬍	⬍	☑	--- ⬍	☐	
lastName	VARCHAR ⬍	100	None ⬍	⬍	⬍	☐	--- ⬍	☐	
occupation	VARCHAR ⬍	100	None ⬍	⬍	⬍	☐	--- ⬍	☐	

I haven't done much, but I've defined more than you think. Each row in this table corresponds to a field in our database table. The first column is for the field titles, and you can see, I've set them up to match our example table image above. The next column is the data type. The default is **VARCHAR**, which stands for variable character. Basically, **VARCHAR** is the data type for strings. For the **id** field, we'll change the type to **INT**: the ideal data type for integers, or whole numbers. The rest of the fields will stay at the default of **VARCHAR**: however, we need to define the longest length the string can be (databases are strict this way). As you can see, we choose 100 characters in the third column. The fourth column is titled *Default*: it lets us define what the default value for these fields should be. The default default (ha!) is **None**, which means the field will be left blank, and we won't change that.

For now, don't worry about the *Collation* and *Attributes* columns. But, what's the *Null* column about? If that box is checked, the field is allowed to be blank. In our case, only the **firstName** field can be blank.

The *Index* column is important; there are several different types of indices; remember the primary key we talked about? Well, that's also known as the primary index; make the **id** field our "primary" index. Finally, we have the *A_I* column, which stands for "auto-increment." We'll check this box for the **id** field. This way, when we make a new record but don't fill the **id** field, it will automatically be

filled with the next number. This is how we get this field to be the unique primary key: this is very important, because the primary key must be a unique identifier: no two records can have the same primary key.

All right! Click *Save* at the bottom, and you should see an overview of our newly-created table:

	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	**id**	int(11)			No	*None*	AUTO_INCREMENT	
☐	**firstName**	varchar(100)	latin1_swedish_ci		Yes	*NULL*		
☐	**lastName**	varchar(100)	latin1_swedish_ci		No	*None*		
☐	**occupation**	varchar(100)	latin1_swedish_ci		No	*None*		

Now, let's insert a record or two. Up near the top, click the *Insert* tab. You'll see forms for two records; so, fill 'em out!

After clicking *Go*, you'll get some confirmation. Head up to the *Browse* tab and click that. You should see the data that we've added to our table.

Now, we're ready to head back to the PHP and write some code to pull this data out. Let's start with this:

Example 4-7 (a)

```
## for MAMP
$mysqli = mysqli_connect("localhost", "root", "root",
  "php_book");
```

```
## or, for WAMP
$mysqli = mysqli_connect("localhost", "root", "",
  "php_book");
```

Woah, woah, right? I'll explain: we're using the MySQL Improved (`mysqli`) extension here; it a pretty powerful PHP extension designed specifically for working with databases. What we're doing above is creating a connection to the MySQL database that we just made via the `mysqli_connect` function. We're passing four parameters to it:

- **host –** Remember how I said MySQL is a server? Well, that means it has a host name or IP address that points to it. Since MySQL server is running on the same computer that is running the PHP, we just use the host name `localhost`.

- **username –** This is the name of a user on the server, the one you are logging in as. By default, there's only one user: `root`. On a real website, you'll want to create a different user with the right privileges, but for now, using `root` is fine.

- **password –** Every user has a password: by default, the password for the `root` user is `root`. On WAMP, however, there's no password: it's a blank string.

- **database name –** This is the name of the database you want to connect to. In our case, that's `php_book`.

This should connect successfully. The `$mysqli` variable now holds our connection object. Oh, that's right: we aren't really discussing objects in this book. For now, think of an object as a collection of

related variables and functions. Its variables are called *properties* and the functions are called *methods*.

Now we're ready to start querying the database. We do so by calling the `query` method on the `$mysqli` object, like so:

Example 4-7 (b)

```
$result = $mysqli->query("SELECT * FROM characters");
```

You call a method with this syntax:

```
$object->method_name();
```

Between the object variable and the method name, you use a dash and greater-than sign. So, in our `$mysqli->query` call, we're calling the `query` method, passing it a single string as our parameter. We're storing the return value in the variable `$result`.

But what about that string parameter? Let's look at it again:

```
"SELECT * FROM characters"
```

This is an SQL query. There's no way I could make you an SQL query expert in this book, so I'll just explain this one for now. The `SELECT` keyword tells the database what action we want to perform: we want to select records from the database; this means we want to read them. The next part tells the database what fields we want to get. In our case, we want to get them all, and so we use the asterisk, which stands for "all fields" in this position. The `FROM` keyword prefaces the part of the query that tells the database which data table we want to get the records from; in this query, we're saying we want to find values within the `character` table.

So, after all that, the `$results` variable will be a `mysqli_result` object that has several useful properties and methods. If the query failed for some reason, `$results` would be `false`. Therefore, we should make sure we have our object:

Example 4-7 (c)

```
if ($results) : ?>
  <table>
    <tr>
      <th> ID </th>
      <th> First Name </th>
      <th> Last Name </th>
      <th> Occupation </th>
    </tr>
  </table>
<?php endif; ?>
```

I'm again using the alternate syntax for the if-statement, because we're mixing PHP and HTML. As you can see, if `$results` is the object, then we output the table opening and closing. In between that, we'll output the table rows. Put this after the closing `</tr>` and the closing `</table>`.

Example 4-7 (d)

```
<?php while ($row = $result->fetch_object()) : ?>
  <tr>
    <td><?php echo $row->id; ?></td>
    <td><?php echo $row->firstName; ?></td>
    <td><?php echo $row->lastName; ?></td>
    <td><?php echo $row->occupation; ?></td>
  </tr>
<?php endwhile; ?>
```

This code is a little more complex than you're used to, but hang on. First, we're using a control structure that you haven't learned yet: the while-loop. Here's the normal syntax:

```
while (condition) {
  // code
}
```

While the condition is true, the code in the block will continue to execute. In our case, we're not really using a normal condition: we're assigning a variable.

```
$row = $result->fetch_object();
```

The `fetch_object` method will return each of the records we got from the database, one at a time. So, the first time you call the method, you'll get the first record; the second time, you'll get the second record, and so on. When we're out of records, `fetch_object` will return `NULL`. You can imagine how this works with our `while` loop: when the condition is first evaluated, we call `fetch_object` for the first time and assign its value to `$row`. Since that will be a record object, which equates to true, we'll execute the code in the loop. Next time you come around to evaluating the condition, `$row` will be assigned the next record, and the loop repeats. When we're out of records, `$row` will be assigned to `NULL`, and the `while` loop will stop.

Inside the loop, we've got some HTML, with some PHP inside that. You can see how the `$row` has a property (just like a variable) for every field in our database table, that we're outputting into the table.

To finish off the example, throw a bit of CSS at the top of the file:

Example 4-7 (e)

```
<style>
table {
  font-family: helvetica;
  border-collapse:collapse;
}
th {
  background: #ccc;
}
td, th {
  border:1px solid #aaa;
```

```
    padding: 10px;
  }
  </style>
```

Now, when you launch this in the browser, you should see this:

ID	First Name	Last Name	Occupation
1	Sherlock	Holmes	Private Detective
2	John	Watson	Doctor

How about adding data to the database using PHP? Well, there are a bunch of ways to do it, and unfortunately, the more secure your chosen method is, the more complex the code will be. I'm going to show you an extremely basic way that you probably shouldn't use very often; I'll mention a better way to do it afterwards, and you'll be able to research it yourself.

Remember how we used the `query` method of the `$mysqli` object to select records from our database? Well, we can pass any type of SQL query to that method. So, let's learn about inserting data with SQL.

First things first: we need to choose which table in our database to insert the data into. It's not a hard choice here, because we have only one table: `characters`. Here's the start:

```
  INSERT INTO characters
```

Next, we can choose which fields we're submitting data for. You might think we're submitting data for all fields, but we're actually going to ignore the `id` field, since that gets set automatically. We can also ignore the order of the fields in the table, if we want, but we do have to get the field names right. Now we're up to this:

```
  INSERT INTO characters (firstName, lastName, occupation)
```

If we plan to insert a value for every field, and we're inserting them in the order that the fields are in the database, then that part — the field names in parentheses — is actually optional.

Finally, we need to include the values that we want to put into the database. They must be added in the same order as the fields you listed in the last step:

```
INSERT INTO characters (firstName, lastName, occupation)
  VALUES ("Mycroft", "Holmes", "auditor");
```

So, how do we use this with our PHP? Check this out:

Example 4-8

```
<?php
$mysqli = new mysqli("localhost", "root", "root",
  "php_book");
$firstName = "Irene";
$lastName = "Adler";
$occupation = "Singer";
$result = $mysqli->query("INSERT INTO characters
  (firstName, lastName, occupation) VALUES ('$firstName',
  '$lastName', '$occupation')");
if ($result) :  ?>
  <p>Added a record for Irene Adler, the singer.</p>
  <a href="./e4-7.php">Now go see your new record in
    action.</a>
<?php endif; ?>
```

We begin as before, by connecting to the database. Then, we create three variables to hold the data that we want to put into our database; sure, we don't need to put them into variables, but remember that most often you'll be getting this data from the user, so it will come from a form (most likely via **$_POST**). Next, we write our insert SQL query in the **$mysqli->query** method. Notice that although our variables are strings, we still need to put single

quotes around them in the **VALUES** section; this is so that the database knows that they're strings.

If the data was inserted correctly, you'll get a link that will take you to our table output page. If you click that, you should see that we now have a third row.

ID	First Name	Last Name	Occupation
1	Sherlock	Holmes	Private Detective
2	John	Watson	Doctor
3	Irene	Adler	Singer

As I said, to teach you everything about PHP and databases would take a tome and a half. I've shown you the basics here. There's so much more you can learn, and there are better, more advanced, and more secure ways to do a lot of what we've done. Check Appendix A at the end for a list of database topics to research.

ROCK* TIP

In the real world, it's a terrible practice to take data that the user has given you and stick it right into an SQL query without doing anything to it. You absolutely must sanitize it first, and you probably want to validate it too. We'll discuss these things in the next chapter.

!

Summary

Well, that's another chapter wrapped up. We've covered some pretty important topics in this chapter. Without those **$_GET** and **$_POST** superglobals, without cookies and session, without databases, PHP wouldn't be much more than a toy that glitzes up your sites. However, PHP isn't all fun and games. For our final chapter, we'll dive into the more serious aspects of PHP: staying secure and dealing with errors.

5

Chapter 5

This is it: we're at the last chapter. But, before we wrap things up, there are a few more basics to learn, so let's not waste any time!

Keeping Things Safe

You might recall from just a few minutes ago (What? You aren't reading the book in one sitting?) that I said never to take input from the user and put it directly into the database. In fact, it's never a good idea to simply accept user input, no matter what you're going to do with it. Why? Two reasons:

1. Users will make mistakes in their input. They aren't being malicious; they're just making mistakes. But those mistakes can cause problems later on.

2. Hackers will try to wreak havoc on your site by using trick-ily-worded input. They **are** being malicious: they want to do things like destroy your database or read whatever private data is in there.

So, how can we make sure that the input the user has given us is safe before we start using it? Well, for starters, you can get pretty far by just making sure that the input is of the type you expected. If you expect a certain piece of input to be a number, pass it to the `is_numeric` function, to see if it is:

```
is_numeric("123.45"); # true
```

```
is_number("two"); # false
```

There are several other functions in the same category as `is_numeric`: for example, `is_array`, `is_string`, `is_null`, `is_bool`, `is_float`, and `is_int`. However, these aren't quite the same as `is_numeric`. For example `is_bool("true")`, `is_float("3.14")` and `is_int("123")` are all false: they don't try to convert the way `is_numeric` does. These functions aren't

as useful for validation, because we usually get data from the user in the form of a string (even if that string holds a number, for example).

So, how do we deal with that form of data? Like I said, the problem here is that most of them will be a specially formatted string. For example, here are a few things you might ask a user for that would come to you as a string:

- a name
- a date
- an email address
- a link
- a phone number

I'm sure you can think of others. So now the question is, how can we verify that a piece of user input is in the format we want, and not some malicious hack?

This is where PHP's handy `filter_var` function comes in handy. For our purposes, this function will take two parameters. The first is the piece of data we want to check out. The second is the ID of a filter. Let's check out a few of these filters.

If we want to make sure our input is a valid email address, we can use `FILTER_VALIDATE_EMAIL`.

Example 5-1

```
$email = filter_var($_POST['email'], FILTER_VALIDATE_EMAIL);
```

You should know that the `filter_var` function will return the data (the first parameter) if the validation passes; otherwise, it will return `false`. This method will make sure that a user has submitted a properly formatted email address.

How about a URL?

Example 5-2

```
$link = filter_var($_POST["url"], FILTER_VALIDATE_URL);
```

Similar to the email address validation, this example will make sure that a user has submitted a properly formatted URL.

Then, there are filters for validating integers and floating point numbers:

Example 5-3

```
filter_var("123", FILTER_VALIDATE_INT);
```

```
filter_var("123.45", FILTER_VALDATE_FLOAT);
```

Notice how all of those filter IDs were prefixed with `FILTER_VALIDATE`. They didn't change the data in any way; they just told you whether the data you gave them matched a certain pattern. But there are a couple of other filter IDs that actually do clean up the data for you. And they all start with the `FILTER_SANITIZE` prefix.

Want to remove unwanted characters from an email address?

Example 5-4

```
filter_var("john★doe@gmail.com", FILTER_SANITIZE_EMAIL);
  # johndoe@gmail.com
```

How about replacing special characters with their HTML-safe entities?

Example 5-5

```
filter_var("<em> Holmes & Watson </em>", FILTER_SANITIZE_  ▸
  SPECIAL_CHARS); # &#60;em&#62; Holmes &#38; Watson
  &#60;/em&#62;
```

If you want to get rid of those HTML tags altogether, you should use this `FILTER_SANITIZE_STRING`:

Example 5-6

```
filter_var("<em> Holmes & Watson </em>", FILTER_SANITIZE_  ▶
    STRING); # Holmes and Watson
```

And for numbers, We've got **FILTER_SANITIZE_NUMBER_INT**, which takes out everything except numbers and plus or minus signs.

Example 5-7

```
filter_var("+123,4a5b6.56", FILTER_SANITIZE_NUMBER_INT);
    # +12345656
```

Now, here's something a bit more complex, There's a **FILTER_SANITIZE_NUMBER_FLOAT** that is for more complex numbers. However, by default, it puts out the same thing that **FILTER_SANITIZE_NUMBER_INT** would:

Example 5-8

```
filter_var("+123,4a5b6.56", FILTER_SANITIZE_NUMBER_FLOAT);
    # +12345656
```

This is where a few options (called flags) will come in handy. There are a few ways to use the options, but we can do it by adding the options as an extra parameter. There are three option IDs that go with **FILTER_SANITIZE_NUMBER_FLOAT**:

- **FILTER_FLAG_ALLOW_FRACTION –** This option keeps any periods (decimal points) it finds.

- **FILTER_FLAG_ALLOW_THOUSAND –** This option keeps any commas it finds.

- **FILTER_FLAG_ALLOW_SCIENTIFIC –** This options keeps the characters **e** and **E**, for numbers in scientific notation.

Let's use the first two flags. We just add them as the third parameter, separating each flag with a vertical bar (" **|** "):

```
filter_var("+123,4a5b6.56", FILTER_SANITIZE_NUMBER_FLOAT,
  FILTER_FLAG_ALLOW_FRACTION | FILTER_FLAG_ALLOW_THOUSAND);
  # +123,456.56
```

Many of the other filters offer flags, and other options as well.

I leave the topic of `filter_var` with a word of warning: use the sanitization filters cautiously. It isn't always wise to change the input the user gives you. For example, if a user's email address has taboo characters, then you probably want to ask them to re-enter it, so you can be sure it's correct. On the other hand, you can probably safely take extra characters out of a number.

There's one more sanitization function you need to know about; well, it's a set of functions, kind of. Whenever you're using user input with a database query, you always want to use that database's escape function on the input. In our examples, we've been using a MySQL database, so we would use the `mysql_real_escape_string`. This escapes any questionable characters so that no one can attack our database by typing malicious code into our form fields. For more on this, check out SQL Injection Attacks (https://en.wikipedia.org/wiki/SQL_injection).

Example 5-10

```
mysql_real_escape_string("' OR ''='"); # \' OR \'\'=\'
```

Just run any values you get from the users through that, and they'll be safe for use. Of course, if you're using a different type of database, you'll use a different function. For example, if you're using PostgreSQL, you'd use `pg_escape_string`; if you're using Sqlite, you'd use `sqlite_escape_string`.

We talked about filtering data just a moment ago, so I want to wrap up our discussion of sanitization and validation with a look at a more complex way of validating. Let's say that the standard methods of validation that PHP gives us aren't quite enough. Let's say we want to make sure we receive a date string in this format:

YYYY-MM-DD. This is a good place to use PHP's regular expression functions. If you aren't familiar with regular expressions, you'll find plenty of good tutorials online; they're pretty complex, and not something I can teach you here.

For our use, the `preg_match` function will work fine. We pass it two parameters: the regular expression pattern we want to match, and the string we want to match it in. If the pattern is found, the function returns 1; otherwise, it returns 0. For example:

Example 5-11

```
preg_match('/^\d{4}-\d{2}-\d{2}$/', "2012-05-12"); # return 1
```

```
preg_match('/^\d{4}-\d{2}-\d{2}$/', "May 12, 2012"); # return 0
```

There are more ways to use this function, as well as other regular expression functions, but, really, regular expressions go beyond the scope of this book.

When Things Go Wrong

As a PHP beginner, you're going to make mistakes. Back in Chapter 1, I showed you how to make sure MAMP or WAMP would display any errors. Now, let's talk briefly about errors in PHP.

First off, PHP offers more than just errors. There are actually three message types: *errors*, *warnings*, and *notices*. Let's look at what each looks like, what it does, and what you should do about it.

Errors

An error is the most severe message you can get: when PHP *throws* (yes, that's the technical term) an error, the execution of code stops, and any code after the code that causes the error will not be executed.

What could cause such an error? Well, often it will be a mistake on your part: you know, typos and such. For example, if you mistype

some syntax, leave off a semicolon, or try to use a function that hasn't been defined, you'll get an error. See here:

Example 5-12

```
require "file_does_not_exist.php";
echo "This text won't ever appear";
```

Running that code results in the following message:

Warning: require(file_does_not_exist.php) [function.require]: failed to open stream: No such file or directory in **/Applications/MAMP/htdocs/ggwp4/index.php** on line **3**

Fatal error: require() [function.require]: Failed opening required 'file_does_not_exist.php' (include_path='.:/Applications/MAMP/bin/php/php5.3.6/lib/php') in **/Applications/MAMP/htdocs/ggwp4/index.php** on line **3**

We're actually getting both a warning and an error in this case, but both point to the same problem: PHP couldn't find the file that I was requiring. Notice that the message after the `require` line is never executed, because errors stop execution. This is also a good time to say that PHP errors can sometimes be pretty cryptic, especially when you're learning. However, they always include a line number, which is a good place to start looking for problems. Of course, read the error message: that'll give you a good start. In this case, I can see that PHP "Failed opening required 'file_does_not_exist.php'" and that there's "No such file or directory."

Warnings

A warning is a bit less severe than an error; when part of your code outputs a warning, the file will continue to execute: the problem isn't bad enough to grind everything to a halt. What causes a warning? Well, `include`-ing a file that doesn't exist, using an incorrect parameter when connecting to a database, and dividing by zero are good examples. See here:

Example 5-13

```
echo date("F j, Y", 1234567890, "something else");
```

```
echo date("F j, Y", 1234567890);
```

Here we're using the **date** function. This function takes a format string (where "**F**" stands for the month name, etc.) and an optional timestamp. However, the function doesn't take a third parameter. When we give it a third, we get a warning:

Warning: date() expects at most 2 parameters, 3 given in **/Applications/MAMP/htdocs/ggwp4/index.php** on line **3** February 13, 2009

Notice that the second line above still executes, because warnings don't stop the execution of the file.

Notices

Notices are a step down from warnings; these are for things that might indicate an error (or might not). Trying to use a variable or array item that doesn't exist will cause a notice. Also, using a deprecated function will cause a notice (although, it's not labeled that way):

Example 5-14

```
$a = split(" ", "a b c");print_r($a);
# Array ( [0] => a [1] => b [2] => c )
```

Here we're using the **split** method, which has been deprecated; this means that it's not recommended to use this function, it's been replaced with something better, and it will probably be taken out of future versions of PHP. What it does is split the second parameter (a string) by the regular expression pattern or string given as the first parameter. If you run the above code you'll see that we get a "Deprecated" message.

Handling Errors

So, what can we do about these errors, warnings, and notices? You obviously don't want errors showing up when visitors are coming to your site, right? To start with, as much as possible you should follow the clues that the messages and line numbers gives you and try to eradicate those errors.

In some cases, though, you can't be sure that you'll never get an error. For example, when using a database, what happens if the database server is down? You can't connect. But you don't want the users to see that warning. Well, there is a way to suppress warnings, but that's still not enough in this case, because we can't perform the action the user is expecting. This is why many functions like `mysqli_connect` will return `false` if something goes wrong. That lets you do something like this:

```
$mysqli = mysqli_connect(/* params */);

if ($mysqli) {
  # standard code
} else {
  # user-friendly explanation
}
```

If you're really in a fix, there's the error control operator. Just put an at-sign (`@`) at the beginning of an expression (remember, an expression is anything that returns a value). However, be careful with this: you're usually better off fixing the error, or refactoring your code so you can fix the error.

.htaccess

Here's a thought: you won't always have access to your PHP settings file. If you're using a shared hosting service, for example, you're just one user among many others using their PHP installation: any settings you change will affect everyone else on that

server. But what if you want to make a few changes to your PHP settings?

There's a neat little thing called an *.htaccess* file, and here's how it works: an *.htaccess* file that goes inside any directory on your server can hold settings for your web server, Apache. And, when you're running PHP as an Apache module (which is usually the case on a shared host), you can configure many PHP options from that *.htaccess* file. This can get pretty complicated, well beyond beginner-level stuff, but I mention it for this reason. If you need to turn error reporting on or off on your website, for testing or production, this will be a good way to do it. Try this: you should have error reporting turned on (we did that in the first chapter). Now, create a file named `.htaccess` (starting with the dot) and put this in it:

Example 5-15-htaccess

```
php_flag display_startup_errors off
php_flag display_errors off
php_flag html_errors off
```

Then, save that file into a directory in your `htdocs` folder. Create a PHP file with a warning or notice, and you won't see the message. Create a PHP file with an error, and you'll get a 500 server error.

There's a lot more you can do with *.htaccess* files, but that's so far out of the scope of this book; I'd encourage you to look 'em up, though. They can be handy.

ROCK*
TIP

`htaccess`? `htdocs`? *See a pattern. The "ht" stands for Hyper Text… just like HTML.*

Frameworks

Perhaps you've heard of some of the many PHP frameworks… or maybe you haven't. In that case, listen up: you could think of a

framework as a collection of pre-written code that works together to make your job of making a website much easier. They're more useful when you're building a full web-app: many of them offer the MVC (Model, View, Controller) architecture. While you can certainly use them for a smaller website, you might find them a bit hefty.

You already know that we won't be getting into frameworks in this book, but you should know that they exist, and that they're extremely helpful for large projects. Besides including their custom application code, they usually have many other helpful methods that make writing an app pretty simple.

If you're interested in learning about frameworks, there are reams of great tutorials on the web; in fact, they all have great documentation as well. Check these ones out sometime:

- CodeIgniter (http://codeigniter.com/)
- Kohana (http://kohanaframework.org/)
- Zend Framework (http://framework.zend.com/)

Deploying

When building PHP websites, you'll probably do so locally: on your own computer, running it with a package like MAMP or WAMP. What happens when you're ready to release it out into the wild? When you're just beginning, this will probably mean firing up your FTP client and moving the site to your web server. Pretty much every web host you'll find today will support PHP, and, for the most part, you shouldn't have any problems. However, programming is never perfect, and issues are sure to come up sooner or later. It's impossible for me to help you with every situation right here, but let me give you some tips that will help:

- First, make sure things are set up correctly. For example, if you're having trouble connecting to the database on your hosting server, check out your host's documentation, or see

if they have a support email address, chat room, or even phone number. If you're using a good host, they should be happy to help.

- If the very same code that worked fine on your local machine is throwing errors on your server, figure out what's causing the errors. This might involve configuring something via an `.htaccess` file, or contacting your host to see if they can configure something on their end.

- If you're still can't figure something out, do a few web searches. Often, searching for "PHP" plus the error message you're getting will bring up a workable solution.

- If all else fails, reach out to the PHP community. Ask about your problem on forums or on Twitter: chances are, you'll find someone who's both friendly and helpful.

Conclusion

That brings this whirlwind beginner's guide to PHP to a close. I hope you've enjoyed the trip, and that you're ready to start using some PHP on your own projects. But remember, it's a huge language that seems almost endless: there's so much more that you can learn, and if you're ready to start, you can check Appendix A for a list of topics to search for.

Well, my job is done. But your job, as a PHP developer, is only just beginning.

APPENDICES

Appendix A: What We Didn't Cover

I mentioned a few times that there's no way we could cover every PHP topic, so here's a super-short list of topics you might want to look into if you're interested in pursuing PHP. Don't forget, you can also learn so much more about the topics we did discuss.

- Headers
- Regular Expressions
- Image Processing (with ImageMagick or other extensions)
- Object Oriented PHP
- PDO (PHP Data Objects)
- XML Manipulation
- Encryption
- SQL Injection Attacks
- Mail: sending and receiving via IMAP or POP3, etc.
- Internationalization / Localization
- PHP on the Command Line

Appendix B: Further Resources

- **PHP.net** (http://php.net/) is, without doubt, the best resource for information about what's what in PHP. Unparalleled documentation.

- **PHP for Absolute Beginners**, written by Jason Lengstorf and published by Apress, is a great book for beginners. It'll take you from knowing nothing to almost a little bit of everything as you build a blog in PHP. It's pretty big — 408 pages — but you'll learn a lot. http://www.apress. com/9781430224730

- **PHP Cookbook, 2nd Edition** is another great resource. Written by Adam Trachtenberg and David Sklar and published by O'Reilly, this 816-pager covers both basic and advanced material: everything from strings to using and building REST and SOAP web services: http://shop.oreilly. com/product/9780596101015.do. O'Reilly has been kind enough to put the first edition up on the web for free: http:// commons.oreilly.com/wiki/index.php/PHP_Cookbook

- **PHP Tutorials at Nettuts+:** (http://net.tutsplus.com/category/php/) Nettuts+ offers some of the best PHP tutorials around. Okay, maybe I'm somewhat biased as a staff writer, so you be the judge.

- All over the web, there are bunches of **great sites with great PHP tutorials**. If you're ever stuck, just do a search, and you're almost guaranteed to find a solution.

About the Author

Andrew Burgess is a Canadian web developer, university student, and staff writer for Nettuts+, where he has published numerous popular tutorials and screencasts. Andrew is also the author of the Rockable titles "*Getting Good with Git,*" and "*Getting Good with JavaScript.*" As a web developer, he specializes in JavaScript and Ruby. Andrew lives with his family in Oshawa, Canada.

Check out Andrew's personal site: http://andrewburgess.ca

Or follow him on Twitter: @andrew8088

Now that you've finished
Getting Good with PHP
check out these related eBooks from the
Rockable Press library:

Getting Good with JavaScript

by ANDREW BURGESS

Getting Good with Git

by ANDREW BURGESS

MORE EBOOKS