LEARN PHP in one day and LEARN IT WELL

The only book you need to
start coding in PHP immediately

PHP for Beginners with Hands-on Project

LEARN CODING FAST

JAMIE CHAN

Learn PHP in One Day and Learn It Well

PHP for Beginners with Hands-on Project

The only book you need to start coding in PHP immediately

By Jamie Chan

https://www.learncodingfast.com/php

Copyright © 2020

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Preface

This book is designed to help you learn PHP fast and learn it well. While the book is suitable for absolute beginners in PHP, you do need to be familiar with HTML and SQL (for Chapter 11).

The book covers a wide range of topics in PHP, carefully selected to give you broad exposure to the language. If you are an absolute beginner in PHP, you'll find that this book explains complex concepts in an easy to understand and concise manner.

At the end of the book, you'll be guided through a project that gives you a chance to put everything you've learned to use and see how it all ties together.

You can download the source code for the examples and project at https://www.learncodingfast.com/php

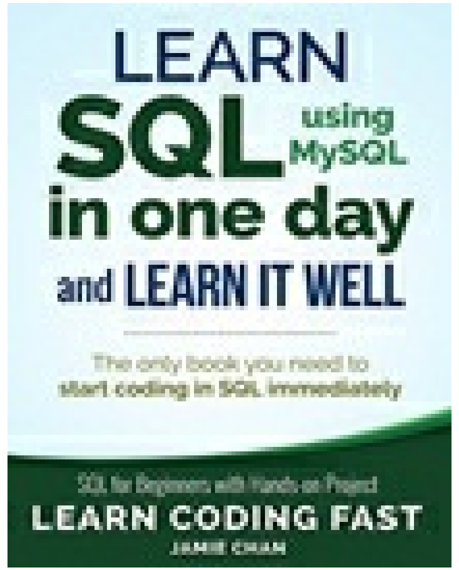
Any errata can be found at https://www.learncodingfast.com/errata

Contact Information

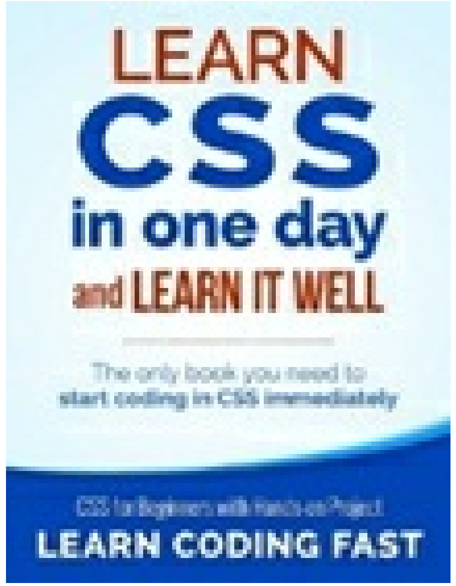
I would love to hear from you.

For feedback or queries, you can contact me at
jamie@learncodingfast.com.

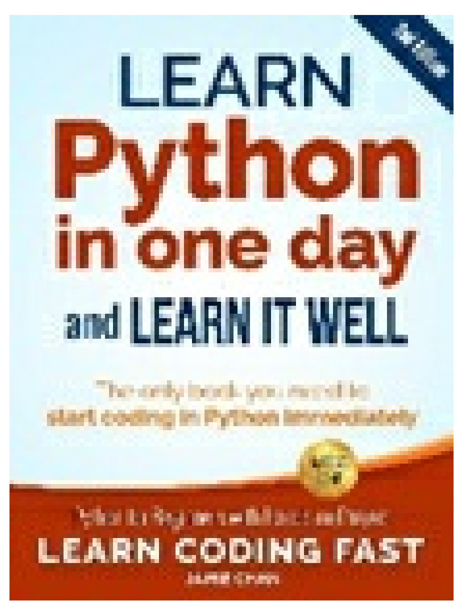
More Books by Jamie



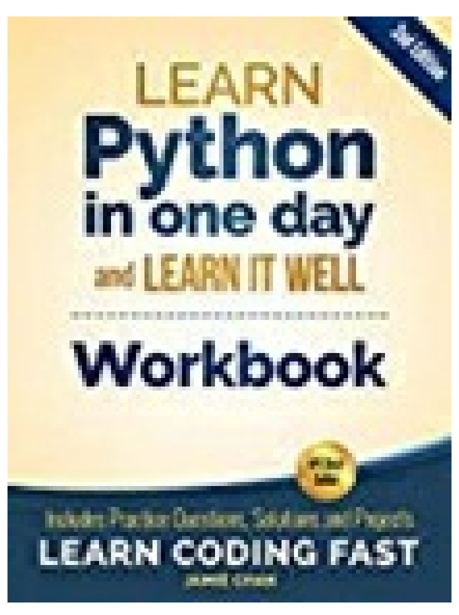
Learn SQL (using MySQL) in One Day and Learn It Well



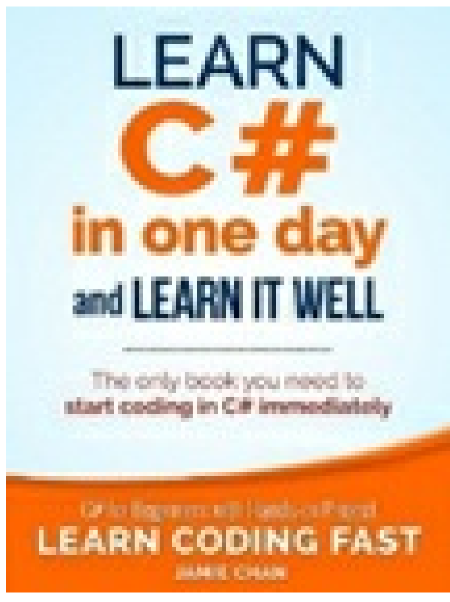
Learn CSS (with HTML 5) in One Day and Learn It Well



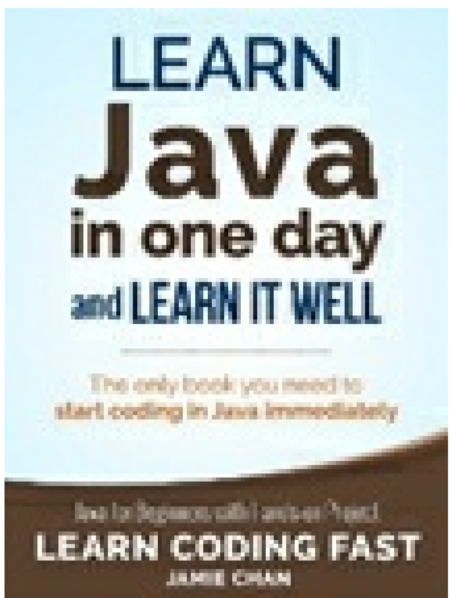
[Learn Python in One Day and Learn It Well (2nd Edition)]



[Learn Python in One Day and Learn It Well (Workbook)]



[Learn C# in One Day and Learn It Well]



[Learn Java in One Day and Learn It Well]

Contents

Chapter 1: Introduction to PHP
- 1.1 What is PHP?
- 1.2 Why Learn PHP?

Chapter 2: Installing XAMPP
- 2.1 Configuring php.ini
- 2.2 Important Links
- 2.3 Coding our first Web Page

Chapter 3: Basic PHP Tasks
- 3.1 Displaying Outputs
  - 3.1.1 echo
  - 3.1.2 print
  - 3.1.3 Escaping Characters
- 3.2 Duplicating Code
  - 3.2.1 include
  - 3.2.2 require
  - 3.2.3 include_once, require_once
- 3.3 Redirecting Users

Chapter 4: Constants, Variables, Data Types and Operators in PHP
- 4.1 Constants
- 4.2 Variables
- 4.3 Basic Data Types in PHP
- 4.4 Type Casting
- 4.5 Operators in PHP
  - 4.5.1 The Assignment Operator

4.5.2 Arithmetic operators
4.5.3 Combined Assignment Operators
4.5.4 Increment/Decrement operators
Chapter 5: More Data Types in PHP
5.1 Strings
5.1.1 Commonly used String Functions in PHP
5.2 Using Strings to Represent Dates
5.2.1 The strtotime() function
5.2.2 The date() function
5.2.3 Setting the timezone
5.3 Arrays
5.3.1 Creating an Array
5.3.2 Displaying the Content of Arrays
5.3.3 Adding Elements to Arrays
5.3.4 Deleting Elements from Arrays
5.3.5 Commonly used Array Functions in PHP
Chapter 6: Control Structures in PHP
6.1 Comparison operators
6.2 Logical Operators
6.3 Control Structures
6.3.1 If Statement
6.3.2 Ternary Operator
6.3.3 Switch Statement
6.3.4 For Loop
6.3.5 Foreach Loop
6.3.6 While Loop
6.3.7 Do-while Loop

6.4 Other Topics in Flow Control
6.4.1 Booleans
6.4.2 Break, Continue
6.4.3 Alternative Syntax
6.4.4 Displaying HTML code
Chapter 7: Functions
7.1 Defining our own Functions
7.2 Type Declaration
Chapter 8: PHP Superglobals
8.1 PHP Form Handling
8.1.1 The isset() function
8.1.2 get and $_GET
8.1.3 post and $_POST
8.1.4 Keeping The Values in The Form
8.1.5 Filtering User Input
8.1.6 Cross-Site Scripting
8.2 $_SESSION
8.3 $_COOKIE
Chapter 9: Object-Oriented Programming
9.1 What is OOP?
9.2 Writing our own class
9.3 Creating an Object
9.4 Accessing Class Members
9.5 Access Modifiers
9.6 Getter and Setter
9.7 Printing a String Representation of the Object
Chapter 10: Inheritance

10.1 Writing the Child Classes
10.2 Creating a Child Class Object
10.3 Access Modifiers Revisited
10.4 Overriding
Chapter 11: Interacting with a Database
11.1 The PDO library
11.2 Connecting to the Database
11.3 SQL Injection
11.4 Prepared Statements
11.5 Putting it all Together
Chapter 12: Managing Errors and Exceptions
12.1 Handling Exceptions
12.1.1 What is an exception?
12.1.2 try-catch-finally
12.1.3 Throwing Exceptions
12.1.4 Exception Handler
12.2 Handling Errors
12.2.1 What are errors?
12.2.2 Error Reporting Settings in PHP
12.2.3 Error Handler and Shutdown Function
12.3 Putting it All Together
Chapter 13: Project
13.1 About the Project
13.2 Acknowledgements and Requirements
13.3 Structure of the Project
13.4 Creating Database, User Account and Tables
13.5 Editing The classes Folder

- 13.5.1 Helper.php
- 13.5.2 Database.php
- 13.5.3 BlogReader.php
- 13.5.4 BlogMember.php
- 13.5.5 Admin.php

13.6 Editing The process Folder
- 13.6.1 p-index.php
- 13.6.2 p-admin.php
- 13.6.3 p-signup.php
- 13.6.4 p-write.php
- 13.6.5 p-read.php
- 13.6.6 messagecard.php

13.7 The includes Folder

13.8 Editing The phpproject Folder
- 13.8.1 UI_include.php
- 13.8.2 User Interface Files

13.9 Running the Code

Chapter 1: Introduction to PHP

Thank you for picking up this book. I'm so glad you chose this book, and I sincerely hope the book can help you master PHP and introduce you to the world of dynamic web programming.

In this book, we'll be covering most of the major topics in PHP. These topics are carefully chosen to give you a broad exposure to PHP while not overwhelming you with unnecessary details. We'll also be building a dynamic website together at the end of the book.

Excited?

Before we dive into PHP proper, note that this book requires you to have a basic understanding of HTML and MySQL.

If you are not familiar with these languages, you are encouraged to refer to my books "Learn CSS (with HTML 5) in One Day and Learn It Well" and "Learn SQL (using MySQL) in One Day and Learn it Well".

If you are familiar, let's move on.

1.1 What is PHP?

PHP is a general-purpose programming language used mostly for web development. Created by Rasmus Lerdorf in 1994, it allows developers to create dynamic web pages with ease.

For instance, developers can create a form in HTML and process it using PHP. Depending on the inputs entered into the form, developers can use PHP to display different outputs to users.

Most of the time, PHP is used as a server-side language. This means that PHP code is not processed on the user's computer (also known as a client).

In other words, when you access a PHP page on your browser, the code is not processed on your computer. Instead, your browser sends a request to a web server, which then processes the code and returns the result to the browser in the form of a web page.

More often than not, this web server is a remote computer where the PHP files are stored. For the web server to process PHP code, a special software known as the PHP interpreter needs to be installed.

We'll learn to set up our web server and install the PHP interpreter using a free software called XAMPP in the next chapter.

1.2 Why Learn PHP?

There are many reasons for learning PHP.

Firstly, PHP is one of the most widely used web programming languages and is used in many popular content management systems such as Wordpress, Drupal and Joomla. As such, the demand for PHP programmers is high. If you plan on working as a freelance developer, PHP is an essential skill to have.

Next, PHP is designed to be beginner-friendly and easy to learn. In addition, due to the popularity of the language, if you run into any issues with your PHP code, you can find help easily. A simple search on the internet will likely help you resolve most of the problems you face.

Last but not least, the syntax of PHP is very similar to other programming languages such as Java or C. Once you are familiar with PHP, you'll find it much easier to master other languages.

Ready to get started? Let's do it!

Chapter 2: Installing XAMPP

As mentioned in the previous chapter, we'll learn to set up our web server and install the PHP interpreter in this chapter. In addition, as we'll be using PHP to interact with a database later in this book, we need to set up our database server too.

To set up our web and database servers, we need to install three software: the Apache web server, the PHP interpreter and the MarieDB database server.

Installing all three software can be tedious if we do them one by one. Fortunately, some kind folks at Apache Friends created a free package that contains all the software we need. This package is known as XAMPP, which stands for Cross-platform (X), Apache web server (A), MarieDB database server (M), PHP (P) and Perl (P).

The MarieDB database server is a community-developed fork of the MySQL server. Although M in XAMPP officially stands for MarieDB, you'll see that XAMPP labels the database server as MySQL in the software. We'll follow this label and refer to the database server as MySQL in our book.

For detailed instructions on installing XAMPP and using it to set up our servers, check out https://learncodingfast.com/how-to-install-xampp-and-brackets.

Instructions are available on the accompanying site of this book so that whenever there are any changes to XAMPP, you can find the updated instructions on the site. This will ensure that you'll always get the latest installation instructions.

Besides installing XAMPP, you are strongly encouraged to install an advanced text editor to do your coding in. While you can definitely code in a basic text editor (such as Notepad), an advanced text editor offers features like syntax highlighting that makes your code easier to read and debug.

A recommended free editor is Brackets. For instructions on installing

Brackets, head over to https://learncodingfast.com/how-to-install-xampp-and-brackets as well.

2.1 Configuring php.ini

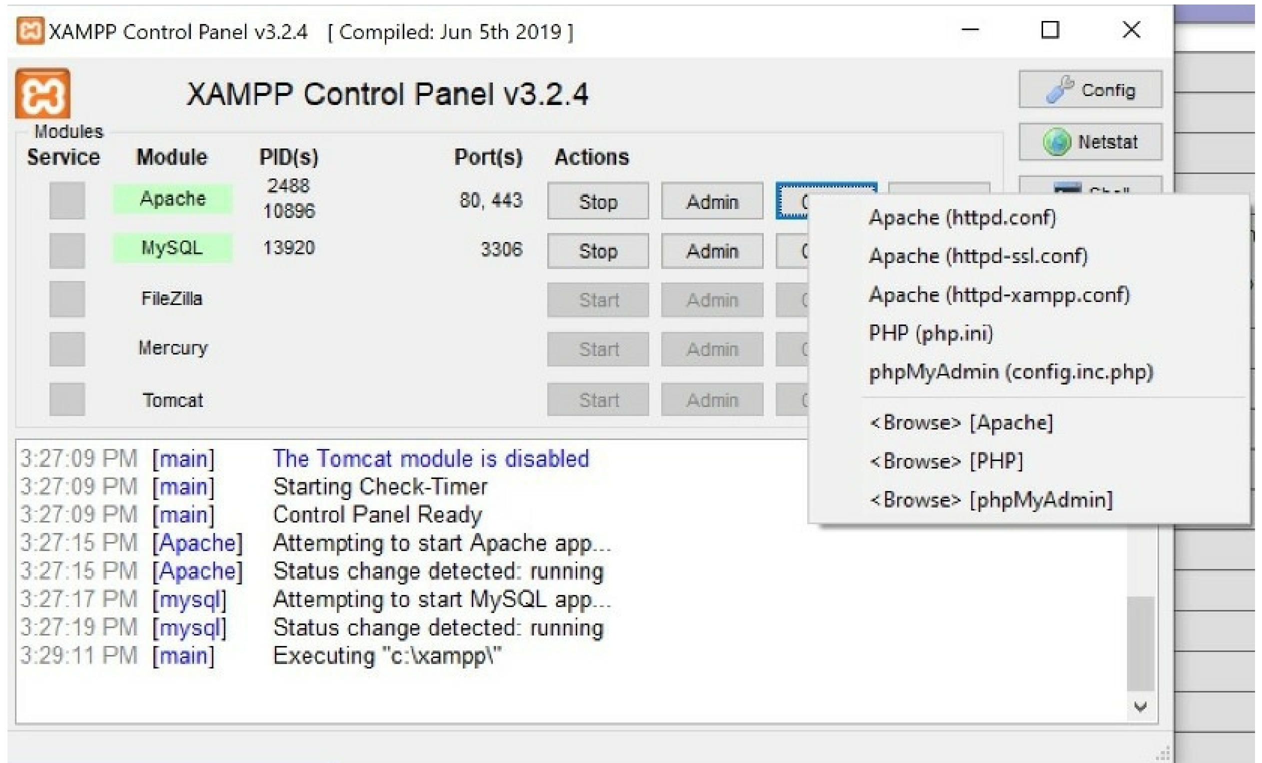
After you have installed XAMPP and Brackets on your computer, you are ready to start creating your dynamic website.

However, before we do that, we need to make some changes to the error settings on our PHP server. This ensures that whenever there's any mistake in our code, an error message will be displayed on the browser.

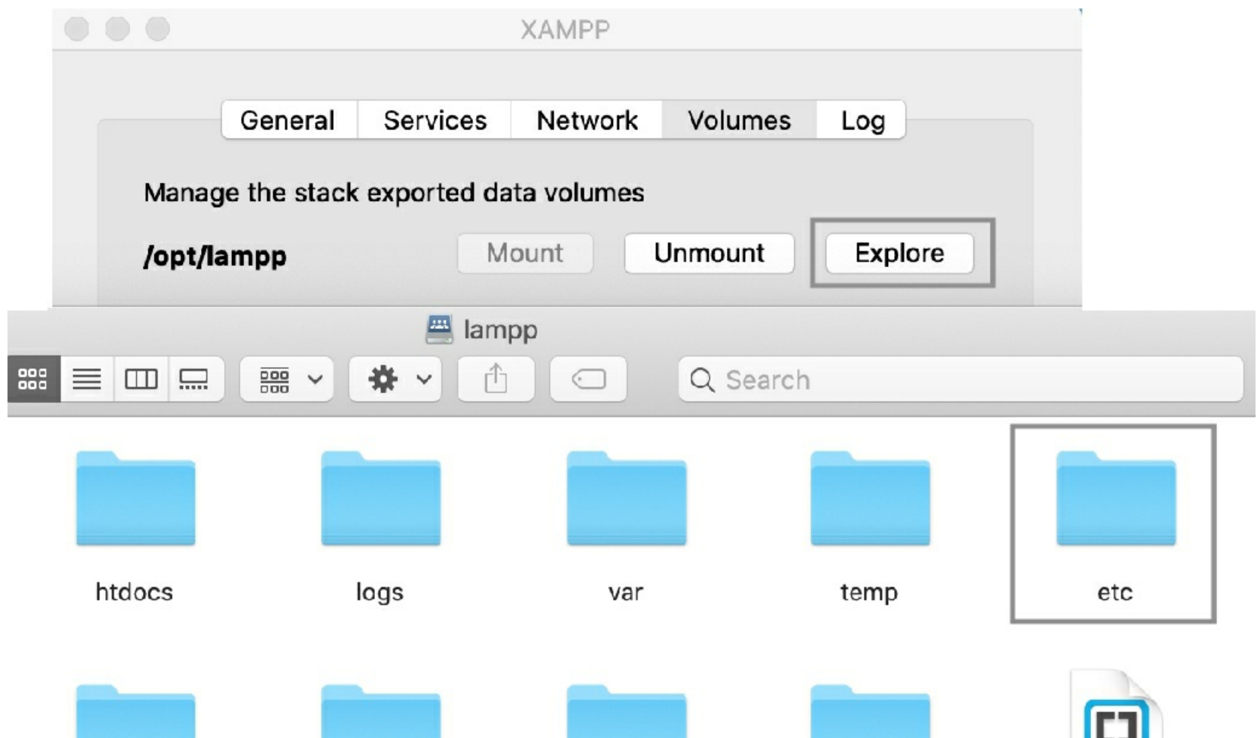
To configure these settings, the most direct way is to modify a file called **php.ini**. The instructions below are valid at the time of writing. If there are any changes to the instructions, you can find the updated instructions at https://learncodingfast.com/php.

To modify **php.ini**, we need to locate the file first.

To do that on Windows, launch XAMPP and click on the "Config" button for Apache. Next, select **php.ini**. The file will be opened in Notepad.



On Mac, launch XAMPP and click on the “Start” button in the “General” tab. Wait for the status to turn green. Next, select the “Volumes” tab and click on the “Mount” button. Finally, click on “Explore” and look for the “etc” folder. You’ll find **php.ini** inside this folder. Open it using Brackets.



Once **php.ini** is open, scroll to the bottom of the page and add the following lines to it:

```
error_reporting=E_ALL
display_errors=On
```

These lines are added to the bottom of the page so that if the settings have been set previously (in earlier parts of the **php.ini** file), the new lines will override the previous settings. We'll discuss these two settings in Chapter 12 later.

Next, save the file and restart Apache. On Windows, just stop the Apache server (if it is currently running) and start it again. On Mac, select the "Services" tab and click on "Apache". If Apache is currently running, click on the "Restart" button. Else, click on the "Start" button. Wait for Apache to restart and your settings will be updated.

You'll now get an error message whenever there's an error in your code. Depending on the PHP version you are using, some of the errors you encounter when testing the code in this book may differ from what is described. Don't worry about that. We'll discuss errors in depth in Chapter 12.

2.2 Important Links

Before proceeding to code our first web page, I would like to bring your attention to two important links.

The first is https://learncodingfast.com/php. This book uses lots of examples to illustrate different concepts. While you are encouraged to type out these examples in Brackets yourself, if you prefer not to, you can download the source code at the link above. The project files and any additional notes or updates to the book can also be found there.

Next, we have the errata page. While every effort has been made to ensure that there are no errors in the book and all source code has been tested extensively on multiple machines, if there are any errors we missed, you can find the errata at https://learncodingfast.com/errata.

2.3 Coding our first Web Page

Great! We are now ready to start coding. First, ensure that you have started Apache in XAMPP.

Note*: If you have any problems starting Apache, it is likely due to a port conflict. Follow the instructions at https://learncodingfast.com/how-to-install-xampp-and-brackets to resolve the conflict.*

Next, create a new file in Brackets and add the following code to it:

```
<!DOCTYPE html>
<html>
<head>
  <title>My first PHP page</title>
</head>
<body>
<h1>My first PHP page</h1>
<?php
  #Simple hello world page
  echo "Hello World!";
?>
```

```
</body>
</html>
```

If you are using an ebook reader, some lines above may wrap to the next line due to the limited width of the reader. If that occurs, try changing your device to the landscape mode or using a smaller font size. The same applies to all the other examples in this book.

Save the file above as **hello.php** to your **htdocs** folder. Any file that contains PHP code must be saved with a .php extension. A file that ends with a .php extension is also known as a PHP script.

You can find the **htdocs** folder inside the XAMPP (Windows) or LAMPP (Mac) folder.

htdocs is the root directory of your local web server. On Windows, to load the files stored in **htdocs**, you simply type *http://localhost/<filename>* into your browser's address bar. For instance, to load **hello.php**, type http://locahost/hello.php.

On Mac, the easiest way to load files stored in **htdocs** is to enable port forwarding. This can be done under the "Network" tab in XAMPP. Suppose you have enabled "localhost:8080->80(Over SSH)" in XAMPP, to load **hello.php**, type http://locahost:8080/hello.php into your browser's address bar.

Got it? You can refer to https://learncodingfast.com/how-to-install-xampp-and-brackets for more detailed instructions.

For ease of reference, from this point forward, I will use http://localhost to refer to your localhost. If you are using Mac, remember to add the port number to the URL (i.e., http://localhost:8080).

Let's analyze the code in **hello.php** now. Most of the code should look familiar to you except the following:

```
<?php
  #Simple hello world page
  echo "Hello World!";
?>
```

The `<?php … ?>` tags above are known as PHP tags; they tell the

server that code enclosed within should be treated as PHP code.

In our example, we have two simple lines of PHP code. The first line

```
#Simple hello world page
```

is known as a comment. Comments are written by programmers to explain their code to other programmers; these comments are ignored by the PHP interpreter.

To add a single line comment to our code, we precede the comment with `#` or `//`.

For instance,

```
# This is a comment
// This is also a comment
```

To add multiple lines comment, we enclose the comment in `/*...*/`.

For instance,

```
/*This is an example of a
multi-line comment. */
```

After the comment, we use the statement

```
echo "Hello World!";
```

to display the words "Hello World!" on the browser. This is known as an `echo` statement; we'll learn more about it in the next chapter.

Notice that we end the `echo` statement with a semicolon (;)? All statements in PHP must end with a semicolon. This is similar to how we end sentences with a period (.) in English.

If you load **hello.php** now, you'll get

My first PHP page

Hello World!

as the output.

That’s it. We have just coded our first PHP web page. Simple? Great! Let’s move on.

Chapter 3: Basic PHP Tasks

In the previous chapter, we installed XAMPP and Brackets and coded our first PHP script.

In this chapter, we're going to learn to perform three very fundamental tasks in PHP - displaying outputs to users, duplicating code across multiple pages and redirecting users.

3.1 Displaying Outputs

Before we learn to display outputs, let's create a PHP file to serve as a template for us to test the PHP statements in this section.

Create a file in Brackets and save it as **output.php** to your **htdocs** folder. Add the following code to **output.php**:

```
<?php
  //Add code here
```

Whenever you want to test any PHP statement in this section, you can add the statement to **output.php** (after the `<?php` opening tag) and load it in your browser to check the output.

Notice that we did not add a closing tag (`?>`) to **output.php**? This is because **output.php** only contains PHP code.

According to the standard recommended by the PHP Framework Interop Group, any PHP script that only contains PHP code should not end with a closing tag. This is to prevent unwanted whitespaces at the end of the file, which can cause issues such as an "Headers already sent" warning when modifying response headers.

Do not worry if the statement above does not make any sense; we'll learn about the "Headers already sent" warning later. For now, let's move on to learn how to display outputs in PHP.

3.1.1 echo

The easiest way to display outputs in PHP is to use an `echo`

statement. To do that, we enclose the text that we want to display in a pair of matching quotation marks after the `echo` keyword. The `echo` keyword is not case sensitive. Hence, `echo`, `Echo` or `ECHO` will all work.

Using an `echo` statement is straightforward. We did that in Chapter 2 when we used it to display the words “Hello World!”.

In addition to using `echo` statements to display simple text messages, we can use them to display text with HTML markup. For instance, we can write

```
echo 'PHP is fun<BR>and easy!';
```

This gives us

```
PHP is fun
and easy!
```

as the output; the text “and easy!” is displayed on the second line due to the <BR> tag.

We can also combine multiple messages in a single `echo` statement. To do that, we separate the messages with commas. For instance,

```
echo "ABCD", "EFGH";
```

gives us

```
ABCDEFGH
```

as the output. Combining messages is useful when our message is more complex, such as when the message includes variables and functions. We’ll learn about variables and functions in later parts of the book.

Last but not least, we can use an `echo` statement with or without parentheses. You may have seen some code that does the following:

```
echo ("ABCD");
```

This is fine in most cases. However, if you use parentheses, you are not allowed to combine multiple messages in a single `echo` statement.

For instance,

```
echo ("ABCD", "EFGH");
```

will give us an error. In most cases, you are better off using `echo` statements without parentheses.

3.1.2 print

Besides using `echo` statements to display outputs, we can use `print` statements.

`print` statements are VERY similar to `echo` statements. Let's look at an example:

```
print "My name is Brody.";
```

gives us

```
My name is Brody.
```

as the output. As you can see, the `print` statement works just like an `echo` statement. While there are some differences between them, in most cases, there is no need to use the `print` statement.

In PHP, it is fairly common that there is more than one way to perform a certain task. This is one of the main complaints developers have about PHP, as it complicates things unnecessarily. In most cases, the different ways are very similar to each other. In order not to overwhelm you with too many options, I'll stick with one option in this book and briefly mention other options if necessary. In this book, I'll be using `echo` statements to display outputs.

3.1.3 Escaping Characters

Next, let's learn to escape characters in PHP.

In the previous sections, we learned to enclose text in quotation marks when we use an `echo` (or `print`) statement. What happens if we want to display an apostrophe?

For instance, suppose we have the following statement:

```
echo '<BR>Today is Friday, we're going to the zoo.';
```

We’ll get an error when we try to run the statement above as PHP does not differentiate between an apostrophe and a quotation mark by default. Hence, it treats the apostrophe in the word “we’re” as a closing quotation mark. In other words, it sees the statement as

```
echo '<BR>Today is Friday, we'
```

and expects a semicolon or comma after the apostrophe. When that does not happen, PHP gets confused and gives us an error.

There are two ways to deal with the error above. The first is to use double quotation marks to enclose the text and rewrite the statement as

```
echo "<BR>Today is Friday, we're going to the zoo.";
```

As PHP uses a double quotation mark to close another double quotation mark, the apostrophe in “we’re” is not mistakenly interpreted as a closing quotation mark.

Alternatively, we can escape the apostrophe in the original statement by preceding it with a backslash character (`\`).

Escaping a character alters the meaning of the character. Without the backslash character, the apostrophe is interpreted as a closing single quote. With it, it is treated as an apostrophe meant to be displayed on the browser.

For instance, to display the apostrophe in the previous example, we can use the statement below:

```
echo '<BR>Today is Friday, we\'re going to the
zoo.';
```

This gives us the following output:

```
Today is Friday, we're going to the zoo.
```

Other common uses of the backslash character include:

<u>Using it to display a double quotation mark</u>

```
echo """;
```

gives us an error, while

```
echo "\"";
```

gives us

```
"
```

Using it to display the backslash character itself

```
echo "\";
```

gives us an error, while

```
echo "\\";
```

gives us

```
\
```

3.2 Duplicating Code

Next, let’s move on to learn how to duplicate code in our PHP scripts.

In PHP, generally speaking, scripts are independent of one another. Hence, code written in one PHP script (say, index.php) is not available once users navigate to another page.

If we want multiple pages on our website to have the same HTML or PHP code, we need to use the `include` statement.

3.2.1 include

Suppose you want multiple pages on your website to have the same `<h1>` heading, you do not need to copy and paste the code into all the pages manually. Instead, you can use the `include` statement.

Let’s try that out. Create a file called **heading.html** (in **htdocs**) and add the following code to it:

```
<h1>Welcome to PHP</h1>
```

Next, create another file called **includedemo.php** and save it to

htdocs too. Add the following code to **includedemo.php**:

```
<!DOCTYPE html>
<html>
<body>
  <?php

    include 'heading.html';
  ?>
</body>
</html>
```

Notice the statement

```
include 'heading.html';
```

in **includedemo.php**?

When PHP sees this statement, it goes to **heading.html**, “copies” the code inside, and “pastes” it into **includedemo.php** for us automatically. This saves us the trouble of having to copy and paste the code ourselves.

If you launch **includedemo.php** now, you’ll see the words “Welcome to PHP” displayed as a `<h1>` heading. This is due to the code in **heading.html**. Got it?

To use an `include` statement, we need to give PHP the **path** of the file we want it to include. This is similar to what we do in HTML when we link CSS or other external files to our webpages. In our example above, as **heading.html** is stored in the same folder as **includedemo.php**, we simply give it the filename.

If **heading.html** is stored in another folder (say in a folder called “includes” inside **htdocs**), we need to use the statement below:

```
include 'includes/heading.html';
```

3.2.2 require

Besides the `include` statement, PHP also has the `require` statement. The difference between the two is that `require` produces

a fatal error and stops the script if PHP fails to find the file required.

`include`, on the other hand, will only produce a warning without stopping the script.

If the content you want PHP to "copy and paste" into your file is crucial to the correct execution of the page, you should use the `require` statement. Otherwise, you can use the `include` statement.

3.2.3 include_once, require_once

Both the `include` and `require` statements come with a `_once` version.

As the names suggest, the `include_once` and `require_once` statements only "copy and paste" the code into the file once. These statements are useful in cases where the same file may be included more than once accidentally (for instance, when the script is so long you have lost track of whether you have included the file previously).

If you add another

```
include 'heading.html';
```

statement to **includedemo.php** (<u>before</u> the `?>` tag) and load the page, you'll get "Welcome to PHP" displayed as a `<h1>` heading **twice**.

On the other hand, if you change the `include` statements to

```
include_once 'heading.html';
include_once 'heading.html';
```

you'll only get the `<h1>` heading displayed once.

3.3 Redirecting Users

Great! We've come to the last section of this chapter. In this section, we'll learn to redirect users using PHP.

In HTML, we know that we can use a `<meta>` tag to redirect users to another page. In PHP, we use a built-in function called `header()`.

First off, what is a function?

A function is a block of code that performs a certain task. For an analogy, think of the functions in MS Excel. If we want to add the values of some cells in a spreadsheet, we can use a built-in function called sum(). This function is pre-written by other programmers and comes included with Excel.

Similar to the pre-written functions in Excel, PHP comes with many built-in functions. Most functions require us to provide certain information for them to perform their tasks. In PHP, we provide this information by enclosing them inside a pair of parentheses (in a pre-defined order) after the function name. The information that we pass to the function is known as arguments.

To use the `header()` function, we need to pass the word “Location”, followed by a colon and the redirect URL to the function. Using a function is also known as calling the function.

To call the `header()` function, add the following code to the start of **includedemo.php** (before the `<!DOCTYPE html>` tag):

```
<?php
  header('Location: http://example.com');
?>
```

If you load **includedemo.php** now, you’ll be redirected to http://example.com. Got it?

When you use the `header()` function, note that it must be called before your script generates any output to the browser, either by normal HTML tags, blank lines, or from PHP. If there’s any output before the function is called, the function will not work. For instance, it will not work in the following two examples:

Example 1

```
<?php
  echo 'Hello';
  header('Location: http://example.com');
?>
```

Example 2

```
<p>This is some text.</p>

<?php
  header('Location: http://example.com');
?>
```

In the first example, output is generated by the `echo` statement (`echo 'Hello';`) before the `header()` function is called.

In the second example, output is generated by the two lines of code before the `<?php` tag. These two lines generate an HTML paragraph and an empty line before the `header()` function is called.

If your script generates any output before the `header()` function is called, you'll get the "Headers already sent" warning mentioned previously in Chapter 3.1 and users will not be redirected. The code may work when you are working on your local computer. However, when you upload it to your hosting company's server, it will fail to work. Got it?

Chapter 4: Constants, Variables, Data Types and Operators in PHP

Cool! We've come to Chapter 4. In this chapter, we are going to learn about constants and variables. In addition, we'll talk about the data that variables store and the different operations we can perform on them.

Before we proceed, you may want to create a new PHP script called **chap4.php** and use it to test the examples in this chapter.

Done? Great! Let's move on. We'll start with constants.

4.1 Constants

Constants in PHP are similar to constants in Mathematics. Recall the number 3.14159 (rounded to 5 decimal places) we learned in Mathematics? This number is called Pi (π), and its value never changes.

Much like Pi in Mathematics, we can define our own constants in PHP and give them names for easy reference. To do that, we use a built-in function called `define()`. Once we define a constant, we are not allowed to change its value.

To use the `define()` function, we need to pass two arguments (separated by a comma) to the function - the name of the constant (passed with quotation marks) and the value.

The name of a constant is case-sensitive by default and can consist of letters, numbers, or underscores. However, it must start with a letter or an underscore. It is a convention to use all uppercase when naming constants.

Let's look at an example.

```
define("BASIC_MEMBER", 1112020);
echo BASIC_MEMBER;
echo '<BR>';
```

```
define("BASIC_MEMBER", 16932);
echo BASIC_MEMBER;
```

Here, we define a constant called `BASIC_MEMBER` and specify its value as `1112020`. Next, we reference it using its name and use an `echo` statement to echo its value.

After echoing its value, we try to redefine `BASIC_MEMBER` and change its value to `16932`. Finally, we echo its value again. If you run the code above, you'll get the following output:

```
1112020
```

```
Notice: Constant BASIC_MEMBER already defined in ...
1112020
```

If you study the output carefully, you'll see that we did not manage to change the value of `BASIC_MEMBER` after defining it the first time. Hence, its value remains as `1112020` when we echo it a second time. Got it? Good!

In the example above, note that we did not enclose `BASIC_MEMBER` in quotation marks when we echo its value. If we use quotation marks (e.g., `echo 'BASIC_MEMBER';`), we'll get

```
BASIC_MEMBER
```

as the output instead, which is not what we want. We enclose the names of constants in quotation marks only when we define them (i.e., when we call the `define()` function).

4.2 Variables

Next, let's move on to variables.

Variables are used to store data in our programs. They have names, and their values can be changed when necessary.

Like constants, the names of variables need to follow certain naming rules.

Firstly, a variable name must start with the `$` symbol. After this

symbol, the name cannot start with a number and can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).

Names like $userName, $age1 and $user_email are fine while names like $4ever (starts with a number), $unit+NY (includes the + sign) and $var 1 (includes a space) are not.

Next, variable names are case sensitive; $userName is not the same as $username.

Finally, variable names should be meaningful. Using variables like $x, $y and $z to store the name, age and password of your user will make your code confusing and less intuitive. While not mandatory, you should definitely use more meaningful names such as $name, $age and $password to make your code more readable.

There are two commonly used styles when naming a variable in PHP. We can use either camel case or underscores. Camel case is the practice of writing compound words in mixed case, beginning each word (except the first) with an uppercase (e.g., $thisIsAVariable). Another common style is to use underscores (_) to separate the words (e.g., $this_is_a_variable).

Choosing one style over the other is a matter of personal preference. What is more important is to be consistent throughout your code. In this book, we'll be using camelCase for variables.

To declare a variable in PHP, we write something like:

```
$x = 7;
```

Here, we declare a variable called $x and give it the value 7. Giving a value to a variable is known as assigning a value to it.

Assigning an initial value to a variable when declaring it is known as initializing the variable. This is not mandatory in PHP. However, it is considered bad practice not to do so as it can lead to unexpected behavior in the script.

When we declare and initialize a variable, PHP allocates a certain area on the server's storage space to store this data. You can

subsequently access and modify the data by referring to it by its name. For instance, to display the value of `$x`, we write

```
echo $x;
```

This gives us `7` as the output. If we want to change the value of `$x`, we simply assign a new value to it. For instance,

```
$x = 5;
echo $x;
```

updates the value of `$x` and gives us `5` as the new output.

In the examples above, note that we did not enclose `$x` in quotation marks when we use the `echo` statement.

In general, we do not enclose the names of constants, variables or functions in quotation marks when we use the `echo` statement. This is because the `echo` statement treats anything enclosed in quotation marks as text or HTML code and echoes it literally. For instance, if we write

```
echo '$x';
```

we'll get

```
$x
```

as the output. The only exception is when it comes to variable names enclosed in **<u>double</u>** quotation marks.

If we enclose a variable name in double quotation marks, the `echo` statement does not echo the variable name literally. Instead, it replaces the name with the value of that variable and echoes the value. For instance, if we write

```
echo 'The value is $x.';
echo '<BR>';
echo "The value is $x.";
```

we'll get

```
The value is $x.
```

```
The value is 5.
```

as the output. When we use single quotation marks, the `echo` statement outputs the text literally. In contrast, when we use double quotation marks, it replaces `$x` with its value. Got it?

4.3 Basic Data Types in PHP

Next, let's talk about data types in PHP.

Variables in PHP can be used to store different types of data. The type of data a variable stores is known as its data type. In this section, we'll discuss three basic data types - `int`, `float` and `bool`.

int

`int` refers to integers, which are numbers with no decimal parts (such as `-5`, `0`, `4` and `7`).

This data type has an upper and lower boundary, both of which are platform-dependent. In other words, it can only store numbers within a certain range. Integers that exceed these bounds (i.e., very large or very small numbers) will be interpreted as floats.

float

`float` refers to floating-point numbers, which are numbers with decimal parts or numbers in exponential form.

Examples include `5.79`, `-3.56` and `7E11` (i.e., 7×10^{11}).

bool

`bool` refers to boolean and is a special data type that can only store one of two values - `true` or `false`.

These two values are not case-sensitive. In other words, `true`, `TRUE` and `True` all refer to the same value. Similarly, `false`, `FALSE` and `False` refer to the same value.

The `bool` data type may seem redundant at the moment. Its significance will be apparent when we discuss comparison operators and condition statements in Chapter 6.

PHP is considered to be a loosely typed language. When we declare a variable, we do not need to state its data type.

To declare a variable that stores integers, we simply assign an integer to it. To declare one that stores floats or booleans, we assign a float or boolean to it respectively.

If we have the following variables

```
$x = 5;
$y = 2.1;
$z = true;
```

`$x` is an integer, `$y` is a float, and `$z` is a boolean.

To verify the data type of a variable, we can use a built-in function called `var_dump()`. This function requires us to provide the variable name and gives us the data type and value of the variable. The following statements

```
var_dump($x);

var_dump($y);

var_dump($z);
```

give us

```
int(5) float(2.1) bool(true)
```

as the output.

4.4 Type Casting

In the previous section, we learned about three basic data types in PHP - `int`, `float` and `bool`. We can easily convert one data type to another. This is known as type casting.

To cast a value/variable to an integer, we add `(int)` or `(integer)` in front of the original value/variable.
To cast to a boolean, we add `(bool)` or `(boolean)`.
To cast to a float, we add `(float)` or `(double)`.

Let's look at an example:

```
$p = (int)4.6;
var_dump($p);
```

Here, we declare a variable called `$p` and assign `(int)4.6` to it. Next, we use the `var_dump()` function to display the data type and value of `$p`.

If we run the code above, we'll get

```
int(4)
```

as the output. The value of `$p` is `4` as PHP casts a float (`4.6`) to an integer by truncating the decimal part of the float.

Besides casting values or variables to integers, floats or booleans, we can also cast them to strings, arrays or objects. We do that by adding `(string)`, `(array)` and `(object)` in front of the values respectively. We'll discuss these advanced data types in subsequent chapters.

4.5 Operators in PHP

PHP comes with many operators that we can use with variables. Let's discuss the assignment operator first.

4.5.1 The Assignment Operator

Previously, we learned to assign values to variables using the `=` sign. This sign is known as an assignment operator and is different from the equals sign we learned in Mathematics.

In PHP (and most programming languages), an assignment works from right to left. In other words, we assign the value (or variable) on the right side of the assignment operator to the variable on the left. Hence,

```
$x = 7;
```

is not the same as

```
7 = $x;
```

While both statements are acceptable in Mathematics, the second statement is not acceptable in PHP.

In the first statement, we assign `7` to `$x`. This is all right and will not give us any error. On the other hand, in the second statement, we assign `$x` to `7` (from right to left). This does not make any sense (as `7` is a constant) and will give us an error.

Always remember that assignment works from right to left.

4.5.2 Arithmetic operators

Next, let's talk about arithmetic operators.

These operators are for performing basic arithmetic operations. They include operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulo (`%`) and exponentiation (`**`).

Suppose you have

```
$x = 5;
$y = 2;
```

`$x + $y` gives us `7`,
`$x - $y` gives us `3`,
`$x*$y` gives us `10`,
`$x/$y` gives us `2.5`,

```
$x%$y
```

gives us `1` because the modulo operator gives us the remainder when `5` is divided by `2`, and

```
$x**$y
```

gives us `25` because the exponentiation operator gives us the value of `5` raised to the power of `2`.

4.5.3 Combined Assignment Operators

Besides the set of arithmetic operators mentioned above, we have a set of operators that combines arithmetic operations with assignment. Suppose you have

```
$x = 5;
```

and you want to add `3` to `$x`, you can do it as follows:

```
$x = $x + 3;
```

An assignment always happens from right to left. Hence, the expression on the right (`$x + 3`) is evaluated first to give us `5+3`. This value is then assigned back to `$x`, which is the variable on the left. In other words, the value of `$x` becomes `8`.

In addition to writing

```
$x = $x + 3;
```

we can use the `+=` operator. This operator is a shorthand that combines the assignment operator with the addition operator. If you write

```
$x = 5;
$x += 3;
echo $x;
```

you'll get `8` as the output.

`$x += 3` simply means `$x = $x + 3`.

Besides the `+=` operator, we have the `-=` operator, which combines the `-` operator with the assignment operator. The same applies to all the arithmetic operators mentioned in the previous section.

Hence, `$x -= 4` is the same as `$x = $x - 4`, `$x *= 2` is the same as `$x = $x*2`, etc.

4.5.4 Increment/Decrement operators

Last but not least, let's talk about the increment (`++`) and decrement (`--`) operators. These operators increase or decrease the value of a variable by 1 and return the new value.

Let's try some code (line numbers are added for reference).

```
1 $q = 3;
```

```
2 echo "<BR>$q<BR>";
3 echo ++$q;
```

If you run the code above, you'll get the following output:

```
3
4
```

Line 2 uses the `echo` statement to output the original value of `$q`. Line 3 does the following:

1. Increment the value of `$q` by 1
2. Echo the new value of `$q`

Hence, we get `4` as the output.

The `++` operator is known as the increment operator. It can be placed in front of (known as pre-increment) or behind (post-increment) the variable name. Try changing line 3 above to

```
echo $q++;
```

and rerun the code, you'll get

```
3
3
```

as the output this time. Surprised? This is because line 3 does the following now:

1. Echo the original value of `$q`.
2. Increment the value of `$q` by 1

In other words, the order of execution of the two tasks (increment and echo) is reversed. The increment is done <u>after</u> the `echo` statement is executed (hence the name post-increment). To prove that `$q` is indeed incremented, you can echo its value once more after line 3. You'll get `4` as the output this time. Got it?

In addition to the increment operator, we have the decrement operator (`--`). This operator consists of two minus signs and decreases the

value of a variable by 1. It can also be used as a pre-decrement or post-decrement operator.

Chapter 5: More Data Types in PHP

In the previous chapter, we looked at some of the basic data types in PHP. In this chapter, we'll discuss two more data types - strings and arrays. You can create a new file named **chap5.php** to test the examples in this chapter.

5.1 Strings

A string refers to a piece of text. In PHP, strings must be enclosed in single or double quotation marks. For instance, if we write

```
$msg = 'Hello';
$greeting = "Good Morning";
$emptyStr = "";
```

`$msg` and `$greeting` store the strings `'Hello'` and `"Good Morning"` respectively.

`$emptyStr`, on the other hand, stores a special string known as an empty string. This is because there is no text enclosed within the pair of double quotation marks we assign to it.

In PHP, you can combine multiple strings using the concatenate operator (.). Let's look at an example:

```
$areacode = "(208)";
$contact = '+1' . $areacode . '1234567';
echo $contact;
```

Here, we use the concatenate operator twice to concatenate the string `'+1'`, the variable `$areacode` and the string `'1234567'` to form a new string. I've added spaces before and after the concatenate operators so that you can see the three components more clearly; you do not need to add these spaces.

After concatenating, we assign the new string to a variable called `$contact` and use the `echo` statement to echo its value.

If you run the code above, you'll get

```
+1(208)1234567
```

as the output.

5.1.1 Commonly used String Functions in PHP

Next, let’s discuss some of the commonly used string functions in PHP. PHP comes with a large number of built-in functions for working with strings. Most functions return a result after performing their tasks. We can assign the result to a variable or use the `echo` statement to display the result directly.

A function may have different variations as some functions allow us to provide optional arguments to modify the function’s behavior. The examples below discuss the most common and useful approach to using each function.

strlen()

The `strlen()` function gives us the length of a string.

Example:

```
$str1 = 'Good Day!';
echo strlen($str1);
```

Output:

```
9
```

Here, we declare a variable called `$str1` and assign the string `'Good Day!'` to it. Next, we pass `$str1` to `strlen()` and use the `echo` statement to display the function's result directly.

We get `9` as the output because 4 (`'Good'`) + 1 (`' '`) + 3 (`'Day'`) + 1 (`'!'`) = 9.

strtolower(), strtoupper()

The `strtolower()` and `strtoupper()` functions convert a string to lowercase and uppercase respectively and return the new string. <u>The original string is not changed.</u>

Example:

```
$str2 = 'Hello World';
$str3 = strtolower($str2);
$str4 = strtoupper($str2);
echo '<BR>'.$str2;
echo '<BR>'.$str3;
echo '<BR>'.$str4;
```

Here, we declare a variable called `$str2` and assign the string `'Hello World'` to it. Next, we pass `$str2` to `strtolower()` and `strtoupper()` and assign the results to `$str3` and `$str4` respectively.

Finally, we use three `echo` statements to echo the values of `$str2`, `$str3` and `$str4`. In each case, we concatenate `'<BR>'` with the variable so that the output will start on a new line.

If you run the code above, you'll get

```
Hello World
hello world
HELLO WORLD
```

as the output.

trim()

The `trim()` function removes whitespaces from the front and end of a string by default and returns the new string. You can specify other characters for it to remove by passing a second optional argument to the function.

Example 1:

```
$str5 = ' is ';
echo 'PHP'.$str5.'Fun<BR>';
echo 'PHP'.trim($str5).'Fun<BR>';
```

Output:

```
PHP is Fun
PHPisFun
```

Here, we first concatenate `$str5` with the strings `'PHP'` and `'Fun<BR>'` and use an `echo` statement to echo the resulting string directly. We get “PHP is Fun” as the output.

Next, we use the `trim()` function to remove whitespaces from `$str5` and concatenate it with `'PHP'` and `'Fun<BR>'` again. We get “PHPisFun” as the output this time.

Example 2:

```
$str6 = '**Hello**World***';
echo trim($str6, '*');
```

Output:

```
Hello**World
```

Here, we pass `'*'` to the `trim()` function as the second argument. As a result, asterisks (instead of whitespaces) are removed from the front and back of `$str6` and we get “Hello**World” as the output.

substr()

The `substr()` function returns a substring. To use this function, we need to pass two arguments to it - the string to extract the substring from and the position to start extracting it. We can also pass a third argument to specify the length of the substring to extract. If we do not provide this argument, the function extracts the substring starting from the specified starting position to the end of the string.

Positions in strings start from 0, not 1. For instance, if we have a string `'ABCDEF'`, `'A'` is at position 0, `'B'` is at position 1 and so on.

Positions can also be negative. If it is negative, it is counted from the back of the string. In `'ABCDEF'`, `'F'` is at position -1, `'E'` is at position -2 etc.

Example:

```
$str7 = 'ABCDEF';
echo substr($str7, 2).'<BR>';
echo substr($str7, -3).'<BR>';
```

```
echo substr($str7, 2, 1);
```

Output:

```
CDEF
DEF
C
```

Here, we declare a variable called `$str7` and assign the string `'ABCDEF'` to it. Next, we pass `$str7` to `substr()` thrice to extract different substrings from it.

In the first two examples, the `substr()` function extracts substrings starting from (and including) positions 2 and -3 respectively. Hence, we get “CDEF” and “DEF” as the outputs.
In the third example, we specify the desired length of the substring by passing a third optional argument to `substr()`. As we specify the desired length to be 1, we get “C” as the output. Got it?

We’ve covered several string functions in PHP. For a full list of all the string functions available, you can check out the page https://www.php.net/manual/en/ref.strings.php.

5.2 Using Strings to Represent Dates

Next, let’s learn to use strings to represent dates. To do that, we use a built-in function called `strtotime()`.

5.2.1 The strtotime() function

As the name suggests, the `strtotime()` function converts a string to time. It accepts a string describing a specific date (and time) and tries to convert that to a timestamp.

If you pass a string to the function that it is unable to convert, it returns `false`. On the other hand, if you pass a string that it is able to convert, it parses the string and returns the UNIX timestamp.

A UNIX timestamp is an integer that gives us the number of seconds that have elapsed since January 1 1970 00:00:00 UTC.

For instance, if you run the statement

```
echo strtotime("next Monday");
```

you'll get something similar to

```
1587340800
```

This gives the UNIX timestamp for next Monday (at the time of writing).

The `strtotime()` function is quite smart and is able to convert various English textual datetime descriptions into UNIX timestamps. For instance, it has no problem converting strings like `"now"`, `"tomorrow"`, `"next Monday"`, `"15 Nov 2019"` or `"+1 week"`.

However, while the function is easy to use, you'll probably agree that the timestamp it returns is not very useful in most cases. If you want to get a more meaningful format for a date or time, you can use another built-in function called `date()`.

5.2.2 The date() function

The `date()` function accepts an optional timestamp and formats the timestamp into a more readable string. If the timestamp is not provided, it formats the current timestamp.

Suppose we want to format a timestamp that is 10 hours from the current time, we can use the statement below:

```
echo date('d-M-Y', strtotime("+ 10 hours"));
```

Here, we pass two arguments to the function.

The first argument specifies how we want the timestamp to be formatted. To do that, we use a string consisting of punctuation marks and **predefined characters** (as defined on https://www.php.net/manual/en/function.date.php).

In our example, we use the string `'d-M-Y'`.

The character 'd' indicates that we want the day to be represented as a two-digit number, using a leading zero if necessary (e.g., 3 is output as 03). The characters 'M' and 'Y' indicate that we want the month and year to be represented as a three-letter text (e.g., Jan) and a four-digit

number respectively. Last but not least, the punctuation mark ‘-’ indicates that we want the day, month and year to be separated by hyphens.

After specifying the desired format, we pass a second argument (`strtotime("+ 10 hours")`) to the `date()` function. This argument is optional and specifies the date we want the function to format.

If you run the `echo` statement above, you’ll get an output similar to

```
03-Apr-2020
```

This gives the date 10 hours from now (at the point of writing). Got it?

5.2.3 Setting the timezone

The sections above discussed two of the many built-in date/time functions in PHP. While most of these functions are easy to use, do note that their results are affected by the timezone set in our PHP server.

For instance, while

```
echo date('d-M-Y', strtotime("+ 10 hours"));
```

gives us `03-Apr-2020` in one timezone, it may give us `04-Apr-2020` in another.

To avoid any discrepancy in results, it is strongly recommended that you manually set the timezone in your PHP server if you want to use any date/time function in PHP.

To do that, you need to modify the `date.timezone` setting in **php.ini**. Refer to Chapter 2 for instructions on locating this file.

Once you have located the file, open it, and scroll to the bottom of the page.

Next, head over to https://www.php.net/manual/en/timezones.php for a list of valid timezone identifiers. Say you want PHP to use the New York timezone, you need to add the line

```
date.timezone=America/New_York
```

to the bottom of **php.ini**. Once that is done, save the file, restart Apache, and the setting will be updated. Got it?

If you do not have permission to update the **php.ini** file, you can use the `date_default_timezone_set()` function. This function takes a timezone identifier and sets the default timezone used by all date/time functions in a PHP script.

For instance, to set the default timezone to the New York timezone, simply add

```
date_default_timezone_set('America/New_York');
```

to the start of your PHP script. This affects the timezone for that particular PHP script. If you want another PHP script to use the same timezone, you have to set it in that script too.

5.3 Arrays

Great! Now that we are familiar with strings and dates, let's move on to another commonly used data type in PHP - arrays.

An array is a special data type that allows us to store related values together as a single variable.

For instance, suppose you want to store the test scores of 5 students. Instead of storing them as `$marks1`, `$marks2`, `$marks3`, `$marks4` and `$marks5`, you can store them as an array.

5.3.1 Creating an Array

There are a few ways to create an array in PHP; the most common way is to use the `array()` function.

Example 1

```
$firstArr = array();
```

This creates an empty array.

Example 2

```
$secondArr = array(11, 16, 4, 9, 12);
```

This creates an array with 5 elements (i.e. 5 values).

The array created is known as an indexed array, as each element in the array has an index.

Indexes start from 0. In other words, the first element (`11`) has an index of 0, the second (`16`) has an index of 1, and so on.

To access the individual elements in the array, we use its index and a pair of square brackets. For instance, if you write

```
echo $secondArr[3];
```

you are accessing the 4th element; you'll get `9` as the output. You can also use the index to update the value of an element.

```
$secondArr[3] = 20;
```

updates the 4th element to `20`.

`$secondArr` becomes `(11, 16, 4, 20, 12).`

Example 3

In addition to storing numbers, arrays can be used to store other data types. In the example below, `$fruitsArr` is used to store strings.

```
$fruitsArr = array('Apple', 'Banana', 'Coconut');
```

Example 4

Besides indexed arrays, we can create associative arrays. An associative array is one where each value in the array is associated with a key. For instance, in the example below, the value `16` is associated with `'Jane'` (known as its key).

```
$assocArr = array(
  'Peter' => 11,
  'Jane' => 16,
  'Paul' => 12
);
```

To access the values in an associative array, we use its key. If we write

```
echo $assocArr['Paul'];
```

we'll get `12` as the output.

Example 5

Next, arrays can be used to store arrays. This is known as a multidimensional array.

```
$simpleMDArr = array(
  array(1, 2, 1, 4, 5),
  array(0, 5, 1, 3, 4),
  array(4, 1, 7, 8, 9)
);
```

In the example above,

`$simpleMDArr[0]` stores the array `(1, 2, 1, 4, 5)`,
`$simpleMDArr[1]` stores `(0, 5, 1, 3, 4)`, and
`$simpleMDArr[2]` stores `(4, 1, 7, 8, 9)`.

If you want to access the elements in the "inner" arrays, you use two pairs of square brackets. For instance,

```
echo $simpleMDArr[2][3];
```

gives us the 4th (3+1) element in `$simpleMDArr[2]`. We'll get `8` as the output.

Example 6

Associative arrays can also be used to store arrays. In the example below, `$assocMDArr` is an associative array used to store three indexed arrays.

```
$assocMDArr = array(
  "first array" => array(1, 2, 6, 1, 3),
  "second array" => array(3, 5, 1, 8, 9),
  "third array" => array(1, 0, 9, 4, 7)
);
```

`$assocMDArr["first array"]` stores the array `(1, 2, 6, 1, 3)`.

If we write

```
echo $assocMDArr["first array"][2];
```

we'll get the 3rd element in `$assocMDArr["first array"]`. In other words, we'll get `6` as the output.

Example 7

Last but not least, we can use an associative array to store associative arrays.

```
$anotherAssocMDArr = array(
  "first player" => array("name" => 'John', "age"
=> 25),
  "second player" => array("name" => 'Tim', "age"
=> 35)
);
```

In the example above, `$anotherAssocMDArr["first player"]` stores the array `("name" => 'John', "age" => 25)`.

If we write

```
echo $anotherAssocMDArr["first player"]["age"];
```

we'll get `25` as the output.

5.3.2 Displaying the Content of Arrays

Next, let's learn two convenient ways to display the content of arrays.

One way is to use the `var_dump()` function covered in the previous chapter. This function displays the data type and value of a variable. If we write

```
$myArray = array(2, 5.1, 'PHP', 105);
var_dump($myArray);
```

we'll get

```
array(4) { [0]=> int(2) [1]=> float(5.1) [2]=>
string(3) "PHP" [3]=> int(105) }
```

as the output.

“array(4)” indicates that `$myArray` is an array with 4 elements.

The text inside the pair of braces that follows indicates the data type and value of each element. For instance, “[0] => int(2)” tells us that the element at index `0` is an integer of value `2`, “[1] => float(5.1)” tells us that the element at index `1` is a float of value `5.1` and so on.

Next, we have the `print_r()` function.

This function is similar to the `var_dump()` function. However, its output is more concise as it does not give us the data type of each element. For instance,

```
print_r($myArray);
```

gives us the following output:

```
Array ( [0] => 2 [1] => 5.1 [2] => PHP [3] => 105 )
```

5.3.3 Adding Elements to Arrays

After creating an array, we can add new elements to it. There are a few ways to do it. The easiest is to use the square bracket notation. This notation allows us to add one element at a time to an array.

Example 1

```
$addDemo = array(1, 5, 3, 9);
$addDemo[] = 7;
```

In the example above, we add `7` to `$addDemo`. The array becomes `(1, 5, 3, 9, 7)`.

Example 2

```
$addDemoAssoc = array('Peter'=>20, 'Jane'=>15);
$addDemoAssoc['James'] = 17;
```

Here, we add `'James'=>17` to `$addDemoAssoc`. The array becomes

('Peter'=>20, 'Jane'=>15, 'James'=>17).

5.3.4 Deleting Elements from Arrays

Besides adding elements to arrays, we can delete elements from them. One way is to use the `array_splice()` function.

To use this function, we need to pass two arguments to it - the array to remove elements from and the position to start removing them.

We can also pass a third argument to specify the number of elements to remove. If we do not provide this argument, the function removes all elements from the specified starting position to the end of the array.

Let’s look at some examples.

Example 1

```
$colors = array("red", "black", "pink", "white");
array_splice($colors, 2);
```

Here, we indicate that we want `array_splice()` to remove all elements from `$colors`, starting from (and including) the element at position 2.

`$colors` becomes `("red", "black").`

Example 2

```
$awardwinners = array("Gold"=>"Max",
"Silver"=>"Boots", "Bronze"=>"Dora");
array_splice($awardwinners, 1);
```

Here, we want `array_splice()` to remove all elements from `$awardwinners`, starting from (and including) position 1.

`$awardwinners` becomes (`"Gold"=>"Max"`).

Example 3

```
$pets = array("corgi", "poodle", "golden retriever",
"jack russell");
array_splice($pets, 1, 2);
```

Here, we want to remove 2 elements from `$pets`, starting from (and including) position 1.

`$pets` becomes `("corgi", "jack russell").`

5.3.5 Commonly used Array Functions in PHP

We've covered several array concepts in the previous sections. Before we end this chapter, let's look at some commonly used array functions in PHP.

count()

The first is the `count()` function. This function accepts an array and returns the number of elements in the array.

Example:

```
$countDemo = array(1, 4, 5, 1, 2);
echo count($countDemo);
```

Output:

```
5
```

array_search()

Next, we have the `array_search()` function. This function searches for a particular **value** in an array. If the value is found, the function returns its corresponding index or key. Else, it returns `false`. If more than one instance of the value is found, the index or key of the first matching value is returned.

Example:

```
$indexArrDemo = array(11, 4, 5, 1, 2, 5, 6);
$assocArrDemo = array('A'=>12, 'B'=>5, 'C'=>20);

echo array_search(5, $indexArrDemo).'<BR>';
echo array_search(20, $assocArrDemo).'<BR>';
var_dump(array_search('B', $assocArrDemo));
```

Output:

```
2
C
bool(false)
```

In the first `echo` statement above, we use the `array_search`() function to search for `5` in `$indexArrDemo`. Although there are two values of `5` in the array, only the index of the first matching value is returned.

In the second `echo` statement above, we use `array_search`() to search for `20` in `$assocArrDemo`. We get “C” as the output.

In the last `echo` statement, we use `array_search`() to search for `'B'` in `$assocArrDemo`. We get `false` as `'B'` is a key in the array, not a value.

in_array()

The `in_array()` function is similar to the `array_search()` function. However, instead of returning the key or index, it returns `true` if the stated value is found in the array. Else, it returns `false`.

With reference to `$indexArrDemo` and `$assocArrDemo` defined above,

```
var_dump(in_array(5, $indexArrDemo));
var_dump(in_array(20, $assocArrDemo));
var_dump(in_array('B', $assocArrDemo));
```

give us

```
bool(true) bool(true) bool(false)
```

as the output.

array_merge()

Finally, we have the `array_merge()` function. This function merges two or more arrays and returns the merged array.

Example 1:

```
$num1 = array(100, 111, 120);
```

```
$num2 = array(100, 3, 5);
$num3 = array(1, 10);
```

```
$newArray1 = array_merge($num1, $num2, $num3);
```

`$newArray1` becomes `(100, 111, 120, 100, 3, 5, 1, 10)`.

Example 2:

```
$names1 = array(5 => "Peter", 24 => "Aaron");
$names2 = array(5 => "Zac", 4 => "Alfred", 7 =>
"Avi");
$newArray2 = array_merge($names1, $names2);
```

Keys are renumbered in the merged array if any of the arrays passed to the `array_merge()` function is an associative array with integer keys.

In the example above, `$newArray2` becomes `(0 => "Peter", 1 => "Aaron", 2 => "Zac", 3 => "Alfred", 4 => "Avi")`.

The key for `"Peter"` is renumbered from `5` to `0`. The same applies to all the other keys.

Example 3:

```
$str1 = array('A'=> 12, 'B' => 5, 'C' => 8);
$str2 = array('A' => 15, 'D' => 10);
$newArray3 = array_merge($str1, $str2);
```

If two or more array elements have the same string keys, the last one overrides the earlier ones.

In the example above, `$newArray3` becomes `('A'=> 15, 'B' => 5, 'C' => 8, 'D' => 10)`.

`'A'=> 12` is replaced by `'A'=> 15`.

We've covered some of the commonly used array functions in PHP. For a complete list of all the built-in array functions, check out https://www.php.net/manual/en/ref.array.php.

Chapter 6: Control Structures in PHP

Congratulations on making it to Chapter 6; this is where the fun begins!

In this chapter, we are going to look at various control structures in PHP, such as the `if` statement, `for` loop and `while` loop. These structures allow us to control the flow of our programs.

However, before we do that, we need to discuss two sets of operators - the comparison and logical operators. Controlling the flow of our programs involves evaluating the results of these operators and proceeding accordingly.

6.1 Comparison operators

Let's look at comparison operators first. These operators compare two values and return `true` or `false` based on the result of the comparison. They include:

Equal (`==`)
Returns `true` if the values on both sides are equal

(e.g., `5 == 5`, `'Hello' == 'Hello'` and `5 == 5.0` all return `true`)

Identical (`===`)

Returns `true` if the values on both sides are equal and of the same date type

(e.g., `5 === 5` returns `true` while `5 === 5.0` returns `false`)

Not equal (`!=` or `<>`)

Returns `true` if the values on both sides are not equal

(e.g., `5 != 7` and `5 <> 7` both return `true`)

Not identical (`!==`)

Returns `true` if the values on both sides are not equal **or** not of the same data type

(e.g., `5 !== 5.0` returns `true` as 5 and 5.0 are of different data types)

Greater than (`>`)

Returns `true` if the value on the left is greater than the value on the right

(e.g., `7 > 2` returns `true`)

Greater than or equal to (`>=`)

Returns `true` if the value on the left is greater than or equal to the value on the right

(e.g., `8 >= 5` and `6 >= 6` both return `true`)

Less than (`<`)

Returns `true` if the value on the left is less than the value on the right

(e.g., `9 < 12` returns `true`)

Less than or equal to (`<=`)

Returns `true` if the value on the left is less than or equal to the value on the right

(e.g., `10 <= 14` and `13 <= 13` both return `true`)

Spaceship (<=>)
Introduced in PHP 7.
Returns `0` if the values on both sides are equal (not necessarily identical)
Returns `1` if the value on the left is greater
Returns `-1` if the value on the left is smaller

(e.g., `5 <=> 7` returns `-1` while `5 <=> 5.0` returns `0`)

6.2 Logical Operators

Next, we have the set of logical operators. The commonly used ones include:

NOT

The NOT (`!`) operator returns `true` when it precedes an expression that is `false`.

For instance, `!(5 > 10)` returns `true` as `5 > 10` is `false` (i.e., 5 is not greater than 10).

AND

The AND operator allows us to combine two comparisons and returns `true` when both comparisons return `true`. We can use either the `and` keyword (case-insensitive) or the `&&` symbol. The two are similar but have different precedence.

Let's look at some examples:

Example 1:

```
$a = 5 > 3 and 4 < 10;
$b = 5 > 3 && 4 < 10;
$c = 5 > 1 && 13 < 5;

var_dump($a);
var_dump($b);
var_dump($c);
```

Output:

```
bool(true) bool(true) bool(false)
```

In the example above, `$a` and `$b` are `true` as both comparisons (`5 > 3` and `4 < 10`) are `true`. In contrast, `$c` is `false` as the second comparison (`13 < 5`) is `false`.

Example 2:

```
$d = 3 > 2 && 3 < 1;
$e = 3 > 2 and 3 < 1;
var_dump($d);
```

```
var_dump($e);
```

Output:

```
bool(false) bool(true)
```

This example illustrates the difference in precedence between the `and` keyword and the `&&` symbol.

`&&` has higher precedence than the assignment (`=`) operator. Hence, the statement

```
$d = 3 > 2 && 3 < 1;
```

is evaluated as

```
$d = (3 > 2 && 3 < 1);
```

`$d` is `false` as `3 < 1` is `false`.

In contrast, the `and` keyword has lower precedence than the assignment operator. Hence, the statement

```
$e = 3 > 2 and 3 < 1;
```

is evaluated as

```
($e = 3 > 2) and 3 < 1;
```

`$e` is `true` as only the result of the first comparison (`3 > 2`) is assigned to it.

In most cases, using `&&` is preferred due to its higher precedence.

OR

Next, we have the OR operator. This operator returns `true` when at least one comparison returns `true`. We can use either the `or` keyword or the `||` symbol. However, I encourage you to use `||` as it has higher precedence.

Example

```
$f = 4 < 7 || 10 > 3;
$g = 3 < 2 || 3 > 1;
```

```
$h = 10 > 15 || 12 < 1;

var_dump($f);
var_dump($g);
var_dump($h);
```

Output

```
bool(true) bool(true) bool(false)
```

`$f` is `true` as both comparisons are `true`.

`$g` is `true` even though the first comparison (`3 < 2`) is `false`. This is because `||` only requires at least one comparison to be `true`.

`$h` is `false` as none of the comparisons are `true`.

6.3 Control Structures

Now that we are familiar with comparisons, we are ready to discuss the various control structures in PHP.

6.3.1 If Statement

The first is the `if` statement. This statement allows the program to evaluate if a certain condition is met and perform an appropriate action based on the result of the evaluation.

The syntax of an `if` statement is as follows:

```
if (condition 1 is met)
{
     //Do task A
}elseif (condition 2 is met)
{
     //Do task B
}elseif (condition 3 is met)
{
     //Do task C
}else
{
     //Do task D
```

```
}
```

Let’s look at an actual example (line numbers are added for reference and are not part of the actual code).

```
1  $a = 7;
2
3  if ($a < 0)
4  {
5     echo 'if block<BR>';
6     echo '$a is smaller than 0';
7  }
8  elseif ($a < 5)
9     echo 'First elseif block';
10 elseif ($a < 10)
11    echo 'Second elseif block';
12 else
13    echo 'Else block';
```

Here, we first declare a variable called `$a` and assign the value `7` to it.

Next, from lines 3 to 7, we have the `if` block.

This block tests if the condition `$a < 0` (on line 3) is `true`. If it is, the program executes everything inside the pair of curly braces that follows (i.e., lines 4 to 7), skipping the rest of the `if` statement (from lines 8 to 13).

On the other hand, if `$a < 0` is not true, the program skips lines 4 to 7 and moves on to test the first `elseif` condition (`$a < 5`) on line 8.

If this condition is `true`, it executes the statement on line 9, skipping the rest of the `if` statement. Notice that we did not enclose line 9 in curly braces? This is because curly braces are optional if there is only one statement to execute.

If the condition `$a < 5` is not true, the program moves on to test the next `elseif` condition (`$a < 10`) on line 10. If this condition is also not true, it proceeds to the `else` block and executes line 13.

There can be as many `elseif` blocks as you want in an `if`

statement. We have two `elseif` blocks in the example above.

In addition, `elseif` and `else` blocks are optional. If we omit them, the `if` statement will just do nothing if the `if` condition is not met. If you run the code above, you'll get the following output:

```
Second elseif block
```

As `$a` equals `7`, it fails the `if` condition (`$a < 0`) and the first `elseif` condition (`$a < 5`). However, it passes the second `elseif` condition (`$a < 10`). Hence, line 11 is executed. Try changing `$a` to other values on line 1 and rerun the code to fully appreciate how it works. The output for some possible values of `$a` is shown below:

If `$a` equals `-2`, we'll get

```
if block
$a is smaller than 0
```

If it equals `3`, we'll get

```
First elseif block
```

If it equals `11`, we'll get

```
Else block
```

6.3.2 Ternary Operator

Next, let's move on to talk about the ternary operator. In the previous example, we have a relatively complex `if` statement with two `elseif` blocks.

If you only want to do a simple `if-else` test (without any `elseif` blocks), you can use the ternary operator (`?`). The syntax is as follows:

```
Condition ? Task A : Task B
```

The ternary operator first checks the condition on its left. If it is `true`, it performs task A. Else, it performs task B.

Let’s look at an example:

```
$a = (7 == 7 ? 'Yes' : 'No');
echo $a;
```

Here, the condition to test is `7 == 7`.

As this condition is `true`, the ternary operator performs the first task and returns the string `'Yes'`. We then assign this string to `$a` and use the `echo` statement to display its value on the next line.

If we run the code above, we’ll get “Yes” as the output. In contrast, if we change the ternary statement to

```
$a = (7 > 10 ? 'Yes' : 'No');
```

we’ll get “No” as the output as `7 > 10` is `false`.

6.3.3 Switch Statement

Next, we have the `switch` statement. This statement is similar to an `if` statement and can be faster if you have many conditions to test. It is typically used when the condition involves comparing a variable against a single value (instead of a range of values).

The syntax is as follows:

```
switch (variable used for switching) {
  case firstCase:
    Task A;
    break;
  case secondCase:
    Task B;
    break;
...
  default:
    Default task;
}
```

Let’s look at an example (line numbers are added for reference):

```
1  $b = 20;
```

```
2
3  switch ($b)
4  {
5     case 10:
6         echo 'Chocolate<BR>';
7         break;
8     case 20:
9         echo 'Lemon<BR>';
10    case 25:
11        echo 'Strawberry<BR>';
12        break;
13    default:
14        echo 'None of the above<BR>';
15 }
```

In the example above, we first declare a variable called `$b` and assign `20` to it.

Next, we have a `switch` statement from lines 3 to 15.

On line 3, the `switch` statement uses the value of `$b` to decide which case to execute. When a certain case is satisfied, everything starting from the next line is executed until a `break` statement is reached.

If `$b` is `10`, case `10` is satisfied. The program executes everything after line 5 until it reaches the `break` statement on line 7. In other words, it executes line 6 and gives us “Chocolate” as output.

On the other hand, if `$b` is `20`, case `20` is satisfied. However, this case does not have a `break` statement. Hence, the program executes everything starting from line 9, until it reaches a `break` statement on line 12. In other words, it executes cases `20` and `25` and gives us “Lemon”, followed by “Strawberry” as output.

Next, if `$b` is `25`, case `25` is satisfied. The program executes line 11 and gives us “Strawberry” as output.

Finally, if `$b` is not `10`, `20` or `25`, the `default` case is executed and we’ll get “None of the above” as output.

The `default` case is optional. In addition, we can have as many cases as we want in a `switch` statement. In our example, we have three cases.

If you run the `switch` statement above, you'll get

```
Lemon
Strawberry
```

as output. Try changing `$b` to other values on line 1 and run the code again to fully appreciate how the `switch` statement works.

6.3.4 For Loop

Next, let's move on to the `for` loop. This control structure executes a block of code repeatedly until the test condition is no longer valid.

The syntax is as follows:

```
for (initial value; test condition; modification to value)
{
  //Do Some Task
}
```

Let's look at an example:

```
for ($c = 1; $c < 5; ++$c){
  echo 'The value of $c is '.$c.'<BR>';
}
```

The main focus of a `for` loop is the first line. There are three parts to this line, each separated by a semicolon. In our example, we have the line

```
for ($c = 1; $c < 5; ++$c)
```

The first part declares a variable `$c` and initializes it to `1`.

The second part tests if `$c` is smaller than `5`. If it is, the statement(s) inside the curly braces that follow will be executed. In our example, the curly braces are optional as there is only one statement to

execute.

After executing the statement(s) in the curly braces, the program returns to the third part (`++$c`). This part modifies the value of the variable used for testing. In our example, we increase the value of `$c` by `1` using the increment operator. As a result, `$c` becomes `2`.

After the increment, the program tests if the new value of `$c` is still smaller than 5. If it is, it executes the code in the curly braces again.

This process of testing and updating the value of `$c` is repeated until the condition `$c < 5` is no longer true. At this point, the program exits the `for` loop and continues to execute other statements after the `for` loop.

If you run the code above, you'll get

```
The value of $c is 1
The value of $c is 2
The value of $c is 3
The value of $c is 4
```

as the output.

6.3.5 Foreach Loop

Next, we have the `foreach` loop. This loop is similar to the `for` loop but is used to loop over arrays. There are two syntaxes:

```
foreach ($array as $value) {
 //code to be executed;
}
```

```
foreach ($array as $key=>$value) {
 //code to be executed;
}
```

The names `$array`, `$value` and `$key` in the syntaxes above are chosen by us; we can use other names if we want.

Let's look at an example. This example uses the first syntax:

```
$arr1 = array(11, 12, 13, 14, 15);

foreach ($arr1 as $num){
  echo 'The number is '.$num.'<BR>';
}
```

Here, we declare an array called `$arr1` with values `11`, `12`, `13`, `14` and `15`.

Next, we have the `foreach` loop. This loop loops through the elements in `$arr1` one by one and assigns each element to a variable called `$num`.

The first time the loop runs, the first element in `$arr1` is assigned to `$num`. In other words, `11` is assigned to `$num`.

After assigning `11` to `$num`, the program executes the code in the pair of curly braces that follows. This gives us “The number is 11” as the output.

Once that is done, the `foreach` loop moves on to the second element and assigns it to `$num`.

`$num` becomes `12` and the `echo` statement is executed again to give us “The number is 12” as output.

This keeps repeating until all the elements in `$arr1` have been accounted for. If you run the code above, you’ll get

```
The number is 11
The number is 12
The number is 13
The number is 14
The number is 15
```

as the output.

Next, let’s look at a `foreach` loop that uses the second syntax. This syntax gives us both the values and keys of the elements in an array and is very useful when working with associative arrays. Suppose we have

```
$arr2 = array('Aaron'=>12, 'Ben'=>23, 'Carol'=>35);
```

To get the keys and values of the elements in `$arr2`, we can use the `foreach` loop below:

```
foreach ($arr2 as $name=>$age){
  echo $name.' is '.$age.' years old.<BR>';
}
```

This gives us

```
Aaron is 12 years old.
Ben is 23 years old.
Carol is 35 years old.
```

as the output.

6.3.6 While Loop

Next up is the `while` loop. This loop performs a task repeatedly while a specific condition remains valid. The syntax is as follows:

```
while (condition is true)
{
     //do A
}
```

As usual, let's look at an example:

```
1 $d = 1;
2
3 while ($d < 5)
4 {
5       echo 'The value of $d is '.$d.'<BR>';
6       $d++;
7 }
```

Here, we first declare a variable called `$d` and assign the value `1` to it.

Next, we have the `while` loop from lines 3 to 7.

On line 3, the `while` loop checks if the condition `$d < 5` is `true`. If it is, it executes the statements inside the curly braces that follow. In

other words, it executes the `echo` statement on line 5 and increments `$d` by `1` on line 6. As a result, `$d` becomes `2`.

Next, the `while` loop returns to line 3 to check if `$d` is still smaller than `5`. If it is, it executes the code inside the curly braces again. This process of checking and updating the value of `$d` continues until `$d` is no longer smaller than `5`. If you run the code above, you'll get the following output:

```
The value of $d is 1
The value of $d is 2
The value of $d is 3
The value of $d is 4
```

At this point, you may notice that the `while` loop is very similar to the `for` loop. Indeed, in most cases, both of them serve the same purpose.

However, when using a `while` loop, one important difference is that we must remember to update the variable used for looping (`$d` in this example). If we forget to do that, the `while` loop will run indefinitely, resulting in an infinite loop.

For instance, in our example, if we omit line 6 (`$d++;`), the `while` loop will never end as the value of `$d` will always be `1`, which is smaller than `5`. Got it? Good!

6.3.7 Do-while Loop

Last but not least, let's move on to the `do-while` loop. This loop is very similar to the `while` loop except that the test condition is placed at the end of the loop. This means that the code within the curly braces of the loop will always be executed at least once.

The syntax of a `do-while` loop is

```
do {
     //some tasks
} while (condition is true);
```

Note that a semicolon (`;`) is required after the test condition. Here's an

example of how the loop works.

```
1 $e = 100;
2
3 do {
4      echo 'The value is '.$e;
5      $e++;
6 } while ($e<0);
```

On line 1, we first declare and initialize a variable `$e` with the value `100`.

Next, we have the `do-while` loop from lines 3 to 6. Within the loop, the program executes the `echo` statement on line 4 and increments the value of `$e` to `101` on line 5.

Finally, it reaches the test condition on line 6.

As the value of `$e` is not smaller than `0`, the test fails. The program exits the `do-while` loop and does not repeat the tasks in the loop. If you run the code above, you will get

```
The value is 100
```

as the output. Although the original value of `$e` does not meet the test condition `($e < 0)`, the code inside the curly braces is executed once as the test condition comes after the closing curly brace.

6.4 Other Topics in Flow Control

Great! We've covered all the main control structures in PHP. Before we end this chapter, I would like to cover a few more miscellaneous topics related to flow control in PHP.

6.4.1 Booleans

First, let's talk about the `bool` data type. We encountered this data type in Chapter 4.3 and learned that it can only store one of two values – `true` or `false`.

In PHP, most values can be converted to `true` or `false` (i.e.,

converted to the `bool` data type).

Values that convert to `false` include:
- numbers that represent zero (such as `0` and `0.0`)
- the empty string (such as `''`, which is made up of two single quotes with no text enclosed)
- the string `"0"`
- an array with zero elements, and
- a special value called `NULL`.

On the other hand, most other values (such as `1, 3.4` and `"Hello"`) convert to `true`.

We've seen how control structures decide whether to execute a certain block of code depending on whether a condition evaluates to `true` or `false`. Suppose we have the following `if` statement:

```
if ("hello")
  echo 'if block';
else
  echo 'else block';
```

If we run the statement above, we'll get

```
if block
```

as the output.

This is because the string `"hello"` is converted to `true`. For any control structure, as long as a condition evaluates to `true`, regardless of whether it is the result of a comparison (e.g., `$a < 5`) or a value converted to `true`, the corresponding block of code will be executed.

In the `if` statement above, the `if` block is executed as the `if` condition (`"hello"`) evaluates to `true`. Got it? Good!

6.4.2 Break, Continue

Next, let's discuss `break` and `continue` statements. These statements can be used to modify the behavior of our control

structures.

We've already encountered the `break` statement when we talked about the `switch` statement previously. When a particular switch case is satisfied, the `switch` statement executes everything that follows until it encounters a `break` statement.

Besides using it in a `switch` statement, we can use the `break` statement in a loop. A `break` statement ends the execution of the loop it is in. Let's look at an example:

```
for ($i = 0; $i < 50; ++$i)
{
  echo "$i<BR>";
  if ($i == 4)
    break;
}
```

Here, we have an `if` statement inside a `for` loop. It is fairly common for us to nest one control structure inside another in programming.

Within the `for` loop, we first echo the value of `$i`. Next, we check if `$i` equals `4`. If it equals, we want the program to break out of the `for` loop. If you run the code above, you'll get the following output:

```
0
1
2
3
4
```

If we do not have the `break` statement, the `for` loop should run 50 times, from `$i = 0` to `$i = 49`.

However, as we have the `break` statement, this loop only runs from `$i = 0` to `$i = 4`. This is because when `$i` equals `4`, the `if` condition (`$i == 4`) evaluates to `true`. The `break` statement that follows then causes the program to break out of the `for` loop, skipping the rest of the iterations (from `$i = 5` to `$i = 49`). Got it?

Next, we have the `continue` statement. This statement does not

cause a loop to end prematurely. Instead, it causes the rest of the loop to be skipped for that particular iteration. Let's look at an example:

```
for ($i = 0; $i < 6; ++$i)
{
   echo '$i = '.$i.', ';

   if ($i == 4)
      continue;

   echo 'First.';

   echo 'Second.<BR>';
}
```

If you run the code above, you'll get the following output:

```
$i = 0, First.Second.
$i = 1, First.Second.
$i = 2, First.Second.
$i = 3, First.Second.
$i = 4, $i = 5, First.Second.
```

Notice that when `$i` equals `4`, the text "First.Second." does not appear? This is because when `$i` equals `4`, the `continue` statement caused the program to skip the statements

```
echo 'First.';

echo 'Second.<BR>';
```

for that iteration. Other than that, the code runs as per normal. Clear?

6.4.3 Alternative Syntax

Next, let's move on to talk about an alternative syntax for control structures in PHP.

PHP offers an alternative syntax for some of its control structures, specifically the `if`, `while`, `for`, `foreach`, and `switch` structures.

This syntax involves changing opening braces to colons (`:`) and the

last closing brace to `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;` respectively.

Let’s use an `if` statement to illustrate this. Suppose we want to display different outputs based on the value of `$a`, we can use the following `if` statement:

```
$a = 5;

if ($a == 5){
   echo '<BR>The value of $a is<BR>';
   echo $a;
}else{
   echo 'Not five';
}
```

Alternatively, we can do it as follows:

```
$a = 5;

if ($a == 5):
   echo '<BR>The value of $a is<BR>';
   echo $a;
else:
   echo 'Not five';
endif;
```

Compare the two `if` statements carefully. Notice that in the second `if` statement above, we replaced both opening braces with a colon? In addition, we replaced the last closing brace with an `endif;` statement.

If you run the code above, you’ll get

```
The value of $a is
5
```

as the output for both syntaxes.

6.4.4 Displaying HTML code

Last but not least, let’s discuss a neat trick for outputting HTML code

in control structures. So far, we have been using `echo` statements to output HTML code (such as the `<BR>` tag) in our control structures.

This technique is easy to use and works well with simple HTML code. However, it can get cumbersome if we have more complicated HTML code to output, especially if the code uses lots of single and double quotation marks. For instance, if we want to output the following HTML code

```
<a href = "names.php" target="_blank">The names are
'Aaron', 'Peter', 'James', 'Max', 'Jenny' and 'Don'.
</a>
<img src="names.jpg" alt="Names" height="42"
width="42">
```

using an `echo` statement, we need to escape a lot of quotation marks. In cases like that, we can use a neat trick to switch between PHP and HTML code.

For demonstration purposes, let's suppose we want to use a `for` loop to output the HTML code `<h1>Hello</h1>` three times, the code below shows how we can do it (using the alternative syntax mentioned in the previous section):

```
1 <?php
2    for ($i = 0; $i < 3; ++$i):
3        echo '<h1>Hello</h1>';
4    endfor;
5 ?>
```

This loop uses an `echo` statement to output the HTML code. Alternatively, if we do not want to use `echo` statements, we can do it as follows:

```
1 <?php
2    for ($i = 0; $i < 3; ++$i): ?>
3        <h1>Hello</h1>
4    <?php endfor;
5 ?>
```

Study the two loops carefully, what differences do you notice?

First, notice that we added `?>` to the end of line 2 in the second loop? This `?>` tag closes the `<?php` opening tag on line 1.

When that happens, any code after line 2 in the second loop is no longer interpreted as PHP code. Instead, it is interpreted as HTML code. This explains why we do not need to use an `echo` statement to output `<h1>Hello</h1>` on line 3.

Next, on line 4, we open the PHP tag again. When PHP sees this opening tag, it knows that what follows is PHP code.

When it encounters the `endfor;` statement, it realizes that this is a `for` loop and searches for the condition to evaluate. When it finds it on line 2, it increments `$i` by `1` and runs line 3 again. This keeps repeating until the condition `$i < 3` is no longer valid. Got it?

If you run the loops above, you'll get "Hello" displayed as a `<h1>` heading three times in both cases.

Study the two loops carefully to appreciate how this technique works. It can be a bit confusing in the beginning. The main idea is to close the PHP tag before switching to HTML code and open it again after the HTML code. This makes it much more convenient to insert HTML code without having to use `echo` statements.

Chapter 7: Functions

In this chapter, let’s move on to talk about functions.

We have already encountered functions in previous chapters. For instance, in Chapter 5, we learned to use various PHP functions to work with strings and arrays. In this chapter, we’ll learn to define our own functions.

7.1 Defining our own Functions

To define our own functions, we use the syntax below:

```
function functionName(list of parameters) {
  //code to be executed;
  //return statements, if any
}
```

All function declarations must start with the `function` keyword.

Next, we have the function name. Function names in PHP are not case-sensitive and can contain letters, numbers, or underscores. However, they must start with a letter or an underscore. Similar to variable names, we commonly use camelCase or underscores when naming functions. In this book, we’ll use camelCase.

After the function name, we enclose the list of parameters (if any) that the function needs in a pair of parentheses. Parameters are variables used for storing values that we pass to the function; the values that we pass are known as arguments.

After declaring the function, we use curly braces to enclose the code that we want the function to perform. If the function returns a result, we use the `return` keyword. Got it?

Let’s look at some examples.

Example 1

```
function displayGreetings(){
```

```
  echo 'Hello';
}
```

Here, we declare a function called `displayGreetings()` that has no parameters (as indicated by the pair of empty parentheses after the function name). This function simply echoes the word “Hello”. To call the function, we use its name followed by a pair of parentheses:

```
displayGreetings();
```

This gives us

```
Hello
```

as the output.

Example 2

```
function displayGreetings2($name, $message){
  echo "Hello $name, $message";
}
```

Here, we declare a function called `displayGreetings2()` that has two parameters - `$name` and `$message`.

If we call the function using the following statement

```
displayGreetings2('Jamie', 'good morning');
```

the arguments `'Jamie'` and `'good morning'` will be assigned to the parameters `$name` and `$message` respectively. When we run the code above, we’ll get

```
Hello Jamie, good morning
```

as the output.

Example 3

We can provide default values for parameter(s) when we declare our functions.

```
function displayGreetings3($name, $message = 'have a good day'){
  echo "Hello $name, $message";
```

```
}
```

In the example above, we declare `'have a good day'` as the default value for the parameter `$message` (refer to the underlined code). To call the function, we can use the statements below:

```
displayGreetings3('Jamie');
echo '<BR>';
displayGreetings3('Jamie', 'how are you?');
```

In the first function call, we omitted the second argument. When we do that, the function uses the default value `'have a good day'` as the value for `$message`.

If we run the code above, we'll get

```
Hello Jamie, have a good day
Hello Jamie, how are you?
```

as the output.

When we declare functions with default values for parameters, note that parameters with default values must come after parameters without default values. If we declare a function as

```
function redundantDefault($a = 1, $b){
  //some code
}
```

the default value for `$a` will not work as it comes before `$b` (which does not have a default value); we'll still need to provide two arguments when calling the function.

Example 4

Last but not least, functions can return a result after they complete their tasks.

```
function addNumbers($num1, $num2, $num3){
  return $num1 + $num2 + $num3;
  echo 'Hello';
}
```

The function above returns the sum of `$num1`, `$num2` and `$num3` using a `return` statement. Once a `return` statement is executed, the function exits, and any statement after the `return` statement is not executed.

We can assign the result returned by the function to a variable or use the `echo` statement to display the result directly.

If we call the function above using the following statement:

```
echo addNumbers(9, 6, 1);
```

we'll get

```
16
```

as the output. The statement

```
echo 'Hello';
```

is not executed as it comes after the `return` statement.

7.2 Type Declaration

The examples in the previous section illustrate the main points we need to know when defining our own functions. Before we move on to the next chapter, I would like to discuss type declaration (also known as type hinting) in this section.

Type declaration is a feature added to newer versions of PHP. This feature is optional but good to implement if you are certain your code will only run on PHP 5 (preferably PHP 7) and above.

First, what is type declaration?

As mentioned previously, PHP is a loosely typed language. In the examples above, when we declared the parameters of our functions, we did not indicate their data types. If the arguments that we pass to the function are of invalid data types, PHP will try its best to execute the function.

For instance, with reference to Example 4 in the previous section, if we try to run the following statement

```
echo addNumbers('9', '6', '1');
```

we won’t get any errors.

Although we pass three strings to the `addNumbers()` function, PHP converts them to integers for us automatically and executes the function. Hence, we’ll get `16` as the output.

Such flexibility makes PHP a very easy language to program in. However, depending on the requirements of the site you are building, you may not want the function to run when the data type is incorrect. In cases like that, you can use type declaration.

Type declaration allows us to state the data types of a function’s parameters when we declare it.

This feature is available from PHP 5 onwards. However, in PHP 5, type declaration does not work for scalar data types. Scalar data types refer to data types that store a single value, such as `int`, `float`, `bool` and `string`. Type declaration only works for data types like arrays.

To use type declaration with scalar data types, we need PHP 7 and above. In addition, we need to add a `strict_types` declaration to our script and set `strict_types` to `1`.

To illustrate how this works, let’s look at an example. This example works in PHP 7.

Create a new file in Brackets and name it **typedec.php**. Add the following code to it.

```
<?php
   declare(strict_types=1);

   function addNumbersStrict(int $num1, int $num2,
int $num3){
      return $num1 + $num2 + $num3;
   }

   echo addNumbersStrict('9', '6', '1');
```

Here, we set `strict_types` to `1` using a built-in function called `declare()`. This statement must be the first statement in our script.

Next, we declare a function called `addNumbersStrict()` with three parameters `$num1`, `$num2` and `$num3`. We indicate that these parameters must be of `int` type by adding `int` in front of their names. Finally, we call the function by passing `'9'`, `'6'` and `'1'` to it.

If you run the code above, you'll get an output similar to what is shown below:

```
Fatal error: Uncaught TypeError: Argument 1 passed
to addNumbersStrict() must be of the type int,
string given...
```

Next, change the `echo` statement to

```
echo addNumbersStrict(9,6,1);
```

and rerun the code. As `9`, `6` and `1` are of `int` type, we won't get any errors now. Instead, we'll get `16` as the output. Got it?

Besides doing type declaration for parameters, we can do it for the return type if we are using PHP 7 and above. To do that, we indicate the return type using a colon after the parentheses in the function declaration.

Add the following code to **typedec.php**:

```
function addTwoNums($a, $b): int {
  return $a + $b;
}

echo '<BR>'.addTwoNums(3, 1);
```

Here, we indicate that the return value of `addTwoNums()` should be of `int` type. Next, we call the function using `3` and `1` as the arguments.

If you run the code above, you'll get `4` as the output. Next, change the last line to

```
echo '<BR>'.addTwoNums(3.9, 1);
```

and run the page again. You’ll get the following output:

```
Fatal error: Uncaught TypeError: Return value of
addTwoNums() must be of the type int, float
returned...
```

This is because 3.9 + 1 gives us 4.9, which is not of `int` type.

Chapter 8: PHP Superglobals

We’ve covered a lot in the preceding chapters. Most of the topics covered so far are fundamental concepts common to other programming languages. In this chapter, we are going to learn something specific to PHP - PHP superglobals.

We’ll learn to use these superglobals to interact with HTML forms and pass information from one PHP script to another.

8.1 PHP Form Handling

First and foremost, this topic requires you to have a basic understanding of HTML form elements. If you are not familiar, https://www.w3schools.com/html/html_form_elements.asp provides a quick reference that you can refer to.

If you are familiar, let’s start by creating a simple HTML form to work with later.

Open Brackets and create a new file called **form.php** in your **htdocs** folder. Add the following code to it:

```
<!DOCTYPE html>
<html>
<head><title>PHP Form Handling</title></head>
<body>
<form action = "" method = "get">
  Enter Name <BR>
  <input type = "text" name = "studentname" value =
"Your Name">
  <BR><BR>
  Favorite Subject(s)<BR>
  <input type = "checkbox" name = "subj[]" value =
"EL">English
  <input type = "checkbox" name = "subj[]" value =
"MA">Math
  <input type = "checkbox" name = "subj[]" value =
```

```
"PG">Programming
  <BR><BR>
  Gender <BR>
  <input type = "radio" name = "gender" value =
"M">Male
  <input type = "radio" name = "gender" value =
"F">Female
  <BR><BR>
  <input type = "submit" name="sm" value = "Submit
Form">
</form>
</body>
</html>
```

Study the code above carefully. Notice that we use the following `<form>` tag to create a HTML form?

```
<form action = "" method = "get">
```

This tag has two attributes - `action` and `method`.

The `action` attribute specifies the file that we want to use to process the HTML form. In our example, we assign an empty string to it. This means that we want to use the current file (i.e., **form.php**) to process the form.

If we want to use a different file to process the form, we need to assign the path of that file to the `action` attribute. For instance, suppose we want to use **process.php** and **process.php** is in the same folder as **form.php**, we need to write

```
<form action = "process.php" method = "get">
```

Next, we have the `method` attribute. This attribute specifies the method that we want to use to send the form data. There are two methods available: `get` and `post`. If we do not specify any method, the default is `get`. In our example, we specify that we want to use the `get` method. We'll discuss both methods later.

Next, let's look at the `<input>` tags in our form; these tags are used to create various form elements.

Notice that all `<input>` tags have a `name` attribute? This attribute is used to name the respective form elements. For instance, the first form element (which is a text box) is named “studentname”, as shown in the code below:

```
<input type = "text" name = "studentname" value =
"Your Name">
```

It is mandatory for us to name form elements if we want to use PHP to process those elements. Similar to how variables in PHP have names, all form elements must have a name so that we can reference them in our code later.

Next, notice that all checkboxes (`type = "checkbox"`) in our form are named “subj” and both radio buttons (`type = "radio"`) are named “gender”?

Both radio buttons have the same name as they belong to the same group (i.e., they are both options for “Gender”).

Similarly, the three checkboxes have the same name as they are all options for “Favorite Subject(s)”. As users can choose more than one option for checkboxes, we need to add a pair of square brackets to indicate that the option(s) selected will be stored as an array.

Last but not least, notice that all `<input>` tags have a `value` attribute? This attribute behaves differently depending on the form element. For text boxes, it specifies the initial value displayed in the box. For “submit” buttons, it specifies the text on the button.

For checkboxes, radio buttons and drop down lists, it specifies the value that will be submitted for each selected option. For instance, if the following checkbox is selected

```
<input type = "checkbox" name = "subj[]" value =
"EL">English
```

the string `"EL"` will be submitted to the PHP server, as we assigned `"EL"` to its `value` attribute.

Got it? Great! Let’s move on to discuss the `isset()` function.

8.1.1 The isset() function

The isset() function is an important function that is commonly used in PHP form handling. This function checks if a variable has been declared and is not NULL.

NULL is a special value that, ironically, is used to indicate that a variable has no value.

Suppose we have two variables $a and $b as shown below:

```
$a = 5;
$b = NULL;
```

The following statements

```
var_dump(isset($a));
var_dump(isset($b));
var_dump(isset($c));
```

give us

```
bool(true) bool(false) bool(false)
```

as the output.

$a is considered set (i.e., isset($a) returns true) as it has been declared and has a non NULL value. In contrast, $b and $c are both considered unset (i.e., isset($b) and isset($c) return false) as $b has a NULL value and $c has not been declared.

The isset() function is commonly used to determine if a button has been clicked. We'll learn to do that later.

Next, let's talk about the get and post methods.

8.1.2 get and $_GET

Both the get and post methods can be used to send form data to the PHP server for processing. The main difference is that when the get method is used, form data is appended to the URL. In addition, there is a limit to the amount of data that can be transmitted using the get

method.

The form in **form.php** currently uses the `get` method. Suppose we fill in the form with the following data:

Enter Name
Alex

Favorite Subject(s)
☑ English ☑ Math ☐ Programming

Gender
◉ Male ○ Female

Submit Form

When we click on the "Submit Form" button, the URL changes to

```
http://localhost/form.php?
studentname=Alex&subj%5B%5D=EL&subj%5B%5D=MA&gender=
```

Notice the question mark (`?`) in the URL? This question mark is used as a separator; what follows the question mark in the URL is known as a query string.

The query string contains the form data that we submitted, with each data presented as a `name=value` pair, separated by ampersands (`&`).

For instance, as we entered the string "Alex" into the text field named "studentname", the pair

```
studentname=Alex
```

gets appended to the URL.

Next, we selected the options "English" and "Math" for "Favorite Subject(s)". Hence, the pairs

```
subj%5B%5D=EL&subj%5B%5D=MA
```

get appended to the URL. These pairs use URL encoding to encode special characters that cannot be displayed in an URL. `%5B` encodes the character “[” while `%5D` encodes “]”.

If you decode

```
subj%5B%5D=EL&subj%5B%5D=MA
```

you’ll get

```
subj[]=EL&subj[]=MA
```

This indicates that we submitted the values “EL” and “MA” for the “subj” form element.

Now, suppose we want to use the information stored in the query string in our PHP scripts, how do we do that?

Simple. We use the `$_GET` superglobal. `$_GET` is a superglobal that stores the query string as an associative array.

To view all data in `$_GET`, we can use the `print_r()` function. To see how that works, add the following lines of code to **form.php**, between the `</form>` and `</body>` tags:

```
<?php
  if (isset($_GET['sm']))
    print_r($_GET);
?>
```

Here, we use the `isset()` function to check if `$_GET['sm']` is set. Before we click on the “Submit Form” button (named “sm”), `$_GET['sm']` is considered to be unset.

When we click on the button, PHP assigns the string `'Submit Form'` to `$_GET['sm']` and the variable becomes set.

When that happens, `isset($_GET['sm'])` returns `true` and the `if` block is executed.

Reload **form.php** (without the query string) and enter the same data we entered previously into the form. Next, click on the “Submit Form” button; you’ll see the following line appended to the bottom of the

page:

```
Array ( [studentname] => Alex [subj] => Array ( [0]
=> EL [1] => MA ) [gender] => M [sm] => Submit Form
)
```

This gives us the content of the `$_GET[]` array.

In other words, `$_GET['studentname'] = 'Alex'`, `$_GET['subj'] = array('EL', 'MA')`, `$_GET['gender'] = 'M'` and `$_GET['sm'] = 'Submit Form'`.

Got it? Great! Let's move on to the `post` method now.

8.1.3 post and $_POST

The `post` method is similar to the `get` method, except that form data is not appended to the URL. This makes the `post` method suitable for sending sensitive information to the server, as information will not be visible in the URL.

Let's look at an example. Change the line

```
<form action = "" method = "get">
```

in **form.php** to

```
<form action = "" method = "post">
```

Next, change the PHP code to

```
if (isset($_POST['sm']))
  print_r($_POST);
```

Finally, load **form.php** again without any query string and enter the same data we entered previously into the form. Click on the 'Submit Form' button.

You should see the URL in your address bar remains unchanged. Other than that, everything else is the same. In other words, you'll get the same output appended to the bottom of the form.

Processing a form using the `get` vs `post` method is very similar. The

main difference is one appends a query string to the URL while the other doesn't. In addition, with the `get` method, we use the `$_GET` superglobal. With the `post` method, we use the `$_POST` superglobal. Clear?

8.1.4 Keeping The Values in The Form

In the previous sections, we learned to process HTML forms using the `get` and `post` methods.

Regardless of which method we use, notice that whenever we click on the "Submit Form" button, the form refreshes itself and data entered is not shown on the form anymore?

This is all right if the form is processed successfully. However, if the form is not processed successfully and we need users to resubmit the form, this can be very troublesome as users need to fill out all the fields again. If we do not want that to happen, we can use `echo` statements.

For text boxes, we use `echo` statements to modify the `value` attribute. This attribute allows us to prefill a text box with text. In **form.php**, we prefilled the text box with the text "Your Name". Alternatively, we can use PHP to prefill the text box with user-entered data. To do that,

Change

```
<input type = "text" name = "studentname" value =
"Your Name">
```

to

```
<input type = "text" name = "studentname" value = "
<?php
  if (isset($_POST['studentname']))
    echo $_POST['studentname'];
?>">
```

If you analyze the code above carefully, you'll notice that we replaced the text "Your Name" with the following block of PHP code:

```
<?php
  if (isset($_POST['studentname']))
    echo $_POST['studentname'];
?>
```

This code first checks if `$_POST['studentname']` is set.

If it is (i.e., the user has submitted data for this text field previously), we use an `echo` statement to prefill the text box with the information submitted. Got it?

It is relatively straightforward to prefill a text box with user-entered data. Things get a bit more complicated when we want to prefill elements like checkboxes, radio buttons and drop down lists. To prefill these elements, we need to know which option has been selected and use PHP to select these options dynamically. Let’s look at an example using checkboxes.

A checkbox is selected when you add the word “checked” (without quotes) to its tag. For instance,

```
<input name = "subj[]" type = "checkbox" value =
"EL" checked>English
```

results in the checkbox labelled “English” being selected. If we want PHP to dynamically select this checkbox, we need to replace the code above with:

```
<input type = "checkbox" name = "subj[]" value =
"EL"
<?php
  if (isset($_POST['subj']) && in_array('EL',
$_POST['subj']))
    echo 'checked';
?>
>English
```

Here, we use PHP to check if `$_POST['subj']` is set and if the current option is selected. If it is, the string `'EL'` should be found in the `$_POST['subj']` array. In other words, `in_array('EL', $_POST['subj'])` should return `true`.

If that's the case, we echo `'checked'` to select this checkbox. This dynamically selects the checkbox labelled "English". Got it?

Great! We'll be writing a function to prefill forms in our project later.

8.1.5 Filtering User Input

Next, let's move on to discuss how we can filter user inputs in our forms. Filtering refers to the sanitization and validation of user inputs.

Sanitization is the process of removing invalid characters from the input. For instance, sanitizing an integer involves removing all characters except digits, plus and minus signs from the input.

Validation, on the other hand, is the process of checking if the input satisfies certain criteria. For instance, if the input is supposed to be an email address, we can use the validation process to check if it is indeed an email address (such as whether it contains the "@" character).

To filter user inputs, we use a built-in function called `filter_var()`. When using this function, we need to specify how we want to filter our inputs. We do that using one of the predefined filters found at https://www.php.net/manual/en/filter.filters.php. Depending on the filter used, this function will either return the filtered data or return `false`.

For instance, if we want to remove invalid characters from an integer, we can use the `FILTER_SANITIZE_NUMBER_INT` filter as shown below:

```
$num = '12.5abc';
echo filter_var($num, FILTER_SANITIZE_NUMBER_INT);
```

If you run the code above, you'll get

```
125
```

as the output.

If we want to validate whether an email address is valid, we can use the `FILTER_VALIDATE_EMAIL` filter. If we pass a valid email address to the function, it returns the email address. Else, it returns `false`. For

instance,

```
$email = 'abc@gmail';
var_dump(filter_var($email, FILTER_VALIDATE_EMAIL));
```

gives us

```
bool(false)
```

as “abc@gmail” is not a valid email address (“.com” is missing).

8.1.6 Cross-Site Scripting

Great! We’ve covered most of the form handling concepts in PHP. Last but not least, let’s move on to cross-site scripting. For this, I need to do a little demonstration.

Change the PHP code in **form.php** (between the </form> and </body> tags) to the following:

```
if (isset($_POST['studentname']))
    echo 'You entered '.$_POST['studentname'].' into
the text field';
```

and reload the page. Type

```
<script>alert('Hacked');</script>
```

into the text field and click on the “Submit Form” button. What do you get?

You get an alert box that says “Hacked” right?

localhost says

Hacked

OK

Surprised?

This example demonstrates a critical security concept in PHP - cross-

site scripting (also known as XSS).

Cross-site scripting occurs when a user enters undesirable scripting code into our PHP form and submits the code. If our site then stores this code (for instance, in a database) and displays it to another user, the malicious code will be executed in the other user’s browser.

In our example, we created a harmless alert box using Javascript. However, users with malicious intent can definitely use this same technique to run harmful scripting code on the victim’s browser.

To prevent this from happening, we need to be careful whenever we display information entered by users. There are a few ways to do it. The easiest is to use a built-in function called htmlspecialchars(). This function converts special characters into HTML entities.

Special characters refer to characters that have special significance in HTML, such as the < and > characters. If you do not want < and > to represent the start and end of an HTML tag, you need to encode them using the htmlspecialchars() function. This function converts < and > to < and > respectively.

<script> will thus be converted to <script> and no longer interpreted as an HTML tag. To see how this works, change the PHP code above to

```
if (isset($_POST['studentname']))
  echo 'You entered
'.htmlspecialchars($_POST['studentname']).' into the
text field';
```

For demonstration purposes, we only encode the text field. In actual coding, you have to encode every form element where users are allowed to enter information themselves.

If you load the page and enter

```
<script>alert('Hacked');</script>
```

into the text field now, you’ll see that the alert box no longer pops out when you submit the form. Instead, you get the following output below

the form:

```
You entered <script>alert('Hacked');</script> into
the text field.
```

<script> and </script> are treated as normal text with no HTML significance.

If you right-click on your webpage and select “View Page Source”, you’ll see the line

```
You entered
&lt;script&gt;alert('Hacked');&lt;/script&gt; into
the text field.
```

below the </form> tag. This shows the encoding of the < and > characters.

8.2 $_SESSION

We’ve covered quite a bit in this chapter so far. In the previous section on PHP form handling, you learned about the $_GET and $_POST superglobals. In this section, we are going to discuss another superglobal - $_SESSION.

In PHP, a session is a way of storing information that can be accessed across multiple pages; this information is stored in the $_SESSION superglobal.

Like the $_GET and $_POST superglobals, the $_SESSION superglobal is an associative array. This array is stored on the server and is available to all pages on the site during the duration of the session.

To demonstrate the use of sessions, we need to create two pages, **session.php** and **session2.php**. Let’s do that now.

First, create a new file called **session.php** in **htdocs** and add the following code to it (line numbers are added for reference).

```
1  <?php
2     session_start();
```

```
3
4     $_SESSION['myFavFood'] = 'Pizza';
5     $_SESSION['myFavDrink'] = 'Cola';
6     $_SESSION['myFavColor'] = 'Orange';
7
8     #updating a session variable
9     $_SESSION['myFavDrink'] = 'Beer';
10
11    #deleting a session variable
12    unset($_SESSION['myFavColor']);
```

Here, we use a built-in function called `session_start()` to start a new session on line 2. This function creates a new session if one has not already been created or resumes the current one if it exists. Each session created is associated with an automatically generated unique session ID.

After creating the session, we store the strings `'Pizza'`, `'Cola'` and `'Orange'` into the `$_SESSION` superglobal, using `'myFavFood'`, `'myFavDrink'` and `'myFavColor'` as the keys respectively.

Next, we update the value of `$_SESSION['myFavDrink']` by assigning a new string to it on line 9.

Finally, on line 12, we use another built-in function called `unset()` to destroy an element in the `$_SESSION` superglobal. To use this function, we pass the name of the element that we want to destroy to the function. After the element is destroyed, we will no longer be able to access it.

Next, let's create another file called **session2.php** (in **htdocs)** and add the following code to it:

```
<?php
  session_start();
  echo '<BR>Food: '.$_SESSION['myFavFood'];
  echo '<BR>Drink: '.$_SESSION['myFavDrink'];
  echo 'Color: '.$_SESSION['myFavColor'];
```

Here, we start by calling the `session_start()` function to resume

the current session. We need to call this function whenever we want to access the `$_SESSION` superglobal.

Next, we use three `echo` statements to output the values stored in `$_SESSION`.

Now, we are ready to demonstrate how sessions work.

To do that, load **session.php** in your browser first. When you do that, **session.php** starts a new session and stores data into the `$_SESSION` superglobal.

Next, load **session2.php** in your browser. What do you get? Depending on the error setting on your PHP server, you will get an output similar to what is shown below:

```
Food: Pizza

Drink: Beer

Notice: Undefined index: myFavColor in …session2.php on line 6

Color:
```

Notice that we get an undefined index notice for "myFavColor"? This is because `$_SESSION['myFavColor']` was destroyed using the `unset()` function in **session.php**.

Other than that, we have successfully retrieved the values of `$_SESSION['myFavFood']` and `$_SESSION['myFavDrink']`. These two values were stored in the `$_SESSION` superglobal in **session.php** and retrieved in **session2.php**.

This demonstrates how we can use the `$_SESSION` superglobal to store and retrieve data across different PHP pages. Got it?

Besides creating and storing data in a session, we can destroy a session. We commonly do that when users log out of our site. To destroy a session, we use another built-in function called `session_destroy()`.

To see how this function works, create a new file called **session3.php**

in **htdocs** and add the following code to it:

```
<?php
  session_start();
  session_destroy();
```

As shown in the code above, we need to call the `session_start()` function to resume an existing session before destroying it.

Next, load **session3.php** in your browser; this destroys the current session.

Finally, load **session2.php** in your browser again. You'll get an output that says indexes "myFavFood", "myFavDrink" and "myFavColor" are all undefined. This indicates that the session no longer exists.

8.3 $_COOKIE

Cool! We've come to the final topic in this chapter - cookies.

A cookie is another way of storing information that can be accessed across multiple pages on the site. It is essentially a small text file that stores data on the user's computer (as opposed to a session that stores data on the server).

As cookies are stored on the user's computer, you should not use cookies to store sensitive data, as a malicious user could potentially manipulate it. Also, cookies may be disabled on the user's browser. Hence, if the information to be stored is crucial or sensitive, you should use sessions instead.

To create a cookie, we use the `set_cookie()` function.

Similar to the `header()` function covered in Chapter 3.3, the `set_cookie()` function must be called before your script generates any output. If you fail to do that, you'll get a "Cannot modify header information - headers already sent" warning and the cookie will not be set. Your code may work when you are working on your local computer, but when you upload it to your hosting company's server, it will fail to work.

The `set_cookie()` function accepts up to seven arguments; only the

first is mandatory. We commonly provide the first three arguments to the function, namely the name of the cookie, the value of the cookie and the expiry date in UNIX timestamp format.

Let’s look at an example. Create a file called **cookie.php** in **htdocs** and add the following code to it:

```
<?php
  setcookie('userName', 'Joy', time() + 120);

  #modifying a cookie
  setcookie('userAge', 25, time() + 3600);
  setcookie('userAge', 26, time() + 3600);

  #deleting a cookie
  setcookie('userLevel', 3, time() + 3600);
  setcookie('userLevel', 3, time() - 3600);
```

Here, we set three cookies - `userName`, `userAge` and `userLevel`.

For the first cookie (`userName`), its value and expiry date are `'Joy'` and `time() + 120` respectively.

`time()` is a built-in function that gives us the current Unix timestamp; `time() + 120` means the cookie will expire 120 seconds (i.e. 2 minutes) after it is set.

Next, we have the `userAge` cookie. This cookie is set twice. When a cookie is set more than once, the newest cookie overwrites the previous ones. Hence, the value of `userAge` is updated from `25` to `26`.

Finally, we have the `userLevel` cookie. This cookie is set with an initial expiry of `time() + 3600`. However, it is subsequently updated to an expiry of `time() - 3600`. When a cookie has an expiry in the past, it gets deleted.

Let’s look at how we can access each of these cookies now. We do that using the `$_COOKIE` superglobal. Create another file called **cookie2.php** in **htdocs** and add the following code to it:

```
<?php
```

```
echo 'User Name is '.$_COOKIE['userName'];
echo '<BR>User Age is '.$_COOKIE['userAge'];
echo 'User Level is '.$_COOKIE['userLevel'];
```

We are now ready to test our scripts. First, launch **cookie.php** in your browser. When you do that, the three cookies (`userName`, `userAge` and `userLevel`) get created.

Next, launch **cookie2.php**. Depending on the error setting on your PHP server, you'll get an output similar to what is shown below:

```
User Name is Joy
User Age is 26
Notice: Undefined index: userLevel in ...cookie2.php
on line 4
User Level is
```

The `userLevel` cookie does not exist as it has been deleted. Wait 2 minutes and reload **cookie2.php**. You'll get an output similar to the following:

```
Notice: Undefined index: userName in ...cookie2.php
on line 2
User Name is
User Age is 26
Notice: Undefined index: userLevel in ...cookie2.php
on line 4
User Level is
```

The `userName` cookie is also non-existent now as it has expired.

Chapter 9: Object-Oriented Programming

In the next two chapters, we are going to cover another important concept in PHP - the concept of object-oriented programming (OOP).

OOP is a major topic. Hence, a full discussion of it is beyond the scope of this book. In this chapter, we'll cover the core concepts in OOP. In the next, we'll talk about inheritance.

Ready? Let's get started.

9.1 What is OOP?

First off, what is OOP?

OOP is an approach to programming where we organize our code by grouping related variables, constants and functions into a class. This class serves as a template from which we can create what is known as objects. Objects can then be used to store data and access functions defined inside the class.

Confused? No worries. The best way to understand OOP is to look at an example. Let's write our own class now.

9.2 Writing our own class

To write our own class, we use the `class` keyword, followed by the name of the class.

The name of the class is not case-sensitive and can contain letters, numbers, or underscores. However, it cannot be a PHP reserved word (i.e., a word that has a predefined meaning in PHP, such as `echo`, `switch`, `break`, etc.) and cannot start with a number.

It is a convention to use pascal case when naming our classes. Pascal case refers to the practice of capitalizing the first letter of each word, including the first word (e.g., ClassName).

To create our class, let's first create a file in Brackets and save it as **Movie.php** to our **htdocs** folder. Next, add the following code to

Movie.php.

```
<?php

class Movie{
  //Add class members here
}
```

Here, we use the `class` keyword to declare a class called `Movie`.

Within the `Movie` class (inside the pair of braces `{}`), we are going to add variables, constants and functions. Variables and functions declared inside a class are known as properties and methods respectively. Collectively, these properties, constants and methods are known as **class members**.

We'll add properties to our `Movie` class first. To do that, add the following code to **Movie.php** (**inside** the pair of braces):

```
private $id;
public $title;
public $rentalPrice;
```

Here, we declare three properties: `$id`, `$title` and `$rentalPrice`. Notice that we did not initialize them in the code above? This is because we'll be initializing them in a special method known as the constructor later. Also, note that we preceded the property declarations with the words `public` or `private`. Don't worry about these keywords at the moment; we'll come back to them later.

Next, let's add a constant to our `Movie` class. To do that, add the following code to **Movie.php** (**inside** the pair of braces):

```
const DISCOUNT = 10;
```

Here, we define a constant called `DISCOUNT` and assign the value `10` to it. Notice that we define a constant differently here (compared to what we learned in Chapter 4.1)? Indeed, to define a constant inside a class, we do not use the `define()` function. Instead, we use the `const` keyword as shown above.

Finally, let's add some methods to our class. As mentioned previously,

a method is a function that is defined inside a class. We'll start with the constructor.

A constructor is a magic method in PHP. Magic methods are methods that have special functionalities in PHP; their names are predefined and always start with **two** underscores.

The constructor is named `__construct()` and is the first method to be called whenever we create an object from the class. We typically use this method to initialize the properties in the class.

Add the following code to the `Movie` class after the constant (but before the closing brace of the `Movie` class):

```
public function __construct($pId, $pTitle,
$pRentalPrice){
  $this->id = $pId;
  $this->title = $pTitle;
  $this->rentalPrice = $pRentalPrice;
}
```

Here, we define a constructor with three parameters: `$pId`, `$pTitle` and `$pRentalPrice`.

Inside the constructor, we initialize `$id`, `$title` and `$rentalPrice` with the values of `$pId`, `$pTitle` and `$pRentalPrice` respectively.

Notice a new keyword `$this` in the code above? We'll explain this keyword later when we learn to create objects in the next section. For now, just know that whenever we want to access the **properties and methods** of a class inside the class, we need to use the `$this` keyword, followed by the `->` operator.

Next, let's add a regular (i.e., non-magic) method to our `Movie` class. Add the following method after the `__construct()` method (but before the closing brace of the `Movie` class):

```
public function conversion($country){

  $rate = 1;
  switch($country){
```

```
        case 'UK':
          $rate = 0.76;
          break;
        case 'Japan':
          $rate = 110;
          break;
    }

    return round($rate*$this->rentalPrice, 2);
}
```

Here, we define a method called `conversion()`. This method has one parameter - `$country` - and converts USD to pounds or yen depending on the value of `$country`.

Within the method, we declare and initialize a variable called `$rate` and use a `switch` statement to update its value based on the value of `$country`.

Next, we multiply `$rate` with the `$rentalPrice` property (`$rate*$this->rentalPrice`) and pass the product as an argument to a built-in function called `round()`. This function accepts two arguments and rounds the first argument off to the precision indicated by the second. In our method, we round the product off to 2 decimal places.

Finally, we use the `return` keyword to return the result of the `round()` function.

Within the method, notice that we did not use the `$this` keyword to access `$country` and `$rate`?

This is because `$country` is a parameter while `$rate` is a local variable (i.e., a variable declared inside the method). In other words, they are not class properties. **We use the `$this` keyword only when accessing class properties and methods.** For instance, we use the `$this` keyword to access the `$rentalPrice` property.

Got it? Once you have added the `conversion()` method to the `Movie` class, our class is complete. We are now ready to make use of

this class. To do that, we need to create an object from it.

9.3 Creating an Object

To create an object, we use the `new` keyword. An object is also known as an instance of the class, and the process of creating an object is known as instantiating the class.

Create a new file in Bracket and save it as **chap9.php** to your **htdocs** folder. Add the following code to **chap9.php**:

```
<?php
  include 'Movie.php';
  $mov1 = new Movie('N0001', 'Lusso', 4.99);
```

Here, we first use the `include` statement to include **Movie.php** (which contains the code for the `Movie` class).

Next, we use the `new` keyword to create a `Movie` object called `$mov1`, passing `'N0001'`, `'Lusso'`and `4.99` as arguments to the `__construct()` method.

As `__construct()` is a magic method, we do not call it using its name.

Instead, when we create a new `Movie` object, PHP looks for the `__construct()` method and executes it for us automatically.

Remember the `$this` keyword in our `Movie` class constructor? `$this` refers to the current object. When we use the statement

```
$mov1 = new Movie('N0001', 'Lusso', 4.99);
```

to create `$mov1`, `$this` refers to `$mov1`. In other words, when PHP executes the constructor, the statement

```
$this->id = $pId;
```

in the constructor becomes

```
$mov1->id = 'N0001';
```

As a result, the value `'N0001'` gets assigned to the `$id` property of

$mov1. The same applies to the other two assignment statements in the constructor. Hence, the values 'Lusso' and 4.99 get assigned to the $title and $rentalPrice properties of $mov1 respectively.

Next, let's create a second Movie object called $mov2 using the statement below:

```
$mov2 = new Movie('P0002', 'Junior', 5.99);
```

When PHP creates $mov2 and calls the constructor, $this refers to $mov2. Hence, the statement

```
$this->id = $pId;
```

in the constructor becomes

```
$mov2->id = 'P0002';
```

The same applies to the other two assignment statements in the constructor. As a result, the values 'P0002', 'Junior' and 5.99 get assigned to the $id, $title and $rentalPrice properties of $mov2 respectively.

Got it? Good! Let's move on.

9.4 Accessing Class Members

After creating an object, we can use the object name and the -> operator to access its properties and methods.

For instance, to access the properties and methods of $mov1 in **chap9.php**, we use the code below:

```
echo $mov1->title.'<BR>';
echo $mov1->conversion('Japan').'<BR>';
```

Here, we first use the -> operator to access the $title property of $mov1. Next, we use the -> operator to call the conversion() method, passing 'Japan' as an argument to the method.

Add the code above to **chap9.php** (after the instantiation statements) and load the page, you'll get

```
Lusso
548.9
```

as the output. Pretty straightforward, right?

Now, suppose we want to access the `DISCOUNT` constant defined in the `Movie` class, how do we do that?

To access a class constant, we do not use the `->` operator. Instead, we use the `::` operator.

This is because class constants are different from class properties; they are allocated once per class, not once for each object. This means that all objects of the same class (`$mov1` and `$mov2` in our example) share the same constants. In other words, even if there are 100 `Movie` objects, there is only one memory location allocated to store the `DISCOUNT` constant.

To access the `DISCOUNT` constant in the `Movie` class, add the following code to **chap9.php**:

```
echo Movie::DISCOUNT.'<BR>';
echo $mov1::DISCOUNT.'<BR>';
echo $mov2::DISCOUNT.'<BR>';
```

If you run the code above, you'll get

```
10
```

displayed three times. This shows that we can use either the class name (`Movie`) or the object name (`$mov1` or `$mov2`) to access a class constant. All three give us the same value as they are accessing the same memory location.

9.5 Access Modifiers

Now that we know how to create objects and access class members using these objects, let's discuss a concept we skipped previously - the `public` and `private` keywords.

These two keywords are known as access modifiers; they serve as gatekeepers controlling where we can access a particular class

member.

`public` class members can be assessed everywhere while `private` class members can only be accessed within the class in which they are declared.

In our `Movie` class, we have two `public` properties (`$title` and `$rentalPrice`) and one `private` property (`$id`).

Previously, we learned to access the `$title` property of `$mov1` in **chap9.php**. As `$title` is a `public` class property, we had no problems accessing it in **chap9.php**. Now, let's try accessing the `$id` property. Add the line

```
echo $mov1->id.'<BR>';
```

to **chap9.php** and load the page again. What do you get? You get

```
Fatal error: Uncaught Error: Cannot access private
property Movie::$id...
```

added to the output, right? This is because `$id` is a `private` property. As `private` class members can only be assessed within the class in which they are declared, we are not allowed to access the `$id` property in **chap9.php**, which is outside the `Movie` class.

This is the gist of how access modifiers work; they basically control whether we can access a particular class member outside the class in which it is declared. Got it?

In PHP, access modifiers are mandatory for properties but optional for methods. If we fail to declare the modifier for a property, we'll get an error message. If we do not declare the modifier for methods, the default modifier is `public`. As of PHP 7.1.0, we can also add access modifiers for class constants. If we do not declare the modifier for constants, the default is `public`.

Besides `public` and `private` members, PHP also has `protected` members. These members can be accessed inside the class in which they are declared and any subclass that inherits from that class. We'll discuss `protected` members and inheritance in the next chapter.

At this point, some of you may be wondering why we want class members to be `private`. For instance, why bother declaring the `$id` property if we cannot access it?

The reason is that while `private` members cannot be accessed outside the class in which they are declared, they can be accessed inside it. For instance, we can write a function inside the `Movie` class to display the page heading based on the `$id` property. To see how that works, add the code below to **Movie.php** (after the `conversion()` method but before the closing brace of the `Movie` class):

```
public function displayHeading($tag){
  if (substr($this->id, 0, 1) == 'N')
    return "<$tag>Movies</$tag>";
  else
    return "<$tag>Award Winning Movies</$tag>";
}
```

This method checks if `$id` starts with `'N'` using the built-in `substr()` function. If it does, it returns an HTML element (as defined by the parameter `$tag`) with the word “Movies” enclosed. Else, it returns an element with the words “Award Winning Movies” enclosed.

Next, replace the line

```
echo $mov1->id.'<BR>';
```

in **chap9.php** with

```
echo $mov1->displayHeading('H1');
```

As `displayHeading()` is a `public` method, we have no problems accessing it outside the class. If you run the code now, you’ll get “Movies” displayed as an `<h1>` element. Got it?

9.6 Getter and Setter

In the previous section, we talked about the difference between `public` and `private` class members.

Whenever possible, we should declare class members as `private` if code outside the class does not need to access them. This act of preventing code outside from accessing class members unnecessarily is known as encapsulation.

Encapsulation makes it easy for us to make changes to class members without affecting code outside the class. For instance, if we want to change the name of the `$id` property in our `Movie` class to `$movieID`, we only need to make changes within the `Movie` class, any code outside the class is not affected. This is one of the advantages of declaring a class property as `private`.

Another advantage of declaring class properties as `private` is that it helps prevent unauthorized modifications to our object properties. To see why this is so, add the following code to **chap9.php**:

```
$mov1->rentalPrice = -20;
echo $mov1->rentalPrice.'<BR>';
```

Here, we first assign `-20` to the `$rentalPrice` property of `$mov1`. Next, we use an `echo` statement to echo its value. If you run the code above, you'll get -20 as the output.

As you can see, we manage to change the `$rentalPrice` property of `$mov1` to `-20`. This is because `$rentalPrice` is a `public` property. Hence, we can access it outside the `Movie` class and change it to any value we like. This is definitely not desirable as rental price should not be negative.

To prevent such modifications from happening, we should not declare the `$rentalPrice` property as `public`. Instead, we should declare it as `private`. Try changing the `$rentalPrice` property to `private` in **Movie.php** and run **chap9.php** again, what do you get?

You get something similar to the output below, right?

```
Fatal error: Uncaught Error: Cannot access private
property Movie::$rentalPrice in…
```

We are no longer allowed to access and modify the `$rentalPrice` property of `$mov1` as it is now a `private` property.

Whenever possible, we should always declare our class properties as `private`; this helps to prevent any unauthorized access or modifications to them. However, if we declare all our class properties as `private`, what happens if code outside the class needs to access or modify those properties?

In cases like these, we can use getters and setters. These are magic methods that allow us to provide limited and controlled access to our `private` and `protected` properties.

To see how this works, add the following methods to **Movie.php** (after the `displayHeading()` method but before the closing brace of the `Movie` class):

```
public function __get($propertyRequested){
   if ($propertyRequested == 'id')
      return 'You do not have permission to access
id.<BR>';
   else
      return $this->$propertyRequested;
}

public function __set($propertyToModify, $value){
   if ($propertyToModify == 'rentalPrice' && $value
> $this->rentalPrice)
      $this->rentalPrice = $value;
   else
      echo 'Failed to modify
'.$propertyToModify.'<BR>';
}
```

The first method (`__get()`) is known as a getter; it controls which property can be accessed outside the class and has one parameter called `$propertyRequested`. This parameter stores the name of the property we want to access.

Within the `__get()` method, we use an `if-else` statement to check If `$propertyRequested` equals `'id'`. If it equals, the `__get()` method returns a string informing us that we do not have permission to access the `$id` property. Else, it returns the property requested.

For instance, if $propertyRequested equals 'rentalPrice', it returns $this->rentalPrice.

Next, we have the __set() method, which is known as a setter. This method controls which property can be modified. It has two parameters; the first stores the name of the property we want to modify ($propertyToModify), and the second stores the new value ($value) to assign to this property.

Within our __set() method, we use an if-else statement to check if $propertyToModify equals 'rentalPrice' and if $value is greater than the current $rentalPrice value. If both conditions are met, it allows us to modify the $rentalPrice property. Else, it echoes a string informing us that it is unable to modify the property. Got it? Good!

Now, let's look at how we can use the __get() and __set() methods. To do that, we simply use the -> operator, followed by the property name. Behind the scene, as long as we have defined our __get() and __set() methods, PHP will call these magic methods automatically.

First, let's change the access modifiers of all the properties in our Movie class to private.

Next, load **chap9.php** again. Recall that previously, we tried to change the $rentalPrice property of $mov1 to -20 and got a fatal error? If you load **chap9.php** now, you'll no longer get an error. Instead, you'll get the following output:

```
Failed to modify rentalPrice
4.99
```

This is because we have defined our __get() and __set() methods.

When we try to modify the value of the $rentalPrice property, PHP calls the __set() method for us automatically. As -20 is smaller than the current $rentalPrice value (which is 4.99), our __set() method prevented us from changing the $rentalPrice value. When

we use an `echo` statement to echo the `$rentalPrice` value, the `__get()` method gives us 4.99 as the output.

Next, add the following code to **chap9.php** and run the page again:

```
$mov1->id = 'A12387';
echo $mov1->id;
```

You'll get

```
Failed to modify id
You do not have permission to access id.
```

added to the output. Here, the `__set()` method prevented us from modifying the `$id` property. Similarly, the `__get()` method prevented us from accessing the `$id` property.

Last but not least, add the following code to **chap9.php** and run the page:

```
$mov1->rentalPrice = 5.99;
echo $mov1->rentalPrice;
```

You'll get

```
5.99
```

added to the output. Here, we try to change the `$rentalPrice` value to `5.99`. As `5.99` is greater than the current `$rentalPrice` value, the `__set()` method allowed us to make the change. When we use an `echo` statement to echo the value of `$rentalPrice` again, the `__get()` method gives us 5.99 as the output.

Got it? Great!

9.7 Printing a String Representation of the Object

Besides the `__set()` and `__get()` methods, another commonly defined magic method in PHP is the `__toString()` method. Let's add one to our `Movie` class.

Add the following code to **Movie.php** (after the `__set()` method but before the closing brace of the `Movie` class):

```
public function __toString(){
    return
        'Discount = '.self::DISCOUNT.'%'.
        '<BR>Id = '.$this->id.
        '<BR>Title = '.$this->title.
        '<BR>Rental Price (USD) = '.$this->rentalPrice;
}
```

This method simply returns a string containing information about the `Movie` class. Notice a new keyword, `self`, in the method above? This keyword is used to access the `DISCOUNT` constant defined earlier in the class.

Previously, we learned that to access a class constant outside the class in which it is defined, we can use either the class name or the object name. For instance, in **chap9.php**, we used `$mov1::DISCOUNT`, `$mov2::DISCOUNT` and `Movie::DISCOUNT` to access the `DISCOUNT` constant.

What if we want to access this constant inside the `Movie` class itself (i.e., inside the class in which it is defined)? To do that, we can use either the class name or the `self` keyword.

In the `__toString()` method above, we used the `self` keyword. Alternatively, we could have used the class name (`Movie::DISCOUNT`) as well.

After declaring the `__toString()` method, we can use it to print a string representation of our `Movie` class objects. To do that, we simply use the `echo` statement.

To see how this works, add the following lines to **chap9.php** and run the page again:

```
echo '<BR>';
echo $mov1;
```

you’ll get

```
Discount = 10%
Id = N0001
Title = Lusso
Rental Price (USD) = 5.99
```

added to the output.

Chapter 10: Inheritance

In the previous chapter, we covered the fundamentals of OOP. Some of the concepts in OOP can be confusing for beginners. If this is the first time you are exposed to OOP, you may want to read through the previous chapter more than once to fully grasp the concepts.

Once you are comfortable with the topics covered in Chapter 9, we are ready to proceed to a more advanced, but equally important, concept in OOP - inheritance.

As the name suggests, inheritance has to do with the relationship between two or more classes. With inheritance, classes can be declared such that one class is a child class (also known as a subclass) of another (known as the parent class or base class). This enables us to group related classes together.

The easiest way to explain inheritance is to look at an example.

10.1 Writing the Child Classes

We'll use the `Movie` class in Chapter 9 as a parent class to illustrate. Open Brackets and create a new file called **AwardWinningMovie.php**. Save this file to your **htdocs** folder.

Add the following code to **AwardWinningMovie.php**:

```
<?php

include "Movie.php";

class AwardWinningMovie extends Movie{

  private $award;

  public function __construct($pId, $pTitle,
$pRentalPrice, $pAward){
     parent::__construct($pId, $pTitle,
$pRentalPrice);
     $this->award = $pAward;
```

```
    }
}
```

In the code above, we first use the `include` statement to include the code for the `Movie` class. Next, we create a class called `AwardWinningMovie`.

Notice a new keyword `extends` in the `AwardWinningMovie` class declaration? This keyword indicates that `AwardWinningMovie` is a child class of `Movie`.

When one class is a child class of another, it inherits all the `public` and `protected` members of the parent class. In other words, it can access and use those members directly, as if they are part of its own code. We'll illustrate what this means later.

Within the `AwardWinningMovie` class, we declare a `private` property called `$award`. Next, we declare a constructor with 4 parameters, `$pId`, `$pTitle`, `$pRentalPrice` and `$pAward`.

Within the constructor, notice a new keyword, `parent`?

As you may have guessed, this keyword is used to call the parent class constructor. As the child class has its own constructor, we need to use the `parent` keyword and the `::` operator when we want to access the parent class constructor. When we call the parent class constructor, we do not create a new object. Instead, we use the name of the constructor (`__construct()`).

In our example, we pass the values of `$pId`, `$pTitle` and `$pRentalPrice` as arguments to the parent class constructor. These values will be used to initialize the `$id`, `$title` and `$rentalPrice` properties declared in the parent class respectively.

As these properties were declared as `private` in the parent class, we are not allowed to access them directly in the child class. Instead, we can only initialize them using the parent class constructor, which is a `public` method.

Next, we have the statement

```
$this->award = $pAward;
```

This statement is used to initialize the `$award` property declared in the child class.

With that, the child class constructor is complete.

Now, let's add another method to the child class. Add the code below to the `AwardWinningMovie` class, after the `__construct()` method but before the closing brace of the class:

```
public function recommend($country){

   switch ($this->award){
      case 'Best Picture':
         $others = 'The Rail';
         break;
      case 'Best Actor':
         $others = '1729';
         break;
      default:
         $others = 'And so it begins';
   }

   return
      'You might also like:<BR>'.
      '<BR>Movie Title = '.$others.
      '<BR>Rental Price = '.$this-
>conversion($country);
}
```

Here, we declare a method called `recommend()` that has one parameter - `$country`. Within the method, we use a `switch` statement to assign different values to the local variable `$others`, depending on the value of `$this->award`.

Next, we return a string recommending a new movie to users. This string should be quite self-explanatory, except for the last part where we use the `$this` keyword to call the `conversion()` method (refer to the underlined code above) .

Recall that we do not have a `conversion()` method in the `AwardWinningMovie` class? Will we get an error when we try to call this method?

The answer is no. This is because the `AwardWinningMovie` class is a subclass of the `Movie` class. Hence, it can access all the `public` and `protected` members of the parent class directly, as if they are part of its own code.

As the parent class - `Movie` - has a `public` method called `conversion()`, the child class can access this method simply by using the `$this` keyword. Got it?

This is one of the main reasons for using inheritance; it allows us to reuse existing code. In our example, the `AwardWinningMovie` class can use the `conversion()` method directly without having to code it itself.

10.2 Creating a Child Class Object

We are now ready to create a child class object. To do that, create a new file in Brackets and save it as **chap10.php** to the **htdocs** folder.

Add the following code to it:

```
<?php
  include 'AwardWinningMovie.php';
  $awm = new AwardWinningMovie('A12324', 'Max',
6.99, 'Best Picture');

  echo $awm->recommend('Japan');
```

Here, we first use the `include` statement to include the `AwardWinningMovie` class.

Next, we use the `new` keyword to create an `AwardWinningMovie` object, passing `'A12324'`, `'Max'`, `6.99` and `'Best Picture'` as arguments to the constructor.

Finally, after creating the `$awm` object, we use it to call the `recommend()` method. If you run the code above, you'll get

```
You might also like:

Movie Title = The Rail
Rental Price = 768.9
```

as the output. Straightforward?

10.3 Access Modifiers Revisited

Good! Let’s revisit the concept of access modifiers now.

Previously, we mentioned that when one class extends another, it inherits all the `public` and `protected` class members of the class it extends. What this means is that it can access those members directly without any restrictions. However, the same does not apply to `private` class members. Child classes are not allowed to access `private` members of the parent class directly.

To understand what this means, try changing the `conversion()` method in the `Movie` class to a `private` method. In other words, change the line

```
public function conversion($country)
```

in **Movie.php** to

```
private function conversion($country)
```

Next, load **chap10.php** again. What do you get? You get an error message similar to the output below, right?

```
Fatal error: Uncaught Error: Call to private method
Movie::conversion() from context
'AwardWinningMovie'...
```

This is because when `conversion()` is a `private` method, we are not allowed to access it directly outside the class in which it is defined (i.e., outside the `Movie` class). Hence, the `recommend()` method in the `AwardWinningMovie` class is not allowed to access it.

Next, change `conversion()` to a `protected` method and load **chap10.php** again. Everything works now, right? This is because child

classes are allowed to access protected members in their parent class directly. Hence, the recommend() method in the AwardWinningMovie class is now allowed to access the conversion() method in the Movie class. Got it? Great!

After changing conversion() to a protected method, note that we are only allowed to access it directly inside the class in which it is declared and any subclass that inherits from that class. In other words, we can only access it directly inside the Movie and AwardWinningMovie classes.

Recall that previously, we accessed the method in **chap9.php**? This was all right when conversion() was a public method. Now that it is a protected method, you'll get an error if you load **chap9.php** again. If you want to access the conversion() method in **chap9.php** directly, you have to change it back to a public method.

10.4 Overriding

Now that you are familiar with child classes and inheritance, let's move on to discuss the concept of overriding. Overriding occurs when a child class modifies the methods it inherits from its parent class.

To illustrate what this means, add the following line to **chap10.php** and load the page.

```
echo $awm->displayHeading('H1');
```

You'll get the text "Award Winning Movies" displayed as a <H1> heading.

Here, we are calling the displayHeading() method declared in the parent class. This method is a public method inherited by the child class. Hence, we can access it using an AwardWinningMovie object.

Next, let's see what happens if we code a new version of the displayHeading() method inside the child class itself. To do that, add the following method to the AwardWinningMovie class, after the recommend() method but before the closing brace of the class:

```
public function displayHeading($tag){
   $baseMsg = parent::displayHeading($tag);
   return $baseMsg.$this->award;
}
```

Here, we declare a method with the same name (`displayHeading`) and parameter list (`$tag`) as the method in the parent class. When that happens, we say that the child class method overrides the parent class method.

Within the child class method, we use the `parent` keyword to call the parent class `displayHeading()` method. Whenever a child class overrides a parent class method, we need to use the `parent` keyword (instead of the `$this` keyword) if we want to access the parent class method inside the child class.

Next, we assign the result returned by the parent class method to a variable called `$baseMsg`. Finally, we concatenate `$baseMsg` with the `$award` property in the child class and return the result.

If you load **chap10.php** again, you'll see the text "Award Winning Movies" displayed as a `<H1>` heading, followed by a line that says "Best Picture". This additional line illustrates that we are now accessing the `displayHeading()` method in the child class.

As `displayHeading()` has been overridden in the child class, the child class method is called when we access it using a child class object (`$awm`).

If we want to access the `displayHeading()` method in the parent class, we have to use a parent class object. For instance, we can do it as follows:

```
$mv = new Movie('A3244', 'Golden Rose', 3.99);
echo $mv->displayHeading('H1');
```

If you add the code above to **chap10.php** and load the page, you'll only get "Award Winning Movies" (without "Best Picture") added to the output. This illustrates that `$mv` is accessing the `displayHeading()` method in the parent class.

Chapter 11: Interacting with a Database

In this chapter, let's move on to talk about databases. Whenever we work with websites, it is highly likely that we need to store information (such as data entered by users) in a database.

This chapter assumes that you are familiar with MySQL. If you are not, I strongly recommend that you check out my other book, "Learn SQL (using MySQL) in One Day and Learn it Well" before proceeding.

If you are familiar with MySQL, let's proceed.

11.1 The PDO library

To use a database in our PHP programs, we can use one of two built-in libraries.

A library is a collection of code that other developers have written and made available for us to use in our own programs. The two libraries available for connecting to a database in PHP are the `MySQLi` and `PDO` libraries.

Both essentially do the same job, connecting to the database and sending queries to it. However, there are some key differences between the two.

The main difference is that we can use the `PDO` library to connect to 12 different types of database servers - such as a MySQL server, an Oracle server, or a Microsoft SQL Server. In contrast, the `MySQLi` library only works with MySQL database servers.

For this reason, we'll use the `PDO` library in this book.

11.2 Connecting to the Database

To send queries to our database using `PDO`, we need to connect to it first.

To do that, we need four pieces of information - the hostname of the server, the name of the database, the username of the account

authorized to access that database and the password.

In most cases, the hostname is simply “localhost”. To get the other three pieces of information, you need to contact your database administrator. For the examples in this book, we’ll be our own database administrator and create a database and user account using phpMyAdmin in XAMPP later.

For now, let’s suppose you have a database called “pawszone” and a user account with username “pz_admin” and password “ABCD”.

In the following sections, we’ll first cover various concepts you need to know to work with `PDO`. After covering the concepts, we’ll get some hands-on practice with a PHP script that shows how the concepts can be applied. Ready?

Let’s learn to connect to our database first.

`PDO` uses an object-oriented approach. To connect to our database, we create a `PDO` object. For instance, to connect to the “pawszone” database, we use the code below:

```
$pdo = new
PDO("mysql:host=localhost;dbname=pawszone",
"pz_admin", "ABCD");

$pdo->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
```

On the first line, we create a `PDO` object by passing three arguments to the `PDO` constructor.

The first argument is a string indicating the type of database we are using (`mysql:` in our case), the hostname of the server (`host=localhost`), and the name of the database (`dbname=pawszone`).

The second argument is the username (`"pz_admin"`) and the third is the password (`"ABCD"`).

After creating this object, we assign it to a variable called `$pdo` and use `$pdo` to call the `setAttribute()` method on the next line.

The `setAttribute()` method allows us to set the attributes of a `PDO` object. Here, we are trying to set the error mode.

By default, after establishing a successful connection, `PDO` does not inform us of anything that goes wrong subsequently. For instance, if we try to select data from a table that does not exist, `PDO` will not inform us that there's no such table. Instead, we'll get a page with no results. This is very frustrating if we are trying to figure out what is wrong.

To get `PDO` to inform us of any issues, we need to use the `setAttribute()` method to configure the error mode. To do that, we pass two predefined constants to the method.

The first constant tells `PDO` the attribute we want to set, and the second tells `PDO` the value we want to set it to. You can find the list of all predefined `PDO` constants at https://www.php.net/manual/en/pdo.constants.php.

In our example, we want to set the error mode attribute (`PDO::ATTR_ERRMODE`) to `PDO::ERRMODE_EXCEPTION`.

In this mode, `PDO` throws an exception when something goes wrong. We'll learn more about exceptions in the next chapter. For now, just know that we can use the information in this exception to figure out what went wrong.

That's it! That's all that's needed to connect to a database. After connecting to our database, we are ready to send queries to it.

11.3 SQL Injection

There are two main types of queries in MySQL - ones that return data from the database (e.g., `SELECT`) and ones that don't (e.g., `CREATE`, `INSERT`, `UPDATE` and `DELETE`).

The easiest way to send queries that do not return data is to use the `exec()` method, while the easiest way to send queries that do is to use the `query()` method.

However, both methods put us at risk of a form of attack known as an

SQL injection.

Recall that we talked about cross-site scripting in Chapter 8.1.6? SQL injection is similar to that in that hackers can input data into our website and use it to hack our database.

Suppose we have a text field called “ownerName” and we want to select rows from a table called “pets” in our database, we can use the code below:

```
$owner = $_POST['ownerName'];

$sql = "SELECT * FROM pets WHERE owner = '$owner'";
$stmt = $pdo->query($sql);
```

Don’t worry if the code above does not make much sense yet. We’ll discuss a better method to query the database later. For now, just know that in the code above, we use the user input `$_POST['ownerName']` in our SQL `SELECT` statement and use the `query()` method to execute the statement.

If our user enters

```
Jamie
```

into the “ownerName” text field, the `SELECT` statement becomes

```
SELECT * FROM pets WHERE owner = 'Jamie'
```

and everything works as expected. However, suppose the user enters

```
Jamie' OR owner != 'Jamie
```

into the text field instead, the `SELECT` statement becomes

```
SELECT * FROM pets WHERE owner = 'Jamie' OR owner !=
'Jamie'
```

When we use the `query()` method to execute this statement, PHP interprets the word “OR” in the user input as the SQL operator `OR`. Hence, it returns ALL rows from the “pets” table, since `owner = 'Jamie' OR owner != 'Jamie'` covers all possible values for the “owner” column. This is definitely not desirable.

Instead of using the `query()` or `exec()` method to execute SQL statements, a better way is to use prepared statements.

11.4 Prepared Statements

There are a few steps involved when using prepared statements:

1) Create the SQL statement using placeholders
2) Prepare the statement
3) Bind variables or values to the placeholders
4) Execute the statement
5) Fetch any data returned if necessary

Step 1: Creating the SQL statement using placeholders

When using prepared statements, we need to use placeholders to replace any variables that contain user inputs in our SQL statements. These placeholders can either be named or unnamed.

An unnamed placeholder is represented by a question mark. For instance, if we want to select information from the “pets” table using user inputs for two columns named “owner” and “petname”, we can use the SQL statement below:

```
$sqlUnnamed = "SELECT * FROM pets WHERE owner = ?
AND petname = ?";
```

Here, we use question marks (`?`) to replace user inputs in our SQL statement and assign the statement to a variable called `$sqlUnnamed`. Alternatively, we can use named placeholders as shown below:

```
$sqlNamed = "SELECT * FROM pets WHERE owner = :owner
AND petname = :pname";
```

Here, we use `:owner` and `:pname` to replace user inputs and assign the statement to a variable called `$sqlNamed`.

Step 2: Prepare the statement

After creating the statement, we need to prepare it. Preparing a statement means sending the statement to your database server in

advance, giving the server a chance to analyze, compile, and optimize its plan for executing the query. The actual execution does not happen at this step.

To prepare the statements, we use the `prepare()` method. This is a built-in method in the `PDO` class that prepares a SQL statement for execution and returns a `PDOStatement` object.

Suppose we have a `PDO` object called `$pdo`, to prepare the `$sqlUnnamed` statement, we write:

```
$preparedUnnamed = $pdo->prepare($sqlUnnamed);
```

Here, we use `$pdo` to call the `prepare()` method and assign the object returned to a variable called `$preparedUnnamed`.

To prepare the `$sqlNamed` statement, we write

```
$preparedNamed = $pdo->prepare($sqlNamed);
```

Here, we assign the object returned to a variable called `$preparedNamed`.

After preparing the statements, we won't be using the `$pdo` object anymore. Instead, we'll be using the `PDOStatement` objects (i.e., `$preparedUnnamed` and `$preparedNamed`) in subsequent steps.

Step 3: Binding values to the placeholders

In step 3, we need to bind values to the placeholders in our SQL statements. This tells the database server what values to replace the placeholders with when executing the statements.

We can use either the `bindValue()` or `bindParam()` method to bind values to placeholders. Both methods are similar, but `bindValue()` tends to be more versatile. We'll be using the `bindValue()` method in the examples that follow.

To bind values to <u>unnamed placeholders</u>, we typically provide two pieces of information to the `bindValue()` method - the position of the placeholder and the value to bind. The positions of placeholders start from 1.

Recall that we have a `SELECT` statement named `$sqlUnnamed` defined and prepared as follows?

```
$sqlUnnamed = "SELECT * FROM pets WHERE owner = ?
AND petname = ?";
$preparedUnnamed = $pdo->prepare($sqlUnnamed);
```

Suppose we want to bind a user input called `$myPet` to the second question mark in `$sqlUnnamed`, this is how we can do it:

```
$preparedUnnamed->bindValue(2, $myPet);
```

Here, we use `$preparedUnnamed` to call the `bindValue()` method.

The first argument (`2`) indicates that we are binding a value to the second placeholder (i.e., the placeholder for the "petname" column) while the second argument (`$myPet`) indicates the value to bind.

To bind values to named placeholders, we replace the first argument with the name of the placeholder, enclosed in quotation marks. For instance, if we want to bind `$myPet` to the `:pname` placeholder in `$sqlNamed`, we do it as follows:

```
$preparedNamed->bindValue(':pname', $myPet);
```

The first argument (`':pname'`) indicates that we are binding a value to the `:pname` placeholder while the second (`$myPet`) indicates the value to bind.

At this point, some of you may wonder if `$myPet` (which stores a user input) puts us at risk of an SQL injection attack.

The answer is no. This is because `PDO` knows that we're sending it the value for a placeholder; hence, it'll not interpret any part of the string stored in `$myPet` as SQL code.

For instance, even if `$myPet` stores the string

```
Max' OR petname != 'Max
```

the word "OR", the quotation marks and the != operator will not be interpreted as SQL code. Instead, `PDO` considers the entire string to be the value for the "petname" column and simply searches for pets

with that weird name. Got it?

Step 4: Execute the statement

After binding values to our placeholders, we need to use the `PDOStatement` object to call the `execute()` method. To execute `$preparedUnnamed`, we write:

```
$preparedUnnamed->execute();
```

To execute `$preparedNamed`, we write

```
$preparedNamed->execute();
```

Step 5: Fetch the data returned, if any

After executing the statement, we can use either the `fetch()` or `fetchAll()` method to fetch any data returned (if the statement is a `SELECT` statement).

The `fetch()` method fetches the rows one by one while the `fetchAll()` method fetches all rows at once. Both methods return `false` on failure.

When using either method, we can specify the style we want the method to use when fetching the rows. Commonly used styles include the `FETCH_ASSOC` and `FETCH_NUM` styles.

The `FETCH_ASSOC` style fetches the rows as an associative array (using column names in the table as keys) while the `FETCH_NUM` style fetches them as an indexed array (based on the order of the columns returned by the `SELECT` statement).

For instance, to fetch the rows for `$sqlUnnamed`, we can use the following `while` loop:

```
while($row = $preparedUnnamed-
>fetch(PDO::FETCH_NUM)){
  echo 'Pet = '.$row[1].'<BR>';
  echo '<BR>Owner = '.$row[0].'<BR>';
}
```

This loop uses the assignment statement

```
$row = $preparedUnnamed->fetch(PDO::FETCH_NUM)
```

as the condition for looping. This is permissible and is a common approach when working with the `PDO fetch()` method.

The assignment statement uses `$preparedUnnamed` (which is the prepared statement for `$sqlUnnamed`) to call the `fetch()` method, specifying the fetch style as `PDO::FETCH_NUM`.

Next, it assigns the result returned to a variable called `$row`.

This variable controls when the `while` loop ends. The loop keeps looping as long as `$row` is not `false`. Within the `while` loop, we use two `echo` statements to display the values of the array returned by the `fetch()` method.

`row[0]` gives us the first column (i.e., the “owner” column) in `$sqlUnnamed` while `row[1]` gives us the second column (i.e., the “petname” column).

When we reach the end of the result set and the `fetch()` method is unable to fetch any more rows, it returns `false`. The `while` loop then stops looping as `$row` is now `false`.

Besides using the `FETCH_NUM` style, we can use the `FETCH_ASSOC` style to fetch our data. This is how we do it:

```
while($row = $preparedUnnamed-
>fetch(PDO::FETCH_ASSOC))
{
  echo 'Pet = '.$row['petname'].'<BR>';
  echo '<BR>Owner = '.$row['owner'].'<BR>';
}
```

Both loops give us the same output; we’ll have a chance to run them and see the results later.

In addition to using the `fetch()` method, we can use the `fetchAll()` method to fetch all rows from the database at once. We’ll do that in the project at the end of the book.

That’s it. We’ve covered the gist of how prepared statements work. We are now ready to try the different SQL concepts covered above.

11.5 Putting it all Together

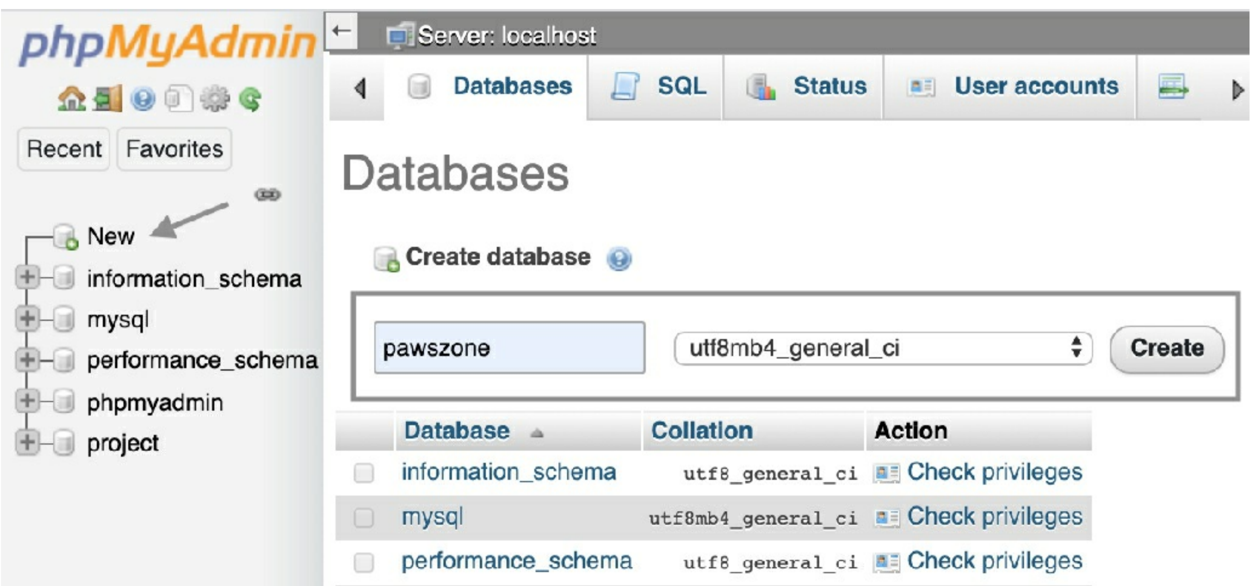
To do that, we need to create a database and user account first.

First, ensure that you have started XAMPP and the Apache and MySQL servers.

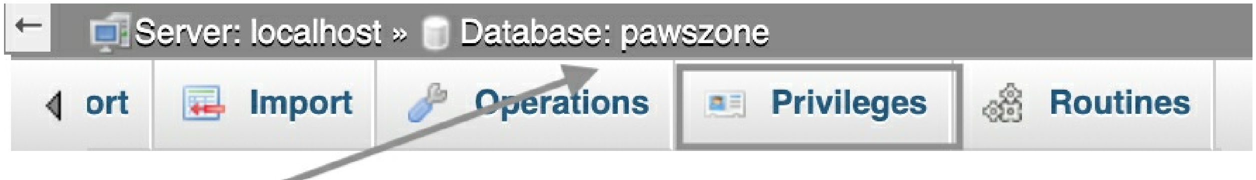
If you have problems starting the MySQL server, it is likely due to a port conflict. Check out https://learncodingfast.com/how-to-install-xampp-and-brackets for instructions on how you can resolve the issue.

Next, launch http://localhost/phpmyadmin/.

Click on “New” on the left and enter “pawszone” (without quotes) into the text field on the right.



Click “Create” to create the database. You’ll be directed to the “pawszone” database. You can verify that by checking the gray bar at the top.



Next, click on “Privileges” and you’ll be directed to the page for creating a new user. At the bottom of the page, you’ll see a link that says “Add user account”. Click on it and enter the information shown in the screenshot below (leave other fields unchanged). Enter “ABCD” (without quotes) into the two password fields.

Login Information

User name: Use text field: pz_admin

Host name: Local localhost

Password: Use text field: ••••

Strength: Extremely weak

Re-type: ••••

Scroll to the bottom of the page and click “Go” to create the user.

Once you are done, you’ll see a message that says “You have added a new user”. You are now ready to connect to the database you have just created.

Create a new file in Brackets and save it as **sql_cud.php**.

Add the following code to it (you can download the code at https://learncodingfast.com/php).

```
<?php

//SECTION A - CONNECT TO DATABASE

$pdo = new PDO("mysql:host=localhost;dbname=pawszone", "pz_admin",
```

```
"ABCD");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

//SECTION B - CREATE TABLE

$sql = "CREATE TABLE IF NOT EXISTS pets ( owner VARCHAR(255) NOT
NULL, petname VARCHAR(255) NOT NULL , breed VARCHAR(255) NOT NULL,
microchip VARCHAR(20), PRIMARY KEY(owner, petname))";

$stmt = $pdo->prepare($sql);

$stmt->execute();

//SECTION C - INSERT DATA

$sql = "INSERT INTO pets (owner, petname, breed)
VALUES (:owner, :petname, :breed)";

$stmt = $pdo->prepare($sql);

$owner = array('Ted', 'Jamie', 'En', 'En');
$pname = array('Angel', 'Max', 'Boots', 'Dora');
$breed = array('Labradoodle', 'Domestic Shorthair', 'Domestic
Shorthair', 'Munchkin');

for ($i = 0; $i < 4; ++$i)
{
   $stmt->bindValue(':owner', $owner[$i]);
   $stmt->bindValue(':petname', $pname[$i]);
   $stmt->bindValue(':breed', $breed[$i]);

   $stmt->execute();
}

//SECTION D - UPDATE DATA

$sql = "UPDATE pets SET microchip = :micro WHERE owner = :owner AND
petname = :petname";

$stmt = $pdo->prepare($sql);

$stmt->bindValue(':micro', '121342345');
$stmt->bindValue(':owner', 'Jamie');
$stmt->bindValue(':petname', 'Max');

$stmt->execute();

//SECTION E - DELETE DATA

$sql = "DELETE FROM pets WHERE owner = :owner AND petname =
:petname";

$stmt = $pdo->prepare($sql);

$stmt->bindValue(':owner', 'Ted');
$stmt->bindValue(':petname', 'Angel');

$stmt->execute();
```

In the code above, we first connect to the “pawszone” database in Section A.

Next, in Section B, we use a SQL `CREATE` statement to create a table called “pets” in our database. As this SQL statement does not have any placeholder, we do not need to call the `bindValue()` method. Instead, we simply prepare the statement and execute it.

In Section C, we have an `INSERT` statement for inserting data into the “pets” table. We first use the `prepare()` method to prepare this statement. Although we’ll be executing the `INSERT` statement four times using a `for` loop later, note that we only need to prepare the statement once. This is another advantage of using prepared statements as it can result in faster execution, especially if the query is complex or needs to be run many times.

After preparing the statement, we declare three arrays - `$owner`, `$pname` and `$breed` - and assign the data to be inserted to these arrays. Next, we use a `for` loop to loop through the arrays.

Each time the loop runs, we bind one value in each array to the relevant placeholder. After binding the data, we call the `execute()` method to execute the `INSERT` statement.

After inserting the data in Section C, we update a row (where owner = “Jamie” and petname = “Max”) in Sections D and delete another (where owner = “Ted” and petname = “Angel”) in Section E. These two sections should be quite self-explanatory.

Read through the code above carefully and fully understand it before proceeding. Once you are clear on how prepared statements work, load the page **sql_cud.php** in your browser to run the code.

Next, head over to http://localhost/phpmyadmin/ and select the “pawszone” database on the left. You should see the “pets” table listed on the right. Click on “Browse” and you’ll get the following table:

				owner	petname	breed	microchip
☐	Edit	Copy	Delete	En	Boots	Domestic Shorthair	NULL
☐	Edit	Copy	Delete	En	Dora	Munchkin	NULL
☐	Edit	Copy	Delete	Jamie	Max	Domestic Shorthair	121342345

As you can see, the “pets” table has been successfully created and data has been added to it. Got it?

Next, let’s learn to select data from this table and display it on our web page using PHP. To do that, create a new file called **sql_select.php** and add the following code to it.

```
<?php

//Section A - Connecting to the database

$pdo = new PDO("mysql:host=localhost;dbname=pawszone", "pz_admin",
"ABCD");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

//Section B - SELECT all rows from pets

$sql = "SELECT petname, owner FROM pets";
$stmt = $pdo->prepare($sql);

$stmt->execute();

while ($row = $stmt->fetch(PDO::FETCH_ASSOC)){

   echo '<BR>Owner = '.$row['owner'].'<BR>';
   echo 'Pet Name = '.$row['petname'].'<BR>';
}

$stmt->execute();

while ($row = $stmt->fetch(PDO::FETCH_NUM)){
   echo '<BR>Owner = '.$row[1].'<BR>';
   echo 'Pet Name = '.$row[0].'<BR>';
}
```

Here, as before, we first connect to the “pawszone” database in Section A. Next, in Section B, we use a `SELECT` statement to select the “petname” and “owner” columns from the “pets” table.

After executing the statement, we use a `while` loop to fetch the rows one by one, using the `FETCH_ASSOC` style. Finally, we use `echo` statements inside the `while` loop to display the data fetched.

Next, we execute the `SELECT` statement again and use another `while` loop to fetch the rows using the `FETCH_NUM` style.

If you load **sql_select.php** in your browser, you'll get the following output displayed twice:

```
Owner = En

Pet Name = Boots

Owner = En

Pet Name = Dora

Owner = Jamie

Pet Name = Max
```

Chapter 12: Managing Errors and Exceptions

We've come to the last chapter before our project. In this chapter, we are going to learn to manage errors and exceptions in our PHP scripts.

Frankly speaking, this is one of the messiest areas in PHP as PHP differentiates between an exception and an error. When something goes wrong, we'll get either an exception or an error.

Internal PHP functions mainly give us errors when something goes wrong. Modern object-oriented extensions (such as the `PDO` extension we learned in the previous chapter), on the other hand, are more likely to use exceptions.

In this chapter, we'll first discuss exceptions and errors separately before consolidating everything into a single file in the last section.

Ready? Let's look at exceptions first.

12.1 Handling Exceptions

12.1.1 What is an exception?

An exception is a predefined object in PHP that allows us to alter the flow of our scripts when something undesirable happens.

For instance, if we try to connect to a database and the connection fails, we can use an exception to help us handle the situation. To see what an exception is, create a file called **connect.php** in **htdocs** and add the following code to it:

```
<?php

$pdo = new
PDO("mysql:host=localhost;dbname=pawszone",
"wrongadmin", "ABCD");
$pdo->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
```

Here, we try to connect to the “pawszone” database using an invalid username (“wrongadmin”).

Next, create a new file called **exceptionHandling.php** in **htdocs** and add the following code to it:

```
<?php
  include "connect.php";
  echo '<BR>Welcome to Pawszone';
```

Here, we use an `include` statement to include the **connect.php** file (which contains the invalid code) and use an `echo` statement to echo the text “Welcome to Pawszone” after the `include` statement.

After creating the two files, try loading **exceptionHandling.php** in your browser (with Apache and MySQL running), what do you get?

You get a message similar to what is shown below, right? I’ve formatted and underlined parts of the message to make it more readable.

```
Fatal error: Uncaught PDOException: SQLSTATE[HY000]
[1045] Access denied for user
'wrongadmin'@'localhost' (using password: YES) in
/opt/lampp/htdocs/connect.php:3


Stack trace:

#0 /opt/lampp/htdocs/connect.php(3): PDO-
>__construct('mysql:host=loca...', 'wrongadmin',
'ABCD')

#1 /opt/lampp/htdocs/exceptionHandling.php(2):
include('/opt/lampp/htdo...')

#2 {main} thrown in /opt/lampp/htdocs/connect.php on
line 3
```

This message informs us that a `PDO` exception (`PDOException`) has occurred and is not caught (i.e., not handled). This exception occurs because of the invalid username we used when connecting to the

“pawszone” database.

When an exception occurs and is not caught, PHP displays a message similar to the one shown above. This output includes a stack trace listing all the files linked to the exception. The trace above tells us that the current exception is linked to line 3 in **connect.php** and line 2 in **exceptionHandling.php**. In addition, the database password, “ABCD”, is revealed in the trace.

Notice that the second statement (`echo '<BR>Welcome to Pawszone';`) in **exceptionHandling.php** is not executed? This is because any exception that is not caught results in a fatal error; the script is terminated immediately before the `echo` statement can be executed.

If we do not want script termination to happen (or the message above to be displayed) when an exception occurs, we need to catch the exception.

12.1.2 try-catch-finally

To catch exceptions, we use the `try`, `catch` and `finally` blocks. The syntax is as follows:

```
try {
  //do something
}
catch (type of exception)
{
  //do something else when an exception occurs
}
finally {
  //do this regardless of whether the try or catch
block is executed
}
```

The `try` block first tries to perform a task.

If the task fails, an exception will be thrown. Throwing an exception simply means PHP will create an object using one of the predefined

classes for dealing with exceptions.

These classes include the `Exception`, `PDOException`, `DOMException` and `OutOfBoundsException` class.

When an exception occurs, PHP creates an object of the relevant exception class. This object contains information about the exception and can be used to call various methods defined in the class.

These methods include the `getMessage()`, `getFile()`, `getLine()` and `getTraceAsString()` methods, which give us the message, file name, line number and stack trace of the exception respectively.

After an exception is thrown (i.e., after PHP creates the exception object), we need to catch it using a `catch` block. We can define multiple `catch` blocks to deal with different types of exceptions.

After the `catch` block(s), we have the `finally` block. This block is optional and always executed regardless of whether the `try` block succeeds or fails.

To illustrate how the `try`, `catch` and `finally` blocks work, replace the original code in **exceptionHandling.php** with the following code:

```
<?php

try{
  include "connect.php";
}catch(PDOException $e){
  echo '<BR>Unable to connect '.$e->getMessage();
}catch(Exception $e){
  echo '<BR>Something else happened'.$e-
>getMessage();
}finally{
  echo '<BR><BR>The finally block is always
executed';
}

echo '<BR>After connecting';
```

Here, we first try to connect to the “pawszone” database inside the `try` block.

If we are unable to connect, PHP throws an exception and one of the `catch` blocks will be executed.

In our example, we have two `catch` blocks. The first `catch` block catches a `PDO` exception (`PDOException`) while the second catches a general exception (`Exception`).

`PDOException` is a predefined class that deals with exceptions when `PDO` is unable to perform its task. `Exception`, on the other hand, is the parent class of all exception classes and deals with general exceptions.

`$e` represents an object of the respective exception class. Inside our `catch` blocks, we use `$e` to call the built-in `getMessage()` method and use an `echo` statement to echo the error message.

If we run the code above, we'll get the following output:

```
Unable to connect SQLSTATE[HY000] [1045] Access
denied for user 'wrongadmin'@'localhost' (using
password: YES)

The finally block is always executed
After connecting
```

The first line of output is from the `catch(PDOException $e)` block. As PHP is unable to connect to the database in **connect.php**, it throws a `PDO` exception.

The second line is from the `finally` block. This block is executed regardless of whether the code in the `try` block succeeds or fails.

Finally, the last line is from the `echo` statement after the `try-catch-finally` blocks. This line illustrates that when an exception occurs and is caught successfully, the script continues executing. Straightforward? Good!

12.1.3 Throwing Exceptions

Next, let’s learn to throw our own exceptions.

In the section above, we learn to catch exceptions thrown by PHP. Besides relying on PHP to throw exceptions for us, we can throw our own exceptions. We do that using the `throw` keyword.

Let’s look at an example. Add the function below to the end of **exceptionHandling.php** (i.e., after the `echo '<BR>After connecting';` statement):

```
function displayUserInput($userInput){
  if ($userInput > 100)
  {
    throw new OutOfRangeException('<BR>User input
is too big');
  }else
  {
    echo '<BR>'.$userInput;
  }
}
```

Here, we define a function called `displayUserInput()` that has one parameter - `$userInput`.

If `$userInput` is greater than `100`, we throw an `OutOfRangeException` exception using the `throw` keyword. `OutOfRangeException` is a predefined class in PHP extended from the `Exception` class. Throwing an `OutOfRangeException` exception involves creating a new object of the class.

When creating a new `OutOfRangeException` object, we can pass a few optional arguments to the constructor. We commonly pass an error message. In our example, we pass the string `'<BR>User input is too big'` to the constructor.

If we call the `displayUserInput()` function using the `try-catch` blocks below:

```
try
{
```

```
  displayUserInput(105);
}catch (OutOfRangeException $e)
{
  echo $e->getMessage();
}
```

an `OutOfRangeException` exception will be thrown in the `try` block as `$userInput` is greater than `100`. This gets caught by the `catch` block. Inside the catch block, we use the `getMessage()` method to get the error message.

If we run the code above, we'll get

```
User input is too big
```

as the output. If we change the function call to

```
displayUserInput(16);
```

no exception will be thrown and we'll get 16 as the output. Got it? Great!

12.1.4 Exception Handler

In the previous sections, we learned to throw and catch exceptions. Some exceptions, such as a `PDO` exception, are easy to pre-empt. Others may be harder. In most cases, despite our best efforts, there are bound to be exceptions we fail to anticipate.

In cases like these, PHP allows us to specify a custom exception handler that gets called whenever there are exceptions we fail to catch. We can use this custom handler to display a custom message to users. However, after the exception handler is called, the PHP script will be terminated.

To see how an exception handler works, add the following code to the end of **exceptionHandling.php**:

```
function myExceptionHandler($e)
{
  echo '<BR>Oppsss... An uncaught exception
occurred.<BR>'.$e->getMessage();
```

```
}
set_exception_handler('myExceptionHandler');
```

Here, we define a function called `myExceptionHandler()` to serve as the exception handler. For a function to qualify as an exception handler, it needs to have one parameter for storing the uncaught exception object. This parameter is `$e` in our example.

Within the function, we use `$e` to call the `getMessage()` method and use an `echo` statement to echo the message.

After coding the function, we need to use a built-in function called `set_exception_handler()` to set `myExceptionHandler()` as the exception handler. To do that, we write

```
set_exception_handler('myExceptionHandler');
```

To test our exception handler, add the following lines to the end of **exceptionHandling.php**:

```
$pdo = new PDO("some invalid database");
echo 'This will not be executed';
```

Here, we try to create a `PDO` object using an invalid argument. This results in a `PDO` exception, which is not caught by any `catch` block.

If you run the code above, you'll get

```
Oppsss... An uncaught exception occurred.

invalid data source name
```

added to the output. The exception handler (`myExceptionHandler`) has successfully handled the exception. However, notice that the `echo` statement after the `PDO` instantiation statement is not executed? This is because the PHP script got terminated after the exception handler is called. Got it?

12.2 Handling Errors

12.2.1 What are errors?

Great! Now that you are familiar with exceptions, let’s move on to discuss errors. As mentioned previously, when something goes wrong in PHP, you will get either an exception or an error.

One major difference between errors and exceptions is that exceptions can be caught while errors, by default, cannot. This is especially true prior to PHP 7. From PHP 7 onwards, PHP throws an `Error` exception when *some* errors occur; this exception can then be caught.

There are four main types of errors defined in PHP - notices, warnings, syntax errors and fatal errors. Each error type is further divided into subtypes and each subtype is represented by a constant and an error code.

Warnings and notices are minor errors that do not lead to script termination. Common causes include trying to divide by zero (which generates a warning) and trying to access an undefined variable (which generates a notice).

Syntax errors, on the other hand, lead to script termination. They are caused by programmers making a mistake in the syntax of their code. Examples include forgetting to close a parenthesis or missing a semicolon.

Last but not least, we have fatal errors, which also lead to script termination. Common causes include trying to call a non-existent function.

Do not worry about the distinction between the different types of errors. When an error occurs, PHP has an excellent reporting system. For instance, if you try to access a non-existent variable using the statement

```
echo $some_undefined_variable;
```

you’ll get an output similar to what is shown below:

```
Notice: Undefined variable: some_undefined_variable
in ...mycode.php on line 6
```

This output informs us that the error is a notice and gives us the

reason for the error. It also gives us the file name of the error and the line number where it occurred.

As you can see, an error message is very informative. Such messages make it very easy for us to debug our code. However, they can reveal too much information to end users.

Therefore, it is a common practice for us to display these messages only during the development stage of our site. When our site goes live, it is strongly recommended that we do not display such messages to end users.

To adjust whether error messages are displayed, we need to update the error settings in our PHP server.

12.2.2 Error Reporting Settings in PHP

There are two relevant settings here - `error_reporting` and `display_errors`.

The `error_reporting` setting affects the type of errors that PHP will report. There are several predefined constants that we can assign to this setting.

If we want PHP to only report fatal errors (and not report warnings and notices etc.), we assign the `E_ERROR` constant to this setting. On the other hand, if we want PHP to report all errors, we assign `E_ALL` to it.

Besides `E_ERROR` and `E_ALL`, we can assign other constants to the `error_reporting` setting. You can find the list of all error constants and codes at https://www.php.net/manual/en/errorfunc.constants.php.

Next, we have the `display_errors` setting. This affects whether errors reported by PHP will be displayed on the browser. During the development stage, we want errors to be displayed. To do that, we set `display_errors` to `On`.

On a live site, we do not want any errors to be displayed. To do that, we set `display_errors` to `Off`.

When `display_errors` is off, we typically log the errors reported by

PHP to a log file. We'll learn to do that later.

To configure the error settings mentioned above, the most direct way is to modify the **php.ini** file. We've already done that in Chapter 2, where we set `error_reporting` to `E_ALL` and `display_errors` to `On`.

If you do not have permission to edit **php.ini**, you can use two built-in functions - `error_reporting()` and `ini_set()` - to modify the settings. To do that, add the following statements to all your PHP scripts.

```
error_reporting(E_ALL);
ini_set('display_errors', '1');
```

In the first statement above, we use the `error_reporting()` function to set `error_reporting` to `E_ALL`. Next, we use the `ini_set()` function to set `display_errors` to `'1'` when the site is in the development stage.

After the site goes live, you should change the `display_errors` setting to `'0'` using the following statement:

```
ini_set('display_errors', '0');
```

Changing the `display_errors` setting using `ini_set()` has its limitations. If there is any syntax error in your script, the `ini_set()` function will not work.

For instance, if you have the following line in your script

```
echo 'Hello;
```

The `display_errors` setting will not be set as the statement above has a syntax error (the closing quotation mark is missing). The page will simply fail to load without any indication of what's causing it. Hence, whenever possible, it is strongly recommended that you update the **php.ini** file instead.

If you do not have access to **php.ini**, you can try modifying the error settings using **httpd.conf** or **.htaccess**. These files allow you to modify the Apache web server settings, which in turn can be used to

modify the PHP settings. Depending on your hosting company, you may also be allowed to add a custom **php.ini** file to your directory and use it to configure the PHP settings.

Check with your hosting company if you have permission to do any of the above and if yes, how to go about doing it. Instruction on using these techniques is beyond the scope of this book.

12.2.3 Error Handler and Shutdown Function

In the previous section, we learned to adjust the `display_errors` setting. On a live site, you are encouraged to turn `display_errors` off.

However, sometimes, not displaying any error message may not be the best approach. For instance, if a fatal error occurs on our site, the site will just fail to load without any information. This can be frustrating for the user.

Instead of leaving users entirely in the dark when something goes wrong, we can provide them with a “user-friendly” error message.

To do that, we need to create a custom error handler. A custom error handler allows us to handle the PHP errors ourselves, instead of relying on the default error handler in PHP. We can use this custom handler to display a custom message to users when something goes wrong.

To see how this works, create a new file in Brackets and save it as **errorHandling.php** to the **htdocs** folder. Add the following code to it:

```
<?php

function myErrorHandler($errno, $errstr, $errfile,
$errline)
{
  echo '<BR>Oppsss... An error occurred.
<BR>'.$errstr;
}

set_error_handler('myErrorHandler');
```

In the code above, we first create a function called `myErrorHandler()` that has four parameters - `$errno`, `$errstr`, `$errfile` and `$errline`. The first two parameters are mandatory for a function to "qualify" as a custom error handler, the last two are optional.

These four parameters are used to store the error code, error message, file name in which the error occurred and the line number respectively. All four pieces of information are provided by PHP when an error occurs.

Within the function, we use an `echo` statement to echo a custom error message to users.

After coding the function, we use a built-in function called `set_error_handler()` to set `myErrorHandler()` as the custom error handler. Got it?

At this point, you'll probably notice that this custom error handler is very similar to the custom exception handler we coded in Chapter 12.1.4. Indeed, the two are similar.

However, while a custom exception handler handles uncaught exceptions, a custom error handler handles errors. In addition, a custom error handler does not result in script termination.

To test the error handler above, add the following code to the end of **errorHandling.php**:

```
echo $a;
echo '<BR>Script is not terminated';
```

If you run **errorHandling.php** now, you'll get the following output:

```
Oppsss... An error occurred.
Undefined variable: a
Script is not terminated
```

The custom error handler is first executed to give us a custom error message when we try to access an undefined variable. After that, the script continues and the second `echo` statement is executed.

The custom error handler replaces the default PHP error handler and gives us a convenient way to modify error messages displayed to users.

However, it is unable to handle all errors. For instance, it is unable to handle fatal errors. These errors lead to script termination.

To handle these errors, we need another built-in function called `register_shutdown_function()`. This function tells PHP which function to run when a script is about to be terminated.

To see how this works, add the following code to the end of **errorHandling.php**:

```
function myShutDownHandler(){

  $lastError = error_get_last();

  if (isset($lastError)) {
    echo '<BR>Oppsss... Script terminated.<BR>';
  }
}

register_shutdown_function('myShutDownHandler');
```

Here, we declare a function called `myShutDownHandler()`. Within the function, we need to determine whether the script termination is due to an error. To do that, we use the `error_get_last()` function.

This function returns `NULL` if no error has occurred (for instance, if script termination is due to the user navigating to another page).

On the other hand, if an error has occurred, it returns the last error as an associative array. The array contains four keys: `type` (which gives the error type), `message` (which gives the error message), `file` (which gives the file where the error occurred) and `line` (which gives the line where the error occurred).

After calling the `error_get_last()` function, we use an `if` statement to check if an error has occurred. If it has, we echo a custom error message to users.

After coding the `myShutDownHandler()` function, we use another built-in function called `register_shutdown_function()` to register it as the shutdown function.

To test our shutdown function, add the following code to the end of **errorHandling.php**:

```
hello();
```

Here, we try to call a non-existent function. If you load **errorHandling.php** now, you'll get the error message below:

```
Fatal error: Uncaught Error: Call to undefined
function hello() in ...errorHandling.php:27 Stack
trace: #0 {main} thrown in ...errorHandling.php on
line 27

Oppsss... Script terminated.
```

The "Fatal error" message is displayed only when `display_errors` is set to `On` (or `'1'`). If you turn `display_errors` to `Off` (or `'0'`), only the last line is displayed. This line is generated by our shutdown function. Got it?

12.3 Putting it All Together

In the previous sections, we talked about exceptions and errors and how to handle them separately. In this section, let's consolidate everything into a single file.

To do that, create a file called **debugging.php** in your **htdocs** folder.

Inside this file, we need to first create a function called `myDebugger()`; this function serves as the main function for handling any uncaught exception or error later.

To create the function, add the following code to **debugging.php**:

```
function myDebugger($msg, $file, $line, $trace = '')
{
```

```
  $message = $trace.'<BR><BR>
<strong>'.$msg.'</strong> found on <u>line
'.$line.'</u> in file <u>'.$file.'</u>';

  if (ini_get('display_errors')) {
    echo $message;
  } else {
    error_log($message);
    header('Location: error.html');
  }
}
```

Here, we define a function called `myDebugger()` with four parameters - `$msg`, `$file`, `$line` and `$trace`. The first three parameters are for storing the message, file name and line number of an error or exception respectively. The last parameter is for storing the stack trace of an exception. All four pieces of information will be provided by PHP when an error or exception occurs.

Within the function, we concatenate the four parameters and format the resulting string using HTML, before assigning it to a variable called `$message`.

Next, we use an `if` statement to determine what gets displayed on our browser. To do that, we use a built-in function called `ini_get()` to get the setting for `'display_errors'`.

When `'display_errors'` is set to `On` (or `'1'`), the `if` condition evaluates to `true` and we use an `echo` statement to display the error message on our browser. Else, we use another built-in function called `error_log()` to log `$message` to our error log file.

The error log file is a text file that contains all the messages we write to it using the `error_log()` function. You can find where the error log file is located on your system by searching for “error_log” (without quotes) in the PHP information page, which can be found at http://localhost/dashboard/phpinfo.php.

If you are using Windows, the log file should be stored in the **C:\xampp\php\logs** folder. However, weirdly, this folder may not exist and you may have to create it yourself. To do that, simply navigate to **C:\xampp\php** and create a **logs** folder. When an error occurs, PHP will automatically create the log file when the `error_log()` function is called.

After logging our error message, we use the `header()` function to redirect users to another page called **error.html** (which we need to create separately). With that, the `myDebugger()` function is complete.

Next, we need to create an exception handler, an error handler and a shutdown function. Inside all three functions, we need to call the `myDebugger()` function.

To do that, add the following code to **debugging.php**:

```
function myExceptionHandler ($e)
{
  myDebugger($e->getMessage(), $e->getFile(), $e->getLine(), $e->getTraceAsString());
}

function myErrorHandler($errno, $errstr, $errfile, $errline)
{
  myDebugger($errstr, $errfile, $errline);
}

function myShutDownHandler(){

  $lastError = error_get_last();

  if (isset($lastError)) {
    myDebugger($lastError['message'], $lastError['file'], $lastError['line']);
  }
}
```

The first function - `myExceptionHandler()` - has a parameter

called `$e` that is used to store an uncaught exception object. Inside the function, we use `$e` to call the built-in `getMessage()`, `getFile()`, `getLine()` and `getTraceAsString()` methods and pass the results as arguments to the `myDebugger()` function.

Next, we have the `myErrorHandler()` function. This function has four parameters - `$errno`, `$errstr`, `$errfile` and `$errline`. Inside the function, we pass `$errstr`, `$errfile` and `$errline` as arguments to the `myDebugger()` function.

Last but not least, we have the `myShutDownHandler()` function. Inside this function, we check if the shutdown is due to an error. If it is, we pass the elements in the associative array returned by `error_get_last()` as arguments to the `myDebugger()` function.

After coding these three functions, we need to set them as our custom handlers and shutdown function. To do that, add the following lines to **debugging.php**:

```
set_exception_handler('myExceptionHandler');
set_error_handler('myErrorHandler');
register_shutdown_function('myShutDownHandler');
```

With that, the **debugging.php** file is complete; we'll be using it in our project later.

In PHP, there are many different approaches to dealing with exceptions and errors, **debugging.php** demonstrates one of the many techniques available.

Regardless of whether an error or an uncaught exception occurs on our site, this file uses one of the custom handlers or the shutdown function to call the `myDebugger()` function. This function displays an error message on the browser when `display_errors` is set to `On` or `'1'`. Else, it logs the error message and redirects users to a custom error page called **error.html**. Got it?

Chapter 13: Project

Congratulations on making it to the end of the book! We've covered a lot in the preceding chapters.

The best way to learn programming is to work on an actual project. In this chapter, we're going to work through a project together. This project covers numerous concepts you've learned in the previous chapters and allows you to see how everything works together. We'll also be covering some new miscellaneous concepts in this project. Excited? Let's do it!

13.1 About the Project

This project involves creating a website that works like a mini blog, where users with an admin account can log in to post. When posting to the blog, the admin user can choose whether to make the post available to non-members or members.

Admin and members need to be logged in while non-members do not. Admin can read and write posts, members can read "Members only" posts after they log in, and non-members can only read "Public" posts.

When a member or admin logs in to the site, the website retrieves the last post he/she has read and notifies him/her which posts are new.

To see a demonstration of how the site works, go to https://learncodingfast.com/php.

13.2 Acknowledgements and Requirements

This project uses HTML, CSS, Javascript, PHP and MySQL. An understanding of HTML (especially HTML forms) and SQL is essential for the project. This chapter only covers PHP and will guide you through all the PHP code used.

All HTML, CSS, Javascript and SQL code will be provided for you. You can download them at https://learncodingfast.com/php.

However, as the project uses Bootstrap (https://getbootstrap.com/) for

the user interface and CKEditor (https://ckeditor.com/ckeditor-4/) for the text editor, their code will not be provided. Instead, links will be provided in the `<head>` element to include these files from their respective content delivery networks (CDN). This means that you need an internet connection when running the code in the project.

Certain instructions (such as creating a database and user account) provided here are specifically for XAMPP. If you are not using XAMPP, you'll have to refer to the software's documentation for specific instructions.

Last but not least, this project uses PHP 5.5 and above. If you have just downloaded XAMPP, the PHP interpreter bundled with it is at least PHP 7.

13.3 Structure of the Project

The main folder of the project is **phpproject**.

Inside this main folder, we have eight files: **admin.php**, **error.html**, **index.php**, **logout.php**, **read.php**, **signup.php**, **UI_include.php** and **write.php**. These files are responsible for the user interface of the blog.

Besides the eight files, we have a sub-folder called **includes**.

Inside **includes**, we have three files: **loadclasses.php**, **header.html** and **debugging.php**.

We also have three sub-folders: **process** (contains files used for processing HTML forms), **classes** (contains files where we define our classes) and **css** (contains files with CSS code).

13.4 Creating Database, User Account and Tables

Before we begin working on the PHP code, we need to create the database, tables and user account.

First, ensure that you have started XAMPP and the Apache and MySQL servers. Next, proceed to

http://localhost/phpmyadmin/index.php and follow the instructions in Chapter 11.5 to **create a database called "project".**

Next, **create a user** with the following information for the "project" database:

User Name: project_admin

Host Name: localhost

Choose your own desired password for the user and click "Go" to create the user.

Once that is done, click on the SQL tab at the top of the page and copy the following SQL code into the editor. This code can be downloaded at https://learncodingfast.com/php.

```
USE project;

CREATE TABLE IF NOT EXISTS members (
     username VARCHAR(100) PRIMARY KEY,
     password VARCHAR(255) NOT NULL,
     is_admin BOOLEAN DEFAULT false,
     last_viewed int DEFAULT 0
);

CREATE TABLE IF NOT EXISTS posts(
     id INT AUTO_INCREMENT PRIMARY KEY,
     post_date TIMESTAMP DEFAULT NOW() NOT NULL,
     username VARCHAR(100) NOT NULL,
     title VARCHAR(255) NOT NULL,
     post TEXT NOT NULL,
     audience INT NOT NULL,
     CONSTRAINT FOREIGN KEY (username) REFERENCES
members(username) ON DELETE CASCADE
);
```

The code above creates two tables, "members" and "posts", for the "project" database. Click on "Go" to execute the code. Once that is done, we are ready to work on the PHP code.

13.5 Editing The classes Folder

First, navigate to your **htdocs** folder and paste the unzipped **phpproject** folder into it. Next, launch http://localhost/phpproject/index.php in your browser. If you see the page below, all is good.

account_box

Login to Your Account

face Username

vpn_key Password

Error Message Here

Log In

Don't have an account? Sign up

Close your browser and navigate to the **htdocs\phpproject\includes\classes** folder on your computer. You should see five PHP files inside.

13.5.1 Helper.php

We'll start with **Helper.php**. Open this file in Brackets; you'll see that we've created a class called `Helper`.

This class has no properties and constructor. Inside the class, our job is to implement five `public` methods - `passwordsMatch()`, `isValidLength()`, `isEmpty()`, `isSecure()` and `keepValues()`.

Let’s start with the `passwordsMatch()` method. This method has two parameters, `$pw1` and `$pw2`. In this project, to keep our code compatible with older versions of PHP, we will not be using type declaration for functions and methods. With this in mind, try declaring the `passwordsMatch()` method yourself.

Next, within the method, we need to check if the values of the two parameters are equal. If they are, we return `true`. Else, we return `false`. Try doing this yourself. Hint: You need to use an `if-else` statement. Once you are done with the `if-else` statement, the `passwordsMatch()` method is complete.

Next, let’s move on to the `isValidLength()` method. This method has three parameters - `$str`, `$min` and `$max`.

`$min` has a default value of `8`, while `$max` has a default value of `20`. You can refer to Chapter 7.1 if you are not familiar with default values for functions (and methods).

Within the method, we need to check if the length of `$str` is smaller than `$min` or greater than `$max`. If it is, we return `false`. Else, we return `true`.

Try coding the method yourself. Hint: You can use the built-in function `strlen()` to get the length of `$str`.

After the `isValidLength()` method, we have the `isEmpty()` method. This method has one parameter, `$postValues`, which stores an array. The method checks if any of the elements in the array is an empty string.

Inside the method, we need to use a `foreach` loop to loop through each element in `$postValues` and use an `if` statement to check if the element equals an empty string. If it equals, we return `true`.

After looping through all the elements, if no empty string is found, we return `false`. In other words, we return `false` <u>outside</u> the `foreach` loop.

Try coding this method yourself. You can refer to Chapter 6.3.5 for

help on using `foreach` loops.

Next, we have the `isSecure()` method. This method has one parameter - `$pw` - and checks if `$pw` contains at least one lowercase character, one uppercase character and one digit. To do so, we need to use regular expressions. A regular expression allows us to translate the requirements above (written in English) into an expression that PHP can understand.

All regular expressions must start and end with a delimiter. This delimiter can be any non-alphanumeric, non-backslash and non-whitespace character. Often used delimiters include forward slashes (/), hash signs (#) and tildes (~). We'll use tildes in our code.

The regular expression for "at least one lowercase character" is `~[a-z]+~`, where `~` is the delimiter, `[a-z]` represents the set of lowercase characters and `+` represents "at least one".

The regular expression for "at least one uppercase character" is `~[A-Z]+~` and that for "at least one digit" is `~[0-9]+~`.

To check if `$pw` satisfies the regular expressions above, we need to use a built-in function called `preg_match()`. This function accepts two arguments - the regular expression and the string that we want to check. The regular expression is passed as a string to the function.

To check if `$pw` contains at least one lowercase character, we write

```
preg_match("~[a-z]+~", $pw)
```

This returns `true` if `$pw` contains at least one lowercase character. Else, it returns `false`.

Our `isSecure()` method needs to apply the `preg_match()` function three times to check if all three requirements are met. If all three are met, it returns `true`. Else, it returns `false`. Try doing this yourself.

Done?

Last but not least, let's move on to the most complicated method in the `Helper` class. This method is used to preserve user input in a

form when the form is not processed successfully (you can refer to Chapter 8.1.4 for more details). To do that, we need to prefill the form with the user's previous input when the form reloads.

To prefill textboxes, we use the `value` attribute. For instance,

```
<input type="text" value = "Hello">
```

prefills a textbox with "Hello".

To prefill textareas, we enclose the text between the `<textarea>` opening and closing tags. For instance,

```
<textarea>Hello</textarea>
```

prefills a textarea with "Hello".

To preselect drop-down lists, we add the word "selected" to the selected option. For instance,

```
<select>
  <option value = 'P'>Public</option>
  <option value = 'M' selected>Members
Only</option>
</select>
```

preselects the "Members Only" option.

Our job now is to write a method to prefill/preselect user input for us. This method is called `keepValues()` and has three parameters - `$val`, `$type` and `$attr`.

`$val` represents the value submitted by the user. For textboxes and textareas, the value submitted is the text entered into the respective form elements. For drop-down lists, the value submitted is the string assigned to the `value` attribute of the selected option. For instance, in the drop-down list above, if the second option is selected, the value submitted is `'M'` (not "Members Only").

`$type` represents the type of form element.

`$attr` is only applicable for drop-down lists and **has a default value of** `''` (an empty string). It represents the string assigned to the `value`

attribute of a drop-down list's option. For instance, for the drop-down list above, `$attr` is `'P'` for the first option and `'M'` for the second.

Try declaring this method yourself.

Next, inside the method, we have the following `switch` statement:

```
switch ($type){
  case 'textbox':
    echo "value = '$val'";
    break;
  case 'textarea':
    //Add code here
  case 'select':
    //Add code here
  default:
    echo '';
}
```

This statement uses the value of `$type` to determine what string to echo. The first case has been completed for you. Based on the description above, try completing the other two cases yourself by echoing a suitable string for the respective form elements.

Hint: Refer to the underlined text for each HTML form element in the description above.

For the `'select'` case, you need to use an `if` statement to compare the value of `$val` (which is the value submitted by the user) with `$attr` (which is the string assigned to the `value` attribute of an option) to decide whether to echo anything for a particular option. Got it?

You may need to read through this section more than once to complete this `switch` statement.

Once you are done with the `switch` statement, the `keepValues()` method is complete and so is the `Helper` class. Remember to close the braces for the `switch` statement, the `keepValues()` method and the class itself.

13.5.2 Database.php

Next, let's move on to the **Database.php** file. Inside this file, we've created a class called `Database` with three constants (`SELECTSINGLE`, `SELECTALL` and `EXECUTE`) and one private property (`$pdo`). In addition, we have a constructor that is used to create a new `PDO` object.

You need to modify two things in the constructor. First, on line 13, you need to change "Your Password" to your actual password.

Next, notice that `$pdo` is a property of the `Database` class? We learned in Chapter 9.2 that to access any property of a class, you need to use the `$this` keyword inside the class.

Hence, on line 13, you need to change

```
$pdo = …
```

to

```
$this->pdo = …
```

The same applies to line 14. Try doing this yourself. Once that is done, the constructor is complete.

Next, we need to add a `public` method called `queryDB()` to the `Database` class. To code this method, you need to be familiar with using prepared statements in PHP. You can refer to Chapter 11.4 for reference if you are not familiar.

The `queryDB()` method has three parameters, `$sql`, `$mode` and `$values`.

`$sql` represents the SQL statement to be executed, `$mode` indicates whether the method needs to fetch any row(s) from the database and `$values`, which has a **default value of `array()`** (i.e., an empty array), is used for binding variables to the placeholders in `$sql`.

Try declaring the method yourself.

Inside the method, we need to use the `$pdo` property (reminder: you

access it using `$this->pdo`) to prepare the SQL statement (`$sql`) and <u>assign the result to a variable called `$stmt`</u>. Try doing this yourself.

Next, we need to bind values to the placeholders in the SQL statement. The placeholders and values for binding are passed as a two-dimensional array (`$values`) to the method. Refer to Chapter 5.3.1 if you are not familiar with multidimensional arrays.

Suppose the placeholders are `:username` and `:password` and the variables to bind are `$uname` and `$pwd` respectively, users need to pass the following array to the `queryDB()` method:

```
$values = array(
  array(':username', $uname),
  array(':password', $pwd)
);
```

To process this array, we use the `foreach` loop below:

```
foreach($values as $valueToBind){
  $stmt->bindValue($valueToBind[0],
$valueToBind[1]);

}
```

When this loop runs for the first time, the array `(':username', $uname)` gets assigned to `$valueToBind`. Inside the loop, we use the `bindValue()` method to bind `$uname` (`$valueToBind[1]`) to `':username'` (`$valueToBind[0]`).

When the loop runs for the second time, we use the `bindValue()` method to bind `$pwd` to `':password'`. Got it?

Copy the `foreach` loop above into the `queryDB()` method and make sure you understand it before proceeding.

After binding values to placeholders, we need to use `$stmt` to call the `execute()` method. Try doing this yourself.

Finally, we need to determine if there are any values to be fetched.

We do that using the second parameter - `$mode`.

`$mode` can take one of three constants - `SELECTSINGLE`, `SELECTALL` or `EXECUTE`. These three constants were defined in the class previously. We use an `if` statement to determine whether we should fetch any results. The `if` statement works similar to the pseudocode below:

```
if ($mode is not equal to SELECTSINGLE, SELECTALL
and EXECUTE){
  throw an Exception

  using 'Invalid Mode' as the error message
}else if ($mode equals SELECTSINGLE){
  use $stmt to call the fetch(PDO::FETCH_ASSOC)
method

  and return the result using the return keyword
}else if ($mode equals SELECTALL){
  use $stmt to call the fetchAll(PDO::FETCH_ASSOC)
method

  and return the result using the return keyword

}
```

First, we check if the value of `$mode` is valid (i.e., it must be either `SELECTSINGLE`, `SELECTALL` or `EXECUTE`).

Next, we check if `$mode` is `SELECTSINGLE` or `SELECTALL` and use the `$stmt` variable to call the `fetch(PDO::FETCH_ASSOC)` or `fetchAll(PDO::FETCH_ASSOC)` method respectively. We then use the `return` keyword to return the results fetched.

Try converting the pseudocode to PHP code yourself.

Hint:

To access the constants defined in the `Database` class, you need to use the `self` keyword or the class name followed by the `::` operator. For instance, to access `EXECUTE`, you can write

Database::EXECUTE. Refer to Chapter 9.7 for reference on using constants in classes.

To throw an exception, you create a new Exception object and pass the error message to the constructor. Refer to Chapter 12.1.3 for reference on throwing exceptions.

Once you are done with the if statement, the queryDB() method is complete and so is the Database class.

13.5.3 BlogReader.php

Now, let's proceed to the BlogReader class. This class has two constants, READER and MEMBER, with values 1 and 2 respectively. In addition, it has two protected properties $db and $type. The constructor of the class has been coded for you.

```
public function __construct(){

   $this->db = new Database();
   $this->type = BlogReader::READER;
}
```

This constructor initializes the values of $db and $type.

Now, we need to add a method called getPostsFromDB() to the class. This method is public and has no parameter; try declaring it yourself.

A blog reader refers to readers of the blog who are not logged in. Readers who are not logged in can only read posts from the "posts" table where the "audience" column has a value smaller than or equal to 1. Hence, inside the getPostsFromDB() method, we need to use the following statement to query the "posts" table:

```
$sql = "SELECT id, unix_timestamp(post_date) as
`post_date`, username, title, post, audience FROM
posts WHERE audience <= :audience ORDER BY id DESC";
```

Add the statement above to the getPostsFromDB() method; we'll use the queryDB() method to execute it later.

This statement has one placeholder `:audience`.

Based on what we mentioned when we coded the `Database` class, we need to declare a two-dimensional array and use it to bind a value to the placeholder. This is done with the following statement:

```
$values = array(
  array(':audience', $this->type)
);
```

Here, we declare a two dimensional array called `$values` with one inner array - `(':audience', $this->type).`

We use this inner array to bind the `$type` property to the `:audience` placeholder. As we've previously assigned `BlogReader::READER` (which equals `1`) to the `$type` property in the constructor, we are essentially binding the value `1` to the `:audience` placeholder. Got it?

Add the `$values` array above to the `getPostsFromDB()` method. Next, we need to use the `Database` object (`$db`) to call the `queryDB()` method in the `Database` class. To do that, we use the statement below:

```
$result = $this->db->queryDB($sql,
Database::SELECTALL, $values);
```

Here, we use `$this->db` to access `$db` as it is a class property. After accessing `$db`, we use it to call the `queryDB()` method, passing the SQL statement (`$sql`), the mode (`Database::SELECTALL`) and the `$values` array to the method.

The mode is `Database::SELECTALL` as we are retrieving more than one row from the "posts" table. Next, we assign the result returned to a variable called `$result`.

Add the statement above to the `getPostsFromDB()` method.

Finally, we need to check if there are any rows returned by the `queryDB()` method. To do that, we check if there are any elements in `$result`. If there aren't, we return `false`. Else, we return the `$result` array. Try writing this `if` statement yourself.

Hint: You can use the `count()` function to get the number of elements in `$result`.

Once that is done, the `getPostsFromDB()` method is complete and so is the `BlogReader` class. Read through the `getPostsFromDB()` method carefully and make sure you understand it before proceeding; we'll be coding many methods similar to it later.

13.5.4 BlogMember.php

Now, let's write a class that extends the `BlogReader` class.

Open **BlogMember.php** and create a new class called `BlogMember`. This class extends the `BlogReader` class and has a `private` property called `$username`.

Try declaring the class yourself.

Inside the class, we have a constructor that has one parameter - `$pUsername`.

Inside the constructor, we need to call the parent class constructor to initialize the inherited property `$db`. Next, we need to assign `$pUsername` to the `$username` property and `BlogMember::MEMBER` (this constant is inherited from the `BlogReader` class) to the inherited property `$type`.

Try implementing the constructor yourself. Hint: You need to use the `parent` keyword, followed by the `::` operator, to call the parent class constructor.

Got it? Good! After you have implemented the constructor, you need to implement six more methods. The methods are:

```
public function isDuplicateID()
public function insertIntoMemberDB($pPassword)
public function isValidLogin($pPassword)
private function getLatestPostID()
public function updateLastViewedPost()
public function getLastViewedPost()
```

Let's start with the `isDuplicateID()` method. This method is `public` and has no parameter. It is called whenever a new user signs up and returns `true` if the username selected by the new user already exists in the "members" table. Try declaring the method yourself.

Inside the method, we need to execute the following SQL statement:

```
SELECT count(username) AS num FROM members WHERE
username = :username
```

This statement returns `0` if the username bound to `:username` is not found.

Try using the `queryDB()` method in the `Database` class to execute this SQL statement. You need to bind the `$username` property to `:username` and use the `Database::SELECTSINGLE` mode (as we are only selecting one row from the database) to call the `queryDB()` method.

Try doing this yourself. You can refer to the `getPostsFromDB()` method in the `BlogReader` class for reference on using the `queryDB()` method. Got it?

Once that is done, assign the result returned by `queryDB()` to a variable called `$result`.

Next, use an `if` statement to check if `$result['num']` is equal to zero. If it is, return `false`. Else, return `true`.

Try coding this method yourself.

Once you have completed the `isDuplicateID()` method, we can move on to the `insertIntoMemberDB()` method.

This method is `public` and has one parameter - `$pPassword`. It is used to insert a new row into the "members" table when a new user signs up.

`$pPassword` stores the password entered by the user when he/she submits the sign-up form.

Inside the method, we need to use the `queryDB()` method with the

`Database::EXECUTE` mode (as we are not fetching any data from the database) to execute the following SQL statement:

```
INSERT INTO members (username, password) VALUES
(:username, :password)
```

To execute this statement, we need to bind `$username` (the class property) to `:username` and `$pPassword` (the parameter) to `:password`.

However, as `$pPassword` stores the password selected by our user, we should not store it in the database in its original form. Instead, we should hash it first. Hashing is similar to encrypting and can be done using a built-in function in PHP called `password_hash()`. This function is available from **PHP 5.5 onwards** and accepts two arguments - the string to hash and the algorithm to use.

The algorithm to use can be any of the predefined constants found at https://www.php.net/manual/en/password.constants.php.

In our project, we'll use `PASSWORD_DEFAULT` as the constant. This constant indicates that we'll use the default algorithm in PHP. To hash `$pPassword`, we use the code below:

```
password_hash($pPassword, PASSWORD_DEFAULT)
```

Hence, to bind values to the `:username` and `:password` placeholders in our SQL statement, we use the `$values` array below:

```
$values = array(
  array(':username', $this->username),
   array(':password', password_hash($pPassword,
PASSWORD_DEFAULT))
);
```

Try using this array, the `Database::EXECUTE` constant and the SQL `INSERT` statement above to call the `queryDB()` method. Once that is done, the `insertIntoMemberDB()` method is complete.

There is no need to return any result for this method as the `queryDB()` method does not return any result when the mode is

Database::EXECUTE.

Done? Great!

Let's move on to the isValidLogin() method. This method is public and has one parameter - $pPassword. It checks if the username and password entered by the user are valid when he/she tries to log in.

Inside the method, we need to use the queryDB() method to execute the following SQL statement:

```
SELECT password FROM members WHERE username =
:username
```

To do that, you need to bind the $username property to :username and use the Database::SELECTSINGLE mode (as we are only selecting one row from the database) to call the queryDB() method.

Once that is done, assign the result to a variable called $result.

Try doing this yourself.

Next, we need to check if $result['password'] is set.

$result['password'] is set only if the fetch() method in queryDB() manages to fetch the password. If there is no user with the username stated in the SQL query, $result['password'] will not be set.

Besides checking if $result['password'] is set, we also need to check if the password entered by the user matches the password fetched from the database.

Recall that the password stored in the database is hashed and no longer in its original form?

To check if the hashed password matches the password entered by the user, we need to use another built-in function called password_verify(). This function accepts two arguments and returns true if the first argument matches the second. The second argument has to be a password hashed using the password_hash()

function.

In our `isValidLogin()` method, to determine if the two passwords match, we use the following `if` statement:

```
if (isset($result['password']) &&
password_verify($pPassword, $result['password']))
  return true;
else
  return false;
```

Add the statement above to the `isValidLogin()` method and the method is complete.

Great! Let's proceed to the `getLatestPostID()` method. This is a `private` method with no parameters and will be used in the `updateLastViewedPost()` method later. Try declaring it yourself.

Inside the method, we need to use the `queryDB()` method to execute the following SQL statement:

```
SELECT max(id) AS max FROM posts
```

Decide on the appropriate mode to use for this SQL statement and try calling the `queryDB()` method yourself. Note that for this SQL statement, there is no need to pass any array to the `queryDB()` method as the statement does not contain any placeholder.

Once you have executed the `queryDB()` method, assign the result to a variable called `$result`.

The SQL statement above returns `NULL` if there is no post in the "posts" table.

Hence, we need to first check if `$result['max']` is set. If it is, we return its value. Else, we return `0`. Try doing this yourself. Once this is done, the `getLatestPostID()` method is complete and we can move on to the `updateLastViewedPost()` method.

The `updateLastViewedPost()` method is `public` and has no parameters. Try declaring it yourself.

Inside the method, we need to update the “last_viewed” column of the “members” table to reflect the latest post viewed by a member. Whenever a member logs into our website, the “last_viewed” value of that member will be updated to the id of the latest post in the “posts” table.

This id is given by the `getLatestPostID()` method we coded earlier. To use this method, add the following line to the `updateLastViewedPost()` method:

```
$max = $this->getLatestPostID();
```

Here, we use the `$this` keyword to call the `getLastestPostID()` method and assign its result to `$max`. Next, we need to use the `queryDB()` method to execute the following SQL statement:

```
UPDATE members SET last_viewed = :max WHERE username
= :username
```

To execute this statement, we need to bind the `$username` property to `:username` and the variable `$max` to `:max`. In addition, we need to decide on the appropriate mode to use when calling `queryDB()`.

Try doing this yourself. Once this is done, the method is complete.

Finally, we move on to the `getLastViewedPost()` method. This method is `public` and has no parameters. It uses the `queryDB()` method to execute the following SQL statement:

```
SELECT last_viewed FROM members WHERE username =
:username
```

To execute this statement, we need to bind the `$username` property to `:username` and decide on the appropriate mode to use when calling the `queryDB()` method. Try doing this yourself and assign the result to a variable called `$result`.

Next, we need to check if `$result['last_viewed']` is set. If there is no user with the username stated in the SQL query, `$result['last_viewed']` will not be set.

If `$result['last_viewed']` is set, we return its value. Else, we

return 0.

Try doing this yourself. Once this is done, the getLastViewedPost() method is complete and so is the BlogMember class.

13.5.5 Admin.php

We are left with one more class to code - the Admin class.

The Admin class has two private properties - $db and $username.

Inside the class, we have a constructor with one parameter - $pUsername. This constructor initializes $username with $pUsername and $db with a new Database object. Try declaring the class and properties and implement the constructor yourself.

Next, we have a public method called isValidLogin() that has one parameter - $pPassword. This method is very similar to the isValidLogin() method in the BlogMember class except that it executes the following SQL statement:

```
SELECT password FROM members WHERE username =
:username AND is_admin = true
```

Try coding this method yourself.

Once that is done, we can move on to the final method - insertIntoPostDB().

This method is public and has three parameters - $title, $post and $audience.

Inside the method, we use the queryDB() method to execute the following SQL statement:

```
INSERT INTO posts (username, title, post, audience)
VALUES (:username, :title, :post, :audience)
```

You need to bind the $username property to :username and the parameters $title, $post and $audience to :title, :post and :audience respectively. In addition, you need to choose the correct

mode for `queryDB()`.

Try coding this method yourself.

Done? Great! All our classes are now complete. We are ready to move on to the **process** folder.

13.6 Editing The process Folder

The files in the **process** folder are for processing HTML forms.

13.6.1 p-index.php

We'll start with **p-index.php**. This file is for processing **index.php**, which contains a form for members to log in to read "Members Only" posts.

index.php has two input boxes named "username" and "password" and one button named "submit". If an error occurs when processing **index.php**, the page echoes an error message stored in a variable called `$msg`.

If you open **p-index.php** (inside the **process** folder), you'll see that the code has already been completed for you. This is done so that you can refer to this file when working on other processing files later. Let's run through the code together.

First, we declare a variable called `$h` and assign a new `Helper` object to it. Next, we declare two variables `$msg` and `$username` and assign an empty string to each of them.

After declaring and initializing the variables, we are ready to process the form in **index.php**. We use the following `if` statement to check if the "submit" button has been clicked.

```
if (isset($_POST['submit']))
{
}
```

Inside the `if` block, we first assign `$_POST['username']` to `$username`. We need to do this as we'll be using `$username` in **index.php** later.

Next, we use an `if-else` statement to check if users have entered data into both the “username” and “password” input boxes.

To do that, we use `$h` to call the `isEmpty()` method in the `Helper` class. This method accepts an array that contains all the variables we want to check. We pass the following array to the method:

```
array($username, $_POST['password'])
```

If any of the elements in the array is an empty string, the `isEmpty()` method returns `true`. When that happens, the `if` block is executed and the string `'All fields are required'` is assigned to `$msg`.

On the other hand, if none of the elements are empty, the `else` block is executed.

Inside the `else` block, we create a `BlogMember` object called `$member`. Next, we have another `if-else` statement. This inner `if-else` statement uses the `$member` object to call the `isValidLogin()` method in the `BlogMember` class.

If the method returns `false`, the condition

```
!$member->isValidLogin($_POST['password'])
```

evaluates to `true` (as `!false` equals `true`) and the `if` block is executed. The string `'Invalid Username or Password'` will then be assigned to `$msg`.

On the other hand, if the method returns `true`, the `else` block is executed. Within this `else` block, we assign `$username` to a session variable called `$_SESSION['username']` and use the `header()` function to redirect users to **read.php**. Once that is done, the **p-index.php** page is complete.

Go through **p-index.php** carefully and make sure you understand it before proceeding. Got it? Great!

13.6.2 p-admin.php

Let's move on to the **p-admin.php** file now. This file is for processing **admin.php**, which contains a form for admin to log in.

admin.php has two input boxes named "username" and "password" and one button named "submit". If an error occurs when processing **admin.php**, the page echoes an error message stored in a variable called `$msg`.

As you may have guessed, **p-admin.php** is very similar to **p-index.php**. Here's what we need to do in **p-admin.php**:

First, declare and initialize `$h`, `$msg` and `$username` (similar to what was done in **p-index.php**).

Next, check if the "submit" button has been clicked. If it has, assign `$_POST['username']` to `$username`.

Next, ensure that all input boxes are not empty.

If any of the boxes are empty, assign an appropriate error message to `$msg`. Else, create an `Admin` object and use it to call the `isValidLogin()` method. (Refer to the `Admin` class to find out what needs to be passed to the constructor and the `isValidLogin()` method when calling them.)

If `isValidLogin()` returns `false`, assign an appropriate error message to `$msg` to inform users that the login credentials are invalid. Else, assign `$username` to `$_SESSION['username']`. In addition, declare a session variable called `$_SESSION['is_admin']` and assign `true` to it. Finally, redirect users to **write.php** using the `header()` function.

Got it? Try coding **p-admin.php** yourself. You can refer to **p-index.php** for reference.

Once you are done, we are ready to move on to **p-signup.php**.

13.6.3 p-signup.php

p-signup.php is for processing **signup.php**, which contains a form for members to sign up.

To keep this project short, we'll only create a sign-up form for blog members, not for admin. We'll learn to convert a blog member to an admin later using phpMyAdmin.

The sign-up form for blog members (**signup.php**) has three input boxes named "username", "password" and "confirm_password" and one button named "submit". If an error occurs when processing **signup.php**, the page echoes an error message stored in a variable called `$msg`.

Here's what we need to do in **p-signup.php**:

First, declare and initialize three variables `$h`, `$msg` and `$username` (similar to what was done in **p-index.php**).

Next, check if the "submit" button has been clicked. If it has, we first assign `$_POST['username']` to `$username`. Next, we need to ensure the following:

1. All input boxes in **signup.php** have been filled out
2. `$username` has a length of between 6 and 100 characters (inclusive)
3. `$_POST['password']` has a length of between 8 and 20 characters (inclusive)
4. `$_POST['password']` contains at least one lowercase character, one uppercase character and one digit.
5. `$_POST['password']` and `$_POST['confirm_password']` match

If any of the cases above are not met, we assign an appropriate error message to `$msg`. Else, we do the following:

Create a `BlogMember` object and use it to call the `isDuplicateID()` method.

If the method returns `true`, assign an appropriate error message to `$msg`, informing users that the username is already in use. Else, use the `BlogMember` object to call the `insertIntoMemberDB()` method and use the `header()` function to redirect users to **index.php**.

Append the query string `new=1` to **index.php**.

Got it? Try coding **p-signup.php** yourself.

Hint: Refer to **p-index.php** for help on processing forms. If you are not familiar with query strings, you can refer to Chapter 8.1.2.

To check the five cases listed above, you can use the `isEmpty()`, `isValidLength()`, `isSecure()` and `passwordsMatch()` methods in the `Helper` class.

Refer to **Helper.php** and **BlogMember.php** to figure out what needs to be passed to the various methods in the `Helper` and `BlogMember` classes when calling them.

Once you are done with **p-signup.php**, we can move on to **p-write.php**. This file is for processing **write.php**, which contains a form for admin to write their posts.

13.6.4 p-write.php

write.php has an input box named "title", a textarea named "post", a drop-down list named "audience" and a button named "submit". If an error occurs when processing **write.php**, the page echoes an error message stored in a variable called `$msg`.

*If you analyze the code in **write.php**, you'll notice that we added a `<script>` element after the textarea. This is for replacing the textarea with a more advanced text editor known as the CKEDITOR (available for free at https://ckeditor.com/ckeditor-4/). We won't go into details on how to use CKEDITOR as it only affects the user interface. Using CKEDITOR does not affect our PHP code in any way.*

Here's what we need to do in **p-write.php**:

First, we need to retrieve two session variables - `$_SESSION['username']` and `$_SESSION['is_admin']` - and check if they are set. If either of the session variables is not set, we know that the user is trying to access **write.php** without logging in with an admin account.

If that’s the case, we use the `header()` function to redirect them to **admin.php**. Else, we do the following:

First, declare and initialize a `Helper` object called `$h`. Next, declare four variables `$title`, `$post`, `$audience` and `$msg` and assign an empty string to each of them.

After that, check if the “submit” button has been clicked.

If it has, assign `$_POST['title']`, `$_POST['post']` and `$_POST['audience']` to `$title`, `$post` and `$audience` respectively.

Next, check that all fields in **write.php** have been filled out. If there is an empty field, assign an appropriate error message to `$msg`.

Else, create an `Admin` object using `$_SESSION['username']` as the argument. Use this object to call the `insertIntoPostDB()` method. (Refer to the `Admin` class to decide what arguments to pass to the method.) Once that is done, assign the string `'Message saved successfully'` to `$msg`.

Clear? Try coding **p-write.php** yourself. Once you are done, we are ready to move on to the most complex processing file - **p-read.php**.

13.6.5 p-read.php

This file is for processing **read.php**, which is used for displaying posts to readers. We need to implement two essential features in **p-read.php**:

First, depending on whether the user is logged in, we need to display different posts. If the user is logged in, we display all posts. Else, we only display “Public” posts. Next, if the user is logged in, we need to display a “New” icon for posts posted to the database after the user’s last login.

To achieve the above, open **p-read.php** and do the following:

First, declare a variable called `$h` and assign a `Helper` object to it. Next, declare a variable called `$update` and assign `false` to it.

Last but not least, declare a variable called `$is_member` and assign `isset($_SESSION['username'])` to it.

Here, we use the `isset()` function to check if the session variable `$_SESSION['username']` is set. If it is (i.e., `isset()` returns `true`), we know that the reader accessing **read.php** is logged in.

Try doing the above yourself.

Next, we need to use a couple of `if-else` statements in **p-read.php**. The structure is shown below:

```
if ($is_member){
  //Create a BlogMember object and use it to call
the getLastViewedPost() method
}else{
  //Create a BlogReader object
}

// Call getPostsFromDB() method

if ($posts == false){
  //include the blankcard.html file
}else{
  //use a foreach loop to process the elements in
$posts
  //include the messagecard.php file
}

if ($is_member)
{
  //include the logout.html file

  if ($update)
    //use the BlogMember object to call the
updateLastViewedPost() method
}
```

In the first `if-else` statement, we check if the user is logged in. If the user is logged in (i.e., `$is_member` is `true`), we initialize a `BlogMember` object and assign it to a variable called `$reader`. We

then use `$reader` to call the `getLastViewedPost()` method in the `BlogMember` class and assign the result to a variable called `$lastPost`.

On the other hand, if the user is not logged in, we initialize a `BlogReader` object and assign it to `$reader`.

Try coding this `if-else` statement yourself. Refer to the respective classes to find out what needs to be passed to the various methods when calling them.

Once the `if-else` statement is complete, we need to use the `$reader` object to call the `getPostsFromDB()` method; this method is defined in the `BlogReader` class.

Recall that `BlogMember` is a subclass of `BlogReader`? Hence, regardless of whether `$reader` is a `BlogMember` or `BlogReader` object, it can access the `getPostsFromDB()` method. Try calling this method yourself and assign the result to a variable called `$posts`.

Next, we need to use a second `if-else` statement to check if `getPostsFromDB()` returned any results.

If there are no results (i.e., `$post` is `false`), we want to display the HTML code in **blankcard.html** (located in the **output_code** folder). To include this file, we use the following statement:

```
include "output_code/blankcard.html";
```

Next, inside the `else` block (if there are results), we use a `foreach` loop to loop through `$posts`.

`$posts` is a two-dimensional array fetched using the `getPostsFromDB()` method in the `BlogReader` class. This method uses the `queryDB()` method in the `Database` class, which in turn uses the built-in `fetchAll()` method in the `PDO` class. The `fetchAll()` method fetches data from a table as a two-dimensional array, where each element in the array is an array representing a row from the table.

In our **p-read.php** file, we assign the data fetched from the “posts”

table to `$posts`. To process `$posts`, we can loop through it using the following `foreach` loop.

```
foreach($posts as $result)
{
  $msgid = $result['id'];
  $title = htmlspecialchars($result['title']);
  $post = strip_tags($result['post'], "<strong><em>
<p><ol><ul><li><a>");
  $username =
htmlspecialchars($result['username']);
  $postdate =
htmlspecialchars($result['post_date']);


  include "output_code/messagecard.php";
}
```

For each iteration, we assign the array in `$posts` to `$result`. Next, inside the loop, we assign the elements in `$result` to various variables.

For instance, we assign `$result['title']` to `$title`. However, before we do that, we apply the `htmlspecialchars()` function to `$result['title']` first. The same applies to `$result['username']` and `$result['post_date']`.

The reason for this is we'll be displaying the values of these elements in **read.php** later. Hence, we have to convert any special characters in these elements to HTML entities to prevent cross-site scripting. Refer to Chapter 8.1.6 if you have forgotten what cross-site scripting is.

However, notice that we did not apply the `htmlspecialchars()` function to `$result['post']`?

This is because we want to allow certain HTML tags in `$result['post']`. Specifically, we want to allow the `<strong>`, `<em>`, `<p>`, `<ol>`, `<ul>`, `<li>` and `<a>` tags.

To do that, we need to use another built-in function called

`strip_tags()`. This function strips a string of all HTML tags <u>except</u> those passed as a second argument to the function. Hence,

```
strip_tags($result['post'], "<strong><em><p><ol><ul>
<li><a>");
```

strips `$result['post']` of all HTML tags except the ones we want. Got it?

Good! After assigning all the elements in `$result` to their respective variables, we use an `include` statement to include the **messagecard.php** file. **messagecard.php** is stored in the **output_code** folder and contains code for displaying the values of those variables above.

Once that is done, the `foreach` loop is complete. Based on the code and description given above, try completing the second `if-else` statement yourself.

Once you are done, we can move on to the last `if` statement. This statement checks if `$is_member` is `true`. If it is, we use an `include` statement to include the **logout.html** file. This file is stored in the **output_code** folder and contains HTML code with a logout link. In addition, we use an inner `if` statement to check if `$update` is `true`. If it is, we use the `$reader` object to call the `updateLastViewedPost()` method in the `BlogMember` class.

Try completing this `if` statement yourself. Once that is done, the **p-read.php** file is complete.

Great! You have completed the hardest file in this project; we just need to tie up some loose ends now.

13.6.6 messagecard.php

First, we need to add some code to the **messagecard.php** file. This file is found inside the **output_code** folder. As mentioned above, this file contains code for displaying posts retrieved from the “posts” table.

When a user is logged in to our blog, we need to display a “New” icon in **read.php** for posts that were posted after the user’s last login.

As posts are displayed using the **messagecard.php file**, we need to make some modifications to this file.

To do that, replace the comment (`//add PHP code here`) in **messagecard.php** with the following `if` statement:

```
<?php
  if ($is_member and $lastPost < $msgid)
  {
    echo '<span class = "new-post">NEW</span>';
    $update = true;
  }
?>
```

Here, we use an `if` statement to check if the user is logged in. In addition, we check if `$lastPost` is smaller than `$msgid`.

`$lastPost` stores the id of the last post viewed by the user. We got that by calling the `getLastViewedPost()` method in the first `if-else` statement in **p-read.php**.

`$msgid` stores the id of the current post in the `foreach` loop.

If `$lastPost` is smaller than the id of the current post, we know that this is a new post. Hence, we use an `echo` statement to echo a `<span>` tag with the word “NEW”. In addition, we set the value of `$update` to `true`.

`$update` is used to indicate whether we need to call the `updateLastViewedPost()` method (in the last `if` statement in **p-read.php**). If `$update` is `true`, we call the method to update the value of the “last_viewed” column in the “members” table to the latest post id in the “posts” table.

Got it? Refer to the `BlogMember` class (**BlogMember.php**) if you are not sure how the `updateLastViewedPost()` method works.

After inserting the `if` statement, we need to replace some text in **messagecard.php** with PHP code. Specifically, we need to replace “post_title”, “user_name”, “post_date” and “post_text” with the values of `$title`, `$username`, `$postdate` and `$post` respectively.

We do that using `echo` statements. For instance, to replace “post_title”, we write

```
<?php echo $title; ?>
```

Try replacing “user_name”, “post_date” and “post_text” yourself. However, before replacing “post_date”, you need to convert `$postdate` to a string. This is because `$postdate` currently stores a UNIX timestamp.

To display `$postdate` in a more human-readable format, you need to use the `date()` function to convert the timestamp to a datetime string. Try doing this yourself, using `'d-M-Y g:i a'` as the first argument to the function. Refer to Chapter 5.2.2 if you need help with the `date()` function.

Once you have updated **messagecard.php**, the **process** folder is complete.

13.7 The includes Folder

We are now ready to discuss the three remaining files in the **includes** folder. These three files are **header.html**, **debugging.php** and **loadclasses.php**.

header.html contains HTML code for the `<head>` element and has already been completed for you.

debugging.php contains code for handling errors and exceptions. This file was explained in detail in Chapter 12.3. Hence, we won’t be going through it here.

However, in this project, note that we did not try to catch any exceptions. Instead, we use **debugging.php** to handle all exceptions. This is because it is pointless for the site to proceed when an exception occurs. For instance, if we fail to connect to the database, the rest of the site will not work. Therefore, it makes sense to simply use **debugging.php** to handle the exception.

If an exception or error occurs, **debugging.php** displays a detailed message on the browser when `display_errors` is set to `'1'` (i.e.

when we are developing the site). When `display_errors` is set to `'0'` (i.e. on a live site), it logs the message and redirects users to **error.html** (which is stored in the main **phpproject** folder).

Last but not least, we have the **loadclasses.php** file. As the name suggests, this file is for autoloading classes. Did you notice that in all the previous files we coded, I asked you to create objects without asking you to include the relevant class files? For instance, in **p-admin.php**, I asked you to create a `Helper` object without asking you to include **Helper.php**.

This will typically lead to a fatal error as each PHP script is unaware of code written in another PHP script. Hence, **p-admin.php** is unaware of the class defined in **Helper.php**. To prevent that error, we need to include **Helper.php** before we create a `Helper` object.

However, instead of including this file ourselves, PHP provides us with a convenient alternative known as an autoloader. If we define an autoloader in our PHP script, whenever we create an object in that script, the autoloader includes the file for us automatically.

To use an autoloader, we need to ensure that the file name matches the class name. For instance, if the file name is **MyClass.php**, the class defined inside has to be called `MyClass`.

Besides that, using an autoloader is straightforward. If you open **loadclasses.php**, you'll see that we've defined a function called `myAutoloader()` that has one parameter - `$class`. Inside this function, we use an `include_once` statement to include the relevant class file.

`INC_DIR` is a constant that gives us a direct path to the **includes** folder in our project; we'll talk more about this constant in the next section.

Inside our `myAutoloader()` function, we concatenate `INC_DIR` with the string `'classes/'`, the variable `$class` and the string `'.php'` to get a direct path to the respective class files.

For instance, if `$class` equals `'Helper'`, the concatenation gives us

the string

```
INC_DIR.'classes/Helper.php';
```

which is a direct path to the **Helper.php** file. We then use the `include_once` statement to help us include this file. Got it?

After we code `myAutoloader()`, we need to use a built-in function called `spl_autoload_register()` to register it as the autoloader.

Once this is done, we simply need to include **loadclasses.php** in all our PHP scripts and PHP will autoload classes for us whenever we create an object.

That's it! We are now ready to go back to the main **phpproject** folder.

13.8 Editing The phpproject Folder

Inside this folder, we have eight files, seven of which are user interface files. The seven files are **signup.php**, **admin.php**, **write.php**, **index.php**, **read.php**, **logout.php** and **error.html**.

Besides these user interface files, we have a file called **UI_include.php**.

13.8.1 UI_include.php

We'll start with the **UI_include.php** file. If you open this file, you'll see that it has already been completed for you. Let's go through the code.

Inside the file, we first define a constant called `INC_DIR` and assign the string

```
$_SERVER["DOCUMENT_ROOT"]. "/phpproject/includes/"
```

to it. `$_SERVER['DOCUMENT_ROOT']` is a predefined variable that stores the document root directory under which the current script is executing. If you are using XAMPP, this root directory refers to the **htdocs** folder.

When we concatenate `$_SERVER['DOCUMENT_ROOT']` and `"/phpproject/includes/"`, we get a direct path to the **includes** folder.

If you rename the **phpproject** folder, you have to edit the path assigned to `INC_DIR` accordingly. For instance, if you rename the folder to **myproject**, you have to assign

```
$_SERVER["DOCUMENT_ROOT"]. "/myproject/includes/"
```

to `INC_DIR` instead.

Next, we have two `include` statements for including **loadclasses.php** and **debugging.php**. As mentioned previously, we use **loadclasses.php** to autoload our classes and **debugging.php** to handle all errors and exceptions. These two files have to be included in most of our user interface files later.

With that, the **UI_include.php** file is complete and we are ready to move on to the user interface files.

13.8.2 User Interface Files

The first is the **error.html** file. This file contains HTML code for displaying a custom message to users when an error or exception occurs on our site. It has already been completed for you.

Next, we have five very similar files - **admin.php**, **write.php**, **index.php**, **signup.php** and **read.php**.

Inside each file, we need to add PHP code to do the following: start a new session and include the **UI_include.php** and **header.html** files.

In addition, each of the files contains a HTML form that is processed by the file itself. To process the form, we need to use an `include` statement to add the relevant processing file to the user interface file. For instance, to process the form in **admin.php**, we need to include the **p-admin.php** file.

The above has already been done for you in **admin.php**.

With reference to the code in **admin.php**, try doing the same for **write.php**, **index.php**, **signup.php** and **read.php**. In each case, note that the **UI_include.php** file must be included before the other two files (as we need `INC_DIR` to be defined before using it). In addition, you need to change the processing file accordingly. Got it?

Great! Once you are done with the above, we need to make some modifications.

For **signup.php**, there is no need to start a new session as **p-signup.php** does not use any session variable. Hence, we should remove the `session_start();` statement from **signup.php**.

Next, for **read.php**, as **p-read.php** includes code for displaying output, we should not include it at the start of the file. Instead, we should include it between the `<body>...</body>` tags.

Last but not least, for **signup.php**, **admin.php**, **index.php** and **write.php**, we need to replace the text "Error Message Here" (without quotes) in the HTML code with the actual error message stored in a variable called `$msg`. To do that, we use the PHP code below:

```
<?php echo $msg; ?>
```

Try doing all the modifications above yourself.

Now, we need to make one more modification to **index.php**. If you refer to **p-signup.php**, you'll notice that we added the query string `new=1` when directing users to **index.php** after a successful sign-up, we want to make use of this query string inside **index.php**.

To do that, add the following code to **index.php** after the line `<div class="form">`:

```
<div class = "new">
  <?php
    if (isset($_GET['new']))
      echo 'ACCOUNT CREATED SUCCESSFULLY';
  ?>
</div>
```

Here, we use the query string to check if users are directed to **index.php** after a successful sign-up. If yes, we echo the string `'ACCOUNT CREATED SUCCESSFULLY'`.

Got it? Great. Let's move on to the last user interface file - **logout.php**. As the code for logging out is very straightforward, we did not create a separate file for processing **logout.php**. Instead, we'll

add the processing code to **logout.php** directly.

To do that, we need to do a few things in **logout.php**. First, we need to resume the existing session (using `session_start()`) and destroy all variables in that session (using `session_destroy()`). Next, we need to include the **UI_include.php** and **header.html** files.

Try doing the above yourself. Once that is done, load the page http://localhost/phpproject/logout.php in your browser; you'll notice two identical links that say, "Click here to log in again".

This is not a mistake. The first link points to **admin.php** while the second points to **index.php**.

If you refer back to **write.php** and study the code carefully, you'll notice that we added a query string (`admin=1`) to the logout URL near the end of the page. This query string tells us that the person logging out is an admin user.

When that happens, we want to display the first logout link in **logout.php**. On the other hand, if the person is not an admin user (i.e., there is no query string), we want to display the second logout link. Try using an `if-else` statement in **logout.php** to achieve the above. You can refer to **index.php** for help on using query strings.

Once that is done, the **logout.php** file is complete.

The project is almost complete at this point. In fact, if you have done everything correctly, you can load http://localhost/phpproject/signup.php and everything will work.

Try entering your desired username into the first input box and click "Submit". What do you notice? You should get an error message that says "All fields are required". Notice that the username you entered into the first input box is gone?

The last part of our project involves adding PHP code to your user interface files to ensure that values entered into forms are maintained if there's an error processing the form.

Recall that we wrote a method called `keepValues()` inside the `Helper` class for this purpose? Suppose we have a `Helper` object

called `$h`, here's how we use the method:

If we have a form with a textbox named "tb" and we store the value submitted for "tb" as `$tb`, we write

```
<input type="text" name="tb" <?php $h->keepValues($tb, 'textbox');?> >
```

If we have a textarea named "ta" and we store the value submitted for "ta" as `$ta`, we write

```
<textarea name = "ta"><?php $h->keepValues($ta, 'textarea'); ?></textarea>
```

Finally, if we have a dropdrop list named "sl" with two `value` attributes "1" and "2", and we store the value submitted for "sl" as `$sl`, we write

```
<select name = "sl">
  <option value = '1' <?php $h->keepValues($sl, 'select', '1'); ?>>1</option>
  <option value = '2' <?php $h->keepValues($sl, 'select', '2'); ?>>2</option>
</select>
```

In our project, the names of the variables used to store each input correspond to the names of the input fields. For instance, `$username` stores the input for the textbox named "username".

Based on the description above, modify **index.php**, **signup.php**, **admin.php** and **write.php** so that all values entered, except passwords, are maintained if there's an error processing the form. Got it?

Once the above is done, we need to make some additional changes to **write.php**. For most pages, if there's no error processing the form, the website automatically loads another page based on the URL we passed to the `header()` function.

However, this does not happen for **write.php**. For this page, after the

admin clicks on the “submit” button, he/she will stay on the same page regardless of whether data is submitted to the database successfully or not.

We want to modify **write.php** so that if data is inserted into the database successfully, we’ll clear the form so that the admin can write a new post. In other words, if data is inserted successfully, we do not want to call the `keepValues()` method.

To do that, we need to use the variable `$msg` declared in **p-write.php**. Recall that after data is inserted into the database, we assign the string `'Message saved successfully'` to `$msg`? We can use this string to decide whether we need to call the `keepValues()` method.

The example below shows how it can be done for the first input field (“title”). The underlined code shows the `if` condition to use.

```
<input id = "txttitle" type="text" name="title"
placeholder="Enter Title" autofocus <?php if ($msg
!= 'Message saved successfully') $h-
>keepValues($title, 'textbox'); ?>>
```

Try doing this for the other input fields.

Once that is done, we’ve completed the project. Congratulations! You are now ready to test your code to see if everything works as expected.

13.9 Running the Code

Before running the code, we need to make some changes to **php.ini**. Follow the instructions in Chapter 2.1 to locate **php.ini** and open it in Brackets. Scroll to the bottom of the page and add the following lines to it (if you have yet to do so in previous chapters):

```
error_reporting=E_ALL
display_errors=On
date.timezone=America/New_York
```

Next, we want to add one more line to **php.ini**. Specifically, we want to

add a line to prepend **debugging.php**. Prepending a file means specifying that we want PHP to process this file before it processes any other PHP script.

Previously, we used the `include` statement in **UI_include.php** to add **debugging.php** to our PHP scripts. This works for most errors. However, it will not work if the file that **debugging.php** is included in has syntax errors. If you want **debugging.php** to work even when there are syntax errors, you need to prepend the file. To do that, add the line

```
auto_prepend_file="
<DOCUMENT_ROOT>/phpproject/includes/debugging.php"
```

to **php.ini**, where `<DOCUMENT_ROOT>` refers to the actual path of your **htdocs** folder.

To find this path, load http://localhost/dashboard/phpinfo.php in your browser and search for “DOCUMENT_ROOT” (without quotes). You’ll see the path listed beside. You need to replace `<DOCUMENT_ROOT>` with the path listed. For instance, if the path is `/opt/lampp/htdocs`, the line should be

```
auto_prepend_file="/opt/lampp/htdocs/phpproject/incl
```

The code above may appear as two lines in this book due to the limited width of the book. Do not break it into two lines, it should be written as a single line in **php.ini**. After prepending **debugging.php**, if you are using PHP 7.2 and above, you should also turn `track_errors` off.

`track_errors` is a built-in feature in PHP that has been deprecated since PHP 7.2. However, this feature is set to “On” by default. Leaving it on may lead to a warning that says “Directive ‘track_errors’ is deprecated found on line 0 in file Unknown” on some servers. To turn it off, simply add

```
track_errors=Off
```

to the bottom of **php.ini**. That’s it. You can now save the file and restart Apache.

Next, go to **UI_include.php** and comment out the `include` statement for **debugging.php**. As we have already prepended the file, we should not include it again.

Next, launch http://localhost/phpproject/signup.php in your browser and sign up for **two** new accounts.

Everything works? If yes, congratulations! Give yourself a pat on the back.

If no, it's all right. Figuring out what went wrong is a large part of programming. This, to me, is where the most learning takes place. If something fails to work, you will likely get an error message. Try to use the line number and file name given in the error message to figure out what is wrong.

Alternatively, you can use `echo` statements to determine which part of your code works. For instance, if you add an `echo` statement to line 10 of your code and you don't see the output, you know that something has likely gone wrong before line 10. Similarly, if you add an `echo` statement to an `if` block and don't see the output, you know that the `if` block is not executed.

Try finding the error yourself. If you really can't figure out the issue, you can check the suggested solution in the **phpproject-complete** folder and compare the code with your code. Copy and paste the functions that you suspect may be causing the error from the suggested solution to your solution and rerun your code to see if it works. Study the suggested solution carefully to really understand how it works. Got it?

Once you manage to find the bug and are able to sign up for two accounts successfully, you can proceed to convert one of them to an admin account. To do that, go to http://localhost/phpmyadmin/ and click on the "project" database. Click on the SQL tab and run the following SQL statement, replacing `YOUR_ADMIN_USERNAME` with the username you want to convert to an admin user:

```
UPDATE members SET is_admin = 1 WHERE username =
'YOUR_ADMIN_USERNAME';
```

Once that is done, you can use the admin account to post to the blog. To do that, go to http://localhost/phpproject/admin.php to log in as an admin. Try adding some posts to the blog. Does everything work?

After posting, you can go to http://localhost/phpproject/index.php to log in as a member (using either account) to read the posts.

Alternatively, you can go to http://localhost/phpproject/read.php directly to read "Public" posts without logging in.

Play around with the site to see if everything works as expected. If something does not work, try debugging it. With perseverance, you can definitely find the error and learn a lot in the process. Have fun!