

PHP CRASH COURSE

THE COMPLETE, MODERN, HANDS-ON GUIDE

MATT SMITH



PHP CRASH COURSE

THE COMPLETE, MODERN, HANDS-ON GUIDE

MATT SMITH



CONTENTS IN DETAIL

[TITLE PAGE](#)

[COPYRIGHT](#)

[DEDICATION](#)

[ABOUT THE AUTHOR AND ABOUT THE TECHNICAL REVIEWERS](#)

[ACKNOWLEDGMENTS](#)

[INTRODUCTION](#)

[Who This Book Is For](#)

[Why PHP?](#)

[What You Will Learn](#)

[Online Resources](#)

[PART I: LANGUAGE FUNDAMENTALS](#)

[1](#)

[PHP PROGRAM BASICS](#)

[Two Methods to Run PHP](#)

[The Replit Online Coding Environment](#)

[A Local PHP Installation](#)

[Template Text vs. PHP Code](#)

[Comments](#)

[Variables](#)

[Creating Variables](#)

[Using Variables](#)

[Naming Variables](#)

[Constants](#)

[Operators and Operands](#)

[Arithmetic Operators](#)

[Combined Arithmetic Assignment Operators](#)

[Increment and Decrement Operators](#)

[Summary](#)

[Exercises](#)

2 **DATA TYPES**

[PHP Data Types](#)

[Scalar Data Types](#)

[The Special NULL Type](#)

[Functions to Test for a Data Type](#)

[Type Juggling](#)

[Numeric Contexts](#)

[String Contexts](#)

[Comparative Contexts](#)

[Logical and Other Contexts](#)

[Type Casting](#)

[Summary](#)

[Exercises](#)

3 **STRINGS AND STRING FUNCTIONS**

[Whitespace](#)

[Single-Quoted Strings](#)

[Joining Strings: Concatenation](#)

[Double-Quoted Strings](#)

[Handling the Character After a Variable Name](#)

[Incorporating Unicode Characters](#)

[Heredocs](#)

[Escape Sequences](#)

[Indentation](#)

[Nowdocs](#)

[Built-in String Functions](#)

[Converting to Upper- and Lowercase](#)

[Searching and Counting](#)

[Extracting and Replacing Substrings](#)

[Trimming Whitespace](#)

[Removing All Unnecessary Whitespace](#)

[Repeating and Padding](#)

[Summary](#)

[Exercises](#)

4

CONDITIONALS

[Conditions Are True or False](#)

[if Statements](#)

[if...else Statements](#)

[Nested if...else Statements](#)

[if...elseif...else Statements](#)

[Alternative Syntax](#)

[Logical Operators](#)

[NOT](#)

[AND](#)

[OR](#)

[XOR](#)

[switch Statements](#)

[match Statements](#)

[The Ternary Operator](#)

[The Null-Coalescing Operator](#)

[Summary](#)

[Exercises](#)

5

CUSTOM FUNCTIONS

[Separating Code into Multiple Files](#)

[Reading in and Executing Another Script](#)

[Creating Absolute Filepaths](#)

[Declaring and Calling a Function](#)

[Parameters vs. Arguments](#)

[Errors from Incorrect Function Calls](#)

[Type Juggling](#)

[Functions Without Explicit Return Values](#)

[Returning NULL](#)

[Exiting a Function Early](#)

[Calling Functions from Within Functions](#)

[Functions with Multiple Return and Parameter Types](#)

[Nullable Types](#)

[Union Types](#)

[Optional Parameters](#)

[Positional vs. Named Arguments](#)

[Skipped Parameters](#)

[Pass-by-Value vs. Pass-by-Reference](#)

[Summary](#)

[Exercises](#)

PART II: WORKING WITH DATA

6

LOOPS

[while Loops](#)

[do...while Loops](#)

[Boolean Flags](#)

[break Statements](#)

[for Loops](#)

[Using the Counter in the Loop](#)

[Skipping Loop Statements](#)

[Handling the Last Iteration Differently](#)

[Alternative Loop Syntax](#)

[Avoiding Infinite Loops](#)

[Summary](#)

[Exercises](#)

7

SIMPLE ARRAYS

[Creating an Array and Accessing Its Values](#)

[Updating an Array](#)

[Appending an Element](#)

[Adding an Element with a Specific Key](#)

[Appending Multiple Elements](#)

[Removing the Last Element](#)

[Retrieving Information About an Array](#)

[Looping Through an Array](#)

[Using a foreach Loop](#)

[Accessing Keys and Values](#)

[Imploding an Array](#)

[Functions with a Variable Number of Arguments](#)

[Array Copies vs. Array References](#)

[Treating Strings as Arrays of Characters](#)

[Other Array Functions](#)

[Summary](#)

[Exercises](#)

8

SOPHISTICATED ARRAYS

[Declaring Array Keys Explicitly](#)

[Arrays with Strings as Keys](#)

[Multidimensional Arrays](#)

[More Array Operations](#)

[Removing Any Element from an Array](#)

[Combining and Comparing Arrays](#)

[Destructuring an Array into Multiple Variables](#)

[Callback Functions and Arrays](#)

[Summary](#)

[Exercises](#)

9

FILES AND DIRECTORIES

[Reading a File into a String](#)

[Confirming That a File Exists](#)

[“Touching” a File](#)

[Ensuring That a Directory Exists](#)

[Writing a String to a Text File](#)

[Managing Files and Directories](#)

[Reading a File into an Array](#)

[Using Lower-Level File Functions](#)

[Processing Multiple Files](#)

[JSON and Other File Types](#)

[Summary](#)

[Exercises](#)

PART III: PROGRAMMING WEB APPLICATIONS

10

CLIENT/SERVER COMMUNICATION AND WEB DEVELOPMENT BASICS

[The HTTP Request-Response Cycle](#)

[Response Status Codes](#)

[An Example GET Request](#)

[How Servers Operate](#)

[Simple Web Servers for File Retrieval](#)

[Dynamic Web Servers for Processing Data](#)

[The Routing Process](#)

[Templating](#)

[PHP Tags](#)

[Short Echo Tags](#)

[The Model-View-Controller Architecture](#)

[Structuring a PHP Web Development Project](#)

[Summary](#)

[Exercises](#)

11

CREATING AND PROCESSING WEB FORMS

[Basic Client/Server Communication for Web Forms](#)

[GET vs. POST Requests](#)

[A Simple Example](#)

[The filter_input\(\) Function](#)

[Other Ways to Send Data](#)

[Sending Noneditable Data Along with Form Variables](#)

[Processing Mixed Query-String and POST Variables](#)

[Offering Multiple Submit Buttons](#)

[Encoding Data in Hyperlinks](#)

[Other Form Input Types](#)

[Radio Buttons](#)

[Checkboxes](#)

[Single-Selection Lists](#)

[Multiple-Selection Lists](#)

[Summary](#)

[Exercises](#)

12

VALIDATING FORM DATA

[Writing Custom Validation Logic](#)

[Managing Multiple Validation Errors](#)

[Testing for a Valid Zero Value](#)

[Displaying and Validating Forms in a Single Postback Script](#)

[Simple Validation Logic](#)

[Array-Based Validation Logic](#)

[Summary](#)

[Exercises](#)

13

ORGANIZING A WEB APPLICATION

[Front Controllers and the MVC Architecture](#)

[Separating Display and Logic Files](#)

[Creating the Front Controller](#)

[Writing the Display Scripts](#)

[Moving Website Logic into Functions](#)

[Designing a Secure Folder Structure](#)

[Simplifying the Front-Controller Script](#)

[Writing the Functions](#)

[Generalizing the Front-Controller Structure](#)

[Distinguishing Between Requested Pages](#)

[Building a Multipage Application](#)

[Summary](#)

[Exercises](#)

PART IV: STORING USER DATA WITH BROWSER SESSIONS

14 **WORKING WITH SESSIONS**

[A Web Browser Session](#)

[The session_start\(\) and session_id\(\) Functions](#)

[The \\$_SESSION Superglobal Array](#)

[Updating a Stored Value](#)

[Unsetting a Value](#)

[Destroying the Session and Emptying the Session Array](#)

[Summary](#)

[Exercises](#)

15 **IMPLEMENTING A SHOPPING CART**

[The Shopping Cart File Structure](#)

[Defining the Product List](#)

[Creating the Products Array](#)

[Adding CSS](#)

[Displaying the Star Ratings](#)

[Creating the Template Script](#)

[Updating the Index Script](#)

[Designing the Shopping Cart](#)

[Creating the Front Controller](#)

[Managing the Product and Cart Arrays](#)

[Streamlining the Index Script](#)

[Creating a Header Template](#)

[Creating the Cart Display Template](#)

[Interacting with the Session](#)

[Updating the Cart-Retrieval Function](#)

[Implementing Cart-Manipulation Functions](#)

[Creating the Empty Cart Template](#)

[Finalizing the Front Controller](#)

[Adding Display Functions](#)

[Writing the switch Statement](#)

[Summary](#)

[Exercises](#)

16 **AUTHENTICATION AND AUTHORIZATION**

[A Simple Login Form](#)

- [Creating a Site with a Login Form](#)
 - [Defining the File Structure](#)
 - [Creating the Shared Page Content](#)
 - [Designing the Page Templates](#)
 - [Developing the Login Form](#)
 - [Writing the Front Controller](#)
 - [Implementing the Logic Functions](#)
 - [Creating the Error Page Template](#)
- [Storing Login Data with Sessions](#)
 - [Updating the Front Controller](#)
 - [Writing the Login Function](#)
 - [Updating the Header Template](#)
 - [Updating the Banking Page Template](#)
- [Offering a Logout Feature](#)
 - [Adding the Logout Function](#)
 - [Updating the Front Controller](#)
 - [Displaying the Logout Link](#)
- [Displaying the Logged-in Username](#)
 - [Retrieving the Username](#)
 - [Updating the Navigation Bar](#)
 - [Updating the CSS](#)
- [Summary](#)
- [Exercises](#)

PART V: OBJECT-ORIENTED PHP

17

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

- [Classes and Objects](#)
- [Creating Relationships Between Objects](#)
- [Encapsulation and Information Hiding](#)
- [Superclasses, Inheritance, and Overriding](#)
- [The Flow of Control for Object-Oriented Systems](#)
- [An Example Class Declaration](#)
- [Summary](#)
- [Exercises](#)

18

DECLARING CLASSES AND CREATING OBJECTS

- [Declaring a Class](#)
- [Creating an Object](#)
- [Private Properties with Public Accessor Methods](#)

- [Getting and Setting Private Properties](#)
- [Screening for Invalid Data](#)
- [Overriding Default Class Behavior with Magic Methods](#)
 - [Initializing Values with a Constructor Method](#)
 - [Converting Objects to Strings](#)
- [Object Variables as References](#)
- [Handling Missing Objects](#)
- [Custom Methods and Virtual Attributes](#)
- [Summary](#)
- [Exercises](#)

19

INHERITANCE

- [Inheritance as Generalization](#)
 - [Creating Objects from Subclasses](#)
 - [Using Multiple Levels of Inheritance](#)
- [Protected Visibility](#)
- [Abstract Classes](#)
- [Overriding Inherited Methods](#)
- [Augmenting Inherited Behavior](#)
- [Preventing Subclassing and Overriding](#)
 - [Declaring a Class final](#)
 - [Declaring a Method final](#)
- [Summary](#)
- [Exercises](#)

20

MANAGING CLASSES AND NAMESPACES WITH COMPOSER

- [Namespaces](#)
 - [Declaring a Class's Namespace](#)
 - [Using a Namespaced Class](#)
 - [Referencing Namespaces in Class Declarations](#)
- [Composer](#)
 - [Installing and Testing Composer](#)
 - [Creating the composer.json Configuration File](#)
 - [Creating an Autoloader](#)
 - [Adding Third-Party Libraries to a Project](#)
- [Where to Find PHP Libraries](#)
- [Summary](#)
- [Exercises](#)

21

EFFICIENT TEMPLATE DESIGN WITH TWIG

[The Twig Templating Library](#)
 [How Twig Works](#)
 [A Simple Example](#)
[Manipulating Objects and Arrays in Twig Templates](#)
[Twig Control Structures](#)
[Creating a Multipage Website with Twig](#)
 [The File Structure and Dependencies](#)
 [The Application Class](#)
 [The Twig Templates](#)
 [Twig Features to Improve Efficiency](#)
 [Improved Page Styling with CSS](#)
[Summary](#)
[Exercises](#)

22

STRUCTURING AN OBJECT-ORIENTED WEB APPLICATION

[Separating Display and Front-Controller Logic](#)
[Using Multiple Controller Classes](#)
[Sharing Controller Features Through Inheritance](#)
[Summary](#)
[Exercises](#)

23

ERROR HANDLING WITH EXCEPTIONS

[The Basics of Exceptions](#)
 [Throwing an Exception](#)
 [Catching an Exception](#)
 [Ending with a finally Statement](#)
[Using Multiple Exception Classes](#)
 [Other Built-in Exception Classes](#)
 [Custom Exception Classes](#)
[Call-Stack Bubbling](#)
[Summary](#)
[Exercises](#)

24

LOGGING EVENTS, MESSAGES, AND TRANSACTIONS

[Built-in PHP Resources for Logging](#)
 [Predefined Constants for Severity Levels](#)
 [Logging Functions](#)
[The Monolog Logging Library](#)
 [Organizing Logs with Channels](#)
 [Managing Logs According to Severity](#)

[Logging Exceptions](#)
[Logging to the Cloud](#)
[Summary](#)
[Exercises](#)

25

STATIC METHODS, PROPERTIES, AND ENUMERATIONS

[Storing Class-Wide Information](#)
[Static Properties vs. Class Constants](#)
[Utility Classes with Static Members](#)
[Sharing Resources Across an Application](#)
[Saving Resources with the Singleton Pattern](#)
[Enumerations](#)
 [Backed Enums](#)
 [An Array of All Cases](#)
[Summary](#)
[Exercises](#)

26

ABSTRACT METHODS, INTERFACES, AND TRAITS

[From Inheritance to Interfaces](#)
 [Inheriting a Fully Implemented Method from a Superclass](#)
 [Inheriting an Abstract Method](#)
 [Requiring Method Implementations with Interfaces](#)
[Real-World Applications of Interfaces](#)
 [Caching Approach 1: Using an Array](#)
 [Caching Approach 2: Using a JSON File](#)
 [Caching Approach 3: Creating a Cacheable Interface](#)
[Traits](#)
 [Declaring Traits](#)
 [Inserting Traits](#)
 [Resolving Trait Conflicts](#)
[What to Use When?](#)
[Summary](#)
[Exercises](#)

PART VI: DATABASE-DRIVEN APPLICATION DEVELOPMENT

27

INTRODUCTION TO DATABASES

[Relational Database Basics](#)

[Database Management Systems](#)
[Structured Query Language](#)
[Databases and Web Application Architecture](#)
[Object-Oriented Programming](#)
[The Model-View-Controller Pattern](#)
[Summary](#)
[Exercises](#)

28

DATABASE PROGRAMMING WITH THE PDO LIBRARY

[The PDO Library](#)
[A Simple Database-Driven Web Application](#)
[Setting Up the Database Schema](#)
[Writing the PHP Classes](#)
[Switching from MySQL to SQLite](#)
[A Multipage Database-Driven Web Application](#)
[Managing the Product Information](#)
[Implementing the Controller Logic](#)
[Designing the Templates](#)
[Summary](#)
[Exercises](#)

29

PROGRAMMING CRUD OPERATIONS

[Deleting Data](#)
[Deleting Everything from a Table](#)
[Deleting Individual Items by ID](#)
[Creating New Database Entries](#)
[Adding Products Through a Web Form](#)
[Highlighting the Newly Created Product](#)
[Updating a Database Entry](#)
[Avoiding Double Form Submission with Redirects](#)
[Summary](#)
[Exercises](#)

30

ORM LIBRARIES AND DATABASE SECURITY

[Simplifying Database Code with an ORM Library](#)
[Adding an ORM Library to a Project](#)
[Moving Database Credentials to a .env File](#)
[Relegating Product Operations to the ORM Library](#)
[Adding a New Database Table](#)
[Security Best Practices](#)
[Storing Hashed Passwords](#)

[Verifying Hashed Passwords at Login](#)

[Securing Database Credentials](#)

[The Doctrine ORM Library](#)

[Removing the Previous ORM Library](#)

[Adding Doctrine](#)

[Verifying That Doctrine Is Working](#)

[Creating Database Tables](#)

[Adding Records to a Table](#)

[Integrating Doctrine into the Application Code](#)

[Creating Foreign-Key Relationships](#)

[Summary](#)

[Exercises](#)

31

WORKING WITH DATES AND TIMES

[The ISO 8601 Standard](#)

[Creating Dates and Times](#)

[Formatting the Date-Time Information](#)

[Using DateTimeImmutable vs. DateTime](#)

[Manipulating Dates and Times](#)

[Using Date-Time Intervals](#)

[Looping at Regular Intervals](#)

[Time Zones](#)

[Daylight Saving Time](#)

[Epochs and Unix Time](#)

[Date-Time Information in a Web Application](#)

[The Application Class](#)

[The Supporting Classes](#)

[The Templates](#)

[MySQL Dates](#)

[Summary](#)

[Exercises](#)

A

INSTALLING PHP

[macOS](#)

[Linux](#)

[Windows](#)

[AMP Installations](#)

B

DATABASE SETUP

[MySQL](#)

[macOS and Windows](#)

[Linux](#)

[SQLite](#)

[Confirming the MySQL and SQLite Extensions](#)

C

REPLIT CONFIGURATION

[Changing the PHP Version](#)

[Adding the Composer Tool](#)

[Using the SQLite Database System](#)

[Serving Pages from the public Directory](#)

INDEX

PHP CRASH COURSE

The Complete, Modern, Hands-on Guide

by Matt Smith



San Francisco

PHP CRASH COURSE. Copyright © 2025 by Matt Smith.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

28 27 26 25 24 1 2 3 4 5

ISBN-13: 978-1-7185-0252-9 (print)

ISBN-13: 978-1-7185-0253-6 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock

Managing Editor: Jill Franklin

Production Manager: Sabrina Plomitallo-González

Production Editor: Sydney Cromwell

Developmental Editor: Nathan Heidelberger

Cover Illustrator: Gina Redman

Interior Design: Octopod Studios

Technical Reviewers: Ryan Weaver and Eoghan Ó hUallacháin

Copyeditor: Sharon Wilkey

Proofreader: Daniel Wolff

[Figure 28-1](https://xkcd.com/327/) is reproduced with permission from <https://xkcd.com/327/>.

Library of Congress Cataloging-in-Publication Data

Name: Smith, Matt, 1967-author.

Title: PHP Crash Course : The Complete, Modern, Hands-on Guide / by Matt Smith.

Description: San Francisco : No Starch Press, 2025. | Includes index.

Identifiers: LCCN 2024028603 (print) | LCCN 2024028604 (ebook) | ISBN 9781718502529 (print) | ISBN 9781718502536 (ebook)

Subjects: LCSH: PHP (Computer program language) | Internet programming--Computer programs. | Computer programming.

Classification: LCC QA76.73.P224 S566 2025 (print) | LCC QA76.73.P224 (ebook) | DDC 005.13/3--dc23/eng20241007

LC record available at <https://lcn.loc.gov/2024028603>

LC ebook record available at <https://lcn.loc.gov/2024028604>

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

[E]

I dedicate this book to Ryan Weaver.

Having this book reviewed by someone of your experience and expertise has been a fantastic privilege. Working with a core Symfony developer has helped ensure that the content is accurate and reflects professional PHP best practices. Quite simply, the book is better than it would have been without your input.

My sincere thanks and best wishes to you and your family.

About the Author

Dr. Matt Smith is senior lecturer in computing at Technological University (TU) Dublin, Ireland, specializing in interactive multimedia instruction including web applications, extended reality (XR), and e-learning systems. He holds a bachelor's degree in business computing (University of Huddersfield, UK), a master's in artificial intelligence (University of Aberdeen, Scotland), and a PhD in computational musicology (Open University, UK). He has been teaching for over 30 years, holding full-time positions at the University of Winchester and London's Middlesex University prior to TU Dublin.

Smith introduced Unity-based 3D game development and interactive virtual environments to his computing and digital media students. In the mid-2000s, at the request of his students, he switched the focus of his web development courses from Java to PHP, and he's been a fan of PHP and the Symfony open source web framework ever since.

Smith has a first-degree black belt in Taekwondo and has taught and competed in that martial art, though when his daughter switched to Shotokan karate, so did he, and he hopes to earn his first-degree karate black belt at about the time this book is published. While at school in the 1980s, Smith wrote the lyrics for, and his band played the music on, the B-side of the audio cassette carrying the computer game *Confuzion* by Incentive Software; the game has a Wikipedia page. He also sang some of the backup vocals, for which he apologizes.

About the Technical Reviewers

Ryan Weaver is a member of the Symfony core team and a writer for SymfonyCasts. He has contributed to and created numerous open source PHP packages. He lives in Grand Rapids, Michigan, with his wife, Leanna, and son, Beckett.

Eoghan Ó hUallacháin is the founder and CEO of Glorsoft, a software engineering company specializing in architecting and developing high-availability, high-volume, multiuser, speed-critical systems. He has over 30 years of experience delivering highly scalable systems for clients in e-commerce, telecoms, finance, government, and other sectors. He's been writing PHP code since 1999 and uses it for user interface and automation work. He holds a BS in computer applications from Dublin City University and an MS in advanced software engineering from University College Dublin.

ACKNOWLEDGMENTS

To the students who encouraged me to learn (and then teach) PHP for our web development degree stream, thank you. It has become my favorite programming language.

I'm very grateful to the team at No Starch Press, most especially to Nathan, Jill, Sydney, Sharon, and Sabrina. Of all the books I've worked on over the years, this book (my first with No Starch Press) has been the most enjoyable to write. Many thanks for putting up with the slow pace at times, and for all of your constructive feedback and editorial comments.

Many thanks to Eoghan Ó hUallacháin for helping me get the book over the finish line (and to many more beer-and-pizza coding events).

To my parents, thanks for everything. Here's another thick computing book for your shelves. And thanks to Sinéad, Charlotte, and Luke for putting up with me writing another book. No more books for a while, I promise!

INTRODUCTION



PHP is one of the engines that drive the internet: it plays a role both in what the user sees on web pages and in what happens behind the scenes, such as processing form submissions, talking to other websites, and interacting with databases. The language was first released in 1995, but it wasn't until the late 2000s, when my computing students encouraged me to introduce PHP into their web programming classes, that I began working with it in earnest. They wanted to improve their skill profiles for the job market, since they felt the internet was going to continue to grow in importance in the work of computing. Clearly, they were right.

Who This Book Is For

PHP Crash Course is for anyone wanting to learn PHP programming in a practical, hands-on way, regardless of whether you have previous programming experience. Since most PHP programs are web applications, knowing about the HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) languages used to represent the content and visual style of web pages might prove helpful, but you don't need to know any web programming, such as JavaScript.

Why PHP?

PHP is currently in its eighth major version, so as well as being tried and tested, it's faster and more secure than ever before. It's a free, open source, and well-maintained language. While other popular web programming languages are available, about 70 percent of the web is run by PHP programs, including Etsy, Facebook (using its dialect of PHP, called Hack), Spotify, Wikipedia, and WordPress.

PHP also has a relatively shallow learning curve. We'll start off with just a few lines of code, and once you're comfortable with the basics, we'll move on to larger, more structured web application systems.

NOTE

The original version of the language was released as Personal Home Page Tools (PHP Tools). These days, however, PHP is a recursive acronym that stands for PHP: Hypertext Preprocessor.

What You Will Learn

In this book, you'll learn to program in PHP, from short, simple scripts to multi-file, database-driven, login-secured, object-oriented web applications.

In [Part I](#), Language Fundamentals, you'll start writing small PHP program scripts. This part introduces some fundamentals of the language, including storing different types of values in named variables, working with text, and writing decision-based logic.

[Chapter 1: PHP Program Basics](#) Introduces writing and executing PHP scripts in both an online coding environment and an editor on

your computer.

Chapter 2: Data Types Discusses the different types of data and how PHP automatically converts between them.

Chapter 3: Strings and String Functions Covers working with text, both in your own code and in some of the built-in text functions that the language offers.

Chapter 4: Conditionals Explores conditional elements of the language, like `if...else`, `switch`, and `match`, and offers guidelines for when each is most appropriate. You'll also learn about operators, such as logical comparisons, that assist in the implementation of decision-making logic.

Chapter 5: Custom Functions Introduces functions, which are reusable, self-contained sequences of code to accomplish tasks.

In **Part II**, Working with Data, you'll work with loops to repeat actions and learn about more sophisticated data structures such as arrays and files.

Chapter 6: Loops Covers adding flexibility and avoiding code duplication through structures to repeat actions.

Chapter 7: Simple Arrays Introduces arrays, a mechanism for storing and manipulating multiple data items under a single variable name.

Chapter 8: Sophisticated Arrays Moves beyond the previous chapter into more sophisticated key-value maps and multidimensional arrays.

Chapter 9: Files and Directories Explores how and when to use functions to interact with files in your PHP scripts.

In **Part III**, Programming Web Applications, you'll begin creating web applications, including receiving and validating data submitted through forms.

Chapter 10: Client/Server Communication and Web Development Basics Introduces important concepts about clients, servers, and how PHP-driven web applications work.

Chapter 11: Creating and Processing Web Forms Covers how to design web forms and how to write PHP scripts that process the data

submitted through those forms.

Chapter 12: Validating Form Data Explores ways to validate received data and covers some typical decision logic to take appropriate actions depending on the correctness of the values received or missing.

Chapter 13: Organizing a Web Application Progressively explores the model-view-controller (MVC) software architectural approach, which divides up the responsibilities for maintaining an application among various scripts, allowing the application to grow without becoming unmanageable.

Part IV, Storing User Data with Browser Sessions, introduces sessions, which allow websites to remember data across page requests; this is useful for shopping carts and logins.

Chapter 14: Working with Sessions Introduces the PHP features that allow web applications to remember information over time.

Chapter 15: Implementing a Shopping Cart Covers how to add items to a shopping cart and how to remember the items until the user is ready to check out and pay.

Chapter 16: Authentication and Authorization Implements security authentication (determining the identity of the person using the computer system) and authorization (deciding whether the user is permitted to access a particular part of the system).

In **Part V**, Object-Oriented PHP, you'll learn about the powerful technique of object-oriented programming (OOP).

Chapter 17: Introduction to Object-Oriented Programming Discusses the motivation for moving from functions to an OOP approach and the differences between these options.

Chapter 18: Declaring Classes and Creating Objects Covers the core OOP features of classes and objects.

Chapter 19: Inheritance Describes the powerful OOP concept of inheritance and how to implement it in code.

Chapter 20: Managing Classes and Namespaces with Composer Covers key mechanisms for implementing OOP solutions in the PHP

programming language and shows how the Composer command line tool can help.

[Chapter 21: Efficient Template Design with Twig](#) Explores the Twig library's inheritance-based system for developing page templates, which simplifies the process of “decorating” application data with HTML to be returned to the user.

[Chapter 22: Structuring an Object-Oriented Web Application](#) Introduces a commonly used, scalable web application software architecture.

[Chapter 23: Error Handling with Exceptions](#) Discusses the error-handling mechanism of exceptions, a feature of many programming languages.

[Chapter 24: Logging Events, Messages, and Transactions](#) Shows how to maintain a system log, including outputting to a logfile or to an external, cloud-based logging service, as is common for large-scale web applications.

[Chapter 25: Static Methods, Properties, and Enumerations](#) Covers the OOP features of class-level, static members. Also covers enumerations, the special category of class making it easy to offer a fixed set of possible values.

[Chapter 26: Abstract Methods, Interfaces, and Traits](#) Explores ways to share methods among multiple classes without the normal process of inheritance.

[Part VI](#), Database-Driven Application Development, walks through writing programs that communicate with database systems. It ends with a discussion about working with dates and times, both in your program code and stored in databases.

[Chapter 27: Introduction to Databases](#) Introduces databases and their relationship with web applications.

[Chapter 28: Database Programming with the PDO Library](#) Discusses writing code to communicate with databases.

[Chapter 29: Programming CRUD Operations](#) Shows how to introduce database CRUD (create, read, update, delete) features to a web application.

Chapter 30: ORM Libraries and Database Security Describes the benefits of automating the relationship between code and database structures through object-relational mapping (ORM) libraries, and outlines several best practices for secure, database-driven web development.

Chapter 31: Working with Dates and Times Covers ways to work with temporal information, including how to handle ambiguities like time zones and daylight saving time.

Finally, the appendixes cover how to set up the tools you'll need to get started with PHP.

Appendix A: Installing PHP Goes through the steps to install PHP for macOS, Linux, and Windows computer systems.

Appendix B: Database Setup Covers how to make sure the MySQL and SQLite database management systems are set up on a local computer.

Appendix C: Replit Configuration Discusses how to reconfigure more advanced Replit projects to work with tools such as the Composer dependency manager and a database management system.

Online Resources

The code listings from this book and my suggested exercise solutions are available to download at <https://github.com/dr-matt-smith/php-crash-course>.

Exercises are included at the end of each chapter. I recommend first attempting these yourself before looking at my solutions.

For updates and other information about the book, see <https://nostarch.com/php-crash-course>.

PART I

LANGUAGE FUNDAMENTALS

1

PHP PROGRAM BASICS



In this chapter, you'll learn two ways to create and run PHP programs: using an online coding environment and using an editor installed locally on your own computer. We'll try both techniques to practice key programming tasks like printing out text messages, assigning values to variables, and working with data of different types. We'll also explore core PHP language features including comments, constants, and expressions.

Two Methods to Run PHP

Often the easiest way to learn a programming language is to use an online environment that has everything already set up for you. This lets you start coding right away and see the results instantly, without having to install and configure language engines, code editors, web servers, or other tools. On the other hand, some prefer the customization and control that comes from working in a programming environment installed on their own machine.

In this section, you'll get to explore both approaches as you write your first PHP programs. You can then use either method to follow along with

the examples throughout this book.

The Replit Online Coding Environment

Several online sites facilitate interactive PHP development and can run PHP web servers for you. We'll focus on Replit (<https://replit.com>), a popular service that's free to use for starter projects. To try it out, go to the Replit website and create an account.

NOTE

Replit is named for the read-evaluate-print loop (REPL), a type of computer environment where the programmer enters an expression and the system immediately executes it, prints out the response, and waits for the next input. A command line terminal, where you enter single-line commands and the terminal executes those commands, is a type of REPL.

Replit features two official, preconfigured templates for creating PHP projects: PHP CLI (short for *command line interface*) and PHP Web Server. The former is appropriate for projects that simply output text in a command line terminal window or work with data files, while the latter is for web development. Let's take a look at each of these templates so you can learn a bit about how PHP programs work in the process.

Creating a Command Line Interface Project

To create a PHP command line interface project, go to the home screen of your Replit account and click **Create Repl**. This will launch a pop-up window where you can search for project templates. Enter **PHP** into the search box. The results should include the PHP CLI and PHP Web Server templates from Replit. (Below these official Replit templates, you may also see other templates created by Replit users and tagged for the PHP language.) Choose **PHP CLI** and enter a name for your project, or take the random-words default name offered. Then click **Create Repl** to launch the project.

The website will take a short while to set up your new project, including creating its file and folder structure and starting up a cloud virtual machine to run it. When the project loads, you'll be presented with the three-column screen shown in [Figure 1-1](#).



Figure 1-1: The new PHP CLI project screen, with the default “Hello, world!” script

The left column lists the project’s files and folders, the middle column is an online code editor, and the right column is the command line terminal output (called *Console*) and an interactive terminal (called *Shell*) for the virtual computer that Replit has created. The Run button at the top of the screen runs the project, at which point any output will be displayed in the console.

A typical PHP project includes one or more files, called *scripts*, saved with the *.php* file extension. In this case, the Replit PHP CLI project automatically starts with a prewritten file called *main.php*. The file contains PHP code to output the message `Hello, world!` in the console. Writing a program that displays this message is a tradition when learning a new language. Besides being fun, it provides an opportunity to learn how to name the text files containing your programs, how to write valid statements in your language of choice, and how to execute a program. What’s more, a “Hello, world!” script serves as a basic test of the language tools on the computer system: if the program runs and successfully outputs the `Hello, world!` message, that means PHP is working.

[Listing 1-1](#) shows the boilerplate “Hello, world!” script that Replit provides in *main.php*.

```
<?php
echo "Hello, world!\n";
```

Listing 1-1: The “Hello, world!” program in *main.php*

The `<?php` at the start of the script is an opening PHP tag. This tag signals that what follows is PHP code. In this case, the code uses the `echo` command to print out the text `Hello, world!` in the console, followed by a

line break denoted with a newline character (`\n`). Notice that the text to be printed is enclosed in double quotation marks. These quotes indicate that the text is a *string*, a type of data consisting of a sequence of characters. We'll discuss strings and special characters like `\n` in detail in [Chapter 3](#).

The `echo` line of code is an example of a *statement*, a single command directing the computer to perform a task (in this case, to display some text). Every PHP statement must end with a semicolon (`;`) to indicate the command is over, as this `echo` statement does. Think of the semicolon as the period at the end of a sentence; without it, the statement is considered incomplete.

Running the main.php Script

Click the green **Run** button to run the *main.php* script. You should see the `Hello, world!` message printed to the console. Congratulations, you've just run your first PHP program! But what actually happened when you clicked Run?

PHP is a *scripted* programming language. This means a program called an *interpreter* translates the contents of a PHP file into machine code as the file is being executed. Other scripted languages include Python and JavaScript. Scripted languages are different from *compiled* programming languages like C, C++, and Swift, in which the translation happens in a separate step prior to execution. During this extra step, all program files are compiled and optimized into one or more executable files.

The interpreter that translates PHP scripts into executable code is usually called the *PHP engine*. When you click Replit's Run button to run the *main.php* file, Replit invokes the PHP engine, which then reads the contents of the file and interprets and executes the lines of code inside it. For simple PHP scripts (such as our Replit *main.php*) that consist of just one or more statements meant to be executed in order, this is a straightforward process. Almost all programs involve more complex decision logic, however, performing tests so the code can respond dynamically to events and determine which statements to execute and in which order. The way the PHP engine decides what to do next is referred to as the *flow of control*. We'll explore this concept further when we discuss conditionals and loops in [Chapters 4](#) and [6](#).

Hitting Replit's Run button isn't the only way to run a PHP script. You can also invoke the PHP engine from the command line by using the `php` command followed by the name of the script you want to execute. To try it out, switch over from the Console tab to the Shell tab in the right-hand column of your Replit project to bring up an interactive command line terminal. Then enter the following after the `$` prompt:

```
$ php main.php  
Hello, world!
```

The `php main.php` command instructs the PHP engine to execute the *main.php* script. As before, this outputs the `Hello, world!` message. You can use this same technique to execute PHP files from the command line on your local machine, where you won't necessarily have Replit's convenient Run button.

Creating a Web Server Project

PHP is primarily used for developing web applications, so let's now try creating a basic web-based PHP project by using Replit's PHP Web Server template. Go back to your Replit account home page and create a new project, this time choosing the **PHP Web Server** template after entering **PHP** into the template search box. Your new project should look like [Figure 1-2](#).



Figure 1-2: The new PHP Web Server template project screen

The only file shown in the left column is a boilerplate *index.php* file. An *index* file such as this has special significance: it represents the default file served up when you visit the home page of a website. (We'll discuss how this works in more detail when we explore web programming in [Part III](#).) The file's contents are shown in the middle column. In the right column are the Console and Shell tabs, and this is also where a Webview tab will appear when we run the web server to show the rendered web page.

The *index.php* file should contain the code shown in [Listing 1-2](#).

```
<html>
  <head>
    <title>PHP Test</title>
  </head>
  <body>
    ❶ <?php echo '<p>Hello World</p>'; ?>

  </html>
```

Listing 1-2: The web server script in index.php

The bulk of this file isn't PHP code but rather the HyperText Markup Language (HTML) needed to create a generic web page, as indicated by the opening `<html>` and closing `</html>` tags at the beginning and end of the file, respectively. As we'll discuss further in "Template Text vs. PHP Code" on [page 13](#), many PHP scripts mix dynamic PHP code meant to be interpreted and executed on the fly with static text in a language like HTML. In this case, the only PHP code is an `echo` statement to display the text `Hello World` ❶. The text is wrapped in HTML `<p>` tags, meaning it will be rendered on the resulting web page as a body paragraph, and the entire `echo` statement is enclosed in PHP tags (the opening `<?php` tag and the closing `?>` tag) to indicate that it's actual PHP code, unlike the surrounding HTML.

Running the Web Server Project

Click the **Run** button, and Replit will launch a web server hosting the *index.php* file and running the PHP engine needed to interpret the PHP code in the file. This time, instead of seeing text appear in a Console tab, you

should see `Hello World` displayed as a basic web page in the Webview tab (see [Figure 1-3](#)).

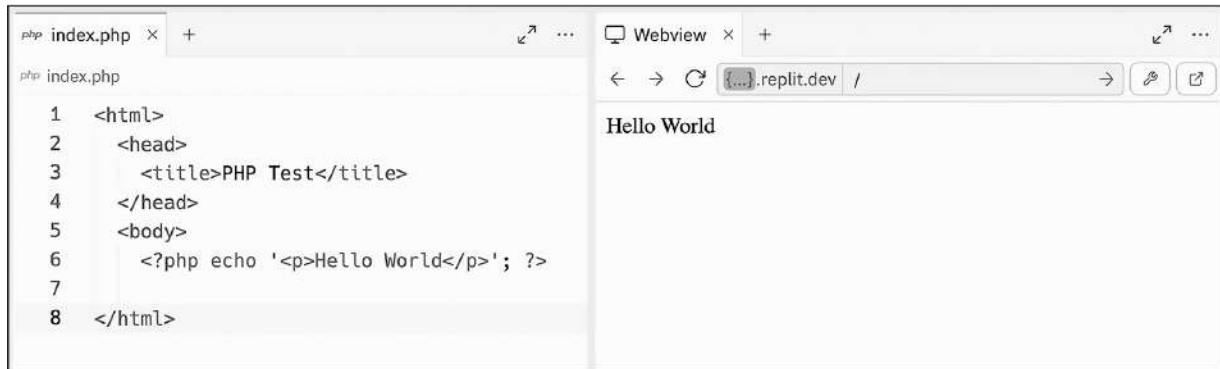


Figure 1-3: Viewing the `index.php` script output in the Replit Webview panel

When running the web server, Replit publishes temporary pages to its *replit.dev* domain. This means it provides publicly served web pages you can view and interact with in a separate web browser tab rather than just through the Replit site itself. To try this, click the green `{...}.replit.dev` URL address bar in the Webview panel. Then copy the URL shown in the pop-up window and paste it into a new tab in your web browser. You should see the same `Hello World` message rendered as its own web page, separate from the Replit interface. Congratulations, you’ve just published your first PHP website!

NOTE

If you choose to use Replit to follow along with this book, you’ll have to do extra configuration to work on some of the more sophisticated projects in later chapters. See [Appendix C](#) for details.

A Local PHP Installation

Online editors like Replit can be fantastic, but they may be slow and restricted on free plans, and they require a reliable, fast internet connection. Many developers instead prefer to work locally on their own machine. To do this, the first step is to install PHP on your computer. If you haven’t already done so, follow the guidelines in [Appendix A](#) to install the latest version of PHP for your operating system.

Once PHP is installed, you'll need an *integrated development environment (IDE)* where you can write your code. An IDE is a powerful text editor that includes useful programming tools like a terminal, sophisticated search-and-replace functions, code spelling correction, and even automatic code generation for common tasks.

In this section, we'll focus on local PHP development with PhpStorm, a popular IDE from JetBrains. Anyone can use it free for 30 days, and from there many people (such as students, teachers, and those in coding bootcamps, user groups, and open source projects) can get a free license. Visit <https://www.jetbrains.com/phpstorm/> to download PhpStorm and follow the installation instructions.

NOTE

If you don't want to use PhpStorm, other free IDEs offer plug-ins to assist your PHP coding, including Visual Studio Code, Eclipse, and Apache NetBeans.

Creating “Hello, world!” with PhpStorm

Let's create a “Hello, world!” project with PhpStorm, similar to the default script that comes with Replit's PHP CLI template. Open the PhpStorm IDE, click **New Project**, and choose **PHP Empty Project** from the list of possible templates. Select a location for the project and change the *untitled* default name to your desired project name. Make sure to include a forward slash before your project's name in the location path, as in */program1*. Then click **Create**.

PhpStorm will set up a new folder with your chosen project name in the desired location. All the files for the project will be contained in this folder; more complex projects might also have subfolders to organize data, program files, configuration files, and so on. With the folder created, PhpStorm will load into the project-editing view shown in [Figure 1-4](#).

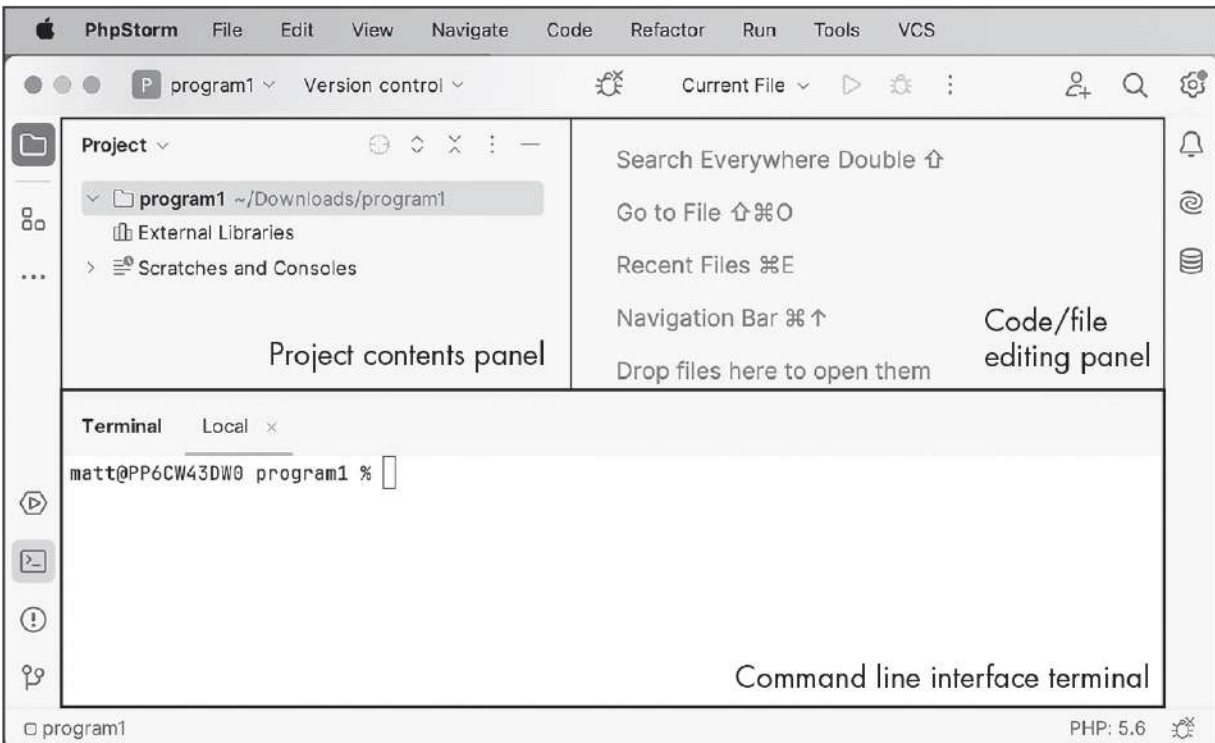


Figure 1-4: The three main PhpStorm panels

The top-left panel in PhpStorm shows the project folder and its contents. The top-right panel is where you edit your code and data files. Click the Terminal (`>_`) icon in the left-hand column of the application window to open a command line terminal at the bottom of the application window, where you can enter commands and see the text output of your programs. This terminal automatically opens at the project folder location.

We're ready to add a basic "Hello, world!" script to the project. Select your project folder in the top-left panel of the application window and then choose **File ► New ► PHP File** from the top menu. Enter **hello** for the filename (PhpStorm will add the `.php` file extension for you) and click **OK**. You should see this new `hello.php` file appear in the project contents panel, and the file should be opened for editing in the code-editing panel, already containing the opening PHP tag (`<?php`) needed to designate the file's contents as PHP code. Now edit the file so it matches the code in [Listing 1-3](#).

```
<?php
print "Hello, world!\n";
```

Listing 1-3: Our “Hello, world!” program in hello.php

As in our Replit command line program, this code simply prints out the text `Hello, world!` followed by a newline character (`\n`). Notice that this time the statement uses `print` rather than `echo` to display the text. The two are largely interchangeable; see the following “`print` or `echo`?” box for more information.

PRINT OR ECHO?

In PHP, you can use either `print` or `echo` to display text. The two have subtle differences, but they aren’t relevant to beginning PHP programming, so for our purposes we can consider `print` and `echo` statements to be essentially equivalent. Which you use comes down to personal preference. Personally, I think the word *print* is a closer match to the intent behind such program statements, so I’ll use `print` throughout this book. On the other hand, older PHP programs (or new code written by older PHP programmers) tend to use `echo`, since `print` was a later addition to the language.

In many programming languages, the equivalent of `print` or `echo` is considered a *function*, and therefore the text to be output must be provided inside a set of parentheses after the function name. In PHP, however, `print` and `echo` aren’t functions but rather *language constructs*. The technical distinction isn’t important, except to say that you don’t need to enclose the text being output by a `print` or `echo` statement in parentheses (although you can if you want).

To run your script, open the Terminal panel (if you haven’t already done so) and enter the following at the command line:

```
% php hello.php
Hello, world!
```

You should see the `Hello, world!` message appear on the next line in the terminal.

A second way to run your script in PhpStorm is to click the green Run button (next to the green “bug” button) at the top right of the application

window. This should execute the file currently being edited. If clicking the button opens a drop-down menu offering a choice of PHP and JS (JavaScript) ways to run the script, choose the PHP option.

If you execute the script this way, a Run panel should open at the bottom of the screen showing the PHP engine you're using and the location of the script being executed. This information is useful if you have multiple versions of the PHP engine on a single computer so that you can test scripts for compatibility with the different engines. Below this should be the output of running the program, followed by an exit code of 0 indicating the program successfully completed execution.

Running a PHP Web Server Locally

When you install PHP, it comes with a built-in web server for testing web development projects locally on your system. You can see information about this web server (and verify that it's working) by using the `phpinfo()` function. This function generates a long string of HTML text reporting details about the current PHP installation. Running a script that calls this function is a useful first step when testing any PHP system for web development, whether on your local machine computer or on a hosted web server.

Using PhpStorm (or another IDE of your choice), create a new project in a folder named *web_project_1*. Then create a new file for the project called *index.php*. As mentioned, the name *index* indicates this will be the default file that the web server hosting the project will return. Edit the file to match the contents of [Listing 1-4](#).

```
<?php
phpinfo();
```

Listing 1-4: Our info web application in index.php

After the obligatory opening PHP tag, you use the statement `print phpinfo();` to display the report that results from calling the `phpinfo()` function. You can view this report as a nicely formatted web page by executing the script in a web browser. In PhpStorm, choose **View ► Open**

in **Browser ▶ Built-in Preview**, or click the PhpStorm icon when your mouse is in the file-editing panel (see [Figure 1-5](#)).

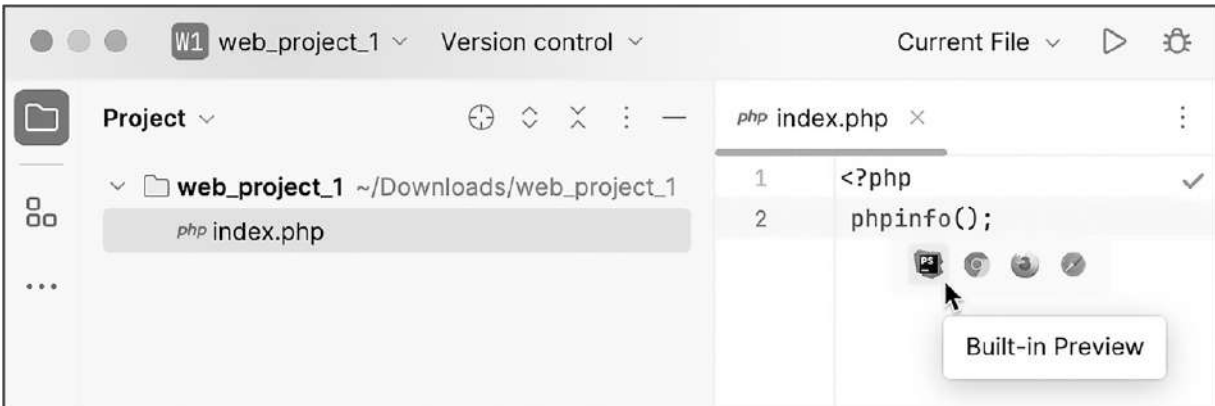


Figure 1-5: Using the PhpStorm web preview

Launching the built-in preview should run the PHP web server and display the results of the *index.php* script in a sample browser window within PhpStorm, as shown in [Figure 1-6](#).

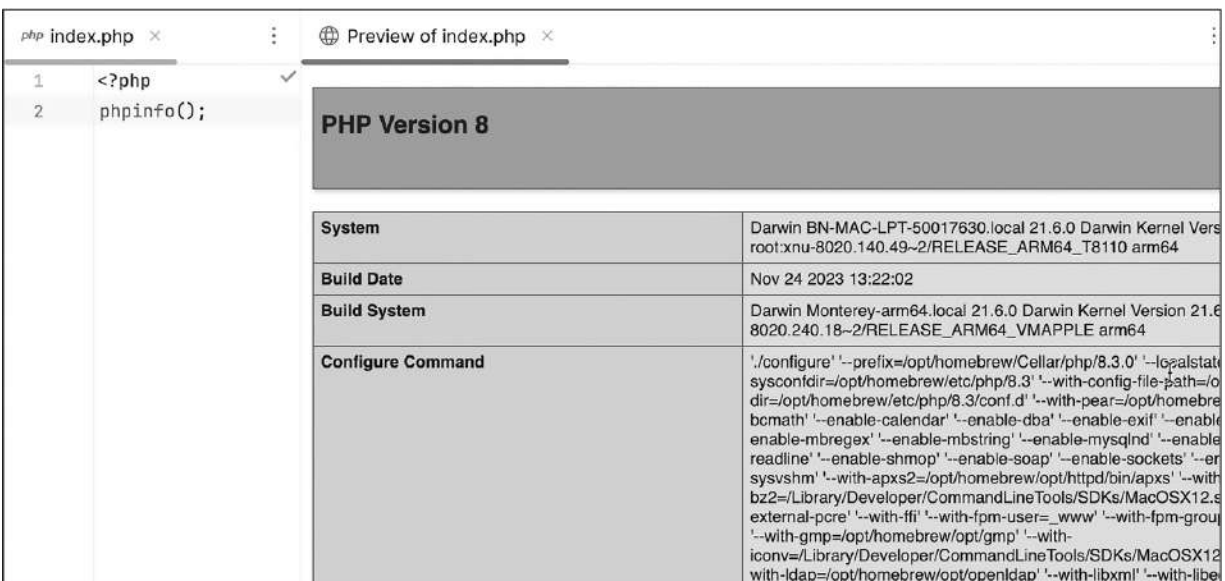


Figure 1-6: The output of the *phpinfo()* function in the PhpStorm preview

You may wish to scroll through this web page to learn more about your system's PHP setup. You'll find the version of the PHP engine, the location

of the *php.ini* configuration file, information about what database extensions (if any) are enabled, the names of key contributors to the PHP language, and more.

You can also view the result of your *index.php* script in a real web browser like Google Chrome or Mozilla Firefox rather than within PhpStorm. (If you're using a different IDE, this may be your only option.) First, open the IDE's terminal and enter the following command:

```
% php -S localhost:8000
```

This tells PHP to launch its built-in web server and make the current project available at *localhost:8000*. Here, *localhost* refers to your local computer system, and *8000* sets the port number. Each application that needs to send and receive messages over the internet needs a unique port number; you can think of these ports as different mailboxes at the same location. Web servers for testing purposes usually use port number 8000 or 8080, whereas production (live) web servers usually run at port 80. Personally, I always use 8000 when developing locally.

With the web server running, open a web browser and enter **localhost:8000** in the address bar. You should see your PHP script's output as before. When you're done, go back to the terminal and press CTRL-C to terminate the web server.

Note that to view a script that isn't named *index.php* in the browser, you'll have to append the script's filename to the end of the URL in the browser address bar. For example, to view the *hello.php* script from [Listing 1-3](#) as a web page, you would first run the PHP web server with the `php -S localhost:8000` command, then navigate to *localhost:8000/hello.php* in your browser.

TROUBLESHOOTING: STARTING THE WEB SERVER

If you try to start your web server on port 8000 and get an error stating that the port is already in use, one solution is to try an alternative port number, such as 8001, 8080, or 8888. However, you also may want to find out which application is already running at port 8000, since your computer might already be running a web server in the background that you don't know about.

Although running several web servers on your computer is perfectly possible, you'll almost always be testing only one web application project at a time, so if several servers are running, you could end up testing the server responses to a different application from the one you're working on. Personally, if I get a message that I can't start a web server on port 8000 because it's already in use, I track down and kill that process (or just reboot the computer) to ensure that I can test the project I'm working on with just one server running on port 8000.

Template Text vs. PHP Code

PHP is a popular language for web development because of how easy it is for PHP scripts to output HTML (or CSS or JavaScript) for display in a web browser. Some parts of the output are typically unchanging *template text*, while other parts are dynamically generated through the execution of PHP program statements. This combination of static template text and dynamic, code-generated text is the backbone of just about any interactive website.

As an example, think about viewing a shopping cart at an online retail site. The web designer doesn't need to write a separate script for displaying every possible configuration of items in a shopping cart. They just have to write one script that mixes the generic template text for the elements that would appear in any shopping cart (the hidden HTML head elements, a navigation bar, the company logo, and the like) with the PHP code needed to dynamically fill in the name, price, quantity, and other details of each specific cart item.

The ability to mix template text with PHP code is why we've needed the opening `<?php` tag in our scripts so far, and why we sometimes need the closing `?>` tag as well. Anything outside these tags is treated as template text and will be output verbatim; anything inside these tags is interpreted as PHP code and executed accordingly. If the script consists entirely of PHP code, as in [Listings 1-1](#), [1-3](#), and [1-4](#), then only the opening `<?php` is needed; if the PHP code is followed by template text, as in [Listing 1-2](#), the closing `?>` tag is needed as well.

To clarify the difference between template text and PHP code, let's write an example script that combines the two. Create a new project (either online with Replit or locally with PhpStorm), and inside this project create a new file called *hello2.php*. Edit the file to match the contents of [Listing 1-5](#) exactly.

```
I am template text, not PHP code.  
print "Hello, world!\n";  
I am more template text.
```

Listing 1-5: The hello2.php script, featuring template text with no PHP code-block tags

The first and third lines of this script are template text and are meant to be output directly when the script is executed. The middle line is PHP code to output the phrase `Hello, world!` followed by a line break. Or is it? Try running this script by entering `php hello2.php` in a command line terminal. Here's the result:

```
% php hello2.php  
I am template text, not PHP code.  
print "Hello, world!\n";  
I am more template text.%
```

The output reproduces all three lines of text verbatim, just as they appear in the file. In particular, the middle line includes PHP code elements like the `print` keyword, quotation marks, and semicolon that weren't meant to be seen. The problem is that we haven't included any opening or closing tags to designate the middle line as PHP code, so the entire script has been interpreted as template text and output directly.

Notice also that the last line of output ends with a new terminal prompt (in this case, a percent symbol). This is because spaces, tabs, and newline characters are copied exactly as they're written when they appear outside the PHP script tags as template text. The terminal then picks up right away with a new prompt wherever the output leaves off, without adding an extra line break. Had we included a blank line at the end of the *hello2.php* script, the new terminal prompt would appear on its own line.

Let's update our script to fix both of these problems. [Listing 1-6](#) shows a revised version of *hello2.php*, with the changes bolded.

```
I am template text, not PHP code.  
<?php  
print "Hello, world!\n";
```

```
?>
```

```
I am more template text.
```

Listing 1-6: Fixing hello2.php to distinguish template text from PHP code

We've added an opening `<?php` tag before the `print` statement and a closing `?>` tag after it. This tells the PHP interpreter that what falls between the tags should be interpreted and executed as PHP code. We've also added a blank line to the end of the script.

If you rerun this script, the PHP engine should now find the starting and ending PHP program tags wrapped around the `print "Hello, world!\n";` statement, and so, as well as outputting template text outside those tags, it will execute that line of code to print out `Hello, world!` and a newline character. Here's the result of executing the script again:

```
% php hello2.php
I am template text, not PHP code.
Hello, world!
I am more template text.
%
```

This time, notice that the first and last lines of the script have been output verbatim as template text, while the middle line contains only the `Hello, world!` message, indicating it has successfully been interpreted as PHP code. The new terminal prompt now also appears on its own line, since we included a blank line at the end of the script in [Listing 1-6](#).

Comments

Comments are a useful feature of any programming language. They're a way to tell the PHP engine to ignore some text written within a PHP code block, so the text will neither be output nor be interpreted as code that should be executed.

Comments can play several roles in a computer program. First, they're a way to embed human-readable notes in the code, such as an explanation of how something works, why you've done something the way you have, or a reminder about something you still need to do. Second, turning one or

more lines of code into a comment is a great way to temporarily disable that code while debugging or trying an alternative way of doing something, without having to delete the code altogether. Finally, comments can also contain special content for preprocessing tools such as documentation generators or code-testing utilities.

As with most languages, PHP provides several ways to define comments. A single-line comment begins with two forward slashes (//) and looks like this:

```
// I am a comment and will be ignored.
```

Everything on the line after the two slashes is treated as a comment and ignored when the code is executed. This means you can place a comment after a program statement on the same line, and the program statement itself will still be executed, like so:

```
print 2 + 2; // Should print 4
```

Here `print 2 + 2` will be executed, resulting in an output of 4, but the PHP engine will ignore the `// Should print 4` comment at the end of the line.

A comment that starts with `/*` and ends with `*/` can span multiple lines. [Listing 1-7](#) illustrates this multiline comment syntax.

```
/*  
I  
am  
a multiline  
comment.  
*/
```

Listing 1-7: A multiline comment example

This style of comment is especially useful if you have a longer block of code that you want to temporarily disable, or *comment out*.

NOTE

You may also meet the older shell-style single-line comment, which begins with the # character rather than two slashes, if you work with or have to maintain legacy code written many years ago. While these shell-style comments are still valid in PHP programs, the // syntax is the preferred style of single-line comment in modern PHP programming.

Variables

A distinguishing feature of computer programs is that they are *dynamic*, meaning their behavior can change each time they're executed based on different data and events. Fundamental to this is the use of *variables*, named values or references to data within code. Variables allow you to store values and refer to them with a meaningful name (an *identifier*).

They're called variables because the value they refer to can change each time a program is executed. For example, a variable might refer to the current date or time, and a program might have logic to do something special based on the value of this variable. Perhaps it will display a greeting on the user's birthday or trigger an alarm each day at 6 AM. Another variable might represent the size of a logfile, and the program might automatically back up the contents of this file and start a new one whenever the size exceeds a certain threshold.

The values of variables don't just change from one run of a program to another; they can also change *during* the course of a program's execution. For example, a variable representing the total value of an online shopping cart will start at 0 and then update as items are added or removed. A variable holding the number of simultaneous users logged into a system would also change as a program runs. At busy times, if the value is very high, more memory or disk space may need to be added to the system.

Creating Variables

You create a PHP variable by giving it a name and assigning it a value. Here, for example, we create a variable called `$age` and assign it a value of 21:

```
$age = 21;
```

PHP variable names must begin with a dollar sign (\$), something that distinguishes PHP code from almost all other programming languages. Assigning a value to a variable hinges around an equal sign (=), known in this context as the *assignment operator*. The variable name goes on the left of the equal sign and its value on the right. Since setting a variable's value is a type of statement, the whole thing ends with a semicolon.

The code to the right of the assignment operator is an expression. An *expression* is something that yields a single value or can be evaluated into a single value. The simplest expression is simply a literal value, like the number 21 in this example. A *literal* is a value that is expressed as itself. Examples are 21 (the whole number twenty-one), 3.5 (the floating-point number three-point-five), true (the truth value true), and "Matt Smith" (the text string Matt Smith).

Other expressions are more complex. They might involve mathematical calculations, incorporate other variables, or even, as you'll see in [Chapter 5](#), feature a call to a function. In these cases, the expression must be *evaluated*, meaning its resulting value is determined, before that value is assigned to the variable. [Listing 1-8](#) illustrates some examples of assignment statements.

```
<?php
$username = "matt";           // A string literal
$total = 3 + 5;               // A calculated expression
$numSlices = $numPizzas * 8;  // A calculation with another
variable
$timestamp = time();          // A function that returns a
value
```

Listing 1-8: Examples of assigning the value of an expression to a variable

We first assign the `$username` variable the value "matt", a string literal. Variables can hold values representing many types of data, as we'll discuss in detail in [Chapter 2](#). We next assign the `$total` variable the value of the calculation `3 + 5`. This variable will therefore hold the number 8. For the value of the `$numSlices` variable, we multiply the value of another variable, `$numPizzas`, by 8 (the `*` symbol denotes multiplication in PHP).

Finally, we set the `$timestamp` variable to the value that results from calling the `time()` function. (You'll learn more about how to get values out of functions in [Chapter 5](#).)

If you try to execute the code in [Listing 1-8](#), it won't quite work. PHP will produce a warning message like the following:

```
PHP Warning: Undefined variable $numPizzas in main.php on line 4
```

The problem here is that the `$numPizzas` variable is *undefined*, meaning it hasn't been given a value. It's important to always assign a value to a variable before you first make use of it.

Using Variables

Once you've created a variable, you can use its name anywhere you need to reference that variable's value. For example, [Listing 1-9](#) shows a program illustrating how to use variables to calculate and print the total number of pizza slices, given a number of pizzas. Create a *pizza.php* file containing the contents of this listing.

```
<?php
$numPizzas = 1;
$numSlices = $numPizzas * 8;
print $numSlices;
print "\n";

$numPizzas = 3;
$numSlices = $numPizzas * 8;
print $numSlices;
print "\n";
```

Listing 1-9: Working with variables in pizza.php

First, we assign the numeric value 1 to the `$numPizzas` variable. Then we multiply the value of `$numPizzas` by 8 to assign the value of the `$numSlices` variable. Remember, variables must be written starting with a dollar sign; you'll get used to this very quickly as you write more PHP. We

next use a `print` statement to show the value inside `$numSlices`, followed by another `print` statement with the `\n` newline character to create a line break.

As mentioned, the value of a variable can change while a program is running, so we next update the value inside the `$numPizzas` variable from 1 to 3. Then we update the value of `$numSlices` by again multiplying `$numPizzas` by 8, and we print out the new value. Here's the result of executing this program at the command line:

```
% php pizza.php
8
24
```

Notice that the value of `$numSlices` changes from 8 to 24 over the course of the program. These values, in turn, are calculated based on the changing values of `$numPizzas`. Try changing the number inside the variable `$numPizzas` yourself to get a different number of slices.

Naming Variables

A few rules and conventions exist for naming variables in PHP. First and foremost, as we've already discussed, all variable names must start with a dollar sign. If you forget the dollar sign when referencing a variable, PHP will usually report an undefined constant fatal error, and the program will crash. (We'll discuss constants in the next section.)

The next character in a variable name after the dollar sign must be a letter of the alphabet (or in certain cases, an underscore). By convention, this letter should be lowercase. While a capital letter is technically permitted, an initial capital is usually reserved for class names rather than variable names. (You'll begin learning about classes and object-oriented programming in [Part V](#).) The remaining symbols in a variable name can be letters, numerals, or underscores.

Single-word variable names should typically be all lowercase, as in `$name` or `$total`. For variable names with multiple words, we follow two common conventions. One is *snake case*: everything is lowercase, and the words are separated by underscores, as in `$game_lives_remaining` or

`$customer_number`. The other is *lower camel case*: the first word is all lowercase, and subsequent words start with a capital letter, as in `$gameLivesRemaining` **OR** `$customerNumber`.

The most important rules of thumb are to be consistent in whatever naming convention you choose, to follow PHP's style recommendations when possible, and above all, to choose names that clearly communicate what the variable represents. A name like `$customerNumber` is clearer than something abbreviated like `$custNo` and certainly clearer than a meaningless variable name like `$x` or `$variable`.

Keep in mind that PHP variable names are case sensitive, so identifiers like `$username` and `$userName` will be treated as separate variables. If you get the capitalization wrong when you reference a variable (or otherwise type the variable name incorrectly), PHP won't know what you mean.

[Listing 1-10](#) shows an example.

```
<?php
$username = "matt";
print $userName;
```

Listing 1-10: Misspelling a variable name

We assign the value "matt" to the `$username` variable, then attempt to print the value of this variable. Because of the incorrect capitalization in `$userName`, however, executing this script at the command line will result in a warning message like the following:

```
PHP Warning:  Undefined variable $userName in main.php on li
ne 3
```

Because PHP variables are case sensitive, the PHP engine interprets `$userName` as a reference to a completely different variable that hasn't previously been given a value. In PHP's eyes, this is therefore the same problem as trying to use the `$numPizzas` variable without first defining it in [Listing 1-8](#).

Keep in mind that while some aspects of PHP, like variable names, are case sensitive, other aspects of the language are case insensitive, meaning

capitalization doesn't matter. These include keywords like `if`, `for`, `switch`, and `print`; data types such as `int` and `string`; values such as `true` and `false`; and function and method names. That said, it's common practice to use lowercase for language keywords and data types, and lower camel case for function and method names. The exercises at the end of the chapter suggest a coding style guide that can help you learn more about these conventions.

VARIABLE NAMES TO AVOID

You must avoid certain names when defining your own PHP variables, since they correspond to variables that are already built into the language. These variables serve special purposes, such as retrieving data from HyperText Transfer Protocol (HTTP) requests or user sessions in the browser. You'll learn about some of these built-in variables in later chapters. For now, it's important to know that the following variable names are off-limits: `$GLOBALS`, `$_SERVER`, `$_GET`, `$_POST`, `$_FILES`, `$_REQUEST`, `$_SESSION`, `$_ENV`, `$_COOKIE`, `$php_errormsg`, `$http_response_header`, `$argc`, and `$argv`. Since most of these don't follow the convention of starting a variable name with a lowercase letter, avoiding them should be easy.

The name `$this` is also off-limits for a user-defined variable, since `$this` carries special meaning in object-oriented programming. We'll see a lot of `$this` beginning in [Part V](#) of this book.

Constants

Some values never change, such as the value of π (always 3.14) or the neutral value on the pH scale (always 7). When referencing such values in code, it's best to use a *constant* rather than a variable. Unlike a variable, once a constant is defined, its value can't be updated. By convention, the names of constants are written in *upper snake case*, with all capital letters and underscores between words, as in `MAX_PROJECTS` or `NEUTRAL_PH`. Unlike variables, constants don't start with a dollar sign.

Some constants are built into the PHP language. [Table 1-1](#) lists a few examples.

Table 1-1: Examples of Built-in PHP Constants

Constant	Description	Value
<code>M_PI</code>	π , the ratio of a circle's circumference to its diameter	3.1415926535898
<code>M_E</code>	e, Euler's number	2.718281828459
<code>PHP_INT_MAX</code>	The largest integer that can be supported by the installed PHP system	Usually 9223372036854775807 for 64-bit systems

You can also create your own custom constants by using the `define()` function. The script in [Listing 1-11](#) shows an example.

```
<?php
define("MAX_PROJECTS", 99);
print "The maximum number of projects is: ";
print MAX_PROJECTS;
print "\n";
```

Listing 1-11: Defining and printing a constant

We call the `define()` function to create a constant named `MAX_PROJECTS` with the value 99. This makes the constant available for use anywhere in the code. We then print out the value of the constant as part of a message. (Multiple `print` statements in a row will output to the same line if they don't contain any line breaks; the final `print "\n"` adds a newline to ensure that the next output—in this case, the next command line prompt—will appear on its own line.) The output of running this script should be as follows:

```
The maximum number of projects is: 99
```

Notice that the constant's name must be enclosed in quotation marks when we create it with the `define()` function. Without these quotation marks, creating the constant wouldn't work. For example, if you were to write `define(MAX_PROJECTS, 99)` without quotes, PHP would interpret `MAX_PROJECTS` as a reference to a previously defined constant and would report

an error. Once you've defined a constant, however, you don't need the quotation marks when you reference it.

NOTE

The PHP documentation rather confusingly refers to numeric and Boolean literals as constants, although it also uses the term string literals. When reading that documentation, it's therefore useful to distinguish between simple constants (literal values themselves) and named constants (such as those created with the `define()` function).

Operators and Operands

Operators are the programming language symbols we use to manipulate data, such as the plus sign (+) for numeric addition and the equal sign (=) for assigning a value to a variable. *Operands* are the values (literals, variables, or complex expressions) that an operator works on. For example, the numeric addition operator expects two operands: one number to the left of the plus sign and another number to the right, as in `2 + 2` or `$price + $salesTax`.

PHP has different operators for working with different types of data. In this section, we'll focus primarily on operators for numeric values. In later chapters, we'll consider other operators as well, such as operators for comparing values ([Chapter 2](#)) and for manipulating logical true/false values ([Chapter 4](#)).

Arithmetic Operators

PHP has *arithmetic operators* for basic mathematical calculations, such as addition (+), subtraction (-), multiplication (*), and division (/). These are all *binary operators*, meaning they require two operands. PHP also has the `**` operator for raising a number to a given power and the modulo operator (`%`), which divides one number by another and reports the remainder. [Table 1-2](#) summarizes these operators.

Table 1-2: The Six Binary (Two-Operand) Arithmetic Operators

Operator	Description	Example expression	Expression value
Addition	Returns the sum of the two operands	<code>3 + 1</code>	4
Subtraction	Returns the difference of the two operands	<code>10 - 2</code>	8
Multiplication	Returns the product of the two operands	<code>2 * 3</code>	6
Division	Returns the quotient of the two operands	<code>8 / 2</code>	4
Modulo	Returns the remainder of the first operand divided by the second	<code>8 % 3</code>	2
Exponentiation	Returns the first operand raised to the power of the second	<code>2 ** 3</code>	8

As in mathematics, these operators have an *order of precedence* that controls the way an expression is evaluated when it contains multiple operations. The arithmetic operators for multiplication, division, and modulo have higher precedence, while the operators for addition and subtraction have lower precedence. Therefore, in the arithmetic expression `1 + 2 * 3`, the `2 * 3` component will first be evaluated to 6, and then `1 + 6` will be evaluated, so the whole expression evaluates to 7. You can use parentheses to force precedence; for example, the expression `(1 + 2) * 3` evaluates to 9 rather than 7 since the parentheses force the addition component to be evaluated to 3 before the multiplication component.

NOTE

See the PHP documentation at

<https://www.php.net/manual/en/language.operators.precedence.php> for a complete list of the order of precedence for all PHP's operators.

Combined Arithmetic Assignment Operators

We've already discussed how the basic `=` assignment operator takes a value and assigns it to a variable. Other assignment operators, such as `+=` and `-=`, combine assignment with arithmetic. These combined operators exist because it's common to want to take the value in a variable, perform a calculation with it, and store the result back in the same variable, replacing the previous value.

To illustrate, say we have a `$total` variable for keeping track of the total cost of the items in an online shopping cart. Every time the user adds or removes an item from the cart, we'd want to change `$total` by the cost of that item. We could do this with the regular `=` operator by writing something like `$total = $total + 25` or `$total = $total - 15`. With combined arithmetic assignment operators, however, we can accomplish the same task by using more concise syntax: `$total += 25` or `$total -= 15`. These statements instruct the PHP engine to update the value of the `$total` variable by adding 25 or subtracting 15 from its previous value.

Similar arithmetic assignment operators are used for the other arithmetic operations. For instance, the `*=` operator multiplies a variable by a given value. The others, `/=`, `%=`, and `**=` for division, modulo, and exponentiation, respectively, are used much less often.

Increment and Decrement Operators

Numbers can also be *incremented* (increased by 1) or *decremented* (decreased by 1) using special operators: a double plus sign (`++`) for incrementing and a double minus sign (`--`) for decrementing. As with the combined arithmetic assignment operators, `++` and `--` provide a more concise syntax for a common programming task. For example, to add 1 to the existing value of the `$age` variable, we could use the basic assignment operator and write `$age = $age + 1`, or use an arithmetic assignment operator and write `$age += 1`. With the increment operator, however, we simply write `$age++`. Likewise, `$age--` subtracts 1 from the value of `$age`. These are examples of *unary operators*, meaning they expect only a single operand.

When you want to use the result of an increment or decrement operation in an expression, the placement of the operator before or after the value it operates on matters. Say `$age` contains an integer value of 21. The expression `$age++` yields the current value of `$age` (21) and then increments the variable. On the other end, `++$age` will first apply the increment and then yield the variable's newly incremented value (22). [Listing 1-12](#) illustrates this distinction.

```
<?php
$person1Age = 21;
print "Person 1 age = ";
❶ print ++$person1Age;
print "\nPerson 1 age (after increment) = ";
print $person1Age;

$person2Age = 21;
print "\nPerson 2 age = ";
❷ print $person2Age++;
print "\nPerson 2 age (after increment) = ";
print $person2Age;
```

Listing 1-12: Demonstrating the difference between pre- and post-increment operators

Here we set both `$person1Age` and `$person2Age` to 21, then use `++` to increment each person's age. However, since we use `++$person1Age` in one case ❶ and `$person2Age++` in the other ❷, the printed results are different, as the script's output shows:

```
Person 1 age = 22
Person 1 age (after increment) = 22
❶ Person 2 age = 21
Person 2 age (after increment) = 22
```

Both times we print `$person1Age`, we see 22 in the output. This is because placing the `++` operator before the variable ensures that its value is incremented from 21 to 22 before it's first printed. By contrast, placing the `++` operator after `$person2Age` allows us to first see its original value of 21

❶, since it forces the increment operation to take place after the variable's value is used in the expression.

To avoid confusion about whether the pre- and post-increment value will be used, many programmers opt to use two separate lines of code: one with a statement incrementing a variable's value, and another with a statement to make use of that new value, as in [Listing 1-13](#).

```
$person1Age = 21;  
$person1Age++;  
print $person1Age;
```

Listing 1-13: Separating the increment operation from the `print` statement

Here we use one statement to only increment `$person1Age` and another statement to only print the variable's value. This unambiguously ensures that the resulting output will be 22.

Summary

We've covered some important PHP programming basics in this chapter. We looked at two ways to create and run PHP programs: using the Replit online environment and using the PhpStorm IDE on your local computer. We covered statements, expressions, variables, constants, and operators, which are all key building blocks of computer programs, and we discussed how to use comments to help make code more readable for humans and to temporarily disable blocks of code when developing and debugging programs. We also took a first look at how to interweave PHP program statements with unchanging template text, a fundamental technique for creating web applications that can dynamically customize the web pages returned to users.

Exercises

1. Visit the PHP website, <https://www.php.net>, and get to know the layout of the language's documentation pages. They will be a great reference when you're programming in PHP.
2. Visit another handy online PHP resource, PHP the Right Way (<https://phptherightway.com>). This website is a collection of best

practices in the PHP programming community. Almost everyone in the community now follows the same code-styling guidelines, which makes PHP code from different programmers easy to read, understand, and contribute to. You can read about these guidelines at https://phptherightway.com/#code_style_guide, where you'll also find links to specific style recommendations from the PHP Framework Interop Group (PHP-FIG), the unofficial group of international PHP professionals driving the language's coding standards.

3. Use comments to disable some of the lines in the following listing so that only `Cat`, `Dog`, and `Helicopter` are printed:

```
<?php
print "Cat\n";
print "Elephant\n";
print "Dog\n";
print "Helicopter\n";
print "Bus\n";
print "Spacecraft\n";
```

4. Write a script that creates a `$name` variable containing your name and uses that variable to print out the message `My name is your name` followed by a newline character.

2

DATA TYPES



In this chapter, we'll explore the data types available in PHP. We'll also consider how to force a value into a specified data type (*type casting*), as well as situations where PHP automatically attempts to convert data types to make the various parts of expressions work together (*type juggling*).

A *data type* is a categorization of a value in a program that specifies how the PHP engine is to interpret that value and therefore which operations can be applied to it. For example, if a value is an integer, the PHP engine knows that operations such as addition and multiplication are permitted and that the outcomes of those operations are themselves integers; meanwhile, the PHP engine knows that the outcome of division on an integer might be another integer or a floating-point (decimal) number.

Understanding which data types are available—and knowing when and how a value's data type can change—is essential as you work with inputs, perform calculations, and output data. If you don't know the type of data you're manipulating or how that data responds to various operations, you might get unexpected results.

PHP Data Types

In [Chapter 1](#), we stored the word `"matt"` in a variable and assigned the number `99` to a constant. These values are of different data types: one is a string, and the other an integer. In all, PHP has 10 built-in data types divided into three categories, as shown in [Figure 2-1](#).

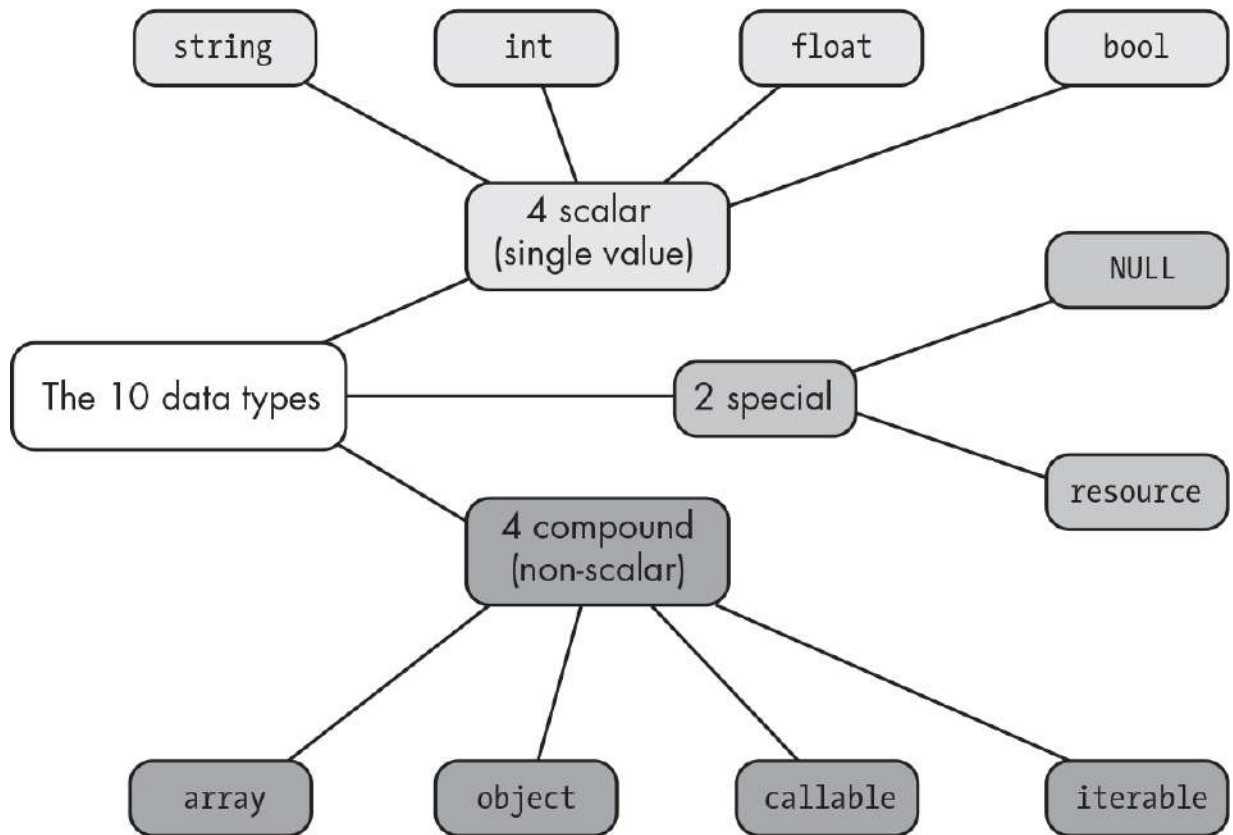


Figure 2-1: PHP data types

For now, we'll mostly focus on the four *scalar* data types, which can hold only one value at a time. We'll also pay some attention to the special `NULL` data type. In later chapters, you'll learn about two of the *compound* data types, `array` (in [Chapters 8](#) and [9](#)) and `object` (in [Part V](#)), which can store and manipulate collections of multiple values. The `resource` special type and the `callable` and `iterable` compound types are used only in complex and special cases and won't be considered in this book.

Scalar Data Types

The four scalar (single-value) data types are `string`, `int`, `float`, and `bool`. The `string` type is for text, the `int` type is for whole numbers (integers), the `float` type is for floating-point (decimal) numbers, and the `bool` type is for Boolean true/false values.

Let's use PHP's *interactive mode* to explore the scalar data types. This mode allows you to enter individual PHP statements at the command line and immediately see the results. We'll use interactive mode in the coming chapters to quickly demonstrate basic concepts and get instant feedback, instead of having to write full PHP scripts. Enter `php -a` at the command line to turn on interactive mode and then enter the following:

```
php > $username = "matt";  
php > print gettype($username);  
string
```

Here we once again assign the value "matt" to the `$username` variable. We then use PHP's built-in `gettype()` function to print out the variable's type. The output confirms that `$username` contains a string.

If you've previously seen or written code in strongly typed languages like Java or C#, you may have noticed that you don't have to specify the data type when assigning a value to a variable. PHP is a *loosely typed* language, meaning that the same variable can store values of different data types at different times and that the PHP engine will automatically infer the data type of an expression.

NOTE

We can also explicitly declare data types in PHP, something we'll do starting in [Chapter 5](#) when we begin writing functions. For now, though, as we work with simple variables, we'll let the interpreter infer data types.

In the case of the `$username` variable, the value "matt" is inferred to be a string. We can similarly assign variables numeric values with or without decimals, and PHP will interpret them as integers or floats as appropriate:

```
php > $age = 21;
php > print gettype($age);
integer
php > $price = 9.99;
php > print gettype($price);
double
```

Here we see that `$age`, with its whole-number value, has been interpreted as an integer, and `$price`, whose value includes a decimal, has been interpreted as ... a *double*? Although the documentation refers to floating-point values as being of the `float` data type, for historical reasons (PHP is an old language!) the function `gettype()` returns `double` when used on floats—a reference to the double-precision format for storing floating-point values. PHP has only one kind of floating-point data type, however, so while other programming languages may have different precision and memory representations for floats, doubles, reals, and so on, all floating-point values in PHP are of the `float` data type (no matter what the `gettype()` function says).

Let's try creating a variable of the `bool` type next. Enter the following:

```
php > $isDutyFree = true;
php > print gettype($isDutyFree);
boolean
php > print $isDutyFree;
1
```

When we use `gettype()` on the `$isDutyFree` variable, we see `boolean` displayed. This is an alias for `bool` in PHP; the two are mostly interchangeable, but to avoid some cases where aliases don't work, always write `bool` in your code (and I'll do so in this book).

More curiously, notice that when we try to print the value of `$isDutyFree`, we see the number `1` rather than `true` in the output. This isn't an error. It's related to the way `bool` values are converted, or *juggled*, into strings. The `print` command expects a string, so whatever we provide after the keyword `print` is automatically converted into a string expression by

the PHP engine. For the `bool` type, `true` is converted to the string `"1"`, and `false` is converted to an empty string (that is, a string with no content, denoted by a set of quotation marks with nothing in between: `""`). We'll discuss conversion to another data type through manual casting and automatic type juggling in more detail later in the chapter.

To see the actual Boolean value of `$isDutyFree`, use the built-in `var_dump()` function instead of `print`. This useful function outputs information about a variable. It's helpful when learning PHP and for debugging purposes to know the value of a variable at a certain point in the execution of your code:

```
php > $isDutyFree = true;
php > var_dump($isDutyFree);
bool(true)
```

The output from `var_dump()` confirms that the data type of `$isDutyFree` is `bool` and that its value is `true`.

The Special NULL Type

PHP has a special data type represented in code by the constant `NULL` or `null` (it's case insensitive). A variable is `NULL` in three situations. In the first, a variable has never been assigned a value, as shown here:

```
php > var_dump($lastName);
Warning: Undefined variable $lastName in php shell code on line 1
NULL
```

When we try `var_dump()` on `$lastName` without giving the variable a value, we first get a warning that `$lastName` is undefined. Then we see that the variable, not having been given a value, evaluates to `NULL`.

Second, a variable is `NULL` if it's been explicitly assigned the constant `NULL` as its value:

```
php > $firstName = NULL;
php > var_dump($firstName);
NULL
```

Here we see an important difference between a variable never having been given a value, as in the previous example, and a variable containing the value `NULL`. In the first case, `var_dump()` produces a warning, whereas in this case we don't get a warning; we just see the variable's value (`NULL`) printed out. Assigning a variable the value `NULL` is fine, just like assigning a variable another value.

Finally, a variable will be `NULL` if it has been *unset*, or cleared of its value, with the built-in `unset()` function:

```
php > $lastName = "Smith"
php > var_dump($lastName);
string(5) "Smith"
php > unset($lastName);
php > var_dump($lastName);
Warning: Undefined variable $lastName in php shell code on line 1
NULL
```

Here we give `$lastName` a value and then use `unset()` to get rid of that value. When we try to use `var_dump()` on `$lastName` after unsetting it, we get the same warning as before and see that it evaluates to `NULL`. Unsetting a variable is the same as never having given it a value in the first place.

When working with variables and data items in more complex programs, you'll sometimes need to design logic to handle encounters with `NULL`. For example, if you're creating a connection to a database but have a problem connecting, the connection variable will be set to `NULL`. In another example, if you expect to be passed a reference to an object (such as the logged-in user) but no such object exists, then a variable will be `NULL`. We'll explore these kinds of situations in [Parts V](#) and [VI](#), when we discuss object-oriented programming and databases.

Functions to Test for a Data Type

PHP has many functions that produce `true` or `false` based on whether the provided variable or expression is of a certain data type. These include `is_string()`, `is_int()`, `is_float()`, `is_bool()`, and `is_null()`. Such functions are useful if you need to confirm that a variable is of a particular type before trying to work with it or, conversely, if you need to check that a variable isn't `NULL`. Here are some examples of these functions in action:

```
php > $gpa = 3.5;
php > var_dump(is_string($gpa));
bool(false)
php > var_dump(is_int($gpa));
bool(false)
php > var_dump(is_float($gpa));
bool(true)
php > $middleName = NULL;
php > var_dump(is_bool($middleName));
bool(false)
php > var_dump(is_null($middleName));
bool(true)
```

Our variable `$gpa` contains a decimal value, so only `is_float()` is `true` for it. Similarly, `$middleName` contains `NULL`, so passing it to `is_null()` yields `true`.

Some of PHP's type-checking functions are `true` for broader categories of data types. For example, the `is_numeric()` function is `true` for variables of type `int` or `float`:

```
php > $gpa = 3.5;
php > $age = 21;
php > var_dump(is_numeric($gpa));
bool(true)
php > var_dump(is_numeric($age));
bool(true)
```

Here we see both the decimal value 3.5 and the whole-number value 21 pass the `is_numeric()` test. The same function is also `true` for strings that contain only numeric characters, but not if non-numeric characters are mixed in:

```
php > $price = "9.99";  
php > var_dump(is_numeric($price));  
bool(true)  
php > $price = "9.99 dollars";  
php > var_dump(is_numeric($price));  
bool(false)
```

When `$price` contains the string "9.99", `is_numeric()` is `true`. When we add the word *dollars* to the end of the string, however, `is_numeric()` becomes `false`.

Type Juggling

In some situations, the PHP engine automatically converts a value from one data type to another. This is known as *type juggling*. Consider this example:

```
php > $answer = "1" + 3;  
php > var_dump($answer);  
int(4)
```

Here we try to add the string "1" and the integer 3. When the PHP engine evaluates this expression and stores the result in `$answer`, it will see the plus-sign operator (+) and assume that numeric addition is meant to take place. The PHP engine will therefore examine the two operands (the values on either side of the plus sign) and try to interpret them as numbers (floats or integers). The 3 is already an integer, but the string "1" will be converted (juggled) into an integer to allow the addition to take place. In the end, we get the integer 4 as an answer.

In six kinds of situations, PHP automatically juggles expressions into different types: numeric contexts, string contexts, comparative contexts,

logic contexts, function contexts, and bitwise (integral and string) contexts. We'll consider some of these contexts next.

Numeric Contexts

When an expression includes an arithmetic operator, PHP will try to juggle the operands to integers or floats. This often happens when one or more of the operands are strings, as in the `"1" + 3` example we just looked at. We'll see an example of this in action when function parameters are coerced into integers in [Chapter 5](#). Boolean values can also be juggled into integers; `true` becomes the integer 1, and `false` becomes the integer 0.

Two important questions arise when juggling in numeric contexts: Will the result become an `int` or a `float`, and what happens when a string includes both numeric and non-numeric characters?

Integer vs. Float

If either operand in an arithmetic expression is a float (or isn't interpretable as an integer), both values will be juggled into floats, and a float operation is performed. Otherwise, both values will be juggled into integers, and an integer operation is performed. For example, when both operands are integers, the result is an integer:

```
php > $answer = (1 + 1);  
php > var_dump($answer);  
int(2)
```

When one operand is an integer, and the other is a numeric string that evaluates to an integer, both operators become integers. The result is also an integer:

```
php > $answer = (1 + "1");  
php > var_dump($answer);  
int(2)
```

Leading and trailing whitespace is ignored when strings are juggled into numbers, so we'll get the same result if we add spaces at either end of the string:

```
php > $answer = (1 + " 1 ");  
php > var_dump($answer);  
int(2)
```

Notice that the extra spaces in the string before and after the 1 character make no difference. The string is still juggled to the integer 1.

When both operands are numbers and one of the operands is a float, both operands become floats. The result is also a float:

```
php > $answer = (1.5 + 1);  
php > var_dump($answer);  
float(2.5)
```

To arrive at this result of 2.5, the integer 1 is juggled to a float behind the scenes, before the addition operation is performed. The same process occurs when one operand is an integer and the other is a numeric string that evaluates to a float. Both operands become floats, and the result is a float:

```
php > $answer = (1 + "9.9");  
php > var_dump($answer);  
float(10.9)
```

In short, if float arithmetic is needed, PHP uses float arithmetic. Otherwise, it uses integer arithmetic.

Numeric, Leading Numeric, and Non-numeric Strings

PHP differentiates between *numeric strings*, whose entire contents evaluate to an integer or a float, and *leading numeric strings*, which begin with numeric characters but also include non-numeric characters such as letters or special symbols. When a leading numeric string is juggled to an integer or float, everything from the first non-numeric character on is dropped. If a string *starts* with a non-numeric character, however, the entire string is considered *non-numeric*, even if it also contains numbers, and it can't be juggled to a number.

NOTE

The exception to this rule about non-numeric characters at the start of a string is the special case of a string beginning with spaces, followed by numeric characters. Such a string will be treated as a numeric string, since leading (or trailing) spaces are ignored.

Let's try numeric addition with a leading numeric string that evaluates to an integer:

```
php > $answer = (1 + "1 dollar");  
Warning: A non-numeric value encountered in php shell code  
on line 1  
php > var_dump($answer);  
int(2)
```

Here "1 dollar" is a leading numeric string. When used in an arithmetic expression, the "1" at the beginning is juggled to an integer, while the " dollar" at the end is ignored. Notice that PHP raises a warning about this but goes ahead with the type conversion anyway. Since both operands can be juggled into integers, the result is an integer.

Addition with a leading numeric string that evaluates to a float works the same way:

```
php > $answer = (1 + "9.99 dollars");  
Warning: A non-numeric value encountered in php shell code  
on line 1  
php > var_dump($answer);  
int(10.99)
```

In this case, "9.99 dollars" is a leading numeric string whose beginning evaluates to the float 9.99. Since both operands can be juggled into floats, the result is a float. Again, PHP raises a warning because of the leading numeric string.

By contrast, if you try to use a non-numeric string in an arithmetic expression, you'll get a `TypeError`, meaning the operation can't be performed with the given data types. This halts the execution of the code. Here's an example:

```
php > $answer = (1 + "April 1");  
Warning: Uncaught TypeError: Unsupported operand types: int  
+ string in  
php shell code:1  
Stack trace:  
#0 {main}  
    thrown in php shell code on line 1
```

Here "April 1" is a non-numeric string because it starts with letters, not numbers. The string can't be evaluated as a number, so it triggers an error. The same error happens if we use an empty string (""):

```
php > $answer = (1 + "");  
Warning: Uncaught TypeError: Unsupported operand types: int  
+ string in  
php shell code:1  
Stack trace:  
#0 {main}  
    thrown in php shell code on line 1
```

An empty string does *not* evaluate to 0 as it does in some other languages. It's a non-numeric string, so it leads to a `TypeError` in arithmetic expressions.

String Contexts

In a few cases, PHP automatically juggles values into strings. First, expressions involving `print` or `echo` statements are juggled to strings, since these commands expect anything that follows to be a string. You've already seen, for example, that values of type `bool` are juggled to the string "1" (for `true`) or "" (for `false`) when they're part of a `print` statement. Similarly, numbers are juggled to their string equivalents.

Second, expressions involving the string concatenation operator (.) are also juggled to strings, as are expressions whose values are being parsed inside strings. We'll discuss these topics in [Chapter 3](#).

Comparative Contexts

Type juggling also occurs when comparing two values of different data types. Comparative expressions evaluate to a Boolean `true` or `false`, which you then typically use for decision-making logic (as we'll discuss in [Chapter 4](#)). The PHP engine knows an expression is comparative when it sees a *comparison operator*, such as `==` for *equal* or `>` for *greater than*. PHP has rules that determine how these expressions are juggled and evaluated, depending on the data types involved.

Identical vs. Equal Values

PHP makes an important distinction between values that are identical and values that are equal. Two expressions are considered *identical* only if (*before* any type juggling) they're of the same data type and contain the same value. By contrast, two expressions are considered *equal* if they contain the same value *after* type juggling.

We use different operators to test for identity and equality. The triple equal sign (`===`) is the *identical operator*, while the double equal sign (`==`) is the *equal operator*. Consider these examples comparing the string `"1"` and the integer `1`:

```
php > var_dump("1" === 1);  
bool(false)  
php > var_dump("1" == 1);  
bool(true)
```

First, we try the comparison with the identical operator. This evaluates to `false`, since the operands are of different data types. Next, we try the comparison with the equal operator. This time it evaluates to `true`, since PHP juggles the string `"1"` into an integer before making the comparison.

PHP also has operators for not-identical (`!==`) and not-equal (`!=`). Consider these comparisons between an integer and a float:

```
php > var_dump(1 !== 1.0);  
bool(true)
```

```
php > var_dump(1 != 1.0);  
bool(false)
```

Using the not-identical operator, the comparison is `true`, since the values are of different data types. Using the not-equal operator, the numbers are first juggled to the same type. This gives them the same value, so the not-equal comparison is `false`.

NOTE

The PHP `<>` operator is equivalent to the `!=` operator; both mean “not equal.” Personally, I always use the `!=` operator, since an exclamation mark (!) by itself means “not” in other contexts too.

Strings vs. Numbers

Since PHP 8.0, when comparing a string with a number, a numeric comparison is made if the string is a numeric string. Otherwise, a string comparison is made. We saw a numeric comparison when we tested the equality of the integer `1` and the numeric string `"1"`. If we try the same comparison with a leading numeric string, the result will be `false`:

```
php > var_dump(1 == "1 dollar");  
bool(false)
```

Though `"1 dollar"` starts with a number, it isn't a fully numeric string. As such, PHP juggles the integer `1` to the string `"1"` and makes a string comparison. The strings aren't equal, so we get `false`.

Two important implications of using only numeric comparisons for fully numeric strings are that any leading or trailing spaces are ignored and that an empty string is *not* considered a numeric string and so is not equal to a numeric `0`, as shown here:

```
php > var_dump(0 == "");  
bool(false)
```

In this comparison, since the empty string isn't numeric, the integer 0 is juggled into the string "0". Then strings "0" and "" are compared and found to be not equal.

Less Than and Greater Than

When working with numbers, using the less-than (<), greater-than (>), less-than-or-equal-to (<=), or greater-than-or-equal-to (>=) operators is straightforward, since it's very clear whether one number is less than, equal to, or greater than another. Here are some examples:

```
php > var_dump(1 < 2);  
bool(true)  
php > var_dump(1 <= 1.01)  
bool(true)  
php > var_dump(2 >= 2);  
bool(true)  
php > var_dump(2 > 2);  
bool(false)
```

You can also use these operators with non-numeric data types, in which case PHP has a variety of rules for evaluating the comparisons. For example, the Boolean `true` is considered greater than `false` and also greater than `NULL`:

```
php > var_dump(true > false);  
bool(true)  
php > var_dump(true > NULL);  
bool(true)
```

Strings are compared with each other one character at a time, with later letters in the alphabet considered greater than earlier letters:

```
php > var_dump("abc" < "acb");  
bool(true)
```

Lowercase letters are considered greater than capital letters, however:

```
php > var_dump("a" > "B");  
bool(true)
```

Strings are typically considered greater than any number, as in these cases:

```
php > var_dump("abc" > 123);  
bool(true)  
php > var_dump("one" > 1000000);  
bool(true)
```

Exceptions to such general rules of thumb exist, however. As we've already discussed, when a fully numeric string is compared to a number, the string is first juggled to a number and then a numeric comparison is made. Here, for example, the string "15" becomes the integer 15 for the purposes of comparison:

```
php > var_dump("15" < 19);  
bool(true)
```

Another exception is strings beginning with a special character such as these: ! # \$ % & ' () * + , - . /. Such a string is always considered less than a number:

```
php > var_dump("*77" < 5);  
bool(true)
```

If a number is compared with a leading numeric string (one that starts with numbers but contains other characters), the number is juggled to a string, and the strings are compared character by character:

```
php > var_dump("1a" > 10);  
bool(true)  
php > var_dump("1a" > 20);  
bool(false)
```

In the first case, the integer 10 is converted to the string "10" before comparison with the string "1a". The first characters are the same, but the character a (a letter) is considered greater than the character 0 (a number). In the second case, the 2 in the string "20" (after juggling) is considered greater than the 1 in the string "1a", so the expression is `false`.

Knowing these string-number and Boolean comparison rules is useful, but relying on them can be dangerous. It's safer to use validation logic to convert strings to numbers first and then make simple numeric comparisons. For example, we'll test that the price received from a web form is numeric in [Chapter 12](#).

The Spaceship Operator

A relatively new addition to the PHP language is the *spaceship operator* (`<=>`). Instead of `true` or `false`, this operator gives an integer value of 0, 1, or -1 depending on the two expressions being compared. If both expressions are the same (after any type juggling), the operator gives a 0; if the first expression is greater than the second, it gives a 1; or if the second expression is greater than the first, it gives a -1. For example:

```
php > var_dump(11 <=> 22);  
int(-1)  
php > var_dump(55 <=> 22);  
int(1)  
php > var_dump("22" <=> 22);  
int(0)
```

In the first case, we see -1 as the output, since 11 is less than 22. In the second case, we see 1 as the output, since 55 is greater than 22. In the last case, we see 0 as the output, since "22" is the same as 22 after type juggling.

The spaceship operator may seem like a strange amalgam of the less-than, greater-than, and equal-to operators. However, it's particularly useful when sorting collections of data into a desired sequence, since certain sorting functions require exactly this 0, 1, or -1 encoding scheme.

Logical and Other Contexts

When a logical value of `true` or `false` is expected, PHP will juggle values of other types to the `bool` type. The three logical type-juggling contexts are as follows:

- Logical operators, such as AND (`&&`) and OR (`||`)
- The ternary operator (`?`)
- Conditional statements such as `if` and `switch`

We'll cover all of these logic contexts in [Chapter 4](#).

Type juggling also may occur in function contexts and bitwise contexts. We'll explore the function context (when arguments are evaluated against function signatures) in [Chapter 5](#). The bitwise context is rarely used in web applications and is beyond the scope of this book.

Type Casting

Type casting refers to explicitly converting an expression or variable to a desired data type. Manual type casting stands in contrast to the type juggling performed automatically by the PHP engine. To cast the value of an expression to a particular type, provide the new data type in parentheses before the expression. For instance, `(float)21` ensures that the value `21` will be treated as a float instead of an integer. Here are some examples of casting various scalar data types:

```
php > $age = (int)20.5;
php > var_dump($age);
❶ int(20)
php > $price = (string)9.99;
php > var_dump($price);
string(4) "9.99";
php > $inventory = (bool)0;
php > var_dump($inventory);
❷ bool(false)
```

Notice that casting from a float to an integer truncates anything after the decimal point, effectively rounding down to the nearest whole number

❶. When casting from a number to a Boolean, 0 becomes `false` ❷, while any other numeric value (including negative numbers!) becomes `true`.

Type casting is one of the lesser-used features of PHP. One example of its use could be to easily obtain the integer part of a float, such as the whole number of seconds when comparing two timestamps.

Summary

This chapter introduced you to PHP's four scalar data types, as well as the special `NULL` type. You saw the difference between variables containing `NULL` and variables evaluating to `NULL` by virtue of being undefined or unset, and you practiced testing whether a variable is of a particular type by using functions like `is_int()` and `is_null()`. Later, this will help you write code that carefully tests values to manage situations that may occur when receiving input from users or external data sources such as databases.

This chapter also showed you how an expression's data type can change, either automatically through juggling or manually through casting. You learned about the contexts where type juggling can occur and rules for evaluating expressions with different data types. Understanding when and how types are juggled will help you avoid unexpected results when working with data of mixed types.

Exercises

1. Write a script to use integer casting to round down a float. Do the following:
 - a. Create a `$scoreFloat` variable containing `55.9`.
 - b. Create a second variable, `$scoreInt`, containing the value of `$scoreFloat` cast into an integer.
 - c. Print out the type of `$scoreFloat`, then its value, then a newline character.
 - d. Print out the type of `$scoreInt`, then its value, then a newline character.

Your program output should look as follows:

```
double scoreFloat = 55.9  
integer scoreInt = 55
```

2. Assign the `$age` variable the integer 21, and `var_dump` its value. Then assign `NULL` to this variable, and `var_dump` it again. Finally, unset the variable and `var_dump` it once more. Note that the output differs when the variable is assigned `NULL` and when it's unset.

3

STRINGS AND STRING FUNCTIONS



In this chapter, we'll take a close look at strings, including how to create them, how to combine them, and how to search, transform, and otherwise manipulate them with PHP's many built-in string functions. Almost every PHP program and web application involves text, so it's important to understand how to create and work with strings.

The chapter introduces PHP's four styles for writing strings: inside single quotation marks or double quotation marks, or in longer, multiline spans as heredocs or nowdocs. These styles come with different features, such as the ability to incorporate variables or represent special symbols. Despite these differences, they all end up with the same result: a value of the `string` data type, which is to say, a sequence of characters.

Whitespace

Before we consider strings with characters that you can see, such as letters and numerals, a word about *whitespace*, the characters you can't see. These are characters that won't use any ink when printing (for example, the space character, the tab character, the newline character, and so on). Sometimes

it's useful to distinguish between *horizontal* whitespace, such as spaces and tabs, and *vertical* whitespace, such as newline characters and vertical tabs.

When you write code, the details of the whitespace you use are often unimportant to the execution of the code. As long as you have a minimum of one whitespace character (such as a space or newline) between expressions, it doesn't matter whether you have additional whitespace characters (such as extra spaces or tabs). For example, the PHP engine will interpret all four of the following statements as exactly the same, ignoring multiple spaces, tabs, and newlines around the variable name and equal sign, and on either side of the string:

```
$lastName = 'Smith';
    $lastName = 'Smith';
    $lastName      =      'Smith'      ;
$lastName
=
    'Smith'
;
```

When you're declaring or manipulating the contents in string expressions, however, you have to be precise about your use of whitespace. Otherwise, words may end up stuck together with no space between them, or some text may end up on a different line. For example, the strings in these four statements are all different, since the extra whitespace is *inside* the quotation marks:

```
$lastName = 'Smith';
$lastName = '    Smith';
$lastName = 'Smith    ';
$lastName = '        Smith    ';
```

Whitespace doesn't come just from the code you write. It can also be introduced when you take a string as input from a user or an external software system such as an application programming interface (API). In this case, you often need to validate that input and trim any unwanted whitespace at its beginning and end. Likewise, you may want to replace any

tabs or newline characters inside a string with single spaces (sometimes users press function keys by accident while typing, adding unintended, invisible whitespace characters to their input). You'll learn to do all these things in this chapter (for example, see "Removing All Unnecessary Whitespace" on [page 60](#)).

Single-Quoted Strings

The simplest type of string in PHP is enclosed in single quotation marks, such as `'matt smith'`. Almost everything that appears inside the single quotes is treated literally, meaning it will be reproduced exactly as written, character for character, if the string is printed out.

PHP has just two special cases for single-quoted strings. First, since single quotes serve to delimit the string, there must be a mechanism for including a single quotation mark inside the string itself. Otherwise, the single quote will be interpreted as the end of the string. The solution is to put a backslash in front of the single quote (`\'`), as in `'matt smith\'s string'`. When this string prints, the single quote will show up, but not the backslash, as you can see here in PHP's interactive mode:

```
php > print 'matt smith\'s string';  
matt smith's string
```

This technique of having a special sequence of characters in a string that PHP will interpret as a certain character is known as *escaping*. Since the backslash is needed to escape single quotation marks, there must also be a way to specify a backslash character in a single-quoted string. For that, write two backslashes: `\\`.

The `\'` and `\\` escape sequences are the only characters that the PHP engine will interpret to mean something else in a single-quoted string. Other escape sequences you might know from other programming languages, such as `\n` for newline, aren't recognized in single-quoted strings.

That doesn't mean you can't include newlines in single-quoted strings, however. For that, simply add line breaks wherever you want them in the string, and PHP will reproduce the line breaks exactly as you've written them. In fact, one reason PHP requires a semicolon to declare the end of a

statement is to allow a single statement to be written over several lines. [Listing 3-1](#) illustrates a string broken across multiple lines.

```
<?php
print 'the
cat
sat on
the mat!
';
```

Listing 3-1: A script with a string containing line breaks

This script includes a single `print` statement for a string that contains several line breaks. Here's the output of running this script at the command line:

```
the
cat
sat on
the mat!
```

The output includes all the newlines that were written into the string.

Joining Strings: Concatenation

Often you'll need to combine several strings, variables, or constants to form a single string expression. You can do this with PHP's string concatenation operator, represented by a period (`.`). Consider this example using PHP's interactive mode:

```
php > $name = 'Matt Smith';
php > print 'my name is ' . $name;
my name is Matt Smith
```

We declare the `$name` variable whose value is the string `'Matt Smith'`. Then we use the string concatenation operator to combine the value of this variable with another string, and we print the resulting longer string. Note the extra space in the string `'my name is '` before the closing single quote.

The concatenation operator doesn't add any spaces between the strings it's joining, so this extra space is to prevent something like `isMatt` from being printed.

When concatenating strings on a single line, it's good programming practice to make the statement more readable by adding a space on each side of the period, as in the preceding example. You can also spread such statements over several lines, which can help make long string expressions more readable, as shown in the script in [Listing 3-2](#). This shows my personal preference of indenting each subsequent line of the statement and beginning each line with the concatenation operator, so it's clear that each line is appending more to the string expression.

```
<?php
$name = 'Matt Smith';
print 'my name is '
    . $name
    . ', I\'m pleased to meet you!'
    . PHP_EOL;
```

Listing 3-2: A script to concatenate and print out several strings

This script prints out a string formed by concatenating four shorter strings together, including the two from the previous example. Notice that we've escaped a single quote to create the apostrophe in the word `I'm` in the third string.

The fourth string at the end of the expression is the special string constant `PHP_EOL`, short for *end of line*. This is a string containing the system-appropriate character (or characters) to move the cursor to the beginning of the next line in a command line terminal (the same as hitting ENTER). Such a special constant used to be needed since different operating systems used slightly different ways to specify the end of a line. It's not such an issue now since applications for most operating systems generally know how to work with each other's files these days, but the constant is still handy for ensuring that the next terminal prompt after a single-quoted string starts on a new line. Here's the output when this script is run at the command line:

```
% php multi_line.php
my name is Matt Smith, I'm pleased to meet you!
%
```

I saved the script from [Listing 3-2](#) in a file named *multi_line.php*. Running the script concatenates the strings and prints the result on a single line. Notice that the next terminal prompt (in this case, the percent character, %) appears on the next line, thanks to the `PHP_EOL` constant.

WHY NOT A PLUS SIGN?

Most programming languages are strongly typed, so the plus sign (+) can be used for both mathematical addition and string concatenation. The compiler knows which operation to implement since it knows whether the variables on either side of the operator are numbers or strings. By contrast, since PHP is loosely typed, the plus sign is used only for mathematical addition. A different character had to be selected for string concatenation, and the period (.) was chosen.

In most object-oriented languages, the period has a different meaning: it allows access to an object's methods and properties. PHP didn't initially support object-oriented programming, however. When this capability was eventually added, the period was already reserved for string concatenation, so PHP had to choose another character sequence for object access. You'll meet this object access operator (->) in [Part V](#), when we discuss object-oriented programming.

The use of dollar signs for variables, periods for string concatenation, and -> for object access makes PHP's syntax a little strange for programmers used to other languages. Once you've been programming in PHP for a while, though, this will all become second nature.

If a variable already contains a string, we can use the concatenating assignment operator (.=) to append another string expression to the end of that variable. Here's an example using PHP's interactive mode:

```
php > $name = 'Matt';
php > $name .= ' Smith';
php > print $name;
Matt Smith
```

First, we initialize the `$name` variable to the string `'Matt'`. Then we use the concatenating assignment operator to append `' Smith'` to the end of the contents of `$name`. When we print out the variable, we can see that it now contains `Matt Smith`.

Double-Quoted Strings

The second type of PHP string is enclosed in double quotes, as in `"Matt Smith"`. Double-quoted strings differ from their single-quoted counterparts in that they are *parsed*, or processed, by the PHP engine, which means they can include PHP variables. When you write a variable (beginning with a dollar sign) in a double-quoted string, the PHP engine will look up the value of the variable and insert it into the string before printing. This is often more convenient than using a period to concatenate single-quoted strings with the contents of variables, as we did in the previous section. Here's an example using PHP's interactive mode:

```
php > $name = 'Matt Smith';  
php > print "My name is $name \nI'm pleased to meet you";  
My name is Matt Smith  
I'm pleased to meet you
```

First, we assign the string `'Matt Smith'` to the `$name` variable, as before. Note that we could have used double quotes for this string, but most PHP programmers use them only for strings that will need parsing. Next, we print a double-quoted string that includes the `$name` variable. The output shows that PHP has successfully parsed this string by inserting `Matt Smith` in place of the variable.

Notice that the double-quoted string includes a single-quote character in the contraction `I'm`. This is perfectly valid in a double-quoted string and doesn't require escaping. Our double-quoted string also includes the escape sequence `\n` to create a newline in the middle of the output. This is one of several escape sequences available for use in double-quoted strings. [Table 3-1](#) lists some of the most common.

Table 3-1: Common Escape Sequences in Double-Quoted Strings

Escape sequence	Description
\\	Backslash
\"	Double-quote
\\$	Dollar sign
\n	Newline
\t	Tab

Notice in particular that since a dollar sign will make the PHP engine parse for a variable, you must use `\$` to include an actual dollar sign in a double-quoted string. You also need to use `\"` to include a double quotation mark.

You can't include constants in double-quoted strings, since the PHP engine can't tell the difference between characters that are part of the string and the name of the constant. (Recall from [Chapter 1](#) that constants don't start with a dollar sign.) One side effect is that you can't include the `PHP_EOL` constant in a double-quoted string to create a newline at the end of the string. Instead, use the `\n` newline escape sequence.

NOTE

On the rare occasions when you need the operating system-independent `PHP_EOL` constant with a double-quoted string, you can use the string concatenation operator to add the constant to the string, much as you saw in [Listing 3-2](#). This situation might arise when a script needs to precisely output the appropriate newline and cursor-to-beginning-of-line character sequence for the system the PHP engine is running on (for example, to ensure that a system file has the correct line endings).

Handling the Character After a Variable Name

When you have a variable name followed by a space in a double-quoted string, such as `"my name is $name \nI'm pleased to meet you"`, the PHP engine can easily identify the variable (`$name`) and recognize that a space should be present after its value. Even punctuation marks such as

periods, commas, and colons are fine when they immediately follow a variable, and so are escape sequences, since these aren't valid characters to have in variable names. Here, for example, using a comma immediately after the `$name` variable is perfectly fine:

```
php > $name = 'Matt Smith';  
php > print "my name is $name, and I'm pleased to meet you";  
my name is Matt Smith, and I'm pleased to meet you
```

If you want other characters to be part of the string immediately after a variable, however, the situation becomes a little more difficult. After its first letter (or underscore character), the characters of a variable name can be letters, numbers and underscores, so if any of these are written immediately after a variable name in a double-quoted string, the PHP engine treats them as part of the variable name. For example, if we have a `$weight` variable and we want its value to be followed immediately by `kg`, as in `80kg`, we can't write something like this:

```
print "$weightkg";
```

The PHP engine would complain, saying there's no variable named `$weightkg`. The solution for these more complex double-quoted-string parsing tasks is to enclose the variable name in curly brackets (braces):

```
php > $weight = 80;  
php > print "my weight is {$weight}kg";  
my weight is 80kg
```

Thanks to the curly brackets, PHP has no problem printing the characters `kg` immediately after the value of `$weight`. Note that this is an example of the string context of *type juggling*, as introduced in [Chapter 2](#). When the string `"my weight is {$weight}kg"` is parsed, the value of `$weight` will be juggled from the integer `80` into the string `'80'` for insertion into the final string.

Incorporating Unicode Characters

Not all characters are available to type directly from your keyboard or are part of the current language settings of your computer system, but that doesn't mean you can't include them in double-quoted strings. *Unicode* is an international standard for declaring and working with a wide range of characters and symbols, including ordinary English letters, emojis, letters from other alphabets, and more. Each Unicode character is defined by a unique hexadecimal code. For example, the code `1F60A` corresponds to one of several smiling emojis.

To use a Unicode character in a double-quoted string, start with the escape sequence `\u`, then provide the character's hexadecimal code in curly brackets. [Listing 3-3](#) shows code to declare and print several Unicode characters.

```
<?php
$smiley = "\u{1F60A}";
$elephant = "\u{1F418}";
$cherokeeTSV = "\u{13E8}";

print "this is a smiley unicode character\n$smiley\n";
print "followed by some elephants of course\n$elephant $elep
hant $elephant\n";
print "Cherokee letter TSV\n$cherokeeTSV\n";
```

Listing 3-3: A script to display various Unicode characters

First, we declare the variables `$smiley`, `$elephant`, and `$cherokeeTSV` to contain double-quoted strings with the Unicode characters for their respective smiley face and elephant emojis, and the Cherokee TSV symbol. Then we print some double-quoted strings that include those variables. Here's the result:

```
this is a smiley unicode character
😊
followed by some elephants of course
🐘 🐘 🐘
Cherokee letter TSV
Ꮝ
```

Note that we could have included the Unicode character escape sequences directly in the double-quoted strings being printed rather than assigning them to variables first. Having them in variables makes it easier to reuse them throughout the script—for example, to print three elephants instead of just one. (The PHP community has a thing for elephants.) For a complete list of Unicode characters and their corresponding hexadecimal codes, visit <https://home.unicode.org>.

Heredocs

Heredocs are an alternative to double-quoted strings. They're just like double-quoted strings in that they're parsed and so can contain variables, but they differ in that heredocs typically span multiple lines. Although double-quoted strings can span multiple lines too, many programmers prefer heredocs for multiline strings since their syntax makes them stand out more obviously from the surrounding code.

To declare a heredoc, start with the heredoc operator (<<<), followed by a sequence of characters of your choice that will serve as a delimiter. Then, on a new line, start typing your string. When you get to the end of the string, repeat your chosen delimiter on its own line, followed by a semicolon to end the statement.

The most commonly used delimiter for heredocs is `EOT` (short for *end of text*), but which delimiter you choose doesn't really matter, as long as that character sequence doesn't appear in the string being declared and as long as the delimiters at the beginning and end of the heredoc match. It makes code more readable to either always use `EOT` or choose something meaningful to the heredoc's contents, such as `SQL` if it contains a SQL statement or `HTML` if it contains HTML. [Listing 3-4](#) shows a script with an example heredoc.

```
<?php
$age = 22;
$weight = 80;
```

```
❶ $message = <<<EOT
  my age is $age
  my weight is {$weight}kg
```

```
❷ EOT;
```

```
print $message;
```

Listing 3-4: A heredoc string declared and printed with the `EOT` delimiter

This code creates the variables `$age` and `$weight` containing 22 and 80, respectively. Then we assign a heredoc expression into the `$message` variable ❶. The heredoc starts with `<<<EOT`, and its content is everything from the next line until a newline character and the `EOT;` at the end ❷. Finally, we print the contents of `$message`. Here's the result:

```
% php heredoc.php
my age is 22
my weight is 80kg
%
```

I saved the script as *heredoc.php* and ran it at the command line. Notice that the variables `$age` and `$weight` were successfully parsed within the heredoc, including when we used curly brackets to allow characters to be output immediately following a variable. Notice also that the next command line prompt starts on its own line. This is because of the blank line in the heredoc before the closing delimiter; heredocs can contain newlines.

Escape Sequences

You can't use the `\"` escape sequence to write a double quotation mark in a heredoc. If you write `\"` inside a heredoc, the backslash will become an ordinary character in the string, just like the double quote. You won't need this escape sequence in heredocs anyway: since you're no longer using

double quotes to delimit the string, using them within the string won't cause confusion.

You *can* include other escape sequences in heredocs, such as `\t` for tabs and `\n` for newlines. These, too, aren't strictly necessary, however, since you can just use the `TAB` and `ENTER` keys when writing the heredoc's contents. You can also type Unicode characters directly into a heredoc (assuming your editor supports Unicode, that is). [Listing 3-5](#) shows an example.

```
<?php
$message = <<<UNICODE
this is a smiley unicode character
😊
followed by some elephants of course
🐘 🐘 🐘
Cherokee letter TSV
Ꮖ

UNICODE;

print $message;
```

Listing 3-5: A script with Unicode characters declared in a heredoc string

We declare this heredoc by using `UNICODE` as the delimiter. The code features Unicode characters typed directly into the string, rather than created via escape sequences. The output of running this script is identical to the output from [Listing 3-3](#).

Indentation

An occasionally useful feature of heredocs is that if indentation (spaces or tabs) appears before the closing delimiter, the PHP engine will attempt to remove that same amount of indentation from all lines of the heredoc. In [Listing 3-6](#), for example, we declare and print a heredoc string that has each of its lines, including the line with the ending `TEXT` delimiter, indented four spaces.

```
<?php
$message = <<<TEXT
    If the closing delimiter is indented
    then that amount of indentation
    is removed from the lines of the string

    TEXT;

print $message;
```

Listing 3-6: A script with an indented heredoc

Since every line of this heredoc has the same indentation as the closing delimiter line, all indentation will be removed from all lines when the string is printed. Here's the output:

```
If the closing delimiter is indented
then that amount of indentation
is removed from the lines of the string
```

If any lines in a heredoc have more indentation than the line with the closing delimiter, that extra bit of indentation will remain in the output.

[Listing 3-7](#) shows an example.

```
<?php
$message = <<<END
    I'm the same indentation as the ending delimiter (4 spaces)
        I have 2 extra spaces
        So have I!
    I'm back to 4 spaces again

    END;

print $message;
```

Listing 3-7: A script that retains extra indentation in a heredoc

The first and last lines of the heredoc are indented four spaces, as is the closing delimiter. The middle two lines of the heredoc have two extra spaces of indentation. In the output, four spaces will be stripped away from the start of each line, leaving two spaces of indentation for the middle two lines, as shown here:

```
I'm the same indentation as the ending delimiter (4 spaces)
  I have 2 extra spaces
  So have I!
I'm back to 4 spaces again
```

This feature of removing indentation from a heredoc is primarily useful when the heredoc is declared as part of a function's body, where it's customary for all code to have some level of indentation. This way, you can write cleaner-looking heredocs that adhere to the indentation conventions of the code around them. We'll meet functions in [Chapter 5](#).

NOTE

If any lines in a heredoc have less indentation than the closing delimiter, or if they have a different kind of indentation (such as a tab character instead of spaces), an error will occur at runtime.

Nowdocs

The last style of PHP string is the *nowdoc*, an unparsed string written with the <<< operator and delimiters. Essentially, the nowdoc is to unparsed single-quoted strings what the heredoc is to parsed double-quoted strings. The only difference between declaring a nowdoc and a heredoc is that the opening delimiter for the nowdoc must be enclosed in single quotes, as in <<<'EOL'. The closing delimiter isn't written with single quotes.

One use of nowdocs is for printing out PHP code. Since nowdocs are unparsed, any code, including variable names, will be reproduced literally in the string expression. [Listing 3-8](#) shows an example.

```
<?php
❶ $name = "Matt Smith";
```



```
❷ $codeSample = <<<'PHP'
    $message = "hello \n world \n on 3 lines!";
    $age = 21;
    ❸ print $name;
    print $age;

    PHP;

print $codeSample;
```

Listing 3-8: A script declaring a nowdoc that contains unparsed PHP code

First, we declare a `$name` variable ❶. Then we declare a nowdoc by using the delimiter `PHP` and assign it to the `$codeSample` variable ❷. (Notice that the starting delimiter is enclosed in single quotes, but the ending delimiter isn't.) The nowdoc contains statements such as a variable declaration (`$age`), strings with escaped characters, and a reference to our `$name` variable ❸. All of this goes unparsed when we print the nowdoc, as you can see in the output:

```
$message = "hello \n world \n on 3 lines!";
$age = 21;
print $name;
print $age;
```

The entire nowdoc has been printed verbatim, including the escape sequences and the characters `$name`. No program statements within the nowdoc were executed; they just became part of the declared nowdoc string. Notice, however, that the indention in the nowdoc has been stripped away, since it matches the indention of the closing delimiter. This works just as with heredocs.

We've finished our overview of the four styles of strings. Deciding which to use is often a matter of personal preference. In general, single quotes are best for short strings that don't include variables that require parsing. For longer, multiline strings with no parsing, consider using

nowdocs. If you need to incorporate parsed variables, use double quotes for shorter strings or heredocs for longer, multiline strings.

Built-in String Functions

PHP has more than 100 built-in functions for manipulating and analyzing strings, ranging from standard tasks like toggling between uppercase and lowercase letters to more specialized tasks such as implementing hash algorithms. We'll look at a few of the most commonly used string functions in this section.

If you're coming to PHP from another language, you may be used to seeing these kinds of operations as methods that are called directly on the strings themselves. Since PHP wasn't originally object-oriented, however, the operations instead exist as stand-alone functions.

NOTE

For a complete list of PHP's string functions, see <https://www.php.net/manual/ref.strings.php>.

Converting to Upper- and Lowercase

When working with user input, you'll often need to standardize strings by ensuring that they all follow the same capitalization rules. This makes it easier to compare strings, or to store text in a database or send it to an API using a consistent format. To that end, PHP has functions for adjusting the capitalization of a string. The `strtolower()` and `strtoupper()` functions convert all the letters to lowercase or uppercase, respectively. Here are some examples illustrated using PHP's interactive mode:

```
php > $myString = 'the CAT sat on the Mat';  
php > print strtolower($myString);  
the cat sat on the mat  
php > print strtoupper($myString);  
THE CAT SAT ON THE MAT
```

We declare a string with a mix of uppercase and lowercase letters. Passing the string to the `strtolower()` function converts everything to

lowercase, while passing it to `strtoupper()` converts everything to uppercase.

PHP's `ucfirst()` function capitalizes just the first letter of a string, if it isn't capitalized already. This is useful when creating messages to be output to the user; capitalizing the first letter helps make the messages look like grammatically correct sentences:

```
php > $badGrammar = 'some people don\'t type with capital le  
tters.';  
php > print ucfirst($badGrammar);  
Some people don't type with capital letters.
```

The related function `lcfirst()` lowercases just the first letter of a string:

```
php > $worseGrammar = 'SOME PEOPLE TYPE WITH ALL CAPS.';  
php > print lcfirst($worseGrammar);  
sOME PEOPLE TYPE WITH ALL CAPS.
```

Often this isn't much of a grammatical improvement, but it can be useful, for example, if you're writing a script to output code. In this case, to follow the naming conventions of the programming language (such as for variables), it can be important to ensure that the first character of a string is lowercase.

The `ucwords()` function capitalizes the first letter of every word in a string. PHP can distinguish between the different words if they're separated by whitespace:

```
php > $mixedCaps = 'some peoPLE use CAPS spoRADically.';  
php > print ucwords($mixedCaps);  
Some PeoPLE Use CAPS SpoRADically.
```

Notice that if any subsequent letters in a word are capitalized, they remain so. Only the first letter of each word is affected. PHP doesn't have an equivalent function for lowercasing the first letter of each word.

Searching and Counting

Several of PHP's built-in string functions serve analytical purposes, such as reporting the length of a string, searching for a character or substring within a string, or counting the number of occurrences of a character or substring within a string. (A *substring* is a portion of a larger string.) Before we examine these functions, though, it's important to distinguish between the *number* of characters in a string and the *position* of each character within that string.

Take the string 'cat scat' as an example. It consists of eight characters (the space in the middle counts), but character positions in PHP are numbered starting from zero. Thus, the character at position 0 is c, at position 1 is a, and so on up to the final t at position 7. This counting from zero is called *zero-based indexing* and is common in computer programming. Using this system, we can say that the substring 'cat' occurs twice in the string, starting at position 0 for the first occurrence and at position 5 for the second:

```
cat scat
01234567
cat  cat
```

With this in mind, let's try out some analytical string functions on the string 'cat scat'. First, the `strlen()` function reports the length of a string, as shown here in PHP's interactive mode:

```
php > $myString = 'cat scat';
php > print strlen($myString);
8
```

As expected, this tells us that 'cat scat' is eight characters long.

The `substr_count()` function counts the number of times a substring appears within a string. Here, for example, we count the instances of the substring 'cat':

```
php > $myString = 'cat scat';  
php > print substr_count($myString, 'cat');  
2
```

We pass two strings to the `substr_count()` function. The first is the string we want to search *in*, which here we’re providing as the `$myString` variable. The second is the string we want to search *for*: in this case, `'cat'`. In computer search terminology, these two strings are often referred to as the *haystack* and the *needle*, respectively, after the expression “looking for a needle in a haystack.”

NOTE

The items entered within the parentheses of a function, such as `$myString` and `'cat'` in the preceding example, are called arguments. They’re pieces of data that the function needs to do its job. We’ll discuss functions in detail in [Chapter 5](#).

Most PHP functions that involve searching within a string, including `substr_count()`, are case sensitive, so it’s important to be careful about capitalization. If we try searching `'cat scat'` for the substring `'Cat'` instead of `'cat'`, for example, we’ll end up with a count of 0:

```
php > $myString = 'cat scat';  
php > print substr_count($myString, 'Cat');  
0
```

The `strpos()` function reports the starting position (counting from zero) of a substring within a string. If the substring occurs multiple times, only the position of the first occurrence is given. Here we search for the first occurrence of the substring `'cat'`:

```
php > $myString = 'cat scat';  
php > print strpos($myString, 'cat');  
0
```

As with `substr_count()`, we provide two strings to the `strpos()` function as arguments, first the haystack and then the needle. The function reports the first occurrence of 'cat' at position 0.

Optionally, you can provide an *offset* to the `strpos()` function as an additional argument, a number that tells it to start searching for the substring from a different position, rather than from the beginning of the string. Here we tell the function to start searching from position 2 onward:

```
php > $myString = 'cat scat';  
php > print strpos($myString, 'cat', 2);  
5
```

This time, since the function isn't searching from the beginning of the string, it identifies the second occurrence of 'cat' at position 5. If the function can't find any occurrences of the needle within the haystack, it reports `false`.

The `count_chars()` function analyzes which characters are, and are not, included in a string. It's a powerful string analysis function that you might use when evaluating the complexity of a password or perhaps for data encryption and decryption tasks. It has a few modes, which you specify as a number when the function is called. In the following example, we use mode 3, which generates a new string consisting of all the unique characters used in the string being analyzed:

```
php > $myString = 'cat scat';  
php > print count_chars($myString, 3);  
acst
```

We call the `count_chars()` function on `$myString`, specifying mode 3. The resulting string features one instance of each character from 'cat scat', arranged in alphabetical order. It shows us that 'cat scat' includes only the letters *a*, *c*, *s*, and *t*.

NOTE

The `count_chars()` function has other modes that count the number of occurrences of each character, but the result is reported as an array, so we won't consider those modes here. We'll discuss arrays in [Chapters 7](#) and [8](#).

Extracting and Replacing Substrings

Other PHP functions manipulate strings by extracting a portion of the string or replacing a portion with something else. For example, the `substr()` function extracts part of a string, starting from a given position. Here's how it works:

```
php > $warning = 'do not enter';  
php > print substr($warning, 7);  
enter
```

We declare a variable with the string 'do not enter', then pass it to the `substr()` function. The number 7 tells the function to extract all the characters from position 7 to the end, giving us `enter` as a result.

If you use a negative number instead, the function makes the extraction counting from the end of the string. For example, here we use -2 to get just the last two characters:

```
php > $warning = 'do not enter';  
php > print substr($warning, -2);  
er
```

You can optionally include a second number in the function call to specify a length for the extraction. Here, for example, we extract just three characters from the string, starting from position 7:

```
php > $warning = 'do not enter';  
php > print substr($warning, 7, 3);  
ent
```

The `strstr()` function provides another technique for extracting part of a string. By default, it searches for the first occurrence of a substring within a string and extracts the contents of the string from that substring on.

Here, for example, we search for the substring '@' to extract just the domain name and extension from an email address:

```
php > $email = 'the.cat@aol.com';  
php > print strstr($email, '@');  
@aol.com
```

The function searches for the first occurrence of an @ sign and reports everything in the string from that point on, including the @ sign itself. We can also use `strstr()` to extract everything in the string *before* the first occurrence of the substring. For that, add `true` to the end of the function call, like this:

```
php > $email = 'the.cat@aol.com';  
php > print strstr($email, '@', true);  
the.cat
```

Again, we search the string for the first occurrence of an @ sign, but this time, thanks to the added `true`, the function reports the contents of the string up to but not including the @ sign. This gives us just the username from the email address.

NOTE

The `strstr()` function is case sensitive, but PHP provides a case-insensitive version called `stristr()`.

The `str_replace()` function finds all occurrences of a substring within a string and replaces them with a different substring. The result of the replacement is reported as a new string, meaning the original string itself isn't modified. Here's an example:

```
php > $foodchain = 'dogs eat cats, cats eat mice, mice eat c  
heese';  
php > print str_replace('eat', 'help', $foodchain);  
dogs help cats, cats help mice, mice help cheese
```

```
php > print $foodchain;  
dogs eat cats, cats eat mice, mice eat cheese
```

When we call `str_replace()`, we need to provide three strings. The first is the substring to search for (in this case, 'eat'). The second is the substring to replace it with (in this case, 'help'). The third is the string to search in, which we've assigned to the `$foodchain` variable. The function generates a new string by replacing all instances of 'eat' with 'help'. Then we print the value of `$foodchain` to confirm that it wasn't affected by the function call.

To make the replacement somewhat permanent, store the result of calling `str_replace()` in a new variable, like this:

```
php > $foodchain = 'dogs eat cats, cats eat mice, mice eat c  
heese';  
❶ php > $friendchain = str_replace('eat', 'help', $foodchain);  
php > print $foodchain;  
❷ dogs eat cats, cats eat mice, mice eat cheese  
php > print $friendchain;  
dogs help cats, cats help mice, mice help cheese
```

This time when we call `str_replace()`, we assign the result to `$friendchain` ❶. The original `$foodchain` variable is still unaffected by the replacement operation ❷, but at least now we have the modified string available in a variable for later use.

Another replacement function is `substr_replace()`. Rather than specifying a substring that should be replaced, this function lets you specify the position in the string at which the replacement should occur. Here's an example:

```
php > $foodchain = 'dogs eat cats, cats eat mice, mice eat c  
heese';  
php > print substr_replace($foodchain, 'help', 5, 3);  
dogs help cats, cats eat mice, mice eat cheese
```

When we call the `substr_replace()` function, we first provide the original string (in `$foodchain`) and the replacement string (`'help'`). Then we provide two numbers as additional arguments. The first, 5, is the position in the original string where the replacement should begin. The second, 3, is the number of characters in the original string that should be replaced, starting from the specified position. This has the effect of replacing the first instance of the word *eat* (which is three characters long and starts at position 5) with the word *help*, while leaving the rest of the string as is.

By setting the replacement length to 0, we can use `substr_replace()` to insert a substring into a string without otherwise altering it, like so:

```
php > $foodchain = 'dogs eat cats, cats eat mice, mice eat cheese';
php > print substr_replace($foodchain, 'don\'t ', 5, 0);
dogs don't eat cats, cats eat mice, mice eat cheese
```

This inserts the word *don't* into the `$foodchain` string starting at position 5, without replacing any characters.

Trimming Whitespace

Often it's necessary to remove whitespace from the beginning or end of a string, a task known as *trimming*. PHP provides three trimming functions:

trim() Removes whitespace from the beginning *and* end of a string

ltrim() Removes whitespace from the beginning of a string (*l* is for *left*)

rtrim() Removes whitespace from the end of a string (*r* is for *right*)

All three functions work in the same way: given a string, they remove any space, tab, vertical tab, newline, carriage return, or American Standard Code for Information Interchange (ASCII) null-byte characters, up to the first non-whitespace character and/or after the last non-whitespace character in the string. Here, for example, we use `trim()` to remove the spaces and newline characters at the start and end of a string:

```
php > $tooSpacey = "    \n\nCAT\nDOG\n\n";
php > print trim($tooSpacey);
CAT
DOG
```

The output shows that the newlines and the beginning and end of the string have been trimmed, but notice that a newline still remains between the words *CAT* and *DOG*. The trim functions have no effect on any whitespace in the middle of a string.

You can optionally control which whitespace characters are trimmed by specifying them in a separate double-quoted string when calling the function. Here's an example:

```
php > $sevenSpacier = "\n\n    CAT\nDOG\n\n";
php > print ltrim($sevenSpacier, ❶ "\n");
CAT
DOG
❷
php >
```

This time we use the string "\n" ❶ to specify that only newline characters should be trimmed. In the output, notice that the spaces before the word *CAT* have been preserved, since the function ignored all but newline characters. The blank line ❷ before the next prompt also indicates that the newlines at the end of the string were left in place, since the `ltrim()` function affects only the start of the string.

Removing All Unnecessary Whitespace

To remove whitespace from throughout a string, not just the edges, use `str_replace()` to find all instances of a particular whitespace character and replace them with empty strings. For example, here we use this technique to get rid of all the tab characters in a string:

```
php > $tooTabby = "\tCat \tDog \t\tMouse";
php > print $tooTabby;
Cat      Dog      Mouse
```

```
php > print str_replace("\t", '', $tooTabby);  
Cat Dog Mouse
```

The string assigned to `$tooTabby` contains several tab characters. Replacing each instance of `"\t"` with `' '` (an empty string) gets rid of the tabs while preserving the regular spaces between each word.

[Listing 3-9](#) pushes this technique even further, repeatedly using `str_replace()` to remove any whitespace from a string except for a single space character between words. This includes getting rid of tabs, newlines, and multiple space characters in a row.

```
<?php  
❶ $string1 = <<<EOT  
the  
    cat      sat  
    \t\t on   the  
mat  
  
EOT;  
  
❷ $noTabs = str_replace("\t", ' ', $string1);  
   $noNewlines = str_replace("\n", ' ', $noTabs);  
  
❸ $output = str_replace(' ', ' ', $noNewlines);  
   $output = str_replace(' ', ' ', $output);  
   $output = str_replace(' ', ' ', $output);  
  
   $output = trim($output);  
  
print "$output";
```

Listing 3-9: Replacing all whitespace (except single space characters) in a string

We use a heredoc to declare the `$string1` variable, which contains tabs, newline characters, and multiple spaces in a row between words ❶. Then we use the `str_replace()` function twice, first to replace all tabs with a single space, and a second time to replace all newline characters with a

single space ❷. (We don't replace them with empty strings in case a tab or newline is the only character between two words.)

Next, we repeatedly use `str_replace()` to replace any instances of two space characters with a single space ❸. It takes three function calls before only single spaces are left. (In [Chapter 6](#), we'll delve into loops, which provide a more efficient way to repeat the same code several times, or until a particular condition is satisfied.) For good measure, we use `trim()` to remove any lingering whitespace at the start or end of the string before printing out the resulting string, enclosed in square brackets so it's easier to see where it starts and ends. Here's the output of running this script:

```
[the cat sat on the mat]
```

The final string has no whitespace before or after it and only single spaces between each word. All the extra whitespace has been removed.

Repeating and Padding

Some PHP string functions work by repeating a character or substring to generate a longer string. For example, to create a new string by repeating a string a given number of times, use `str_repeat()`, like this:

```
php > $lonely = 'Cat';  
php > print str_repeat($lonely, 5);  
CatCatCatCatCat
```

This gives our lonely string some company by repeating 'Cat' five times.

Closely related to repeating is *padding*: a character or substring is repeatedly added to the beginning or end of a string until the string reaches a desired length. Padding is useful, for example, if you're displaying multiple numbers of different lengths and you want their digits to line up nicely. In that case, you might add spaces or zeros as padding in front of the numbers, as shown here:

```
12 // padded with spaces  
1099
```

```
000001 // padded with zeros
000855
```

PHP has a `str_pad()` function for such padding tasks. Here, for example, we pad the string 'Cat' with hyphens (-) until it's 20 characters long:

```
php > $tooShort = 'Cat';
php > print str_pad($tooShort, 20, '-');
Cat-----
```

We call `str_pad()`, providing the original string (`$tooShort`), the desired length (20), and the string to use as padding ('-'). By default, PHP adds the padding to the right of the original string, but you can add the constants `STR_PAD_LEFT` or `STR_PAD_BOTH` to the function call to place the padding on the left or to pad both sides equally instead. Here are some examples:

```
php > $tooShort = 'Cat';
php > print str_pad($tooShort, 20, '-', STR_PAD_LEFT);
-----Cat
php > print str_pad($tooShort, 20, '-', STR_PAD_BOTH);
-----Cat-----
```

In each case, the function adds hyphens until the resulting string is 20 characters long.

Summary

Strings are a core data type that you'll probably use in every web application you create. In this chapter, you learned about the four ways to declare strings: single-quoted strings, double-quoted strings, heredocs, and nowdocs. You saw how double-quoted strings and heredocs are parsed and so can incorporate variables, while single-quoted strings and nowdocs are not. You also tried out PHP's built-in functions for working with strings, and you learned how to combine strings by using the `.` and `.=` operators.

Exercises

1. Write a script that declares a `$name` variable containing your name as a single-quoted string. Then include a `print` statement that uses the string concatenation operator (`.`) to combine the contents of `$name` with the string `' is learning PHP'`. When you run your script, the output should look something like this:

```
Matt is learning PHP
```

2. In a script, create a `$fruit` variable containing the string `'apple'`. Then use a double-quoted string and a `print` statement to output the following message:

```
apple juice is made from apples.
```

Change your script so that `$fruit` contains `orange`, leading to the following output:

```
orange juice is made from oranges.
```

Hint: You'll need to use curly brackets to create the plural fruit names from the `$fruit` variable.

3. Write a script that declares a heredoc string variable `$happyMessage` containing the following (including the newlines):

```
PHP is fun
```



Print out the contents of the `$happyMessage` variable.

4. In a script, create an `$appleJuice` variable containing the string `'apple juice is made from apples.'` Then use the `str_replace()` function to create a new string variable, `$grapefruitJuice`, containing the string `'grapefruit juice is made from grapefruits.'` Try using other PHP functions to further transform the string. For example, capitalize

the first letter of the string so it looks like a grammatically correct sentence.

4

CONDITIONALS



In this chapter, you'll learn about *conditional* elements of the PHP

language, including `if...else` statements, `switch` statements, and `match` statements. These structures, along with language features such as the ternary operator, the null-coalescing operator, and logical operators, make it possible to write dynamic code that decides what to do based on a set of conditions. The conditions might depend on certain inputs (for example, from a user or from a software system such as a database or API) or on other varying data (such as the current date or time, or whether a file exists).

Conditions Are True or False

At the core of any decision-making logic is a *Boolean expression*, or code that is either `true` or `false`. The simplest Boolean expression is a literal value of `true` or `false`. In almost all cases, though, we instead write an

expression containing some kind of test. The test might examine the values of variables, or perhaps call a function and check the value it returns. Either way, the test ultimately evaluates to `true` or `false`. Examples of tests include the following:

- Does a variable contain a particular value?
- Does a variable have any value assigned at all?
- Does a file or folder exist?
- Is the value in one variable greater or less than another value?
- Does a function return `true` or `false` based on the arguments provided?
- Is the length of a string greater than a certain minimum?
- Does a variable contain a value of a particular data type?
- Are two expressions both `true`, or just one, or neither?

Tests such as these all evaluate to `true` or `false`. They form the conditions of the choice statements you can use in your code.

if Statements

Perhaps the most common conditional statement in any programming language is the `if` statement. It allows you to execute a statement only if a certain condition is `true`. Otherwise, that statement is simply skipped. In PHP, `if` statements are written in the following format:

```
if (condition) statementToPerform;
```

Start with the `if` keyword, followed in parentheses by the condition you want to check. This condition is the Boolean expression that evaluates to `true` or `false`. It's common practice to add a space after the `if` keyword, before the opening parenthesis. Next comes the statement that should be executed if the condition is true.

[Listing 4-1](#) shows an example of an `if` statement. It prints the message `Good morning` if the hour of the day is before 12 (assuming a 24-hour clock).

```
<?php
$hourNumber = 10;
if ($hourNumber < 12) print 'Good morning';
```

Listing 4-1: An `if` statement

First, we set the `$hourNumber` variable to 10. Then we use an `if` statement to test our condition: whether the value in `$hourNumber` is less than 12. Since 10 is less than 12, the condition is `true`, so the statement after the condition is executed, printing out the message `Good morning`.

In this example, we had only one statement that we wanted to execute if the condition was `true`. But what if we want to execute multiple statements? We need a way to group the statements so it's clear that they're all part of the `if` statement. For this, enclose the sequence of statements in curly brackets (braces) immediately after the condition. The curly brackets delineate a *statement group*, a PHP construction that can contain zero, one, or many statements, and that PHP treats as a single statement. [Listing 4-2](#) shows an example of a conditional with a statement group.

```
<?php
$hourNumber = 10;
if ($hourNumber < 12) {
    print 'Good';
    print ' morning';
}
```

Listing 4-2: A refactored `if` statement featuring a statement group

This `if` statement produces the same result as [Listing 4-1](#), but we've rewritten it to consist of multiple `print` statements, one for each word in the message. The statements are enclosed in curly brackets to group them together. It's customary to write the opening bracket on the same line as the condition, followed on separate lines by each statement in the group, followed by the closing bracket, also on a separate line. By convention, each statement in the statement group is indented.

NOTE

Even if you have only a single conditional statement to be executed, enclosing that statement in curly brackets to form a statement group is common practice. This way, all `if` statements follow the same style, no matter how many statements are involved.

if...else Statements

Many situations require a program to perform one set of actions if a condition is `true` or another set of actions if the condition is `false`. For these situations, use an `if...else` statement. [Listing 4-3](#) shows an example where we choose between printing `Good morning` and `Good day`.

```
<?php
$hourNumber = 14;
if ($hourNumber < 12) {
    print 'Good morning';
} else {
    print 'Good day';
}
```

Listing 4-3: An `if...else` statement

This code once again checks whether the value of `$hourNumber` is less than 12. If it is, the condition is `true`, so we execute the `if` branch of the statement, printing `Good morning` as before. If the condition is `false`, however, and `$hourNumber` isn't less than 12, we execute the `else` branch of the statement instead, printing the message `Good day`. Notice that the `else` keyword appears after the closing curly bracket of the `if` branch's statement group. Then the `else` branch is given its own statement group enclosed in curly brackets.

In this case, `$hourNumber` is 14 (2 PM), so the condition evaluates to `false` and the `else` branch's statement is executed.

Nested if...else Statements

An `if...else` statement chooses between two courses of action. If you have more than two courses of action to choose from, you have a few options. One is to nest further `if...else` statements inside the original

else branch. [Listing 4-4](#) shows an example. This script encodes the logic that if the hour is before 12, we print `Good morning`; if the hour is between 12 and 17 (5 PM), we print `Good afternoon`; or otherwise, we print `Good day`.

```
<?php
$hourNumber = 14;
❶ if ($hourNumber < 12) {
    print 'Good morning';
} else {
    ❷ if ($hourNumber < 17) {
        print 'Good afternoon';
    } else {
        print 'Good day';
    }
}
```

Listing 4-4: Nested if...else statements

First, we have an `if` statement testing whether the hour is less than 12 ❶. If this condition isn't true, the `else` statement will be executed. The statement group for the `else` statement is a second (nested) `if...else` statement. The condition for this second `if...else` statement is whether the hour is less than 17 ❷. (If we're at this point, we've already determined that the hour isn't less than 12, so in effect we're testing whether the hour is between 12 and 17.) If this new test passes, `Good afternoon` will be printed. Otherwise, we get to the `else` portion of the nested `if...else` statement, where `Good day` is printed. Try playing with different values of `$hourNumber` to see how it affects the output of the script.

if...elseif...else Statements

Choosing between three or more actions is such a common pattern in programming that PHP provides a simpler syntax for it that avoids the need for nesting: between an `if` statement and its `else` statement, place one or more `elseif` statements. The PHP engine will first test the condition for the `if` statement. If the statement is false, the engine will then test the

condition for the first `elseif` statement, then the next `elseif` statement, and so on. When PHP finds a `true` condition, that branch's statements are executed, and the remaining condition checks are skipped. If none of the `if` or `elseif` conditions are `true`, the `else` statement at the end will be executed, if there is one.

[Listing 4-5](#) shows the same logic from [Listing 4-4](#), but it's rewritten using `if...elseif...else`.

```
<?php
$hourNumber = 14;
if ($hourNumber < 12) {
    print 'Good morning';
❶ } elseif ($hourNumber < 17) {
    print 'Good afternoon';
} else {
    print 'Good day';
}
```

Listing 4-5: Simplifying the nested `if...else` statements with `if...elseif...else`

Our second condition now appears in sequence as an `elseif` statement ❶ after the `if` statement, rather than having to be nested inside the `else` statement. You can add as many `elseif` statements as you want between the `if` and the `else`.

Alternative Syntax

PHP offers an alternative syntax for `if`, `if...else`, and `if...elseif...else` statements that uses colons rather than curly brackets to set off the various parts of the code. This syntax is illustrated in [Listing 4-6](#), which reproduces the `if...else` statement from [Listing 4-3](#).

```
<?php
$hourNumber = 14;
if ($hourNumber < 12):
    print 'Good morning';
else:
```

```
    print 'Good day';
endif;
```

Listing 4-6: The alternative syntax for conditional statements

In this alternative syntax, the condition for the `if` statement is followed by a colon (`:`). This line acts like an opening curly bracket, so any statements between it and the `else` (or `elseif`) keyword are considered part of the statement group to be executed if the condition is `true`. The `else` keyword is similarly followed by a colon rather than an opening curly bracket. The statement group for the `else` branch ends with the `endif` keyword signaling that the whole `if...else` structure is over.

This alternative syntax is particularly useful for web applications, where HTML template text could appear between the `if` statement and the `else` statement, and the use of indented curly brackets could be hard to follow in the code. Likewise, the `endif` keyword clearly indicates that the overall conditional is ending.

Logical Operators

PHP's *logical operators* manipulate or combine Boolean expressions, producing a single `true` or `false` value. This way, you can write more sophisticated tests for conditional statements than simply comparing two values, as we've done so far (for example, testing whether two conditions are `true`). These logical operators perform operations such as AND, OR, and NOT. The operators are summarized in [Table 4-1](#).

Table 4-1: PHP Logical Operators

Name	Operator	Example	Description
NOT	!	!\$a	true if \$a is false
AND	and	\$a and \$b	true if both \$a and \$b are true
	&&	\$a && \$b	
OR	or	\$a or \$b	true if either \$a or \$b is true, or if both are true
		\$a \$b	
XOR	xor	\$a xor \$b	true if either \$a or \$b, but not both, is true

Notice that AND and OR operations can be written two ways: with words (`and` or `or`) or with symbols (`&&` or `||`). The two versions perform the same function, but the symbol versions have a higher precedence than the word versions when an expression is evaluated. (We discussed operator order of precedence in [Chapter 1](#), in the context of arithmetic operators.)

NOT

An exclamation mark (!) represents the NOT operator. This operator negates a Boolean expression or tests whether the expression is not `true`. For example, [Listing 4-7](#) uses the NOT operator to test a driver's age. In Ireland, you have to be at least 17 years old to drive a car.

```
<?php
$age = 15;
if (!($age >= 17)) {
    print 'Sorry, you are too young to drive a car in Ireland.
d.';
}
```

Listing 4-7: An `if` statement using the NOT (!) operator

The `if` statement checks whether it is *not* true that the value of `$age` is greater than or equal to 17. Since 15 is not 17 or more, you should see the following message printed out when you run the script:

```
Sorry, you are too young to drive a car in Ireland.
```

Notice that we've placed `$age >= 17` in parentheses to separate it from the NOT operator. This is because the NOT operator normally takes higher precedence than the `>=` operator, but we want to check whether `$age` is greater than or equal to 17 before using `!` to negate that result. If we had written `if (!$age >= 17)` instead, without the inner parentheses, PHP would try to evaluate `!$age` first. The NOT operator requires a Boolean operand, so the value of 15 inside `$age` would be juggled to `true` (as would any other nonzero value). Then, since `!true` is `false`, we would have the expression `false >= 17`.

Next, PHP would try to evaluate the `>=` comparison, and since one of the operands is a Boolean, it would try to make the second operand a Boolean too. The integer `17` would thus be juggled to `true` (since it's nonzero), giving us the expression `false >= true`, which evaluates to `false`. Ultimately, without those extra parentheses, `!$age >= 17` would evaluate to `false` for any nonzero integer value of `$age`.

To avoid all this type juggling and potential for error due to missing parentheses, I often create a temporary Boolean variable for use in an `if` statement before introducing the NOT operator. For example, [Listing 4-8](#) shows an alternate version of the code from [Listing 4-7](#), with an extra variable to avoid any chance of mixing integers and Booleans.

```
<?php
$age = 15;
$seventeenAndOlder = ($age >= 17);
if (!$seventeenAndOlder) {
    print 'Sorry, you are too young to drive a car in Ireland.';
}
```

Listing 4-8: A cleaner version of [Listing 4-7](#), with an extra Boolean variable

We use the `$seventeenAndOlder` variable to store the `true` or `false` value of the `$age >= 17` test. Then the `if` statement uses the NOT operator to test whether `$seventeenAndOlder` is not `true`. While this adds an extra line of code compared to [Listing 4-7](#), it's much clearer to understand since we've separated the age test Boolean expression from the `if` statement condition.

NOTE

Placing expressions like `$age >= 17` inside parentheses isn't necessary when assigning their value to a variable. [Listing 4-8](#) uses parentheses to help make the code clearer to read.

AND

An expression with the AND operator is `true` when both operands are `true`. You can use either the keyword `and` or a double ampersand (`&&`) to create an AND operation. For example, the `if...else` statement in [Listing 4-9](#) uses the AND operator to determine whether a driver meets both conditions to be allowed to take a driving test. In Ireland, you have to pass a theory test and hold a learner's license for at least six months before you're allowed to apply for a driving test.

```
<?php
$passedTheoryTest = true;
$monthsHeldLearnersLicense = 10;
$heldLearnersLicenseEnough = ($monthsHeldLearnersLicense >=
6);

if ($passedTheoryTest and $heldLearnersLicenseEnough) {
    print 'You may apply for a driving test.';
} else {
    print "Sorry, you don't meet all conditions to take a dr
iver's test.";
}
```

Listing 4-9: An `if...else` statement using the AND operator

We declare the `$passedTheoryTest` variable as `true`, and `$monthsHeldLearnersLicense` with value 10. Then we test whether `$monthsHeldLearnersLicense` is greater than or equal to 6 and store the resulting Boolean (`true`, in this case) in the `$heldLearnersLicenseEnough` variable. Next, we declare an `if...else` statement with the condition `$passedTheoryTest and $heldLearnersLicenseEnough`. Since both values are `true`, the AND operation is `true` as well, so the message `You may apply for a driving test` will be printed out.

Try changing `$passedTheoryTest` to `false` or setting `$monthsHeldLearnersLicense` to a value less than 6. The AND operation should then evaluate as `false`, and the message in the `else` branch of the statement should print out.

OR

An OR operation is `true` when either or both operands are `true`. You can use either the keyword `or` or a double vertical pipe (`||`) to write an OR operation. [Listing 4-10](#) illustrates an `if` statement that uses the OR operator to determine whether a password fails basic security rules (by including the string `'password'` or being less than six characters long).

```
<?php
$password = '1234';
$passwordContainsPassword = str_contains($password, 'password');
$passwordTooShort = (strlen($password) < 6);

❶ if ($passwordContainsPassword || $passwordTooShort) {
    print 'Your password does not meet minimal security requirements.';
}
```

Listing 4-10: An `if` statement using the OR operator

We declare the `$password` variable storing the string `'1234'`. Then we declare two Boolean variables to help with our test. First, `$passwordContainsPassword` is assigned the result of passing variable `$password` and string `'password'` to the built-in `str_contains()` function. This function returns `true` if the second string argument (the “needle”) is found anywhere inside the first string argument (the “haystack”), or `false` otherwise. Since in this case the `$password` variable doesn’t contain the string `'password'`, `$passwordContainsPassword` will contain `false`. The other Boolean variable, `$passwordTooShort`, will be `true` if the length of `$password` is less than 6, tested with the built-in `strlen()` function. Since the string `'1234'` in `$password` is less than six characters long, this variable will be assigned the value `true`.

Finally, we declare an `if` statement, using the OR operator (`||`) to create the condition based on the two Boolean variables ❶. Since at least one of the variables is `true`, the `if` statement condition passes, and a message prints indicating the password is insecure:

Your password does not meet minimal security requirements.

Try changing the value of `$password` to be a string six characters or longer (other than 'password')—for example, "red\$99poppy". Then neither `$passwordContainsPassword` nor `$passwordTooShort` will be `true`, so the logical OR test in the `if` statement will be `false` and no message will be printed out.

XOR

An XOR operation (short for *exclusive OR*) is `true` when only one of the operands is `true` but not both. We use keyword `xor` to create an XOR expression. [Listing 4-11](#) illustrates an `if...else` statement using an XOR operation. The code determines whether a dessert is creamy but not too creamy. (Custard *and* ice cream would be too much!)

```
<?php
$containsIceCream = true;
$containsCustard = false;
if ($containsIceCream xor $containsCustard) {
    print 'a nice creamy dessert';
} else {
    print 'either too creamy or not creamy enough!';
}
```

Listing 4-11: An if...else statement with the xor operator

We declare two Boolean variables, `$containsIceCream` and `$containsCustard`, setting one to `true` and the other to `false`. Then we declare an `if...else` statement with the condition `$containsIceCream xor $containsCustard`. Thanks to the XOR operator, if one but not both of these variables is `true`, the condition will evaluate to `true`, and a nice creamy dessert will be printed out. If neither variable is `true`, or if both variables are `true`, then the XOR expression will be `false`, and `either too creamy or not creamy enough!` will be printed instead.

In this example, since only one variable is `true`, we should get the `nice creamy dessert` message. Try playing with the values of the two Boolean

variables and see how the result of the XOR expression is affected.

switch Statements

A `switch` statement is a conditional structure that tests a variable against several possible values, or *cases*. Each case has one or more statements to be executed if its value matches the variable's (after type juggling, so it performs equality tests like `==`). You can also provide a default case if none of the values match. If you need to choose from three or more possible paths, a `switch` statement is a convenient alternative to an `if...elseif...else` statement, as long as the decision hinges on the value of a single variable.

[Listing 4-12](#) shows a `switch` statement that prints an appropriate message about the local currency based on the value of the `$country` variable.

```
<?php
$country = 'Ireland';

❶ switch ($country) {
    ❷ case 'UK':
        print "The pound is the currency of $country\n";
        break;
    ❸ case 'Ireland':
    case 'France':
    case 'Spain':
        ❹ print "The euro is the currency of $country\n";
        break;
    case 'USA':
        print "The dollar is the currency of $country\n";
        break;
    ❺ default:
        print "(country '$country' not recognized)\n";
}
```

Listing 4-12: Using a `switch` statement to print the currency based on the value of `$country`

First, we assign `$country` the value `'Ireland'`. Then we begin a `switch` statement with the keyword `switch` followed by the variable to be tested in parentheses (`$country`) ❶. The remainder of the `switch` statement is enclosed in a set of curly brackets. Within the `switch` statement, we declare the values of `$country` to check, each in its own indented `case` clause. Each `case` clause is defined using the keyword `case`, followed by the value to be tested, followed by a colon (`:`). Then come the statements to execute if that case is a match on new, further indented lines. For example, if the value of `$country` is `'UK'` ❷, the message `The pound is the currency of UK` will print.

If you want the same set of actions to apply to multiple cases, list those cases one after the other, followed just once by the statement(s) to execute. For example, Ireland, France, and Spain all use the euro, so we've listed those cases in sequence ❸. The `print` statement after those cases ❹ will apply to any of them; you don't need to repeat it for each case.

Our script features an additional case for when `$country` has the value `'USA'`. Then the final part of the `switch` statement declares a default case using the `default` keyword rather than `case` ❺. This default will be executed if none of the other cases match the variable being tested. Given that we set `$country` to `'Ireland'`, the script should output the message `The euro is the currency of Ireland`.

Notice that we've included the `break` keyword in each case's statement group, after each `print` statement. This interrupts, or *breaks out of*, the `switch` statement, preventing any further code in that statement from being executed. The role of `break` statements is essential to understand. Once a matching case has been found, all remaining statements in the body of the `switch` statement are executed, even statements from other, nonmatching cases, unless a `break` statement is encountered to interrupt the execution. If we removed all the `break` statements from [Listing 4-12](#), for example, we'd end up with the following output:

```
The euro is the currency of Ireland
The dollar is the currency of Ireland
(country 'Ireland' not recognized)
```

The value of `$country` is `'Ireland'`, not `'UK'`, so the first case isn't a match, and the first `print` statement is skipped. Once we encounter the `'Ireland'` case, however, the remaining three `print` statements execute, since there aren't any `break` statements to interrupt the `switch` statement. This is rarely the behavior you'll want from a `switch` statement, so in almost every situation, you'll need to add a `break` statement to the end of each case (or set of cases), as we've done in [Listing 4-12](#).

match Statements

A `match` statement chooses a value for a variable based on the value of another variable. You could accomplish the same task with a `switch` statement, but `match` statements are written much more compactly. Also, `match` statements rely on strict comparisons (the equivalent to testing for identity with `===`), whereas `switch` statements make comparisons after any relevant type juggling (the equivalent to testing for equality with `==`). Therefore, the time to use a `match` statement over a `switch` statement is when a variable needs to be tested against multiple values of the same type, and when the action to perform based on that test is to assign a value to a variable.

[Listing 4-13](#) shows the same logic as [Listing 4-12](#)'s `switch` statement, implemented with a `match` statement instead.

```
<?php
$country = 'Ireland';

❶ $currency = match ($country) {
    'UK' => 'pound',
    'Ireland' => 'euro',
    'France' => 'euro',
    'Spain' => 'euro',
    'USA' => 'dollar',
    ❷ default => '(country not recognized)'
};

print "The currency of $country is the $currency";
```

Listing 4-13: Using a `match` statement to set `$currency` based on the value of `$country`

We write the `match` statement as part of the assignment of the `$currency` variable ❶. It consists of the `match` keyword, followed by the variable to check in parentheses, followed by a comma-separated sequence of *arms* enclosed in curly brackets. Each arm is written in the form `x => y`, where *y* is the value to assign to `$currency` if the value of `$country` matches *x*. As with the `switch` statement, we provide a default arm in case none of the values match ❷. After the `match` statement, we print out a message including the values of `$country` and `$currency`.

Compared to the `switch` statement in [Listing 4-12](#), this `match` statement is more concise. After assigning a value to `$currency`, we have to write only a single `print` statement, as opposed to including a separate `print` statement for each case of the `switch` statement. We also no longer need all the `break` statements; with a `match` statement, once a match has been found, the rest of the statement is ignored.

The `match` statement is a relative newcomer to the PHP language. Many experienced programmers still use `switch` where `match` would be more efficient. (I'm guilty of this myself sometimes.) In general, if you're testing a variable against multiple values, I recommend trying a `match` statement first. Only if that solution is found inadequate should you change to a `switch` statement.

The Ternary Operator

PHP's *ternary operator* (or *three-part operator*) selects one of two values, depending on whether a test is `true` or `false`. The operator consists of two separate symbols, a question mark (?) and a colon (:), and is written in the following form:

```
booleanExpression ? valueIfTrue : valueIfFalse
```

To the left of the question mark, you write a Boolean expression that evaluates to `true` or `false` (for example, comparing two values). To the right of the question mark, you write two values separated by a colon. If the Boolean expression is `true`, the value to the left of the colon is chosen

(*valueIfTrue*). If the Boolean expression is *false*, the value to the right of the colon is chosen (*valueIfFalse*). Usually, the result is assigned to a variable.

Essentially, the ternary operator provides a more succinct way to write an *if...else* statement, as long as the purpose of the *if...else* statement is to assign a value to a variable (as opposed to performing another sequence of actions). To illustrate, [Listing 4-14](#) shows two ways to choose between two values for *\$currency* based on the value of *\$region*: first using an *if...else* statement and then using the ternary operator.

```
<?php
$region = 'Europe';

❶ if ($region == 'Europe') {
    $currency = 'euro';
} else {
    $currency = 'dollar';
}

print "The currency of $region is the $currency (from if...e
lse statement)\n";

$region = 'USA';
❷ $currency = ($region == 'Europe') ? 'euro' : 'dollar';

print "The currency of $region is the $currency (from ternar
y operator statement)\n";
```

Listing 4-14: Comparing if...else and ternary operator statements

We assign *\$region* the value 'Europe'. Then we declare an *if...else* statement that sets the value of *\$currency* to 'euro' if the region is 'Europe' or to 'dollar' otherwise ❶. We print out a message to verify the result. Next, we change *\$region* to 'USA' and reassign *\$currency* by using the ternary operator ❷. The ternary operator expression follows the same logic as the *if...else* statement: if *\$region* equals 'Europe', the code sets *\$currency* to 'euro', and if not, the code sets *\$currency* to 'dollar'.

Again, we print a message to check the result. Here's the output of running the script:

```
The currency of Europe is the euro (from if...else statement)
The currency of USA is the dollar (from ternary operator statement)
```

The second line shows that the ternary operator has worked as expected, assigning 'dollar' as the value of `$currency` because the value in `$region` wasn't 'Europe'. As you can see, in a case like this that requires a straightforward decision between two possible values, the ternary operator is concise, using just one line of code compared to the four lines of the `if...else` statement.

The Null-Coalescing Operator

Another operator that chooses between two values is the *null-coalescing operator*, indicated with a double question mark (`??`). This operator makes its choice depending on whether a variable is `NULL`. The general form of an expression using the null-coalescing operator is as follows:

```
$variable = value ?? valueIfNull
```

First, the null-coalescing operator checks `value`, the expression on the left of the `??` operator. This could be a variable, or perhaps a function that returns a value. If this expression isn't `NULL`, then `value` is assigned to `$variable`. Otherwise, the value to the right of the null-coalescing operator (`valueIfNull`) is assigned to the variable instead. This provides a fallback in case a variable hasn't been defined (or contains `NULL`), without raising a warning or error. This mechanism is especially useful when you're expecting a value from a user but none has been provided, or when you're looking for a record in a database and the record doesn't exist.

[Listing 4-15](#) shows the null-coalescing operator in action. We use it to test the `$lastname_from_user` variable twice, first before it's been assigned

any value (and therefore is `NULL`), then a second time after it's been given a value.

```
<?php
❶ $lastname = $lastname_from_user ?? 'Anonymous';
   print "Hello Mr. $lastname\n";

   $lastname_from_user = 'Smith';
❷ $lastname = $lastname_from_user ?? 'Anonymous';
   print "Hello Mr. $lastname\n";
```

Listing 4-15: Testing for `NULL` with the null-coalescing operator

First, we use the null-coalescing operator to set the value of `$lastname` ❶. The operator tests the `$lastname_from_user` variable, which hasn't been assigned a value yet, and so is `NULL`. Therefore, `$lastname` should be assigned the value to the right of the `??` operator (the string `'Anonymous'`). We print out a message to check the result. Then, after assigning a value to `$lastname_from_user`, we use the same null-coalescing operator expression to again set the value of `$lastname` ❷. This time, since `$lastname_from_user` contains a non-`NULL` value, that value should be passed along to `$lastname`. Here's the result:

```
Hello Mr. Anonymous
Hello Mr. Smith
```

The first line shows that, since variable `$lastname_from_user` is `NULL`, `$lastname` is assigned the string `'Anonymous'`. The second time around, however, the string `'Smith'` inside `$lastname_from_user` is successfully stored in the `$lastname` variable and printed out.

Summary

In this chapter, you've learned about the keywords and operators for writing code that makes decisions. Much of the power of computers and programming languages is built upon the kinds of operators and choice statements we've discussed. You saw how `if` and `if...else` statements

make a choice based on a single test, although that test may itself combine Boolean expressions with logical operators such as AND or OR. You also saw how to incorporate multiple tests by adding `elseif` branches between the `if` and the `else`. Then you learned about other conditional structures, including `switch` and `match` statements, that test a variable for different possible values. These structures let you define one or more statements to be executed when a particular value is found. Closely related to these are the ternary and null-coalescing operators, which both choose between two possible values.

Exercises

1. Write a script that assigns a name to the `$name` variable and then prints the message `That is a short name` if the length of the string is less than four characters.
2. Write a script that determines the size of the machine you need for your laundry. The script should check the value of the `$laundryWeightKg` variable and print `Fits in standard machine` if the value is less than 9, or print `Needs medium to large machine` otherwise.
3. Use a `switch` statement or a `match` statement to test the value of the `$vehicle` variable and print an appropriate message based on that value. Use the following value/message combinations:

```
bus "Beep beep"
```

```
train "Runs on tracks"
```

```
car "Has at least three wheels"
```

```
helicopter "Can fly"
```

```
bicycle "You never forget once you've learned"
```

```
(None of the above) "You've chosen the road less traveled"
```

4. Write a script that prints the message `You are now logged in` if both `$userNameCorrect` and `$passwordCorrect` are true. Otherwise, print `Invalid credentials, please try again`.

5

CUSTOM FUNCTIONS



In this chapter, you'll learn how to declare and use your own *functions*, which are named, self-contained sequences of code that accomplish a particular task. You'll see how functions promote code reusability, since putting code in a function is much more efficient than having to rewrite the same sequence of code every time you need to perform that function's task. Functions also let you write programs that achieve a lot with a small number of statements, since each statement can invoke the complex logic hidden within one of your functions.

Custom functions are typically declared in a separate file from the main program statements that an application will execute. This stems from the *PHP Standards Recommendations (PSRs)*, a list of guidelines and best practices for PHP programming. According to PSR-1, a file should either declare symbols (such as functions) or cause side effects, but not both. A

side effect is a concrete outcome of executing a piece of code, such as outputting text, updating a global variable, changing the contents of a file, and so on.

While functions themselves can cause side effects such as these, *declaring* a function (defining what the function will do) isn't the same as *calling* the function (having the function actually do that thing). Therefore, functions should be declared in one file and called in another. To adhere to this guideline, this chapter first touches on the basics of how to work with code spread across multiple files before we turn our attention to functions. We'll revisit the topic of working with files in more detail in [Chapter 9](#).

Separating Code into Multiple Files

Even if we set aside the best practice of declaring functions in a separate file, it's still standard to break up an application's code across multiple files. Consider that a sophisticated application might consist of tens of thousands of lines of code. If all that code were in a single large text file, navigating that file and locating a particular section of code to work on would be difficult. Organizing code into different files makes a project much more manageable.

Using multiple files also promotes code reusability. Once you start writing your own functions, you'll see how declaring those functions in separate files makes it easy to reuse the function in different parts of a project or in different projects altogether. To give another example, multipage web applications often include the same elements, such as HTML headers, footers, and navigation lists on many pages. Rather than repeating that code for each page that needs it, the common code can be written once in its own file. This way, if you need to change something about it (for example, updating the image reference for a web logo), you need to make the change in only one place instead of tracking down and updating every instance of the repeated code. Software engineers call this the *don't repeat yourself (DRY)* principle.

Once you start spreading an application's code across multiple files, you need a way to access one file's code from within another file. In this section, we'll look at some PHP language features that make this possible.

Reading in and Executing Another Script

PHP's `require_once` command reads the code in another file and executes it. To see how this command works, we'll create two scripts. One, the main script, will use `require_once` to access the code from the other script. First, create a *main.php* file containing the code shown in [Listing 5-1](#).

```
<?php
print "I'm in main.php\n";

require_once 'file2.php';

print "I'm back in main.php\n";
```

Listing 5-1: A main script to read in and execute code from a different script

In this script, we print out two messages indicating that we're in the main application file. In between, we use the `require_once` command to read in and execute the contents of the *file2.php* script. The filename is specified as a string immediately after the command. Since we haven't specified a directory path along with the filename (for example, *Users/matt/file2.php*), it's understood that the file is in the same folder as this current script. This is known as a *relative path*: the file's location is determined relative to the location of the current script.

Now create *file2.php* containing the code shown in [Listing 5-2](#). Be sure to save this file in the same location as *main.php*.

```
<?php
print "\t I'm printing from file2.php\n";
print "\t I'm also printing from file2.php\n";
```

Listing 5-2: The contents of file2.php to be read in and executed from another script

This script has two `print` statements, printing out messages saying they're from *file2.php*. Notice that each message begins with a tab escape character (`\t`). This way, these messages will be indented, whereas the messages printed from our main script won't be, a visual clue that the messages are coming from separate scripts.

Now enter `php main.php` at the command line to run the main script. Here's the output:

```
I'm in main.php
    I'm printing from file2.php
    I'm also printing from file2.php
I'm back in main.php
```

We see the first message from the main script, followed by the two indented messages from *file2.php*. This confirms that the contents of *file2.php* were read in and executed thanks to the `require_once` statement in our main script. Finally, the program flow of control returns back to the main script after the `require_once` statement, and we see the final printed message from the main script.

NOTE

Besides `require_once`, PHP provides three other commands for reading in and executing code declared in a separate file: `require`, `include`, and `include_once`. They all work similarly; you can read about the differences in the PHP documentation. In 99.99 percent of the web applications I write, I use `require_once`.

Creating Absolute Filepaths

The constant `__DIR__` will always refer to the *absolute filepath* to the script currently being executed, meaning the complete filepath, starting from the root directory. This is one of PHP's *magic constants*, built-in constants whose value changes depending on the context. In the case of `__DIR__`, the value varies based on the location of the file in which `__DIR__` is being evaluated.

It's best to use `__DIR__` whenever possible when writing `require_once` statements: simply concatenate the value of `__DIR__` with any remaining relative path information to access the file you're trying to read in and execute. This avoids any confusion as to whether the path relates to the current script (the one calling the `require_once` command) or to a script that might have required the current script. Consider that you might have a

chain of scripts, with one script requiring another, and that script also requiring another. If these scripts were in different directories, using the `__DIR__` magic constant ensures that wherever you write a `require_once` statement, you'll know the path will be correct to the files you wish to read in and execute.

To try using `__DIR__`, update your *main.php* file as shown in [Listing 5-3](#). The changes are shown in black text.

```
<?php
print "I'm in main.php\n";

$callingScriptPath = __DIR__;
print "callingScriptPath = $callingScriptPath\n";

❶ require_once __DIR__ . '/file2.php';

print "I'm back in main.php\n";
```

Listing 5-3: A main script using `__DIR__` to read in and execute code in a different script

We assign the `$callingScriptPath` variable the value of the `__DIR__` magic constant and print a message containing this variable. Then we use `__DIR__` after the `require_once` command to make it explicit that the *file2.php* script resides in the same directory as this main script ❶. Notice that we use the string concatenation operator (`.`) to combine the value of `__DIR__` with the string `'/file2.php'`, building an absolute path to the other file. Here's the output of running the main script:

```
I'm in main.php
❶ callingScriptPath = /Users/matt/magic
    I'm printing from file2.php
    I'm also printing from file2.php
I'm back in main.php
```

As before, the first message from *main.php* prints out. Then we see the path to the main script (the value of `__DIR__`) printed out ❶. For me, it is */Users/matt/magic*, the path to the directory on my computer for this

example project. The rest of the output is the same as before, featuring the messages from *file2.php* followed by the final printed message from the main script.

Declaring and Calling a Function

Now let's turn our attention to declaring and using our first custom function. The function will determine which of two numbers is smaller. In keeping with best practices, we'll declare the function in one file, *my_functions.php*, and then call it from a separate file, *main.php*. Start a new project and create *my_functions.php* containing the code shown in [Listing 5-4](#).

```
<?php
function which_is_smaller(int $n1, int $n2): int
{
    if ($n1 < $n2) {
        return $n1;
    } else {
        return $n2;
    }
}
```

Listing 5-4: Declaring a function in my_functions.php

Here we declare a function named `which_is_smaller()`. We begin with the keyword `function`, followed by the function name. By convention, function names are written in snake case, in all lowercase letters and with underscores to join multiple words. This enables you to write meaningful, easy-to-read function names (although, unfortunately, not all of PHP's built-in functions follow this naming convention because of choices made in the language's early design).

After the function name comes a set of parentheses containing a comma-separated list of the function's *parameters*. These are inputs that the function needs to do its job. In this case, we have two parameters, `$n1` and `$n2`, representing the two numbers we want the function to compare. Each parameter name is preceded by its data type to ensure that the correct form

of data enters the function. Here, for example, `int $n1` indicates that parameter `$n1` should be an integer.

NOTE

If a function doesn't need any parameters, you still have to include an empty set of parentheses after the function name.

After the parentheses comes a colon (:), followed by the function's return type. Most functions do some work and produce a value as a result, which the function then *returns*, or provides, to the script that called the function. The *return type* specifies the data type of this value. In this case, the function will return the integer `$n1` or `$n2`, whichever is smaller, so we set the return type to `int`.

The code we've written so far has defined the function's *signature*, a combination of its name, parameters (and their types), and return type. The PHP engine uses a function's signature to uniquely identify the function, recognize when we're calling it, validate that appropriate data is being passed to the function's parameters, and ensure that the function is returning an appropriate value.

Next comes the *body* of the function, a statement group enclosed within curly brackets and containing the code that will execute each time the function is called. The body of our `which_is_smaller()` function consists of an `if...else` statement that tests whether integer `$n1` is smaller than integer `$n2`. If `$n1` is smaller, the `return $n1;` statement will be executed. Otherwise (if `$n2` is smaller or the same as `$n1`), `return $n2;` will be executed. In both cases, we use the `return` keyword to make the function provide a value (either `$n1` or `$n2`) to the script that called it. As soon as a function reaches a `return` statement, the function stops executing and gives control back to the calling script. Even if the function body includes additional statements after the `return` statement, they won't execute after the function has returned a value.

Now that we've declared a function, let's use it. Create *main.php* in the same location as *my_functions.php* and enter the code shown in [Listing 5-5](#).

```
<?php
require_once __DIR__ . '/my_functions.php';

$result1 = which_is_smaller(5, 2);
print "the smaller of 5 and 2 = $result1\n";

$result2 = which_is_smaller(5, 22);
print "the smaller of 5 and 22 = $result2\n";
```

Listing 5-5: Calling the `which_is_smaller()` function from `main.php`

We use `require_once` to read in the declaration of our function from *my_functions.php*. This doesn't call the function; it simply makes the function available for use in our *main.php* script. Next, we call our function by writing the function name, followed in parentheses by the values we want the function to compare, 5 and 2. These values are known as *arguments*; they fill in the values of the function's parameters. Notice that we call the function as part of an assignment statement for the `$result1` variable. This way, the function's return value will be stored in `$result1` for later use (in this case, in the next line of code, where it's printed out in a message). When a function has a return value, it's common to follow this pattern of calling a function and assigning the result to a variable.

We conclude the script by calling the function again, this time using 5 and 22 as arguments. This is the beauty of functions: you can call them as many times as you want, with different input values each time. We store the return value of the second function call in the `$result2` variable and again print out a message showing the result. Here's the output of running the *main.php* script:

```
the smaller of 5 and 2 = 2
the smaller of 5 and 22 = 5
```

We can see that our function is working correctly. It returns 2 as the smaller of 5 and 2, and 5 as the smaller of 5 and 22.

Parameters vs. Arguments

The terms *parameter* and *argument* are closely related and often mistaken for each other. When you *declare* a function, the parameters are variables that stand in for the inputs the function will work with. As you saw in [Listing 5-4](#), you list the parameters in the parentheses after the function name. In our `which_is_smaller()` function, the parameters were `$n1` and `$n2`. Each parameter is a temporary variable, local to the function code itself, that will be assigned a value when the function is called. These variables exist only while the function is being executed. Once the function has finished executing, the local parameter variables are discarded from the computer's memory.

The technical term for how long a variable “lives” in a software system is *scope*. The scope of any variable declared in a function, including a parameter, is local to the function itself. As such, you can't expect to access a function's variables from any code outside the function declaration. In our example, we can't use the variables `$n1` and `$n2` in *main.php*. Instead, the way to get a value out of a function is with a `return` statement.

When we *call* a function, the arguments are the specific values we pass to the function in the parentheses after the function name. These arguments supply the values for the function's parameters. When we call `which_is_smaller(5, 22)`, for example, the argument 5 is assigned as the value of parameter `$n1`, and the argument 22 is assigned as the value of parameter `$n2`. The order of arguments matches the order of parameters. In this case, the arguments are literals, but arguments can also be variables, as shown here:

```
which_is_smaller($applesCount, $orangesCount)
```

That's all there is to it. Arguments are the values passed when executing a function, and parameters are the local variables created when the function executes, populated by the arguments received. Each argument passed to a function will therefore have a corresponding local (temporary) parameter variable while that function is executing. (One exception is the special case of pass-by-reference parameters, which we'll cover later this chapter.)

Errors from Incorrect Function Calls

In two common cases, you'll get an error when calling a function: if you don't pass the correct number of arguments, or if you pass arguments of the wrong data type. (See "Errors, Warnings, and Notices" on [page 88](#) for information on errors and other kinds of alerts generated about your code.) Consider this call to our custom `which_is_smaller()` function:

```
$result = which_is_smaller(3);
```

The function requires two integer arguments, but we're providing only one. If you try to execute this expression, the application will halt and you'll see a fatal error similar to the following:

```
PHP Fatal error:  Uncaught ArgumentCountError: Too few arguments to function
which_is_smaller(), 1 passed in /Users/matt/main.php on line 9 and exactly 2
expected in /Users/matt/my_functions.php:2
```

You'll also get a fatal error if you pass arguments of the wrong data type (that is, values that can't be type-juggled into the parameter data types specified in the function declaration). Consider this expression, where we pass non-numeric strings to our `which_is_smaller()` function:

```
$result = which_is_smaller('mouse', 'lion');
```

Trying to execute this statement will produce an error message like the following:

```
PHP Fatal error:  Uncaught TypeError: which_is_smaller(): Argument #1 ($n1)
must be of type int, string given, called in /Users/matt/main.php on line 10
and defined in /Users/matt/my_functions.php:2
```

A fatal `TypeError` has occurred because our function requires two integer arguments but we've provided strings instead.

ERRORS, WARNINGS, AND NOTICES

PHP communicates that it has identified issues in your code in three ways. In order from most to least severe, they are errors, warnings, and notices.

A *fatal runtime error* indicates that a problem that can't be recovered from has occurred while code is being executed. In this case, the PHP engine immediately terminates the execution of the program. An example of a fatal runtime error is attempting to execute a function that the PHP engine can't find (perhaps the function name is misspelled, or the statement to read in the file containing the function is missing). Errors can also occur when the PHP engine is parsing, rather than executing, a file containing code, such as when the PHP engine reads in a function declaration from a file. These *parse errors* can be triggered, for example, by a missing semicolon or closing quotation mark in a function declaration. Fixing errors is crucial, or your code won't run.

Warnings identify nonfatal issues, meaning the PHP engine is able to continue executing your code in spite of the problem. For example, PHP will warn you if you're using a variable that hasn't been assigned a value, but it won't halt the program. Instead, it'll give the variable a default value such as `NULL` and continue executing the code. You should always heed warnings and take steps to fix your code so the issue highlighted in the warning is resolved.

A *runtime notice* indicates something has occurred during code execution that *could* indicate an error but that might also be intended behavior of correctly running code. An example is a *deprecation message*, a notice about a language feature that still works in the current version of the PHP engine but will be removed in a future version. These messages help you monitor your software and plan ahead for a feature eventually becoming unavailable. In fact, since PHP 8, many deprecation messages have been upgraded from notices to warnings as part of the quality improvements to encourage modern PHP programmers to write more robust and correct code.

Type Juggling

To avoid a `TypeError` like the one we just saw when arguments of the wrong type are provided, the PHP engine attempts to juggle those arguments into the expected data type. (For a refresher on type juggling, see [Chapter 2](#).) [Listing 5-6](#) shows some examples where we provide non-integer arguments to our `which_is_smaller()` function. Update your *main.php* file to match the listing.

```
<?php
require_once __DIR__ . '/my_functions.php';

$result1 = which_is_smaller(3.5, 2);
print "the smaller of 3.5 and 2 = $result1\n";

$result2 = which_is_smaller(3, '55');
print "the smaller of 3 and '55' = $result2\n";

$result3 = which_is_smaller(false, -8);
print "the smaller of false and -8 = $result3\n";
```

Listing 5-6: Updating the main.php script to demonstrate type juggling

We call `which_is_smaller()` three times and print the results. None of these function calls will trigger an error, since the arguments can all be juggled to integers. First, we call the function with float 3.5 and integer 2. The float will be juggled to integer 3. Next, we use integer 3 and string '55' as arguments. This time, the string will be converted to integer 55. Finally, we pass Boolean `false` and integer -8 as arguments. The `false` will be converted to integer 0. Here's the output of running the script:

```
PHP Deprecated:  Implicit conversion from float 3.5 to int l
oses precision in
/Users/matt/my_functions.php on line 2

the smaller of 3.5 and 2 = 2
the smaller of 3 and '55' = 3
the smaller of false and -8 = -8
```

When you run the script, the first thing you should see printed out is a deprecation message informing you that you're losing precision when float 3.5 is juggled into integer 3. This message indicates that at some point in the future (possibly PHP 9), PHP will stop automatically juggling floats with fractional components into integers, so the code will someday stop working and trigger an error. After this message, you should see the results

of the three `print` statements, indicating that the three function calls occurred without issue, thanks to PHP’s automatic type juggling.

NOTE

When you encounter a deprecation message, reading a discussion about the upcoming change can be informative. For example, the request for comments (RFC) document explaining the deprecation message output from [Listing 5-6](https://wiki.php.net/rfc/implicit-float-int-deprecate) is available online at <https://wiki.php.net/rfc/implicit-float-int-deprecate>.

These function calls worked despite the incorrect argument data types, but well-written programs should avoid relying on type juggling altogether. Take note of deprecation warnings like the one we just encountered, and look for ways to revise your code to cope with different kinds of values without warnings or errors. In this particular case, we could refactor the function to use union types (discussed in “Union Types” on [page 98](#)), which would allow both integers and floats as arguments.

Functions Without Explicit Return Values

Not every function has to explicitly return a value. For example, you could write a function that simply prints out a message without returning anything to the calling script. When a function doesn’t have an explicit return value, declare its return type as `void`.

To demonstrate, we’ll declare a function that prints out a given number of stars, padded on both sides with another spacer character to achieve a fixed line length. We’ll be able to use the function to create ASCII art, images formed by arranging characters of text. Start a new project and create *my_functions.php* containing the code shown in [Listing 5-7](#).

```
<?php
function print_stars(int $numStars, string $spacer): void
{
    $lineLength = 20;
    $starsString = str_repeat('*', $numStars);
    $centeredStars = str_pad($starsString, $lineLength, $spa
cer, STR_PAD_BOTH);
```

```
    print $centeredStars . "\n";  
}
```

Listing 5-7: Declaring the `print_stars()` function in `my_functions.php`

Here we declare a function named `print_stars()`. The function requires two parameters: `$numStars` and `$spacer`. The integer `$numStars` is the number of stars (`*` characters) to be printed out. The string `$spacer` is the character to use as padding on both sides of the stars. After the parentheses, we use `: void` to indicate that this function won't explicitly return any value.

Inside the function body, we set the length of the line to be printed to 20 characters. (Since this value is *hardcoded* into the function, it will be the same each time the function is called; a more flexible alternative could be to set `$lineLength` as a parameter.) Then we generate a string (`$starsString`) containing the number of asterisks specified by the `$numStars` parameter. Next, we use the built-in `str_pad()` function (discussed in [Chapter 3](#)) to create a string 20 characters long, with `$starsString` centered and padded symmetrically on the left and right with whatever string is in the `$spacer` parameter. If `$numStars` is 10 and `$spacer` is `'.'`, for example, this will produce the string `'.....*****.....'`, 10 asterisks with 5 periods on each side, giving a total length of 20. Finally, we print out the result, followed by a newline character.

Notice that we haven't included a `return` statement in the function body. There's no need, since all the function is doing is constructing and printing a string. If we were to try to return a value from this function, it would trigger a fatal error, since we declared the function as `void`.

Now let's use our function to generate an ASCII art image of a tree. Create *main.php* containing the code shown in [Listing 5-8](#).

```
<?php  
require_once __DIR__ . '/my_functions.php';  
  
❶ $spacer = '/';  
   print_stars(1, $spacer);  
   print_stars(5, $spacer);
```

```
print_stars(9, $spacer);
print_stars(13, $spacer);
print_stars(1, $spacer);
print_stars(1, $spacer);
```

Listing 5-8: A script in main.php to generate a tree shape with the `print_stars()` function

After reading in the function declaration with `require_once`, we set the spacer character to be a forward slash (/) ❶. Then we call our `print_stars()` function six times, printing a tree shape made up of lines with 1, 5, 9, and 13 stars, plus two more lines with just 1 star for the trunk. Here's the output of running the *main.php* script at the terminal:

```
//////////*//////////
//////////*****//////////
//////////*****//////////
///*****//////////
//////////*//////////
//////////*//////////
```

We've created a tree during a heavy rainstorm!

Returning NULL

Even when a function is declared as `void`, it still technically has a return value: `NULL`. If a function finishes executing without returning a value, the function returns `NULL` by default. To prove it, let's try calling our `print_stars()` function again and assigning the result to a variable, as we would with a function that has a return value. Update your *main.php* file to match [Listing 5-9](#). The changes are shown in black text.

```
<?php
require_once __DIR__ . '/my_functions.php';

$spacer = '/';
print_stars(1, $spacer);
print_stars(5, $spacer);
print_stars(9, $spacer);
print_stars(13, $spacer);
```

```
print_stars(1, $spacer);
$result = print_stars(1, $spacer);

var_dump($result);
```

Listing 5-9: Updating main.php to store and print the `print_tree()` function's `NULL` return value

We make the same calls to the `print_stars()` function as before, but this time we store the return value of the last function call in the `$result` variable. We then use `var_dump()` to see the contents of `$result`. Since `print_stars()` doesn't have an explicit return value, `$result` should contain `NULL`. Here's the output of running the *main.php* script:

```
//////////*//////////
//////////*****//////////
//////////*****//////////
///*****//////////
//////////*//////////
//////////*//////////
NULL
```

We can see the ASCII tree again, followed by `NULL` from the call to `var_dump()`. This confirms that the function has returned `NULL` by default, despite being declared as `void`.

Exiting a Function Early

A function declared as `void` can still use a `return` statement, as long as the statement doesn't include a value. As mentioned earlier, a function stops executing as soon as it encounters a `return` statement, so writing `return` without a value provides a mechanism for exiting a function early. This can be useful, for example, if a problem occurs with one of the function's parameters. You can add validation logic to check the parameters at the start of the function, and use `return` to halt the function execution and resume the main calling script if one or more argument values aren't as expected.

The `str_pad()` function we've been using to create centered lines of stars will trigger a fatal error if the padding string is empty. Rather than let

that crash our program, let's update our `print_stars()` function to first check whether the `$spacer` string parameter is empty. If it is, we'll use `return` to exit the function early. Modify *my_functions.php* to match [Listing 5-10](#).

```
<?php
function print_stars(int $numStars, string $spacer): void
{
    if (empty($spacer)) {
        return;
    }
    $lineLength = 20;
    $starsString = str_repeat('*', $numStars);

    $centeredStars = str_pad($starsString, $lineLength, $spa
cer, STR_PAD_BOTH);
    print $centeredStars . "\n";
}
```

Listing 5-10: Adding a `return` statement to exit the `print_stars()` function early

We add an `if` statement to the start of the function body, using the built-in `empty()` function to test whether `$spacer` is an empty string. If so, we use `return` without any value to end function execution early and return program control to the calling script. If the function execution gets past this `if` statement, then we know that `$spacer` isn't empty, so our call to `str_pad()` should work fine.

To see whether the `return` statement is working, update the *main.php* script as shown in [Listing 5-11](#).

```
<?php
require_once __DIR__ . '/my_functions.php';

$spacer = '';
print_stars(1, $spacer);
print_stars(5, $spacer);
print_stars(9, $spacer);
print_stars(13, $spacer);
```

```
print_stars(1, $spacer);  
$result = print_stars(1, $spacer);  
  
var_dump($result);
```

Listing 5-11: Updating main.php to call `print_tree()` with an empty spacer string

We set `$spacer` to an empty string rather than a slash before making our calls to `print_stars()`. The output of running the main script should now simply be `NULL`. The `print_stars()` function returns early each time it is called because `$spacer` is an empty string, so we no longer see our ASCII tree. Then again, we don't see a fatal error either, because our `return` statement prevents us from calling `str_pad()` with an invalid argument. We still see `NULL` in the output, the result of the `var_dump()` call. This indicates that when a function encounters a `return` statement without a value, it returns `NULL`, just as it would if it didn't have a `return` statement at all.

Calling Functions from Within Functions

It's perfectly reasonable to call one function from within the body of another function. In fact, we've done it several times already, calling built-in PHP functions like `str_repeat()` and `str_pad()` inside our `print_stars()` function. It's also possible, and in fact, quite common, to call your own custom functions from within other custom functions.

A lot of the power of programming comes from breaking problems into smaller tasks. You write basic functions to tackle those small tasks and then write higher-level functions that combine the tasks to solve the larger problem. In the end, your main application script looks quite simple: you just call one or two functions. The trick is that those functions themselves call several other functions, and so on.

It took us six calls to our `print_stars()` function to generate an ASCII tree. Let's move those six calls into another function, `print_tree()`. That way, every time we want to print a tree, all we need is one function call in our main script. Add the new `print_tree()` function to *my_functions.php* as shown in [Listing 5-12](#).

```
<?php
function print_stars(int $numStars, string $spacer): void
{
    --snip--
}

function print_tree(string $spacer): void
{
    print_stars(1, $spacer);
    print_stars(5, $spacer);
    print_stars(9, $spacer);
    print_stars(13, $spacer);
    print_stars(1, $spacer);
    print_stars(1, $spacer);
}
```

Listing 5-12: Adding the `print_tree()` function to `my_functions.php`

We declare the `print_tree()` function after our previously declared `print_stars()` function. It requires a string parameter called `$spacer`. In the function body, we write our six original calls to `print_stars()`. Notice that `$spacer`, the parameter of the `print_tree()` function, is also acting as an argument when we call `print_stars()`. This way, we can easily print trees with different padding characters around the asterisks just by changing the string we pass in when we call `print_tree()`.

With this new function, we can now greatly simplify our main script. Update *main.php* as shown in [Listing 5-13](#).

```
<?php
require_once __DIR__ . '/my_functions.php';
print_tree('/');
print_tree(' ');
```

Listing 5-13: Simplifying the `main.php` script with the `print_tree()` function

After reading in the function declaration file, we call `print_tree()` twice to generate two trees. The first time we use a forward slash as the

spacer, as before, and the second time we use a space character. Here's the result:

```
//////////*//////////
//////////*****//////////
//////////*****//////////
///*****//////////
//////////*//////////
//////////*//////////

      *
    *****
  *****
*****

      *
      *
```

Our main script has accomplished with 2 calls to `print_tree()` what would have previously taken 12 calls to `print_stars()`. Of course, those calls to `print_stars()` are still happening, but we've hidden them inside the `print_tree()` definition, making our main script much tidier. You can begin to see the power of functions to organize code and promote reusability.

Functions with Multiple Return and Parameter Types

For straightforward situations, you can usually write a function that does something and returns a value of a single type or that returns no value. Other times, however, you'll want to make a function more reusable by allowing it to return values of different data types depending on the situation. Likewise, you might want a function's parameters to accept values of different data types to ensure that your code can cope with input validation issues. *Nullable types* and *union types* offer elegant ways to permit multiple types, both for a function's return value and its parameters.

Nullable Types

It's quite common to write functions that normally return one kind of value, such as a string or a number, but that sometimes return `NULL` instead. For

example, a function that typically performs a calculation might return `NULL` if it receives invalid inputs, or a function that retrieves information from a database might return `NULL` if it's unable to establish a database connection (we'll see this in [Part VI](#) when we discuss databases). To allow for this, declare the function's return type to be *nullable* by adding a question mark (?) immediately before the return type. For instance, placing `: ?int` at the end of the first line of a function declaration means that the function will return either `NULL` or an integer.

Let's see this in action with a function that attempts to return the integer value of a spelled-out number (such as `1` instead of `'one'`). If the function doesn't recognize the input string, it will return `NULL` instead. Start a new project and create *my_functions.php* containing the contents of [Listing 5-14](#).

```
<?php
function string_to_int(string $numberString): ❶ ?int
{
    return match ($numberString) {
        'one' => 1,
        'two' => 2,
        'three' => 3,
        'four' => 4,
        'five' => 5,
        ❷ default => NULL
    };
}
```

Listing 5-14: A function that returns an integer or `NULL`

We declare the `string_to_int()` function, using the nullable type `?int` to indicate that the function will return either `NULL` or an integer ❶. The function takes in the string parameter `$numberString`. Its body is a single `return` statement that chooses a value to return by using a `match` expression. This is possible because `match` expressions evaluate to a single value. The expression has five clauses matching the strings `'one'` through `'five'` to the corresponding integer. A sixth clause sets the `default` case

❷, returning `NULL` if any other string is provided. In this way, the `match` expression returns an integer or `NULL`, just as the function's nullable return type indicates.

Now we'll write a *main.php* file with a script that calls our function. When you call a function with a nullable return type, it's important to test the return value, in case it's `NULL`. [Listing 5-15](#) shows how.

```
<?php
require_once __DIR__ . '/my_functions.php';

❶ $text1 = 'three';
$number1 = string_to_int($text1);
❷ if (is_null($number1)) {
    print "sorry, could not convert '$text1' to an integer
    \n";
} else {
    print "'$text1' as an integer = $number1\n";
}

$text2 = 'onee';
$number2 = string_to_int($text2);
if (is_null($number2)) {
    print "sorry, could not convert '$text2' to an integer
    \n";
} else {
    print "'$text2' as an integer = $number2\n";
}
```

Listing 5-15: A main.php script calling the nullable-type `string_to_int()` function

We assign string `'three'` as the value of the `$text1` variable, then pass that variable to our `string_to_int()` function, storing the return value in `$number1` ❶. Next, we use an `if...else` statement to test whether the value in `$number1` is empty (`NULL`) ❷. If so, we print a message stating that the string couldn't be converted to an integer. Otherwise, we print a message showing the string and its corresponding integer. We then repeat the process with string `'onee'`. Here's the output:

```
'three' as an integer = 3
sorry, could not convert 'onee' to an integer
```

We can see that the function returns the integer 3 when the argument is the string 'three', but it returns `NULL` when the argument is the misspelled string 'onee'. Declaring our `string_to_int()` function with a nullable return type gives us the flexibility to respond to this problematic input in a meaningful way.

Just as functions can have nullable return types, you can use the same question mark syntax to declare function parameters as nullable, meaning the parameter can be `NULL` or some other type. For example, the parameter list `(?string $name)` means that a function accepts a `$name` parameter that is either `NULL` or a string.

Rather than having to duplicate the `if...else` statement in our *main.php* script each time we call our `string_to_int()` function, as we did in [Listing 5-15](#), we might take the function's `NULL` or integer return value and pass it as an argument to another function to generate an appropriate message. That function therefore needs to be able to accept a parameter that may be `NULL` or an integer. [Listing 5-16](#) shows such a function named `int_to_message()`. Add the function to the end of your *my_functions.php* file.

```
function int_to_message(?int $number): string
{
    if (is_null($number)) {
        return "sorry, could not convert string to an integer\n";
    } else {
        return "an integer = $number\n";
    }
}
```

Listing 5-16: A function with a nullable type for the `$number` parameter

The signature for this function includes a single parameter called `$number` of nullable type `?int`. This means that the argument provided to

the function can be either `NULL` or an integer. The function body uses the `if...else` statement we had in our *main.php* script to return an appropriate message depending on which data type is passed in.

We can now greatly simplify our main script by removing the duplicated `if...else` statements and calling our new function instead.

[Listing 5-17](#) shows the updated script.

```
<?php
require_once __DIR__ . '/my_functions.php';

❶ $text1 = 'three';
❷ $number1 = string_to_int($text1);
❸ print int_to_message($number1);

$text2 = 'onee';
$number2 = string_to_int($text2);
print int_to_message($number2);

❹ print int_to_message(string_to_int('four'));
```

Listing 5-17: Simplifying main.php with the `int_to_message()` function

Notice that our main script is much simpler now that the logic for generating the message has been moved to a function. For each input, we follow a pattern of three basic statements: declaring a string ❶, storing the integer (or `NULL`) returned from calling `string_to_int()` with that string ❷, and printing the string returned by passing this integer or `NULL` value to the `int_to_message()` function ❸.

If we really want to make our code even more succinct, we can put all three of those statements into a single line ❹, calling the `string_to_int()` function inside the parentheses when we call the `int_to_message()` function. This way, the former's return value is passed directly as an argument to the latter, without the need for an intermediary variable. This choice is a matter of programming style. Personally, I prefer to use intermediate variables to prevent a single line of code from becoming too complex.

Union Types

If you want a function to be able to return a range of data types, declare its return value by using a *union type*. This is a list of the value's possible data types, separated by vertical bars. For example, `int|float` indicates that a value could be an integer or a float. Union types can apply to function parameters as well as return values.

Nullable types are essentially a special category of union types, and their question mark syntax provides a convenient shorthand when one of the possible data types is `NULL`. The union type `string|NULL` is the same as the more concise nullable type `?string`, for example. Union types are most useful when your code has multiple non-`NULL` types, like `int|float`, or when there are multiple non-`NULL` types plus `NULL`, like `string|int|NULL`, indicating the data type could be a string, an integer, or `NULL`. This couldn't be expressed with nullable-type syntax, since you can't mix a nullable type with others in a union by writing something like `?string|int`. You also can't include `void` as one of the types in the union.

To demonstrate union types, let's modify our `string_to_int()` function into a `string_to_number()` function that can return an integer, a float, or `NULL`, depending on the string passed in. We'll also update our `int_to_message()` function into a `number_to_message()` function that can take in an integer, a float, or `NULL` as a parameter. Update *my_functions.php* to match [Listing 5-18](#).

```
<?php
function string_to_number(string $numberString): ❶ int|float
|NULL
{
    return match ($numberString) {
        ❷ 'half' => 0.5,
        'one' => 1,
        'two' => 2,
        'three' => 3,
        'four' => 4,
        'five' => 5,
        default => NULL
    };
};
```

```

}

function number_to_message(string $text, ❸ int|float|NULL $number): string
{
    ❹ if (is_int($number)) {
        return "'$text' as an integer = $number\n";
    }

    ❺ if (is_float($number)) {
        return "'$text' as a float = $number\n";
    }

    ❻ return "sorry, could not convert '$text' to a number\n";
}

```

Listing 5-18: Using union types as function return values and parameters

First, we declare `string_to_number()`, a revised version of our `string_to_int()` function. We use the union type `int|float|null` to indicate that the function will return an integer, a float, or `NULL` ❶. Just like `string_to_int()` previously, this function takes in a single string parameter. We add a new clause to the `match` statement in the function body, matching the string `'half'` to the float value `0.5` ❷, hence the need for the union type.

Next, we declare `number_to_message()`, a revised version of `int_to_message()` that returns a string. This function takes in two parameters. The first, the string `$text`, will be the same as the string passed to our `string_to_number()` function. The second, `$number`, will be that function's return value, and so it might be an integer, a float, or `NULL`. We therefore use the same `int|float|NULL` union type for the parameter ❸.

In the function body, we first test whether `$number` contains an integer value ❹, in which case we return a message stating that `$text` is an integer. Next, we test whether `$number` contains a float value ❺, returning an appropriate message if it does. Finally, we return a message stating that `$text` couldn't be converted to a number ❻. Execution wouldn't get this

far if either of the previous `return` statements was executed, so we know at this point that `$number` is neither an integer nor a float. We therefore don't need to place this final `return` statement inside an `else` clause or another `if` statement, although we could.

This choice is a matter of personal programming style. I like to end functions like this with an unconditional `return` statement, so I can clearly see the default to be returned. However, some programmers prefer to end the last `if` statement with an `else` clause as a way to communicate the default. The execution is the same either way.

Now let's test our functions. Update your *main.php* script to match [Listing 5-19](#).

```
<?php
require_once __DIR__ . '/my_functions.php';

$text1 = 'three';
$number1 = string_to_number($text1);
print number_to_message($text1, $number1);
```

Listing 5-19: Calling functions with union type parameters and return values in main.php

We call our `string_to_number()` function, passing in the string `'three'`, and store the result in the `$number1` variable. Then we pass `$number1` along to our `number_to_message()` function and print the message that it returns. This code should output the message `'three'` as an integer = 3.

Optional Parameters

If the value of a parameter will usually be the same each time you call a function, you can set a default value for that parameter when you declare the function. In effect, this makes the parameter optional. You'll need to include an argument corresponding to that parameter only when you know you'll want the value to be something other than the default.

Many of PHP's built-in functions have optional parameters with default values. For example, PHP's `number_format()` function, which takes in a float and converts it into a string, has several optional parameters

controlling how the string will be formatted. Enter `php -a` at the command line to try out the following code in interactive mode:

```
❶ php > print number_format(1.2345);  
1  
❷ php > print number_format(1.2345, 2);  
1.23  
❸ php > print number_format(1.2345, 1, ',');  
1,2
```

The `number_format()` function's first parameter is not optional; it's the float that we want to format. By default, calling the function with just one argument ❶ returns a string version of the number with the decimal portion removed. When we add an integer as an optional second argument ❷, the function uses that integer to set the number of decimal places to include. We've used the value 2 to preserve two decimal places. By default, the decimal separator is represented with a period, but if we add a string as an optional third argument ❸, the function will use that string as the decimal separator instead. In this case, we're using a comma, a common decimal separator in continental Europe.

[Listing 5-20](#) shows the signature for the `number_format()` function, taken from the PHP online documentation, to illustrate how the default values for the parameters are declared.

```
number_format(  
    float $num,  
    int $decimals = 0,  
    ?string $decimal_separator = ".",  
    ?string $thousands_separator = ",",  
): string
```

Listing 5-20: The built-in `number_format()` function, including optional parameters with default values

First, notice that when you have a long list of parameters, you can spread them over several lines to make the code more readable. The function takes up to four parameters, but the second, third, and fourth all

have default values assigned with the assignment operator (=) after the parameter name. For example, the second parameter, `$decimals`, has a default value of 0, so when we call `number_format(1.2345)` without providing a second argument, the function executes with the default value for `$decimals` and formats the number to include zero decimal places. Likewise, the `$decimal_separator` parameter has a period as its default value, and the `$thousands_separator` parameter has a comma.

The order in which the parameters are declared is important. All mandatory parameters (those without default values) must be listed first, followed by the optional parameters. This is because the order of arguments when you call a function must match the order of the parameters. If you had an optional parameter followed by a mandatory one, and you omitted the optional parameter, there'd be no way to know that your first argument was meant to correspond to the second parameter. The only exception to this rule is if you use named arguments, as we'll discuss later in the chapter.

Now that we've seen how optional parameters work, let's add one to a custom function. We'll revisit our `which_is_smaller()` function from earlier in the chapter and add an optional parameter controlling how the function behaves if the values passed in for comparison are the same. Return to the `my_functions.php` file for that project and update the script to match [Listing 5-21](#).

```
<?php
function which_is_smaller(int $n1, int $n2, ❶ bool $nullIfSame = false): ?int
{
    if ($n1 < $n2) {
        return $n1;
    }

    if ($n2 < $n1) {
        return $n2;
    }

    ❷ if ($nullIfSame) {
        return NULL;
    }
}
```

```
    }  
  
    ❸ return $n1;  
}
```

Listing 5-21: Updating the `which_is_smaller()` function to include an optional parameter

We add a third parameter to our function, the Boolean `$nullIfSame`, and give it a default value of `false` ❶. Thanks to this default value, the function will typically return `$n1` if `$n1` and `$n2` are found to be the same ❸. However, if the user overrides this default by passing `true` as the third argument when calling the function, `NULL` is returned instead ❷. To account for this possibility, we use the nullable type `?int` to set the function's return type.

The sequence of `if` and `return` statements matters here. The code will get to `if ($nullIfSame)` ❷ only if `$n1` and `$n2` are equal. Since `$nullIfSame` is `false` by default, this condition will typically fail, so the final `return $n1;` will execute ❸. It's only if the user has set `$nullIfSame` to `true` that the function returns `NULL`.

Update the project's *main.php* file as shown in [Listing 5-22](#) to test the function.

```
<?php  
require_once __DIR__ . '/my_functions.php';  
  
$result1 = which_is_smaller(1, 1);  
var_dump($result1);  
$result2 = which_is_smaller(1, 1, true);  
var_dump($result2);
```

*Listing 5-22: Calling `which_is_smaller()` from *main.php*, with and without the optional parameter*

We call out `which_is_smaller()` twice, using `var_dump()` to show the results. The first time we pass in `1` and `1` and leave out the optional argument, so `$nullIfSame` will be `false` by default. The second time, we

add `true` as a third argument, overriding the default. Here's the output of running the main script:

```
int(1)
NULL
```

The first line indicates that the function followed the default behavior of returning `1` (the value of the first argument) when we omitted the optional argument. When we used the third argument to set `$nullIfSame` to `true`, however, the function returned `NULL`.

Positional vs. Named Arguments

When you call a function, the PHP engine by default interprets the arguments *positionally*, matching them to the function's parameters based on their order. However, you can also call a function by using *named arguments*: you explicitly pair an argument's value with the name of the corresponding parameter. In this case, the order of arguments no longer matters. Named arguments are especially useful when a function has optional parameters.

To use named rather than positional arguments, you don't have to change the function declaration in any way, although it becomes even more important to have meaningful parameter names. Instead, all you have to do is include the parameter name (minus the dollar sign) inside the parentheses when you call a function, followed by a colon (`:`) and the desired argument value. For example, to use a named argument to pass `true` as the value of the `$nullIfSame` parameter when calling our `which_is_smaller()` function, you would include `nullIfSame: true` in the argument list. The convention is to add a space after the colon.

[Listing 5-23](#) shows an updated *main.php* file, adding an extra call to `which_is_smaller()` using named arguments.

```
<?php
require_once __DIR__ . '/my_functions.php';

$result1 = which_is_smaller(1, 1);
var_dump($result1);
```

```
$result2 = which_is_smaller(1, 1, true);  
var_dump($result2);  
❶ $result3 = which_is_smaller(nullIfSame: true, n1: 1, n2: 1);  
var_dump($result3);
```

Listing 5-23: Calling `which_is_smaller()` by using positional and named arguments

The new call to `which_is_smaller()` ❶ is functionally equivalent to the previous call, but we use named arguments. As such, we're able to list the arguments in a different order from the way the parameters were declared: first `$nullIfSame`, then `$n1`, then `$n2`. Here's the result:

```
int(1)  
NULL  
NULL
```

The last two lines of output are both `NULL`, indicating the last two function calls achieved the same result using positional and named arguments.

In this example, each function call used all positional or all named arguments, but you can also mix both styles of arguments in the same function call. In that case, the positional arguments must come first, in the same sequence as the function declaration, followed by the named arguments in whatever sequence you wish. Consider this example:

```
$result = which_is_smaller(5, nullIfSame: true, n2: 5);
```

Here the first argument, `5`, doesn't have a name. PHP will therefore treat it positionally and match it to the first parameter declared, which is `$n1`. The remaining arguments are named and so can appear in any order. By contrast, here's another call to the function:

```
$result = which_is_smaller(nullIfSame: true, 5, n2: 5);
```

This time we've started with a named argument for `$nullIfSame`. Then we have an unnamed argument, `5`, presumably intended for the `$n1` parameter. The PHP engine will have no way of knowing this, however,

since we started with a named argument, and so this function call will trigger an error.

Skipped Parameters

When a function has multiple optional parameters, you can use named arguments to set just the optional parameters that you want while skipping the rest. This works because the named arguments free you from adhering to the order of the parameters. Any parameters you skip will take on their default values. To illustrate, let's create a function that prints customizable greetings. Start a new project and create *my_functions.php* to match [Listing 5-24](#).

```
<?php
function greet(
    string $name,
    string $greeting = 'Good morning',
    bool $hasPhD = false
): void
{
    if ($hasPhD) {
        ❶ print "$greeting, Dr. $name\n";
    } else {
        print "$greeting, $name\n";
    }
}
```

Listing 5-24: A `greet()` function with two optional parameters

We declare the `greet()` function as `void`, since it prints out a message without returning a value. The function has a required string parameter `$name`, as well as two optional parameters with default values, `$greeting` and `$hasPhD`. The body of the function is an `if` statement that outputs the values of `$greeting` and `$name`, inserting the title `Dr.` in between if parameter `$hasPhD` is `true` ❶.

Now we'll look at a few ways to call the `greet()` function. Create *main.php* containing the code shown in [Listing 5-25](#).

```
<?php
require_once __DIR__ . '/my_functions.php';

greet('Matt');
greet('Matt', hasPhD: true);
```

Listing 5-25: A main script calling `greet()` with skipped parameters

The first time we call `greet()`, we pass just the string `'Matt'` as an argument. We don't use named arguments, so this will be matched positionally to the `$name` parameter. The other parameters will use their default values, resulting in the message `Good morning, Matt`.

The second time we call `greet()`, we use the positional argument `'Matt'` and the named argument `hasPhD: true`. Notice that `$hasPhD` is the third parameter in the function declaration; we've skipped over the second parameter! This is perfectly fine. The parameter we skipped, `$message`, has a default value, and thanks to our use of a named argument, the PHP engine will know unambiguously which provided arguments match which function parameters. We should get the message `Good morning, Dr. Matt` as a result.

Here's the output of running the *main.php* script:

```
Good morning, Matt
Good morning, Dr. Matt
```

The output is just as we expect. Thanks to the combination of default parameter values and named arguments, we are able to skip the `$message` parameter without issue.

Pass-by-Value vs. Pass-by-Reference

By default, PHP functions match arguments to parameters by using a *pass-by-value* approach: the values of the arguments are copied and assigned (passed) to the appropriate parameters, which are created as temporary variables limited to the scope of the function. In this way, if the values of any parameters are manipulated while the function is executing, those

changes will have no effect on any values outside the function itself. After all, the function is working with copies of the original values.

Another approach is *pass-by-reference*: instead of receiving copies, the function parameters are passed references to the original variables themselves. In this way, if a variable is passed as an argument to a function, the function can permanently change the value of that variable. To indicate a pass-by-reference parameter, place an ampersand (&) immediately before the parameter name when you're declaring the function.

I don't typically recommend using pass-by-reference parameters; in fact, I can't think of a single one I've written in the last 20 years. Allowing functions to change the variables passed to them makes programs more complex and therefore harder to understand, test, and debug. Still, it's important to be familiar with the concept, since you might encounter pass-by-reference parameters in other people's code, including in third-party libraries you might want to use for your own projects. Calling a function with pass-by-reference parameters without knowing how they work could lead to unintended results.

NOTE

In some programming languages, programmers use several pass-by-reference parameters as a way for a function to “return” multiple values without the need for `return` statements. There are better ways to do this in modern PHP, however, such as returning an array (see [Chapter 7](#)) or an object (see [Part V](#)).

To illustrate the difference between pass-by-value and pass-by-reference parameters, and to show why the latter are often best avoided, we'll create two versions of a function that calculates someone's future age. Start a new project and create *my_functions.php* with the contents of [Listing 5-26](#).

```
<?php
function future_age (int $age): void
{
    $age = $age + 1;
    print "You will be $age years old on your next birthda
```

```
y.\n";  
}
```

Listing 5-26: A pass-by-value version of `future_age()`

Here we declare a function named `future_age()`. It features an integer parameter `$age` declared in the usual way, so this will be a normal pass-by-value parameter. The function is declared `void` since no value is to be returned. In the body of the function, we add 1 to `$age` and print out a message containing the result.

Now create a main script in *main.php* containing the code shown in [Listing 5-27](#).

```
<?php  
require_once __DIR__ . '/my_functions.php';  
  
$currentAge = 20;  
print "You are $currentAge years old.\n";  
future_age($currentAge);  
print "You are $currentAge years old.\n";
```

Listing 5-27: Testing the pass-by-value version of `future_age()`

We assign the `$currentAge` variable an integer value of 20. Then we print out a message showing the value of this variable. Next, we call our `future_age()` function, passing `$currentAge` as an argument. We then print out another message showing the value of the variable. This gives us a look at the value of `$currentAge` before and after the function call. Here's the result:

```
You are 20 years old.  
You will be 21 years old on your next birthday.  
You are 20 years old.
```

The first and last lines of output are the same, indicating that calling `future_age()` has no effect on the value of the `$currentAge` variable. In fact, when the function is called, a local variable `$age` is created within the

scope of the function, and the value of `$currentAge` is copied into it. This way, when the function adds 1 to `$age`, it does so without changing the value of `$currentAge`. That's how pass-by-value parameters work: they don't have any influence outside the scope of the function itself.

Now let's modify our `future_age()` function to use a pass-by-reference parameter and see what difference that makes. Update your *my_functions.php* file as shown in [Listing 5-28](#).

```
<?php
function future_age (int &$age): void
{
    $age = $age + 1;
    print "You will be $age years old on your next birthday.\n";
}
```

Listing 5-28: A pass-by-reference version of `future_age()`

The only change here is adding an ampersand (&) before the parameter name, indicating `$age` is a pass-by-reference parameter. As a result, `$age` will no longer be a local variable containing a copy of the value in the variable passed as an argument when the function is called. Rather, `$age` will be a reference to that variable, so any changes made to `$age` will also be made to that variable. To prove it, run your *main.php* script again. This time you should see the following output:

```
You are 20 years old.
You will be 21 years old on your next birthday.
You are 21 years old.
```

Notice that adding 1 to the `$age` parameter within the function also adds 1 to the `$currentAge` variable outside the function. Unless the user's birthday occurred in the instant between the function call and the final `print` statement, this probably isn't what we want. This illustrates the danger of using pass-by-reference parameters: they can change the value of variables that are normally outside the scope of a function.

Summary

In this chapter, we've explored how to promote code reusability by declaring and calling functions, named sequences of code that accomplish a particular task. You practiced declaring functions in a separate *.php* file and then loading them into your main application file with `require_once`, allowing you to write concise, well-organized scripts. You saw how `return` statements allow functions to send values back to the calling script while also providing a mechanism to terminate a function early, and you explored how nullable and union types give functions the flexibility to take in or output values of various data types.

You learned about the difference between parameters (the variables used within a function) and arguments (the values passed to those variables when you call a function). You saw how to make parameters optional by giving them a default value, and how to use named arguments to pass in values in any order or even skip parameters. Finally, you learned about the difference between pass-by-value and pass-by-reference parameters, in the rare event you want a function to be able to update variables outside its own scope.

Exercises

1. Create a project with separate *main.php* and *file2.php* scripts. The *file2.php* script should print out the string '456'. In your *main.php* script, first print out '123', then read in and execute *file2.php*, then print out '789'. The overall output should be 123456789, but the middle 456 has been printed from *file2.php*.
2. Write a project declaring a `which_is_larger()` function that returns the larger of two integers. Your *main.php* script should read in and execute the file declaring your function, and then print out the results of calling the function with the following arguments:
 - 4 and 5
 - 21 and 19
 - 3 and 3

What happens in the last case, where the parameters are the same?

3. Modify your `which_is_larger()` function to accept either integers or floats, and to return an integer, a float, or `NULL` if both numbers are the same.
4. Create a file *my_functions.php* that declares a `void` function to print out the first letter of your name in ASCII art style. This function should have two parameters, one (`$character`) a string setting the character to use for making the art, and the second (`$spacer`) a string setting the character to fill in the gaps. Assign suitable default values to each parameter. For example, since the first character of my name is `M`, my function might be `capital_m(string $character = 'M', string $spacer = ' ')`, and it might provide the following output when called with no arguments:

```
MM           MM
MMMM        MMMM
MM MMM  MMM MM
MM   MMMM  MM
MM           MM
MM           MM
MM           MM
```

Next, write a *main.php* script to call your function with no arguments (using both default values). Then use named arguments to call the function two more times, once providing just the main character, and then providing just the spacer character.

PART II

WORKING WITH DATA

6

LOOPS



This chapter introduces the *loop*, a control structure that allows you to repeat a sequence of statements over and over again. Once you understand how to use loops, you'll be able to easily create scripts to process large volumes of data and perform repetitive tasks with efficiently written code.

We'll cover two main types of PHP loops: the `while` loop and the `for` loop. A `while` loop is used for repeating a series of actions until a specific condition is met, while a `for` loop is used for repeating a series of actions a fixed number of times. We'll also look at `do...while` loops, a variation on `while` loops. A fourth type of PHP loop, the `foreach` loop, is specifically designed for looping through collections of data, such as arrays. We'll look at `foreach` loops when we introduce arrays in [Chapter 7](#).

while Loops

One kind of PHP loop is the `while` loop: a sequence of statements is repeatedly executed *while* a certain condition is true. [Figure 6-1](#) illustrates how this works.

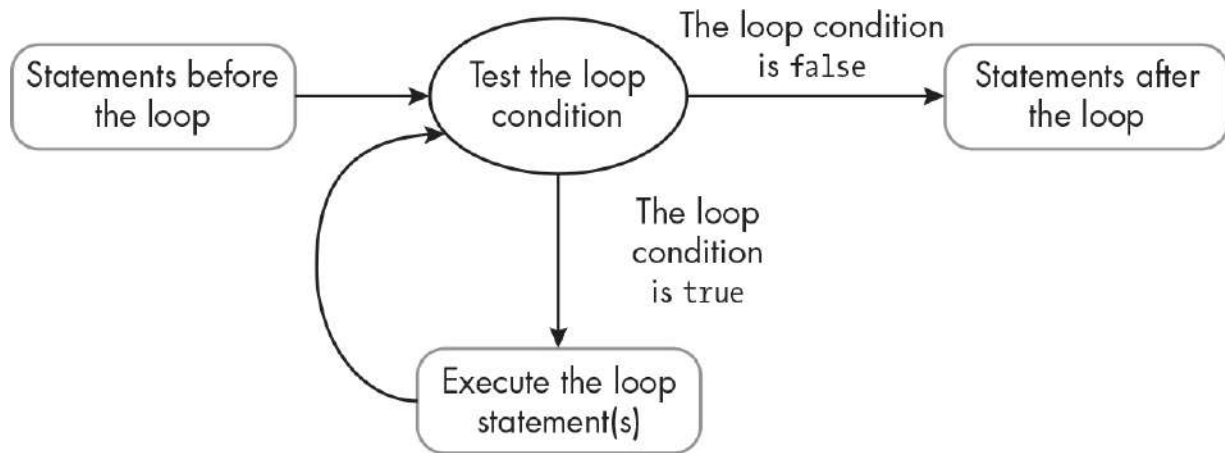


Figure 6-1: The flow of logic for a `while` loop

When a script features a `while` loop, some statements are typically executed before the loop begins. Then the PHP engine checks the condition controlling the loop. If the condition is `false`, the statements in the loop will never be executed, and the program skips ahead to the statements after the loop. If the condition is `true`, the statements in the loop are executed once, then the condition is checked a second time. If it's still `true`, the statements in the loop are executed again. This cycle continues, with the PHP engine checking the condition before each new repetition until it's found to be `false`, and the loop ends.

To write a `while` loop, start with the `while` keyword, followed in parentheses by the Boolean condition that will control the loop. Then enclose the statement that should be repeated in curly brackets. If the loop contains only a single statement, the curly brackets are optional, but most programmers include them anyway to emphasize that the statement is part of the loop.

[Listing 6-1](#) shows a script that uses a `while` loop to test the length of a password. The loop keeps prompting the user to enter a new password until the result is long enough.

```
<?php
$password = "cat";

while (strlen($password) < 6) {
    $password = readline("enter new password (at least 6 cha
```

```
acters): " );  
}  
  
print "password now set to '$password'";
```

Listing 6-1: A `while` loop requesting a password of six characters or more

Before the loop, we set the `$password` variable to an initial value of "cat". Then we declare the `while` loop, using `strlen($password) < 6` as the condition. Before each repetition of the loop, PHP will check the length of `$password`. If it's shorter than six characters, the condition is `true`, so the statement in the loop will execute. If it's six or more characters, the condition is `false`, so the loop will end. Since `$password` starts out being three characters long, we know the loop will execute at least once.

Inside the loop, we use the built-in `readline()` function to take in a password from the user. The function displays the string passed as an argument as a command line prompt, then reads whatever the user types at the command line before pressing ENTER. We store the result back into the `$password` variable, so now we'll have a new value to check before the next repetition of the loop. Once the password is long enough and the loop ends, we confirm the new value of `$password` back to the user.

NOTE

This program is not a good example of secure password protocols since the password is being shown onscreen, where anyone could read it. But it's a handy illustration of a `while` loop.

The following is a sample run of this script. I've entered a few passwords that are too short before providing an acceptable one:

```
enter new password (at least 6 characters): dog  
enter new password (at least 6 characters): whale  
enter new password (at least 6 characters): catdog123  
password now set to 'catdog123'
```

Here's what happens each time the PHP engine checks the `strlen($password) < 6` condition during this run of the script:

1. `$password` contains "cat", so the condition is `true`. The loop executes once.
2. `$password` contains "dog", so the condition is still `true`. The loop executes a second time.
3. `$password` contains "whale", so the condition is still `true`. The loop executes a third time.
4. `$password` contains "catdog123", so the condition is `false`. The loop ends, and the final `print` statement executes.

In all, the PHP engine checks the loop's condition four times and repeats the loop's contents three times. There will always be one more check of the condition than there are repetitions of a `while` loop, since the condition check happens *before* the loop's contents are executed. Try changing the initial value of `$password` to text at least six characters long and running the script again. You'll find that the script skips straight to the final `print` statement, since the first check of the loop condition evaluates to `false`.

***do...while* Loops**

A `do...while` loop is an alternate form of a `while` loop: the condition check happens *after* each repetition of the loop, rather than before. This way, the statement(s) in the loop are guaranteed to be executed at least once, and the number of loop repetitions will match the number of times the condition is checked. [Figure 6-2](#) shows how this works.

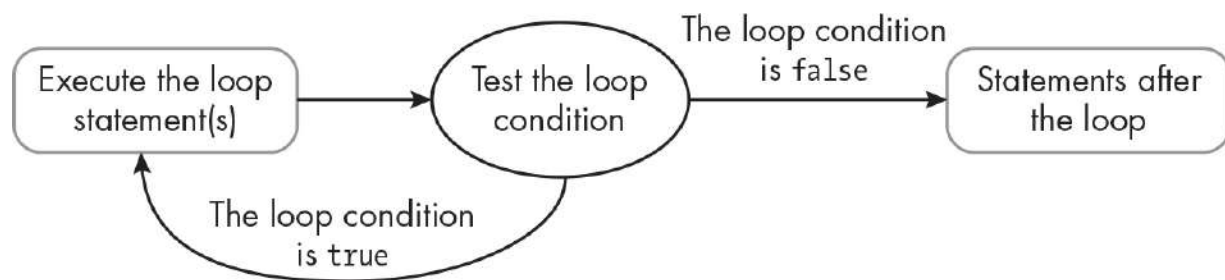


Figure 6-2: The flow of logic for a `do...while` loop

With a `do...while` loop, first we go once through the statements in the loop. Then we test the loop condition. If the condition is `false`, we move on

to the statements after the loop, so the loop will have executed only the one time. If the condition is `true`, the loop statements are executed again, and so on until the condition is `false`.

To write a `do...while` loop, start with the `do` keyword, followed in curly brackets by the statements that should be repeated. After the closing curly bracket, write the `while` keyword, then the loop condition in parentheses. For example, [Listing 6-2](#) shows how to rewrite the password-checking script from [Listing 6-1](#) by using a `do...while` loop instead of a `while` loop. The changes are shown in black text.

```
<?php
do {
    $password = readline("enter new password (at least 6 cha
racters): ");
} while (strlen($password) < 6);

print "password now set to '$password'";
```

Listing 6-2: A modified version of [Listing 6-1](#) using a `do...while` loop

We begin the loop with the `do` keyword. Then, after our statement with the `readline()` function, on the same line as the closing curly bracket, we write the `while` keyword and then, inside parentheses, the same `strlen($password) < 6` loop condition we had before. Notice that we need to include a semicolon after the condition to end the statement.

The key difference between this and the previous version of the script is that we no longer need to set an initial value for the `$password` variable before entering the loop. Instead, we read in an initial password from the user during the first run of the loop before testing its value and deciding whether a repetition is necessary. In general, if you know you'll want the statements in a loop to be executed at least once, a `do...while` loop may be a better choice than a `while` loop.

Boolean Flags

If the logic governing whether to continue looping is more complex than a single condition, it's often clearer to control the loop by using a Boolean

variable known as a *flag*. Typically, you set the flag to `true`, then enter the loop, using the flag as the condition. The loop itself features logic (perhaps a series of `if` statements) to set the value of the flag to `false` when the loop is ready to end.

For example, say we want to repeatedly prompt the user for input until they type either *quit* or *q*. We could accomplish that with a `while` loop that starts this way:

```
while (($userInput != 'q') && ($userInput != 'quit')) {
```

The condition for this loop is borderline hard to read, and it would become even more convoluted if we wanted to watch for a third input. Controlling the loop with a Boolean flag clarifies the code, as shown in [Listing 6-3](#).

```
<?php
$continueLooping = true;
while ($continueLooping) {
    $userInput = readline("type something (or: quit): ");

    if ($userInput == 'quit') {
        $continueLooping = false;
    }
    if ($userInput == 'q') {
        $continueLooping = false;
    }

    print "you typed '$userInput'\n";
}

print '--- I have left the loop! ---';
```

Listing 6-3: Using a Boolean flag variable as the loop condition

We first create the Boolean flag variable `$continueLooping` and set it to `true`. Then we declare a `while` loop with this flag as the loop condition. Notice that this is much clearer than the compound condition shown earlier. Since the condition is initially `true`, we'll enter the loop and execute the

loop statement group at least once. After prompting the user for text and storing it in the `$userInput` variable, we use two `if` statements to check whether the variable contains `'quit'` or `'q'`. These `if` statements take the place of the original compound loop condition; both set the flag to `false` to end the loop when the user asks to quit. After the `if` statements, we print the user's input. Then, outside the loop, we print a message confirming that the loop is over.

Here's a sample run of the script:

```
type something (or: quit): the
you typed 'the'
type something (or: quit): cat sat
you typed 'cat sat'
type something (or: quit): on
you typed 'on '
type something (or: quit): the mat
you typed 'the mat'
type something (or: quit): quit
you typed 'quit'
--- I have left the loop! ---
```

As you can see, the script prints out whatever text the user enters. In this case, the loop ended after I typed *quit*, but it'll also end if you type the letter *q*.

***break* Statements**

The `break` keyword immediately exits a loop, without allowing any remaining statements in the loop to execute. In the previous script's output, you may have noticed that when I typed *quit*, that word was repeated by the loop's final `print` statement (`you typed 'quit'`) before the loop terminated. With a `break` statement, we could immediately halt the loop as soon as the user enters *quit* or *q*, before that `print` statement runs.

Using `break` also eliminates the need for a Boolean flag. Instead, we can simply use the literal Boolean value `true` as the looping condition by writing `while (true)`. Since `true` is always `true`, this loop will theoretically repeat forever, or at least until some conditional logic triggers

a `break` statement. [Listing 6-4](#) shows how to update our user input script with `while (true)` and `break`.

```
<?php
❶ while (true) {
    $userInput = readline("type something (or: quit): ");

    if ($userInput == 'quit'){
        break;
    }

    if ($userInput == 'q'){
        break;
    }

    ❷ print "you typed '$userInput'\n";
}

❸ print '--- I have left the loop! ---';
```

Listing 6-4: An updated version of [Listing 6-3](#), using `break` to exit a `while (true)` loop

We've deleted our `$continueLooping` Boolean flag and replaced the `while` loop's condition with the literal `true` ❶. Inside the loop, we still have our two `if` statements testing whether `$userInput` contains the string `'quit'` or `'q'`, but this time each `if` statement simply contains the `break` keyword to immediately exit the loop. This way, if either `if` statement passes, we'll jump straight to the final `print` statement ❸, without executing the `print` statement inside the loop ❷.

Here's a sample run of this updated version of the script:

```
type something (or: quit): hello
you typed 'hello'
type something (or: quit): world
you typed 'world'
type something (or: quit): quit
--- I have left the loop! ---
```

This time, when I enter the word *quit*, no `you typed 'quit'` message appears. Instead, the loop ends immediately, so the next message is `--- I have left the loop! ---`.

In many cases, it's a matter of personal preference whether you use `break` statements or a Boolean flag in the loop condition to terminate a loop. If you're coming from a language that doesn't support `break` statements, Boolean flags might feel more natural. On the other hand, if lots of conditions are being tested to decide whether the loop should end, `break` statements can be more practical. When writing a code compiler or programming language tool, for example, tens or hundreds of tests might be included. Using `break` statements can save you from having to scroll through pages of code to see what might be happening in the later loop statements, after all the tests have been conducted.

for Loops

A `for` loop is a style of loop that repeats a set number of times. If you know exactly how many times you want to repeat a task (for example, giving a user three chances to enter the correct password, or testing students with exactly 10 questions selected randomly from a test bank), a `for` loop may be a better choice than a `while` loop. A `for` loop hinges around a *counter variable*, customarily called `$i` (short for *iterator*), that governs the number of repetitions. Declaring a `for` loop requires three expressions, all featuring this counter variable:

1. An expression initializing the counter to a starting value
2. An expression testing the counter's value to determine when the loop should stop
3. An expression to increment (or decrement) the counter after each repetition of the loop

To declare a `for` loop, all three of these expressions are written in this sequence on a single line, immediately after the `for` keyword, enclosed in a set of parentheses. Here's an example:

```
for ($i = 1; $i <= 5; $i++) {
```

Here `$i = 1` initializes the counter variable to 1. Then `$i <= 5` sets the looping condition; as long as `$i` is less than or equal to 5, the loop will continue repeating. PHP will check this condition *before* each new repetition of the loop. Finally, `$i++` uses the increment operator (`++`) to tell PHP to add 1 to `$i` *after* each pass through the loop. This way, `$i` gets a new value for each cycle of the loop that can be tested against the looping condition. In this case, `$i` will have a value of 1 the first time through, then 2 the second time through, and so on. The fifth time through the loop, when `$i` has a value of 5, the `$i <= 5` condition still passes, but at the end of the fifth repetition, `$i` will be incremented to 6. At this point, `$i` is no longer less than or equal to 5, so the loop ends after five repetitions.

To verify that the `for` loop works as we expect, let's fill in the loop's body with a simple `print` statement, as shown in [Listing 6-5](#). This script should print the same message five times.

```
<?php
for ($i = 1; $i <= 5; $i++) {
    print "I am a for loop\n";
}
```

Listing 6-5: An example of a `for` loop

We declare a `for` loop, using the same set of expressions with the looping variable `$i` we just discussed. Inside the curly brackets delineating the loop's statement group, we write a `print` statement. Here's the output of running the script:

```
I am a for loop
I am a for loop
I am a for loop
I am a for loop
I am a for loop
```

As you can see, the message indeed prints five times, thanks to the way the `for` loop's counter variable increments from 1 until it's no longer less than or equal to 5.

Using the Counter in the Loop

Part of the power of `for` loops is that the counter variable `$i` is available for use within the body of the loop. This can facilitate working with mathematical tasks, or with organized sets of data indexed or identified by a sequence of integers. For example, we might want to work with all the items in a database table whose IDs are a sequence of integers starting at 1 (see [Part VI](#) for more on databases). Or we might want to loop sequentially through all the elements of an integer-indexed array (see [Chapter 7](#)).

To demonstrate, [Listing 6-6](#) shows an updated version of our original `for` loop that incorporates the value of `$i` into the printed message. Since `$i` has a different value during each repetition of the loop, each message will now be unique.

```
<?php
for ($i = 1; $i <= 5; $i++) {
    print "I am repetition $i of a for loop\n";
}
```

Listing 6-6: Using the counter variable `$i` within a `for` loop

We've updated the loop's `print` statement to include the value of `$i`. Here's the result:

```
I am repetition 1 of a for loop
I am repetition 2 of a for loop
I am repetition 3 of a for loop
I am repetition 4 of a for loop
I am repetition 5 of a for loop
```

Notice that the number in each output line changes based on the value of `$i`. The output helps illustrate how the counter variable is working: it starts at 1 and increases by 1 with each repetition of the loop. Once `$i` gets to 6, the `$i <= 5` condition no longer passes, so the loop ends without printing `I am repetition 6 of a for loop`.

So far we've been initializing `$i` to 1, but you can initialize it to any value you want. In fact, you'll find many examples of `for` loops using a

counter variable that starts at 0. This is because `for` loops are often used in conjunction with arrays, collections of items that are numbered starting from 0. We'll discuss arrays, and how to loop through them, in [Chapter 7](#).

When you initialize the counter variable to 0, it's also common to set the looping condition with the less-than operator (`<`) instead of with less-than-or-equal-to (`<=`), like this:

```
for ($i = 0; $i < 3; $i++) {
```

This loop will run as long as `$i` is less than 3. Since `$i` starts at 0, the loop will repeat three times, when `$i` equals 0, 1, and 2.

Skipping Loop Statements

The `continue` keyword stops the current repetition of a loop, but unlike the `break` keyword, it doesn't end the loop entirely. Instead, the loop immediately jumps ahead to the next repetition. This is useful if you want to skip certain passes through a loop. For example, maybe you're retrieving entries from a database and want to ignore certain values, or you want to use only certain numbers in a sequence, such as a random sample of every third item in a randomized collection of data.

[Listing 6-7](#) shows an example of a `for` loop with a `continue` statement. Here we're using `continue` to skip odd values of counter variable `$i`, so we end up printing only the even values. The listing also illustrates how a `for` loop can decrement the counter variable instead of incrementing it.

```
<?php
for ($i = 8; $i > 0; $i--) {
    ❶ $odd = ($i % 2);

    if ($odd) {
        continue;
    }
    ❷ print "I am an even number: $i\n";
}
```

Listing 6-7: Skipping parts of a `for` loop with `continue`

We declare a `for` loop with the `$i` counter variable starting at 8 and decreasing by 1 after each repetition, thanks to the `$i--` decrementing expression. The loop will count down to 1, then stop when `$i` equals 0. Inside the loop, we use the modulo operator (`%`) to test whether the current value of `$i` is even or odd ❶. If even, `$i % 2` will be 0, or if odd, `$i % 2` will be 1. Either way, we store the result in the `$odd` variable, then use that variable as the condition of an `if` statement.

Since an `if` statement requires a Boolean condition, `$odd` will be type-juggled to a Boolean: `true` for 1 or `false` for 0. This way, when the value of `$i` is odd, we'll execute the `if` statement, which contains just the `continue` keyword to interrupt the current repetition of the loop and skip to the next one. When `$i` is even, we don't execute the `if` statement body, allowing us to complete the current repetition of the loop by executing the `print` statement ❷. The net effect is that we print out only even values of `$i`, as the output shows:

```
I am an even number: 8
I am an even number: 6
I am an even number: 4
I am an even number: 2
```

We've successfully skipped the odd numbers thanks to our conditional logic triggering the `continue` statement in our loop.

NOTE

The `continue` keyword works in `while` loops just as it does in `for` loops. Likewise, the `break` keyword also works to completely stop a `for` loop.

Handling the Last Iteration Differently

Sometimes you might want to do something different during the final repetition of a loop. With a `while` or `do...while` loop, you can't know if it's the final repetition until the loop has already ended, but with a `for` loop, you can anticipate the last repetition with conditional logic and write code that treats that last repetition differently.

For example, say we're using a `for` loop to build up a string containing a list of items entered by the user, and we want to separate each item with a comma. We might be tempted to write something like [Listing 6-8](#).

```
<?php
$message = "go to the market and buy: ";
$numItems = 3;
for ($i = 1; $i <= $numItems; $i++) {
    $item = readline("type something to buy: ");
    $message .= "$item, ";
}

print $message;
```

Listing 6-8: A `for` loop creating a list of items, separated by commas

We initialize a `$message` variable with the string "go to the market and buy: ". Then we assign `$numItems` the value 3. This will be the number of repetitions of our `for` loop. Next, we declare the `for` loop, where `$i` counts up from 1 to `$numItems` (3). Each time through the loop, we prompt the user to enter an item to buy, storing the input in the `$item` variable. We then append the value of `$item` to the `$message` string, followed by a comma and a space. When the loop ends, we print out the `$message` string that we've constructed.

The problem is that we're treating each repetition of the loop the same, so every item in the final message will have a comma after it, including the last one. You can see this in the following sample run of the script:

```
type something to buy: bread
type something to buy: butter
type something to buy: apples
go to the market and buy: bread, butter, apples,
```

Our script has built up a message including the three items entered at the command line, and unfortunately, a comma appears after the last item, apples. We can fix this by adding a test to determine whether we're on the

last repetition of the loop. Then we'll add a comma only if it isn't the last repetition. [Listing 6-9](#) shows how to update the script.

```
<?php
$message = "go to the market and buy: ";
$numItems = 3;
for ($i = 1; $i <= $numItems; $i++) {
    $item = readline("type something to buy: ");
    $message .= $item;
    $lastIteration = ($i == $numItems);
    if (!$lastIteration) {
        $message .= ', ';
    }
}

print $message;
```

Listing 6-9: Updating the script from [Listing 6-8](#) to leave off the final comma

This time we first append just the value of `$item` to the `$message` string, without a comma and space after it. Then we create the Boolean variable `$lastIteration`, giving it the value of the expression `$i == $numItems`. This expression will be `true` only the last time through the loop. Next, we have an `if` statement with `!$lastIteration` as the condition. Thanks to the NOT operator (`!`), this condition will be `true` for all repetitions except the final one. Inside the `if` statement, we append a comma and space to `$message`. This way, all but the last item in the list will have a comma after it.

Here's a sample run of the updated script:

```
type something to buy: bread
type something to buy: butter
type something to buy: apples
go to the market and buy: bread, butter, apples
```

We no longer have a comma after `apples`, the last item in the list, since we're treating the final repetition of the `for` loop differently from the

others.

NOTE

Once we start working with arrays, we'll be able to avoid this kind of last-repetition-is-special loop logic by using the built-in `implode()` function. It intelligently adds a separator between each item in a list, but not after the last item. We'll discuss this in [Chapter 7](#).

Alternative Loop Syntax

PHP provides an alternative syntax for writing `while` and `for` loops, setting off the contents of the loop with a colon (:) rather than enclosed in curly brackets. Just like the alternative syntax for `if` statements we discussed in [Chapter 4](#), this other way of writing loops is often useful when combining PHP statements in a script with templating text such as HTML for web applications. To demonstrate, [Listing 6-10](#) uses the alternative syntax to rewrite the password-setting `while` loop from [Listing 6-1](#).

```
<?php
$password = "cat";

while (strlen($password) < 6):
    $password = readline("enter new password (at least 6 characters): ");
endwhile;

print "password now set to '$password'";
```

Listing 6-10: An alternative syntax for the `while` loop from [Listing 6-1](#)

Notice that the line declaring the `while` loop ends with a colon rather than an opening curly bracket. In lieu of a closing curly bracket, we signal the end of the loop with the `endwhile` keyword.

[Listing 6-11](#) likewise shows the alternative syntax for the `for` loop from [Listing 6-6](#).

```
<?php
for ($i = 1; $i <= 5; $i++):
    print "I am repetition $i of a for loop\n";
endfor;
```

Listing 6-11: An alternative syntax for the `for` loop from [Listing 6-6](#)

Again, notice the colon at the end of the line declaring the loop, and the `endfor` keyword to indicate the end of the loop.

NOTE

PHP has no alternative syntax for a `do...while` loop.

Avoiding Infinite Loops

It's all too easy to accidentally get trapped in an infinite loop that keeps repeating forever because the stopping condition is never met. To avoid this, it's important to make sure the condition controlling the loop can and will be `false` when appropriate. One way to mistakenly write an infinite `for` loop is to set the increment expression to move in the wrong direction relative to the looping condition. For example, consider this code:

```
for ($i = 1; $i > 0; $i++) {
```

This increment expression adds 1 to `$i` after each repetition. Meanwhile, the looping condition tests whether `$i` is greater than 0. Since `$i` is growing with each repetition, it will always be greater than 0, so the loop will never end. The solution is to make `$i` decrement rather than increment, or to change the looping condition to some kind of less-than comparison.

With `while` and `do...while` loops, you can get stuck repeating forever if the variables in the loop condition don't have a chance to be changed correctly in the loop statements. For example, say we want to write a script that totals up prices entered by the user until the total exceeds \$100, then print out the result. We might accidentally create an infinite loop by writing something like [Listing 6-12](#).

```
<?php
$total = 0;

do {
    $costString = readline("enter item cost: ");

    if (is_numeric($costString)) {
        ❶ $total = floatval($costString);
    }
} while ($total < 100);

print "grand total = \$$total\n";
```

Listing 6-12: An unintentionally infinite do...while loop

We set `$total` to 0, then enter a `do...while` loop that keeps repeating while `$total` is less than 100. Inside the loop, we take in a line of input from the user and verify it's numeric. If it is, we convert the input to a float and store the value in `$total` ❶.

Do you see the problem? We should have used something like `$total += floatval($costString);` to add the latest input to the value already in `$total`, but we've used a regular assignment operator (`=`) rather than an addition assignment operator (`+=`). As a result, the value of `$total` will always be the last value entered. If the user ever enters a value greater than 100, the loop will end, and the `print` statement will echo back that last value. Otherwise, we'll be stuck in an infinite loop, without really calculating a running total.

An infinite `while` loop also occurs when the variable tested in the loop condition never changes, so once the loop is entered, it's never exited. Returning to [Listing 6-12](#), for example, we might use a `$grandTotal` variable to set the looping condition, as in `while ($grandTotal < 100)`, but then increment the `$total` variable inside the loop instead of `$grandTotal`. This way, `$grandTotal` would never change, so the loop would run forever.

Summary

In this chapter, we examined `while` loops, `do...while` loops, and `for` loops, which all offer different methods of repeating a sequence of statements. The key to these looping control structures is determining how long the loop should keep repeating, either by setting a condition for when the loop should stop, as in a `while` or `do...while` loop, or by specifying a fixed number of repetitions, as in a `for` loop. In addition to the basic structure of these loops, we discussed `break` and `continue` statements, which provide a mechanism for abruptly ending an entire loop or the current repetition of a loop, respectively. Armed with control structures like loops and choice statements, you'll be able to write sophisticated programs that perform repetitive tasks and make decisions in response to the current conditions.

Exercises

1. Use a `do...while` loop to keep taking in words input by the user until they enter one that begins with a capital letter.
Hint: Compare the string entered with the value returned by the `ucfirst()` function.
2. Use a `break` statement with a `while (true)` loop to keep taking in strings input by the user until one is numeric.
Hint: Use the `is_numeric()` function.
3. Use a `continue` statement in a `for` loop to print out all the multiples of 3 up to 21 (3, 6, 9, and so on).

7

SIMPLE ARRAYS



This chapter introduces *arrays*, a compound data type designed for storing and manipulating multiple data items under a single variable name. Arrays allow you to group related data and efficiently apply the same operations to each data item.

At its heart, an array is a mapping of values to keys. Each *value* is a piece of data you want to store in the array, and its *key* is a unique identifier associated with that value so that you can access it from within the array. In this chapter, we'll focus on simple arrays, which use integers as the keys. You'll learn how to create and manipulate simple arrays, and how to iterate over the items in an array by using a `foreach` loop. In the next chapter, we'll explore how to create more sophisticated arrays by using strings (and other data types) as keys, instead of integers.

Creating an Array and Accessing Its Values

Let's start our exploration of arrays by creating a simple array that stores the monthly rainfall totals for a location ([Listing 7-1](#)).

```
<?php
$rainfall = [10, 8, 12];
```

Listing 7-1: Declaring a simple array

We declare an array called `$rainfall` by using a sequence of comma-separated values inside square brackets: `[10, 8, 12]`. This is a three-element array, containing the values 10, 8, and 12.

By default, PHP gives each array value an integer as a key. The keys are assigned in sequence, starting from zero: the first value (10) has a key of 0, the second value (8) has a key of 1, and the third value (12) has a key of 2. Accepting this default mapping is what makes `$rainfall` a *simple array* (as opposed to the *sophisticated arrays* with custom key-value mappings that we'll explore in [Chapter 8](#)).

Now that we have an array, we can use its keys to access its values individually. In [Listing 7-2](#), we concatenate each value from the `$rainfall` array into a string message and print it out.

```
<?php
$rainfall = [10, 8, 12];

print "Monthly rainfall\n";
print "Jan: " . $rainfall[0] . "\n";
print "Feb: " . $rainfall[1] . "\n";
print "Mar: " . $rainfall[2] . "\n";
```

Listing 7-2: Accessing array elements with their integer keys

We access an item in an array by specifying its key in square brackets, after the array name. For example, `$rainfall[0]` gives us the first value in the `$rainfall` array (10), which we concatenate with the string "Jan: ". Similarly, we access the second element of the array with `$rainfall[1]`. Since the integer keys start at zero, the last element of an n -member array has a key of $n - 1$. In this case, we access the last element of our three-element array with `$rainfall[2]`. Here's the output of running this script:

```
Monthly rainfall
Jan: 10
Feb: 8
Mar: 12
```

The values 10, 8, and 12 have been successfully read from the array and printed using their integer keys 0, 1, and 2.

If you try to access an array element by using a key that hasn't been assigned, you'll get a PHP warning. For example, say we add the following `print` statement to the end of [Listing 7-2](#):

```
print "Apr: " . $rainfall[3] . "\n";
```

This statement will trigger a warning that looks something like this:

```
PHP Warning: Undefined array key 3 in /Users/matt/main.php
on line 8
```

Our array has only three elements, with keys 0, 1, and 2, so no element exists corresponding to `$rainfall[3]`. Later in the chapter, we'll discuss how to avoid warnings like this by first ensuring that an array element with a particular key exists before trying to access it.

THE ARRAY() FUNCTION

In [Listing 7-1](#), we declared an array by writing it out as a literal, enclosing its values in square brackets. Another way to declare an array is to call the `array()` function, passing the array element values as a sequence of comma-separated arguments. Here's how to declare the same `$rainfall` array from [Listing 7-1](#) by using the `array()` function:

```
$rainfall = array(10, 8, 12);
```

Understanding this alternative technique for declaring arrays is important, since you may find it in older programs and some of the PHP documentation pages (<https://www.php.net>). These days, however, the square-bracket notation is more

common (and more succinct), so I'll stick to square-bracket notation in this book. Learn more about this function at <https://www.php.net/manual/en/function.array.php>.

Updating an Array

Often you'll need to update an array after you've created it by adding or removing elements. For example, it's common to start with an empty array, created by assigning an empty set of square brackets (`[]`) to a variable, and then to add elements to it as a script progresses. In this section, we'll discuss common techniques for changing the contents of an array.

Appending an Element

If you're adding a new element to an array, you'll most often want to add it at the end, an operation known as *appending*. This is such a common task that PHP makes it very easy to do as part of a simple assignment statement. On the left side of the equal sign, you write the array name followed by an empty set of square brackets; on the right side of the equal sign, you write the value you want to append to the array. For example, [Listing 7-3](#) shows a script that creates an empty array of animals and then appends elements to the end of it.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';

print "animals[0] = $animals[0] \n";
print "animals[1] = $animals[1] \n";
```

Listing 7-3: Appending elements to the end of an array

We declare an empty array called `$animals` by writing an empty set of square brackets. Then we add two elements to the end of the array, one at a time. For example, `$animals[] = 'cat'` adds the string value 'cat' to the end of the array. PHP automatically gives the new element the next available integer as a key. In this case, since `$animals` is empty when 'cat' is added, it receives a key of 0. When we then use the same notation to add

'dog' to the array, that element automatically gets a key of 1. To confirm this, we print the individual values from the array at the end of the script, resulting in this output:

```
animals[0] = cat
animals[1] = dog
```

The output indicates that 'cat' was successfully mapped to key 0 of the array, and 'dog' to 1. The PHP engine was able to find the array's highest integer key, add 1 to it, and use the result as the next unique integer key when appending to the array.

Adding an Element with a Specific Key

While it's more common to append elements to an array and let PHP do the work of automatically assigning the next available integer key, you can also manually specify an element's key when you're adding it to an array. For example, `$heights[22] = 101` would add the value 101 to the `$heights` array and give it the integer key 22. If a value already exists at that key, that value will be overwritten. As such, this direct assignment technique is often used to update an existing value in an array rather than add a completely new value. [Listing 7-4](#) expands our `$animals` array script to illustrate how this is done.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';
$animals[0] = 'hippo';

var_dump($animals);
```

Listing 7-4: Directly assigning an array element with a specified key

As before, we append 'cat' and 'dog' to the `$animals` array. Then we replace the value of the first array element with 'hippo' by directly assigning this string to key 0 of the array. Here's the output of running this script:

```
array(2) {  
    [0]=>  
    string(3) "hippo"  
    [1]=>  
    string(3) "dog"  
}
```

Notice that 'hippo' is now mapped to key 0, indicating it has replaced the original 'cat' value.

Be careful when adding a new array element with a specific key. This action can break the sequence of integer keys if an array element doesn't exist for the key you provide. This would happen if you used a key beyond the existing size of the array. Making an array with a break in the sequence of integer keys is permissible, but it can cause issues if you've written code elsewhere that relies on having a continuous sequence of keys. We'll explore nonsequential and non-integer keys when we look at sophisticated arrays in [Chapter 8](#).

Appending Multiple Elements

So far we've been able to add only one element to an array at a time, but the built-in `array_push()` function can add several elements at once to the end of an array. The function takes a variable number of parameters. The first is the array you want to update, and the rest are the new values to be appended, and you can append as many as you want. For example, [Listing 7-5](#) revisits the script from [Listing 7-3](#), where we first added elements to the `$animals` array and then printed them, and uses `array_push()` to append two more animals to the end of the array.

```
<?php  
$animals = [];  
$animals[] = 'cat';  
$animals[] = 'dog';  
array_push($animals, 'giraffe', 'elephant');  
  
print "animals[0] = $animals[0] \n";  
print "animals[1] = $animals[1] \n";
```

```
print "animals[2] = $animals[2] \n";  
print "animals[3] = $animals[3] \n";
```

Listing 7-5: Using `array_push()` to append multiple values to the end of an array

We call `array_push()`, passing in the `$animals` array and the two string values we want to add, `'giraffe'` and `'elephant'`. Since the new elements are added to the end of the array, they're automatically assigned the next available integer keys, 2 and 3. We confirm this at the end of the script by accessing the two additional elements by their keys and printing them out, along with the two original elements:

```
animals[0] = cat  
animals[1] = dog  
animals[2] = giraffe  
animals[3] = elephant
```

The output indicates that `'giraffe'` was successfully mapped to key 2 and `'elephant'` to 3.

You may have noticed that when we called the `array_push()` function, we didn't do it as part of an assignment statement, with the function call on the right side of an equal sign and a variable name on the left to capture the function's return value. This is because `array_push()` directly modifies the array passed to it. In this sense, `array_push()` is quite different from the string manipulation functions we looked at in [Chapter 3](#), which created and returned a new string rather than making changes directly to the original string passed to them.

The `array_push()` function can directly modify the provided array because its first parameter has been declared using a pass-by-reference approach. As we discussed in [Chapter 5](#), this means the function is given a direct reference to the value of the argument passed in, as opposed to being given a copy of the argument's value via a pass-by-value approach. We can confirm this by looking at the function's signature in the PHP documentation:

```
array_push(array &$amp;array, mixed ...$values): int
```

The ampersand (&) before the first parameter, `&$array`, indicates that this is a pass-by-reference parameter.

Since `array_push()` is directly modifying the array, there's no need for it to return a copy of the array, or for us to use an assignment statement to capture that return value when we call the function. In fact, `array_push()` *does* have a return value, an integer indicating the new length of the array. This can be useful if you need to keep track of the array's length as you're updating it; we didn't need this return value in [Listing 7-5](#), so we simply made a stand-alone call to the function, without assigning the result to a variable.

Removing the Last Element

The built-in `array_pop()` function returns the last item of an array while also removing that item from the array. This is another example of a pass-by-reference function that changes the provided array. In [Listing 7-6](#), we use `array_pop()` to retrieve and remove the last element of our `$animals` array.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';

$lastAnimal = array_pop($animals);
print "lastAnimal = $lastAnimal\n";
var_dump($animals);
```

Listing 7-6: Using `array_pop()` to retrieve and remove the last array element

We call `array_pop()`, passing the `$animals` array as an argument, and we store the function's return value in the `$lastAnimal` variable. We then print out `$lastAnimal`, as well as the `$animals` array, to see which elements remain. Here's the result:

```
lastAnimal = dog
array(1) {
    [0]=>
```

```
        string(3) "cat"
    }
```

The string in variable `$lastAnimal` is 'dog', since this was the last of the elements appended to the array. The `var_dump` of `$animals` shows that the array contains only 'cat' after the call to `array_pop()`, demonstrating how this pass-by-reference function was able to change the array passed into it.

ARRAYS AS STACKS

One of the classic data structures for solving some types of computer tasks is the *stack*. It treats data like a stack of items, such as a pile of books or blocks. You can *push* an element onto the stack by adding it on top of the existing elements, or *pop* the last (topmost) element off the stack.

If you push A, then B, then C, for example, you have a stack with A on the bottom, B in the middle, and C on top. If you then start popping items, you first get C, then B, then A. The most recent item added to the stack is always the first item to be removed. The PHP functions `array_push()` and `array_pop()` mirror these operations, making it easy to create scripts that solve problems by using simple arrays as stacks.

Retrieving Information About an Array

We've considered some functions for modifying an array, but other functions can return useful information about an array without changing it at all. For example, `count()` returns the number of elements in an array.

This can be useful if you want to check whether an array contains anything at all (a count of zero might indicate that a shopping cart is empty or that no records were retrieved from a database), or whether it has more items than expected (perhaps a customer has more than one address on file).

Sometimes knowing the number of items in an array can be helpful in order to control a loop through that array. In [Listing 7-7](#), we use `count()` to print the total number of items in the `$animals` array.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];
```



```
print count($animals);
```

Listing 7-7: Counting the number of elements in an array

We call the `count()` function, passing the name of the array we want it to count up, and print the result. Since `$animals` has four elements, this script should output the integer 4.

NOTE

The `sizeof()` function is an alias of `count()`. If you see a script that uses `sizeof()`, know that it works the same way as `count()`.

Another analytical array function is `array_is_list()`. PHP distinguishes between arrays that are lists and arrays that aren't. To be considered a *list*, an array of length n must have consecutively numbered keys from 0 to $n - 1$. The `array_is_list()` function takes in an array and returns `true` or `false` based on whether the array meets that definition. All the arrays discussed in this chapter qualify as lists, since they rely on PHP's default behavior of assigning keys sequentially from 0. In the next chapter, however, we'll explore arrays with non-integer keys as well as the `unset()` function, which can remove an element of an array with a given key, potentially breaking the consecutive chain of numeric keys and disqualifying an array as a list. Thus, `array_is_list()` could be useful for evaluating an array before passing it along to code that expects the array to be structured as a list.

The `array_key_last()` function returns the key for the last element of the given array. Assuming the array is a proper list with consecutively numbered keys, the return value of `array_key_last()` should be one less than the return value of `count()`. For example, calling `array_key_last($animals)` at the end of [Listing 7-7](#) would return the integer 3, since that's the key of the fourth (and final) element of the array.

Earlier I mentioned that trying to access an array key that doesn't exist triggers a warning. To avoid this, use the `isset()` function to test whether an array key exists before trying to access it. [Listing 7-8](#) shows the function in action.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

❶ if (isset($animals[3])) {
    print "element 3 = $animals[3]\n";
} else {
    print "sorry - there is no element 3 in this array\n";
}

print "(popping last element [3])\n";
❷ array_pop($animals);

if (isset($animals[3])) {
    print "element 3 = $animals[3]\n";
} else {
    print "sorry - there is no element 3 in this array\n";
}
```

Listing 7-8: Using `isset()` to test the existence of an array key

First, we create our four-element `$animals` array. Then we use an `if...else` statement with `isset()` to access only the element with key 3 if that element exists (at this point, it should) ❶. We next use `array_pop()` to remove the last element from `$animals` (the one at key 3) ❷. Then we repeat the same `if...else` statement. Now no element has key 3, but since we're testing for the element with `isset()` before attempting to access it, we shouldn't get a warning. Take a look at the output of the script:

```
element 3 = elephant
(popping last element [3])
sorry - there is no element 3 in this array
```

The first line of the output indicates the first call to `isset()` returned `true`, triggering the `if` branch of the conditional. The last line shows the second `isset()` call returned `false`, triggering the `else` branch and saving us from trying to access a nonexistent array element.

Looping Through an Array

It's common to have to access the elements of an array, one at a time, and do something with each one. Assuming the array is a list, you can do this with a `for` loop that uses a counter variable as the key for the current array element. By starting the counter at 0 and incrementing it up to the length of the array, you can access each element in turn. [Listing 7-9](#) uses a `for` loop to print each element of our `$animals` array.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$numElements = count($animals);
for ($i = 0; $i < $numElements; $i++) {
    ❶ $animal = $animals[$i];
    print "$animal, ";
}
```

Listing 7-9: Using a `for` loop to loop through an array

We use `count()` to look up the length of the array, storing the result into the `$numElements` variable. Then we declare a `for` loop that increments counter `$i` from 0 up to but not including the value of `$numElements`. (We could hardcode the stopping condition as `$i < 4`, but using a variable makes the code more flexible in case the length of the array changes.) In the loop statement group, we use `$animals[$i]` to retrieve the element whose key is the current value of loop variable `$i`, storing it in `$animal` ❶. Then we print out this `$animal` string, followed by a comma and a space. The output when we run this script in a terminal is as follows:

```
cat, dog, giraffe, elephant,
```

Each of the array elements is printed out in sequence. (Don't worry, we'll fix that final comma shortly.)

Using a foreach Loop

This `for` loop approach works, but cycling through the elements of an array is such a common task that PHP provides another type of loop, the `foreach` loop, to do it more efficiently. At the core of a `foreach` loop is the `foreach ($array as $value)` syntax; `$array` is the name of an array to loop over, and `$value` is a temporary variable that will be assigned the value of each element in the array, one at a time. [Listing 7-10](#) shows an updated version of [Listing 7-9](#), using a `foreach` loop rather than a `for` loop.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

foreach ($animals as $animal) {
    print "$animal, ";
}
```

Listing 7-10: Using a `foreach` loop to elegantly loop through an array

We declare this loop by using `foreach ($animals as $animal)`. Here, `$animal` is a temporary variable that takes on the value of each array element in turn, which we then print in the body of the loop. Notice that we no longer have to worry about determining the length of the array to set the loop's stopping condition, nor do we need to manually access each array element, as we did in the `for` loop version with `$animals[$i]`. The `foreach` loop retrieves each element automatically. The result is the same as the `for` loop version, but the `foreach` loop's syntax is much more elegant and concise.

The `foreach` loop has the added benefit that we don't need to care whether the provided array is a true list. With the `for` loop version, we're relying on the consecutive integer numbering of the array keys; if a key is missing, we'll get a warning when we try to access that key. By contrast, the `foreach` loop simply accesses each element in the array, no matter what the keys are.

Accessing Keys and Values

An alternative syntax for `foreach` loops allows you to access both the key and the value of each array element, instead of just the value. For this, declare the loop in the format `foreach ($array as $key => $value)`. Here, `$array` is the array you want to loop through, `$key` is a temporary variable that will hold the current element's key, and `$value` is a temporary variable that will hold the current element's value. The `=>` operator connects a key to a value. We'll use it more extensively in [Chapter 8](#) when we work with sophisticated arrays whose keys can be strings and other data types.

Gaining access to keys as well as values allows us to eliminate that pesky final comma from the output after the last element in the `$animals` array. Recall that the `array_key_last()` function returns the key of the last element in an array. By comparing the value from this function with the current key in the `foreach` loop, we can decide whether to print a comma after each element. [Listing 7-11](#) shows how.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

foreach ($animals as $key => $animal) {
    print "$animal";
    if ($key != array_key_last($animals)) {
        print ", ";
    }
}
```

Listing 7-11: A revised `foreach` loop that accesses the key and value of each array element

We declare a `foreach` loop by using `foreach ($animals as $key => $animal)`. Each cycle through the loop, `$key` will be the key and `$animal` will be the value of the current array element. Inside the loop, we first print out the string in `$animal`. Then we use an `if` statement to also print a comma and a space if the current element's key is *not* equal to the last key of the array (identified with the `array_key_last()` function). This should produce the following output:

```
cat, dog, giraffe, elephant
```

We've successfully eliminated the comma after the last element in the array.

Imploding an Array

The code in [Listing 7-11](#) is essentially printing a string containing the elements in an array with a separator (in this case, a comma and a space) between them. This is a common task, so PHP provides a built-in function called `implode()` to do it automatically, without the need for any kind of loop.

The function takes two arguments: a string to use as a separator and an array to implode into a string. The separator goes *between* elements, not *after each* element, so the code won't place an extra separator after the last array element. [Listing 7-12](#) shows an updated script that uses `implode()` rather than a `foreach` loop.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

print implode(', ', $animals);
```

Listing 7-12: Using `implode()` to convert an array into a string

Here we print the result of calling `implode()` on the `$animals` array. We use the string `,` as a separator to put a comma and a space between the array elements. The output should be exactly the same as that of [Listing 7-11](#), but `implode()` makes the code much more efficient to write.

The `implode()` function may have rendered our `foreach` loop unnecessary in this case, but don't let that fool you. A `foreach` loop is the right tool to use in plenty of scenarios. In general, when the code you want to apply to each element in an array is more sophisticated than simply printing out that element's value, a `foreach` loop is likely appropriate.

Functions with a Variable Number of Arguments

One important application for arrays is that you can use them to declare functions that accept a variable number of arguments. When we created

custom functions in [Chapter 5](#), we needed to know exactly how many arguments each function would take in so we could define the function with the corresponding number of parameters. This isn't always possible, however.

For example, say we want to declare a `sum()` function that takes in an unspecified quantity of numbers, adds them all up, and returns the result. We don't know whether the user will pass two numbers, three numbers, or more as arguments, so we can't create a separate parameter for each number. Instead, we use a single parameter to represent all the numbers, and we write an ellipsis, or three dots (`...`), before the parameter name. This syntax tells PHP to treat the parameter as an array and to fill it with all the arguments provided, however many there are.

[Listing 7-13](#) shows how this approach works by declaring the `sum()` function just described. Remember that functions should be declared in a separate file from the code that calls them, so create a *my_functions.php* file containing the contents of this listing.

```
<?php
function sum(...$numbers): int
{
    $total = 0;

    foreach ($numbers as $number) {
        $total += $number;
    }

    return $total;
}
```

Listing 7-13: A function to take in a variable number of integer arguments and return their sum

We declare the `sum()` function, which returns an integer. It has a single parameter, `...$numbers`. Thanks to the ellipsis, any arguments the function receives will be assigned as elements to a local array called `$numbers`. Notice that we don't specify a data type for the parameter when using the ellipsis notation; we know the overall `$numbers` variable will be of the

array type, although the individual elements of the array can be of any type. Inside the function body, we initialize `$total` to 0. Then we use a `foreach` loop to cycle through the elements of the `$numbers` array, adding the value of each element to `$total`. Once the loop has finished, we return `$total`, which holds the sum of the arguments.

NOTE

Our `sum()` function doesn't include any logic to confirm that the elements in `$numbers` are actually numbers. A real-world function would need some form of data validation and would perhaps return `NULL` or indicate invalid data some other way if the arguments provided aren't all numbers. Also note that PHP already has a built-in `array_sum()` function that totals up the numbers in an array. We've implemented our own version for demonstration purposes.

[Listing 7-14](#) shows a `main.php` script to read in the `sum()` function declaration and test it out with a variable number of arguments.

```
<?php
require_once 'my_functions.php';

print sum(1, 2, 3) . "\n";
print sum(20, 40) . "\n";
print sum(1, 2, 3, 4, 5, 6, 7) . "\n";
```

Listing 7-14: A main script that calls `sum()` with different numbers of arguments

After reading in and executing `my_functions.php` with `require_once`, we make three calls to `sum()`, each with a different number of arguments, and print the results. The script produces this output:

```
6
60
28
```

The three printed sums have been correctly calculated. This indicates that our `sum()` function has successfully collected the variable number of arguments in an array.

Array Copies vs. Array References

Say you have a variable containing an array, and you assign it as the value of a second variable. In some languages, such as Python and JavaScript, the second variable would be assigned a *reference* to the original array. The two variables would then refer to the same array in the computer's memory, so a change to one variable would apply to the other variable as well. In PHP, however, the default is to assign the second variable a *copy* of the array. Because the two variables have their own separate arrays, a change to one won't impact the other. [Listing 7-15](#) returns to our `$animals` array to illustrate this point.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$variable2 = $animals;
array_pop($variable2);
var_dump($animals);
```

Listing 7-15: Copying an array

We declare our usual `$animals` array, then assign `$animals` to `$variable2`. This creates a separate copy of the array in `$variable2`, so anything we do to one array should have no effect on the other. To prove it, we use `array_pop()` to remove the last element from the `$variable2` array, then print the original `$animals` array. Here's the result:

```
array(4) {
    [0]=>
    string(3) "cat"
    [1]=>
    string(3) "dog"
    [2]=>
    string(7) "giraffe"
```

```
[3]=>
string(8) "elephant"
}
```

All four animal strings are still present in the `$animals` array, even though we deleted the final element ('elephant') from the `$variable2` array, so the variables indeed hold separate arrays.

If you want to assign the second variable a reference to the original array, as is customary in other languages, rather than a copy of the array, then use the reference operator (&) at the time of assignment. [Listing 7-16](#) updates the code from [Listing 7-15](#) to show the difference.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$variable2 = &$animals;
array_pop($variable2);
var_dump($animals);
```

Listing 7-16: Using the reference operator when assigning an array

This time we prefix the `$animals` array with the reference operator (&) when assigning it to `$variable2`. This means a change to one variable will now apply to the other variable as well, since both refer to the same array in memory. The updated script results in this output:

```
array(3) {
    [0]=>
    string(3) "cat"
    [1]=>
    string(3) "dog"
    [2]=>
    string(7) "giraffe"
}
```

The output reveals that popping element 3 from the `$variable2` array also removed element 3 from the `$animals` array. This confirms that both

`$variable2` and `$animals` refer to the same array in memory.

One of these approaches to array assignment isn't inherently better than the other; they're just different. Sometimes it's best to copy an array before manipulating it. For example, a web page might offer the user a chance to edit a shopping list, while providing a Cancel button to undo those edits. In this case, you'll want to work with a copy of the shopping list array until the changes are confirmed, since you may need to revert to the original array if the user clicks Cancel. Other times, it's preferable to have multiple variables referencing the same array in memory. Perhaps the code contains logic that chooses one of several arrays and so needs to set a variable to be a reference to the chosen array.

The key point to take away from this section is that PHP defaults to copying the array unless you use the reference operator (&). If you've learned a different programming language before PHP, or if you learn another language in the future, it's important to understand the difference between assignment of a copy and assignment of a reference, and to know which behavior the language uses by default.

Treating Strings as Arrays of Characters

In a way, you can think of a string as an array of individual characters. This can be useful since you may sometimes want to “navigate” through the string character by character for tasks such as encryption, spellchecking, and so on, just as you'd traverse the elements of an array.

As you saw in [Chapter 3](#), the characters in a string are numbered from 0, just like the elements in a simple array. In fact, you can use the same square-bracket notation for accessing an array element to also access a specific character from a string. For example, if `$name` held the string 'Smith', `$name[0]` would return 'S', `$name[1]` would return 'm', and so on. Strings also support *negative* integer keys for counting characters backward from the end of the string: `$name[-1]` returns 'h' (the last character), `$name[-2]` returns 't', and so on.

NOTE

Unlike strings, arrays themselves don't interpret negative integer keys as counting backward from the end of the array. Instead, `$animals[-1]` would

be interpreted as an element of the `$animals` array with an actual key of `-1`. While you can manually assign negative integers as keys to array elements, I personally can't remember ever needing to do so.

[Listing 7-17](#) shows an example of using array key syntax to access individual characters from a string.

```
<?php
$name = 'Smith';

$firstChar = $name[0];
$secondToLastChar = $name[-2];

print "first character = '$firstChar'\n";
print "second to last character = '$secondToLastChar'\n";
```

Listing 7-17: Using square-bracket notation to access string characters

We assign the string `'Smith'` to the `$name` variable. Next, we copy the string's first character (`$name[0]`) to the `$firstChar` variable and its second-to-last character (`$name[-2]`) to `$secondToLastChar`. We then print out messages with the values of these variables, producing the following output:

```
first character = 'S'
second to last character = 't'
```

Unlike with arrays, we can't pass a string to a `foreach` loop to cycle through all its characters. However, we *can* use PHP's built-in `str_split()` function to convert a string into an actual array of individual characters, then pass that array to a `foreach` loop, as shown in [Listing 7-18](#).

```
<?php
$name = 'Smith';

$characters = str_split($name);
foreach ($characters as $key => $character) {
```

```
        print "character with key $key = '$character'\n";
    }
```

Listing 7-18: Using `str_split()` and `foreach` to loop over the characters in a string

We pass the same `$name` string to `str_split()`. By default, this function breaks the string into individual characters, assigns them as elements of an array, and returns the result, which we store in the `$characters` variable. We then use a `foreach` loop to access each key and value in the array version of the string and print them out. Here's the result:

```
character with key 0 = 'S'
character with key 1 = 'm'
character with key 2 = 'i'
character with key 3 = 't'
character with key 4 = 'h'
```

The output shows that we've successfully looped through the characters from the original string after first converting the string to an array with `str_split()`.

The `str_split()` function has an optional second argument controlling the number of characters for each string element in the resulting array. The argument defaults to 1, splitting the string into individual characters, but if we'd called `str_split($name, 2)`, for example, then the resulting array would contain two-character strings: `['Sm', 'it', 'h']`.

Other Array Functions

We've discussed some built-in functions for working with arrays in this chapter, but PHP has many more. Other useful functions that apply to arrays include the following:

- `sort()`** Sorts an array's values in ascending order (alphabetical for string values, numerical order for number values)
- `usort()`** Sorts the values into a custom order based on a user-defined function
- `array_flip()`** Swaps the keys and values for each array element

array_slice() Returns a new array containing a subsequence of elements from an existing array

array_walk() Calls a user-defined function on each element of an array

array_map() Calls a user-defined function on each element of an array and returns a new array of the results

array_rand() Returns random keys from an array

For a full list of array functions, see the PHP documentation at

<https://www.php.net/manual/en/ref.array.php>.

Summary

In this chapter, we've begun exploring arrays, a compound data type that stores multiple values under a single variable name, with each value having its own identifying key. For now, we've focused on simple arrays, whose keys are integers. The chapter introduced various techniques for adding, subtracting, and accessing array elements, as well as functions for obtaining information about an array, such as `count()` and `isset()`. You also learned how to work with each array element in sequence by using a `foreach` loop. In some cases, PHP provides built-in functions for handling common array tasks, such as the `implode()` function that joins all the elements of an array into a single string. These functions sometimes allow you to replace complete loops and conditional statements with a single function call.

Exercises

1. Create a `$colors` array containing the string names of five colors. Print a random color from the array.

Hint: You can get a valid random key by calling

```
array_rand($colors).
```

2. Write a `foreach` loop to print each of the colors from your array in Exercise 1 on a new line, in the following form:

```
color 0 = blue  
color 1 = red  
...
```

3. Use `array_pop()` to print the last element of your array of colors from Exercise 1. Then use `var_dump()` to show that this item has been removed from the array.
4. Create an array containing integer ages 23, 31, and 55. Use built-in functions to calculate and print out the number of items in the array and their average.

8

SOPHISTICATED ARRAYS



In this chapter, we'll take a more sophisticated approach to PHP arrays and explore how to manually assign an array's keys. This opens up the possibility of using meaningful strings as keys instead of PHP's default behavior of using sequential integers. We'll also discuss multidimensional arrays, where the value of an array element is itself another array, and we'll look at more functions and operators for working with arrays. With this expanded look at PHP arrays, you'll begin to see how they can store and manipulate more complex data structures.

Declaring Array Keys Explicitly

We've discussed how PHP will automatically assign sequential integer keys to array elements, starting from 0, in which case the resulting array will meet the definition of a *list*. Rather than relying on this default behavior,

however, you can use the double-arrow operator (`=>`) when declaring an array to explicitly map a key to each value. Then you aren't obligated to follow the default pattern for keys. For example, you can use nonsequential integers as keys, or start counting from a number other than 0. Either way, the resulting array will no longer be considered a list, but it will be a valid array nonetheless. To illustrate, [Listing 8-1](#) shows a script that explicitly uses nonsequential integer keys in an array.

```
<?php
$rainfallValues = [
    0 => 10,
    4 => 8,
    3 => 12
];

print "-- Monthly rainfall --\n";
foreach ($rainfallValues as $key => $rainfallValue) {
    print "$key: $rainfallValue\n";
}

var_dump(array_is_list($rainfallValues));
```

Listing 8-1: Explicitly declaring integer array keys out of sequence

Here we declare a `$rainfallValues` array. Inside the square brackets of the array, we use the `=>` operator to explicitly assign a key to each array element. For example, `0 => 10` adds an element to the array with a value of 10 and a key of 0. The key/value pairs are separated by commas, just as we separated the array values by commas in [Chapter 7](#) when we weren't explicitly declaring the keys. In this case, we've also placed each key/value pair on its own indented line, for clarity. With the array declared, the script continues by looping through it and printing its key/value pairs.

Notice that the array keys we've declared aren't sequential. The second array element has a key of 4, and the third element has a key of 3. This may not be the most intuitive scheme for assigning keys, but if that's what we want, PHP is perfectly fine with it. The array won't meet the definition of a

list (so the call to `array_is_list()` at the end of the script should return `false`), but the array is still valid. Here's the output of running the script:

```
-- Monthly rainfall --
0: 10
4: 8
3: 12
bool(false)
```

The `foreach` loop works even though the array isn't a proper list, iterating through the array's key/value pairs and printing them out. Notice that the element with key 4 prints before key 3. What matters is the order in which the elements are *declared*, not the numerical order of the keys themselves. The `false` at the end of the output confirms that the array no longer meets the requirements of a list.

Once you start explicitly declaring keys, you don't necessarily have to declare one for *every* array element. If an element is declared without a key, PHP will automatically look for the most recent integer key, increment it, and use that as the new key. This can be useful if you want an array to have sequential keys that don't start from 0.

For example, say you have a class of students and want an array mapping the students' IDs to their grades. Each ID is a seven-digit number, beginning with the year and followed by three digits that increment sequentially. In 2025, for instance, the first student would have a numeric ID of 2025001, the next 2025002, and so on. In this case, you can explicitly declare just the first array key and let PHP automatically assign the rest.

[Listing 8-2](#) shows how.

```
<?php
$studentGrades = [
    2025001 => 'A',
    ❶ 'B',
    'A',
    'D',
    'F'
];
```

```
print "-- Student grades--\n";
foreach ($studentGrades as $studentId => $grade) {
    print "$studentId => $grade\n";
}

var_dump(array_is_list($studentGrades));
```

Listing 8-2: Declaring the first array key explicitly and the rest automatically

Within the `$studentGrades` array, we explicitly give the first element a key of `2025001`. Then, beginning with the second element ❶, we supply only the values. By default, PHP will map these values to the integer keys `2025002`, `2025003`, and so on. As before, we finish the script by looping through and printing the key/value pairs and testing whether the array counts as a list. The output is shown here:

```
-- Student grades--
2025001 => A
2025002 => B
2025003 => A
2025004 => D
2025005 => F
bool(false)
```

Notice that PHP has assigned the remaining keys sequentially, incrementing from the explicitly declared key of `2025001`. However, even though the keys are sequential, they don't start from 0. Therefore, the array isn't a list, as the `false` at the end of the output confirms.

Arrays with Strings as Keys

Let's take our coding a step further: now that we're assigning array keys explicitly, who's to say they have to be integers? They can just as easily be strings, in which case each value in the array can be given a meaningful name as a key. Returning to the `$rainfallValues` array from [Listing 8-1](#), for example, we can use month names as keys instead of integers. This

change will better indicate that each value in the array is a monthly rainfall total. [Listing 8-3](#) revises the script accordingly.

```
<?php
$rainfallValues = [
    'jan' => 10,
    'feb' => 8,
    'march' => 12
];

print "-- Monthly rainfall --\n";
foreach ($rainfallValues as $key => $rainfallValue) {
    print "$key: $rainfallValue\n";
}

var_dump(array_is_list($rainfallValues));
```

Listing 8-3: Using strings as array keys

This time we've assigned the key 'jan' to the value 10, the key 'feb' to the value 8, and the key 'march' to the value 12. We use the same => operator as before to pair keys with values. The only difference is that the keys are now strings. Here's the script's output:

```
-- Monthly rainfall --
jan: 10
feb: 8
march: 12
bool(false)
```

The string keys clarify what the values actually represent. The `false` at the end of the output shows that this array isn't a list. This isn't surprising, since the keys aren't even integers.

Accessing individual values from an array with string keys works just like accessing values from arrays with integer keys: provide the key in square brackets, after the array name. For example, here's how to print the rainfall value for the month of March:

```
print $rainfallValues['march'];
```

Similarly, you can also use square-bracket notation to add or update array elements with string keys. Here we add a new rainfall total for April:

```
$rainfallValues['april'] = 14;
```

This is a simple example, but hopefully you can begin to see the potential power of PHP arrays to build meaningful collections of data. When you don't need the full range of features of object-oriented programming (discussed in [Part V](#)), using arrays with string keys allows you to work with data whose values are naturally associated with keys that make sense for the task (such as dates or months, people's names or IDs, or product names or codes).

Multidimensional Arrays

Up to now, the arrays we've been exploring have been *single-dimensional*: they contain a sequence of elements, and each element is a scalar (single) value mapped to a key. However, you can also declare arrays containing elements that are arrays themselves, resulting in a *multidimensional array*. For example, say you want to create an array of tasks and the time each task will take in minutes. Each element in the array could itself be an array holding the name of a task and its associated duration, as shown here:

```
$tasksAndMinutes = [  
    ['shopping', 30],  
    ['gym', 60],  
    ['nap', 15]  
];
```

Here `$tasksAndMinutes` is a multidimensional array. Its first element, `['shopping', 30]`, is a two-element array holding a string task name and the integer number of minutes to allocate for that task. The other array elements follow this same format. With a multidimensional array like this,

we refer to the overall `$tasksAndMinutes` as the *outer* array and its elements as the *inner* arrays.

One way to work with a multidimensional array is to use a nested set of `foreach` loops, one to iterate over the elements of the outer array and the other to iterate over the elements of each inner array. In the `$tasksAndMinutes` array, however, all the inner arrays have the same structure (which won't always be a given). Therefore, in cases like these, you can use your knowledge of that structure to extract the values from each inner array by using a single `foreach` loop that iterates over the outer array. [Listing 8-4](#) illustrates this approach.

```
<?php
$tasksAndMinutes = [
    ['shopping', 30],
    ['gym', 60],
    ['nap', 15]
];

foreach ($tasksAndMinutes as $item) {
    $task = $item[0];
    $minutes = $item[1];
    print "allow $minutes minutes today for task: $task\n";
}
```

Listing 8-4: Working with a multidimensional array

We declare the `$tasksAndMinutes` array as shown previously. Next, we declare a `foreach` loop that iterates through the elements of `$tasksAndMinutes`, using the `$item` variable to represent the current element. As we've seen, each element is itself an array containing a task name and a time in minutes. We can therefore extract the first element of `$item` (using integer index 0) into the `$task` variable and the second element (index 1) into `$minutes`. Then we print a message about the current task by using these two variables, producing the following output:

```
allow 30 minutes today for task: shopping
allow 60 minutes today for task: gym
```

```
allow 15 minutes today for task: nap
```

The times and task names have successfully been extracted from each inner array during the `foreach` loop.

In this example, the inner arrays use integer keys by default, but as you know, arrays can also use non-numeric keys. Pairing the values in each inner array with meaningful string keys like `'task'` and `'minutes'` will make the code much more readable. For example, we'll be able to access the task from the current element of `$tasksAndMinutes` with `$task = $item['task']` rather than `$task = $item[0]`. [Listing 8-5](#) shows this improvement.

```
<?php
$tasksAndMinutes = [
    ['task' => 'shopping', 'minutes' => 30],
    ['task' => 'gym', 'minutes' => 60],
    ['task' => 'nap', 'minutes' => 15],
];

foreach ($tasksAndMinutes as $item) {
    $task = $item['task'];
    $minutes = $item['minutes'];
    print "allow $minutes minutes today for task: $task\n";
}
```

Listing 8-5: Refactoring [Listing 8-4](#) to use string keys in the inner arrays

This time we explicitly assign the string keys `'task'` and `'minutes'` to the values in each of the arrays inside `$tasksAndMinutes`. Then we use those meaningful keys inside the `foreach` loop to extract the values from the current inner array being processed. The result is exactly the same as before, but the code is easier to read. Before PHP allowed for object-oriented programming, well-labeled multidimensional arrays like this were an integral part of the code for the data-related features of many programs.

More Array Operations

In [Chapter 7](#), we discussed array operations such as adding an element to and removing an element from the end of a simple array. Now that we've explored sophisticated arrays, let's consider more array operations. We'll look at how to remove an element from anywhere in an array, how to use array operators like union (+) and spread (...), and how to extract the elements of an array into separate variables.

Removing Any Element from an Array

You can remove an element from an array by passing the element's key to the `unset()` function. Unlike the `array_pop()` function covered in the previous chapter, which specifically removes the *last* element in an array, `unset()` can remove an element from *any* position. Also unlike `array_pop()`, the `unset()` function doesn't return the deleted element; it's simply gone.

Using `unset()` becomes more appropriate when you start assigning strings rather than integers as array keys. With string keys, the order of the array elements often loses its significance, so it's more meaningful to remove an element based on its key rather than its position in the array.

[Listing 8-6](#) revisits the `$rainfallValues` array as an example.

```
<?php
$rainfallValues = [
    'jan' => 10,
    'feb' => 8,
    'march' => 12
];

unset($rainfallValues['feb']);

print "-- Monthly rainfall --\n";
foreach ($rainfallValues as $key => $rainfallValue) {
    print "$key: $rainfallValue\n";
}
```

Listing 8-6: Using `unset()` to remove an element from an array

We use `unset()` to remove the element with the `'feb'` key from the `$rainfallValues` array. Then we loop through the array to print details for the remaining elements as before. Here's the result:

```
-- Monthly rainfall --
jan: 10
march: 12
```

Notice that no data is printed for an element with the key `'feb'`, since that element no longer exists within the array.

NOTE

Calling `unset()` on a whole array, such as `unset($rainfallValues)`, would delete the entire array, just as calling `unset()` on any other variable would clear that variable.

Combining and Comparing Arrays

You can combine or compare arrays by using some of the same addition, equality, and identity operators that apply to scalar (single-value) variables. [Table 8-1](#) summarizes the six array operators available.

Table 8-1: Array Operators

Name	Symbol	Example	Description
Union	+	<code>\$a + \$b</code>	Returns an array with the elements of arrays <code>\$a</code> and <code>\$b</code> .
Spread	<code>...</code>	<code>[1, ...\$a]</code>	Returns an array that has <code>1</code> as the first element, followed by the elements of array <code>\$a</code> .
Equal	<code>==</code>	<code>\$a == \$b</code>	Returns <code>true</code> if arrays <code>\$a</code> and <code>\$b</code> have the same key/value pairs.
Identical	<code>===</code>	<code>\$a === \$b</code>	Returns <code>true</code> if arrays <code>\$a</code> and <code>\$b</code> are identical: they have the same key/value pairs, and their elements are in the same order and of the same types.
Not equal	<code>!=</code> or <code><></code>	<code>\$a != \$b</code> <code>\$a <> \$b</code>	Returns <code>true</code> if arrays <code>\$a</code> and <code>\$b</code> do <i>not</i> have the same key/value pairs.
Not identical	<code>!==</code>	<code>\$a !== \$b</code>	Returns <code>true</code> if array <code>\$a</code> is <i>not</i> identical to array <code>\$b</code> .

[Listing 8-7](#) shows some of these operators in action.

```
<?php
$cars1 = ['audi' => 'silver', 'bmw' => 'black'];
$cars2 = ['audi' => 'white', 'ferrari' => 'red'];
$names1 = ['matt' => 'smith', 'joelle' => 'murphy'];
$names2 = ['joelle' => 'murphy', 'matt' => 'smith',];

print_r($cars1 + $cars2);
var_dump($names1 == $names2);
var_dump($names1 === $names2);
❶ print_r(['rolls royce' => 'yellow', ...$cars1, ...$names1]);
```

Listing 8-7: Using array operators

First, we declare some example arrays to work with: `$cars1` and `$cars2` have car makes as keys and car colors as values, while `$names1` and `$names2` have first names as keys and last names as values. (Notice that `$names1` and `$names2` have the same elements, but in the opposite order.)

Then we apply operators to these arrays and print the results. We use the union (+) operator to combine `$cars1` and `$cars2`, and we test the equal (==) and identical (===) operators on `$names1` and `$names2`. We also use the array spread operator (...) to create a new array with a key of 'rolls royce' mapped to a value of 'yellow', as well as all the elements of the `$cars1` and `$names1` arrays ❶. Notice that we use `print_r()` to show the results of the operations that return arrays; this function displays arrays more succinctly than `var_dump()`. Running the script results in this output:

```
❶ Array
(
    [audi] => silver
    [bmw] => black
    [ferrari] => red
)
❷ bool(true)
bool(false)
❸ Array
```

```
(  
  [rolls royce] => yellow  
  [audi] => silver  
  [bmw] => black  
  [matt] => smith  
  [joelle] => murphy  
)
```

The first part of the output shows the result of `$cars1 + $cars2` ❶. Both arrays of cars have an element with a key of `'audi'`, but an array can't have two identical keys. As such, the union operator takes the `'audi' => 'silver'` element from `$cars1` but ignores `'audi' => 'white'` from `$cars2`, resulting in a three-element array. Next, the `true` and `false` outputs ❷ indicate that the `$names1` and `$names2` arrays are *equal*, since they have the same keys and values, but not *identical*, since the sequence of elements is different. The final array shows the result of using the spread operator `(...)` ❸. The new array has a `'rolls royce'` element, followed by the elements from `$cars1` and `$names1`.

It's worth underscoring what the spread operator `(...)` is doing here: it extracts the elements from one array and inserts them, one at a time, into another array. Without the spread operator, the entire array would be inserted as a single element into the new array, thus creating a multidimensional array, rather than its individual elements being spread into the new array. To illustrate, say we had omitted the spread operator before `$cars1` ❶ in [Listing 8-7](#), like this:

```
print_r(['rolls royce' => 'yellow', $cars1, ...$names1]);
```

The resulting array would have an element containing the whole `$cars1` array, as shown here:

```
Array  
(  
  [rolls royce] => yellow  
  [0] => Array  
    (  

```

```
        [audi] => silver
        [bmw] => black
    )

    [matt] => smith
    [joelle] => murphy
)
```

Now the second element in the array, with key 0, is itself an array containing the complete contents of `$cars1`. This example also illustrates how an array can mix integer keys with non-integer keys. When the whole `$cars1` array is added as an element to the new array, it's automatically given the first available integer key, 0, since it wasn't given a key manually. Meanwhile, the other elements in the new array all have explicitly assigned string keys. Arrays with mixed keys like this are rare; usually such an array would indicate something has gone wrong, such as the missing spread operator here.

Destructuring an Array into Multiple Variables

Sometimes it can be useful to extract the values from an array and assign them to separate variables, a process known as *destructuring*. If you know the number of elements in the array, you can destructure it in a single statement, as shown in [Listing 8-8](#).

```
<?php
$rainfallValues = [10, 8, 12];

❶ [$jan, $feb, $march] = $rainfallValues;

print "-- Monthly rainfall --\n";
print "Jan: $jan \n";
print "Feb: $feb \n";
print "Mar: $march \n";
```

Listing 8-8: *Destructuring a three-element array into three separate variables*

We declare the `$rainfallValues` array to have three elements. Then we destructure the array into the `$jan`, `$feb`, and `$march` variables ❶. For that, we list the target variables inside square brackets on the left of an assignment operator (`=`) and provide the variable containing the whole array on the right. Finally, we print out the values in the three variables, producing the following output:

```
-- Monthly rainfall --  
Jan: 10  
Feb: 8  
Mar: 12
```

Notice that the values from the array are successfully assigned into, and print out from, the individual `$jan`, `$feb`, and `$march` variables.

Callback Functions and Arrays

A *callback function*, or simply a *callback*, is a function that isn't called directly, but rather is passed as an argument to another function. The other function then calls the callback function for you. PHP has several functions that use callbacks in conjunction with arrays.

For example, `array_walk()` takes in an array and a callback function as arguments and applies the callback function to each element in the array, transforming the original array in the process. Similarly, `array_map()` takes in an array and a callback function, applies the callback to each array element, and returns a new array containing the results. Both `array_walk()` and `array_map()` are known as *higher-order functions*, since they take in a function as an argument.

If you've declared a function in a separate file (as previously discussed in [Chapter 5](#)) or are using one of PHP's built-in functions, you can use that function as a callback by passing a string containing the function's name to a higher-order function. For example, say we've declared a function called `my_function()` and we want to apply it to every element in `$my_array` by using `array_map()`. Here's how to do it:

```
$my_new_array = array_map('my_function', $my_array);
```

We pass the string `'my_function'` (the name of the desired callback) and the array as arguments to `array_map()`, which will call `my_function()` for each element in the array. The results are returned in a new array, which we store in the `$my_new_array` variable.

Rather than declare the callback function separately, another common approach is to define an *anonymous* (unnamed) callback function directly in the argument list for the higher-order function. Before we look at an anonymous function in the context of a higher-order function like `array_map()`, though, let's consider an anonymous function by itself to better understand the syntax. Here's a simple anonymous function that takes in a number and returns double its value:

```
function (int $n): int {return $n * 2;}
```

The function begins with the `function` keyword, followed by the function's signature, `(int $n): int`, which indicates that the function takes a single integer parameter `$n` and returns an integer value. Notice that the function signature doesn't include a name, since the function is anonymous. After the signature comes the anonymous function's body, which is enclosed in curly brackets. The body returns twice the value of the provided `$n` argument.

Another option is to write the anonymous callback as an *arrow function*, using a more concise syntax that uses the double-arrow operator (`=>`) to separate the function's signature and body. This syntax removes the need for the `return` keyword, the curly brackets around the body, and the semicolon to end the statement in the body. Here's the arrow-function version of our doubling operation:

```
fn (int $n): int => $n * 2
```

Instead of `function`, we now begin with `fn`, a reserved keyword for declaring arrow functions. Then comes the function's signature as before. Next, we write the double-arrow operator (`=>`), followed by an expression defining the function's return value (in this case, `$n * 2`). Without the curly

brackets, semicolon, and `return` keyword, the arrow function is extremely compact.

Now let's try using this arrow function as a callback. [Listing 8-9](#) shows how to pass the arrow function to `array_map()` in order to double every value in an array.

```
<?php
$numbers = [10, 20, 30];

$doubleNumbers = array_map(
    ❶ fn (int $n): int => $n * 2,
    $numbers
);

var_dump($doubleNumbers);
```

Listing 8-9: Passing an arrow callback function to `array_map()`

We declare a `$numbers` array containing 10, 20, and 30. We then call the `array_map()` function. For the first argument, we use arrow-function syntax to declare the doubling callback function we just discussed ❶. Notice that the arrow function ends with a comma, since it's part of the list of arguments to `array_map()`. The second argument is the `$numbers` array. The `array_map()` function will automatically apply the arrow function to each element in the array and return a new array containing the results. We store that new array in the `$doubleNumbers` variable. Here's the output of running this script and printing the resulting array:

```
array(3) {
    [0]=>
    int(20)
    [1]=>
    int(40)
    [2]=>
    int(60)
}
```

The `$doubleNumbers` array contains the values 20, 40, and 60. This indicates that the `array_map()` function successfully accessed each value in the `$numbers` array and applied the doubling arrow function to it.

Summary

Arrays are flexible data structures, especially when we begin assigning meaningful string keys to the values of the elements instead of using the default integer keys. In this chapter, you learned how to work with string-keyed arrays. You also saw how to embed arrays inside other arrays to create multidimensional arrays, and how to apply callback functions (written with arrow-function syntax) to every element of an array. Techniques like this are what make the array a sophisticated structure for representing and manipulating complex data.

Exercises

1. Use an array with string keys to store the following pairs of names and heights in meters:

Fred	1.82
Joelle	1.55
Robin	1.70

Write a `foreach` loop to iterate through the array elements and print them out.

2. Create a multidimensional array to represent the following data about movies:

Back to the Future	
duration	116
leadingActor	Michael J. Fox
The Fifth Element	
duration	126
leadingActor	Bruce Willis
Alien	

duration	117
leadingActor	Sigourney Weaver

3. Declare one array containing the odd numbers up to nine (1, 3, 5, 7, 9) and another with the even numbers (2, 4, 6, 8). Use the array spread operator (`...`) to combine the two arrays, and then sort them into numerical order with PHP's built-in `sort()` function.

9

FILES AND DIRECTORIES



Many applications require you to read or write data to and from files. In this chapter, we'll explore how to interact with files via PHP scripts. We'll focus primarily on simple *.txt* files, though we'll also touch on how PHP handles other common text file formats.

PHP provides many built-in functions for working with files. Some read or write files all in one go, while other, lower-level functions provide more granular control, allowing you to open and close files and selectively read or write at specific locations. Not all web applications will require you to work with external files, but knowing your way around these functions is still useful, in case the need arises. Outside of web applications, for example, you may find yourself needing to reformat the data inside files or to move and rename files and directories. With the functions we'll discuss here, you can write a PHP script to automate that process.

Reading a File into a String

If you know a file exists and you want to read all its contents into your script as a single string, you can do so in a single statement, simply by calling the built-in `file_get_contents()` function. To illustrate, let's first

create a file to be read in. [Listing 9-1](#) shows a file containing a programming haiku by Jorge Suarez (found at http://selavo.lv/wiki/index.php/Programming_haiku). Create a new file named *data.txt* containing these lines.

```
what is with this code?  
oh my, looks like I wrote it  
what was I thinking?
```

Listing 9-1: The text file data.txt containing a programming poem

This file contains three lines of text. The line breaks are a sign that the first two lines end with an invisible newline character.

Now that we have a file to work with, we can write a script to read and print its contents. Create a *main.php* file in the same directory as *data.txt* and enter the code in [Listing 9-2](#).

```
<?php  
$file = __DIR__ . '/data.txt';  
  
$text = file_get_contents($file);  
print $text;
```

Listing 9-2: A main.php script to read and print the contents of a file

First, we declare a `$file` variable containing the path and filename for our text file. Since the text file and main script are in the same directory, we create this file location string by concatenating the `__DIR__` magic constant (the path to the location of the main script) with a forward slash and the *data.txt* filename. We then use `file_get_contents()` to read the contents of the file into the `$text` variable. Finally, we print out the string containing the file contents.

Run the main script at the terminal and you should see the haiku printed across three lines, just as it appears in [Listing 9-1](#). This is because the invisible newline characters in the file made it into the `$text` string, just like the visible characters. We can prove these invisible characters exist in a couple of ways: by checking the size of the text file or by replacing the

newlines with visible characters in the string read from the file. To make it really easy to see how newline characters are part of a text file, let's replace the contents of *data.txt* with that of [Listing 9-3](#).

```
a
b
```

Listing 9-3: A simplified data.txt file

Now that the file consists of just two characters, each on a separate line, we can more easily examine the contents of the file. Update *main.php* to match [Listing 9-4](#).

```
<?php
$file = __DIR__ . '/data.txt';

$text = file_get_contents($file);

$numBytes = filesize($file);
$newlinesChanged = str_replace("\n", 'N', $text);

print "numBytes = $numBytes\n";
print $newlinesChanged;
```

Listing 9-4: An updated main.php script to prove the existence of newline characters

As before, we first read the contents of the file into the `$text` variable. Then we read the size of the file with the built-in `filesize()` function, which returns the file's number of bytes. In a text file with basic ASCII characters, each character (including invisible characters) takes up 1 byte, so we should expect the result to be 3. Next, we generate another string that replaces each newline character (`"\n"`) in `$text` with a capital letter `N`, storing the result in the `$newLinesChanged` variable. Finally, we print the file size and the updated string. Here's the output of running this script at the terminal:

```
numBytes = 3
aNb
```

The first line confirms that the file contains just three characters (bytes) of data: the letter `a`, a newline character, and the letter `b`. The second line is the string representing the contents of the file with the newlines made visible: `aNb` again confirms that the file contains just three characters, with a newline character between the two letters.

Confirming that the newlines exist isn't a trivial exercise: later in the chapter, we'll explore functions that work with a file's contents line by line. These functions rely on invisible newlines to know where one line ends and the next begins.

NOTE

The `file_get_contents()` function can also read files from the web rather than from your local machine if you pass it a full URL to the file's location. For example, try storing the URL <https://filesamples.com/samples/document/txt/sample1.txt> in the `$file` variable and then calling `file_get_contents($file)` as in [Listing 9-2](#). You should get back a string of nonsensical Latin text.

Confirming That a File Exists

The previous examples assume that a file named `data.txt` exists. In practice, however, it's a good idea to test that a file exists before attempting to read its contents. Otherwise, if you attempt to open or read a file that can't be found, you'll get a runtime warning such as the following:

```
PHP Warning:  file_get_contents(/Users/matt/nofile.txt): Failed to open
stream: No such file or directory in /Users/matt/main.php on
line 4
```

Execution will continue after the warning, which can lead to further warnings and errors if the script attempts to manipulate the contents of the nonexistent file. To make your code more robust and able to cope with a missing file, you can use the built-in `file_exists()` function. It returns a Boolean value confirming whether the provided file exists. Let's try it out by updating `main.php` with the contents of [Listing 9-5](#).

```
<?php
$file = __DIR__ . '/data.txt';
$file2 = __DIR__ . '/data2.txt';

$text = "file not found: $file";
$text2 = "file not found: $file2";

if (file_exists($file)) {
    $text = file_get_contents($file);
}

if (file_exists($file2)) {
    $text2 = file_get_contents($file2);
}

print $text . "\n";
print $text2 . "\n";
```

Listing 9-5: An updated main.php script to confirm the existence of a file before reading it

Here we add `$file2`, a second variable holding a path to a nonexistent file, *data2.txt*. Before attempting to read anything, we assign a default `file not found` message to the `$text` and `$text2` variables. This way, these variables will still hold something, even if we fail to read the contents of a file. We next use the `file_exists()` function in two successive `if` statements to ensure that we attempt to read the contents of *data.txt* and *data2.txt* only if those files can be found. Then we print the contents of `$text` and `$text2`, each followed by a newline character. Here's the result:

```
a
b
file not found: /Users/matt/data2.txt
```

Since *data.txt* can be found, its contents have been read into `$text` (replacing the default `file not found` message) and printed out. Meanwhile, since *data2.txt* doesn't exist, printing `$text2` ends up displaying a message indicating that the file can't be found.

“Touching” a File

Linux and macOS have a `touch file` terminal command that either updates the last accessed or modified timestamp of the specified file to the current datetime or creates an empty file if that file doesn’t already exist. PHP offers the almost identical `touch()` function, which provides another way to ensure that a file exists before trying to access it. If you don’t mind a file’s contents being empty, you can replace the default `file not found` messages and `if` statements from [Listing 9-5](#) with simple `touch()` statements, as shown in [Listing 9-6](#).

```
<?php
$file = __DIR__ . '/data.txt';
$file2 = __DIR__ . '/data2.txt';

touch($file);
touch($file2);

$text1 = file_get_contents($file);
$text2 = file_get_contents($file2);

print $text1 . "\n";
print $text2 . "\n";
```

Listing 9-6: An updated main.php script to “touch” files before reading them

We now pass each filename to `touch()` before using `file_read_contents()` to read the files. This lets us safely read the files without `if` statements and `file_exists()`, since we know `touch()` will create the files (albeit empty ones) if they don’t already exist.

Ensuring That a Directory Exists

We’ve so far been working with files in the same directory as the executing script, but a file could also be in a different directory. In that case, it’s important to confirm that the directory exists (and perhaps create it if it doesn’t), since just like a missing file, a nonexistent directory will trigger a runtime warning. PHP has two built-in functions for this: `is_dir()` returns

a Boolean value confirming whether a specified directory path can be found, and `mkdir()` attempts to create a directory at the specified path.

NOTE

The `mkdir()` function will throw a runtime warning if the directory it's trying to create already exists or if it can't be created based on the current permissions settings. For more on permissions, see “Directory and File Permissions” on [page 163](#).

To try these functions, update the contents of *main.php* as shown in [Listing 9-7](#).

```
<?php
$dir = __DIR__ . '/var';
$file = $dir . '/data.txt';

if (!is_dir($dir)) {
    mkdir($dir);
}

touch($file);

$text = file_get_contents($file);
print $text;
```

Listing 9-7: An updated main.php script to create a directory if it doesn't exist

We break the desired path and filename into two variables: `$dir` holds the path to the directory where the file is to be read from, and `$file` holds the path plus the filename. We set `$dir` to the */var* subdirectory within the directory where our script is executing (`__DIR__`); this subdirectory doesn't exist. The `if (!is_dir($dir))` statement checks whether `$dir` is *not* a valid directory path and calls `mkdir()` to create the directory if it isn't. We're then safe to call `touch()` on the file, since we now know the directory exists, and then to read the file, since `touch()` creates the file if it, too, doesn't exist.

The default option for `mkdir()` is that the function isn't recursive: it will fail to create a directory if the parent of that directory doesn't exist. However, the function has an optional `recursive` parameter; if it's set to `true`, the function will create any missing parent directories as well. [Listing 9-8](#) shows an example.

```
<?php
$dir = __DIR__ . '/sub/subsub';
$file = $dir . '/data.txt';

if (!is_dir($dir)) {
    mkdir($dir, recursive: true);
}

touch($file);

$text = file_get_contents($file);
print $text;
```

Listing 9-8: An updated main.php script to recursively create directories if they're missing

The directory path now includes a `/subsub` directory inside a `/sub` directory inside the current directory of the executing script. Inside the `if` statement, we call `mkdir()` with the `recursive` argument set to `true`. This ensures that the function won't create just the `/subsub` directory but also its parent `/sub` directory if necessary. We have to set `recursive` as a named argument, since `mkdir()` takes another optional argument to set the new directory's permissions, and this argument comes before `recursive` in the function signature.

DIRECTORY AND FILE PERMISSIONS

The default permission setting for a directory created with `mkdir()` is full *read-write-execute*, meaning anyone can read, write, or execute files in that directory. The best practice, however, is to use the minimum required permissions. To apply a different setting, use the function's optional `permissions` argument. You can also change the permissions for an existing file or directory by using the `chmod(file, permissions)` function. In both cases, you specify the permission as an octal (base-8) integer.

Octal numbers are prefixed with either `0` (zero) or `0o` (zero and a lowercase letter `o`). Here are the octal codes for some commonly used permissions:

`0o777` Everyone can read, write, and execute (default for the `mkdir()` function).
`0o600` The file owner can read and write; no access is given to anyone else.
`0o664` The file owner and group can read and write; any other user can only read.

Writing a String to a Text File

Just as you can use `file_get_contents()` to read the contents of a file into a string, you can write the contents of a string to a text file by using the reciprocal `file_put_contents()` function. This function automatically creates the file being written to if it doesn't exist, so you don't need to worry about testing the filename first. The updated *main.php* script in [Listing 9-9](#) shows how it works.

```
<?php
$content = <<<CONTENT
    the cat
    sat
    on the mat!
CONTENT;

$file = __DIR__ . '/newfile.txt';

file_put_contents($file, $content);
$text = file_get_contents($file);
print $text;
```

Listing 9-9: A main.php script writing data from a string to a file

First, we declare a three-line heredoc string, `$content`, using `CONTENT` as the delimiter. Then we set the `$file` variable to the current directory path plus the filename *newfile.txt*. Next, we call the `file_put_contents()` function, passing it the destination file and the text to write to that file. This should create a file *newfile.txt* containing the text from the `$content` heredoc. To confirm that the file has been created with the text content, we

use `file_get_contents()` to read the text back out of the file and into the `$text` variable, which we then print. Here's the result:

```
the cat
sat
on the mat!
```

The output matches the original heredoc string, indicating we successfully wrote the string to *newfile.txt* and read it back out again.

If the file you're trying to write to already exists, the default behavior of `file_put_contents()` is to completely replace (overwrite) the contents of that file. To avoid this, call the function with the `FILE_APPEND` option. This adds the new text to the end of the file after its existing content.

[Listing 9-10](#) shows an example, updated from [Listing 9-9](#).

```
<?php
$newContent = <<<CONTENT
    the rat
    spat
    on the cat!
    CONTENT;

$file = __DIR__ . '/newfile.txt';

file_put_contents($file, $newContent, FILE_APPEND);
$text = file_get_contents($file);
print $text;
```

Listing 9-10: A main.php script appending text to the end of a file

This time we create a different heredoc string and add it to *newfile.txt* by calling `file_put_contents()` with `FILE_APPEND` as a third argument. This should append the string after the current contents of the file, as the output confirms:

```
the cat
sat
```

```
on the mat!  
the rat  
spat  
on the cat!
```

Try running the code in [Listing 9-10](#) again without the `FILE_APPEND` option. You'll find that only the text from `$newContent` appears in the output, since the existing text in the file is overwritten.

Managing Files and Directories

Beyond reading from and writing to files, PHP offers functions to help manage existing files and directories. For example, you can delete a file with the `unlink()` function or delete a whole directory with `rmdir()`. Both functions return `true` if successful or `false` otherwise. As with reading files, it's important to test for the existence of a file or directory before attempting to delete it. Otherwise, if you call `unlink()` or `rmdir()` on a file or directory that doesn't exist, you'll get a warning (but execution will continue). [Listing 9-11](#) shows these functions in action.

```
<?php  
$dir = __DIR__ . '/var';  
$file = $dir . '/data.txt';  
  
if (!is_dir($dir)) {  
    mkdir($dir);  
}  
  
touch($file);  
  
var_dump(is_dir($dir));  
var_dump(file_exists($file));  
  
unlink($file);  
rmdir($dir);  
  
var_dump(file_exists($file));  
var_dump(is_dir($dir));
```

Listing 9-11: A main.php script to create and then delete a directory and a file

As in some earlier examples, we declare the target directory and filename in two variables, `$dir` and `$file`. We then create the directory if it doesn't already exist and `touch()` the file. At this point, we should be confident that a *data.txt* file exists in a */var* directory; we confirm this by `var_dump()`ing the results of calling `is_dir()` and `file_exists()`. Next, we use `unlink($file)` and `rmdir($dir)` to delete the file and its directory. Finally, we make the same `var_dump()` calls again to make sure that neither the directory nor the file exists when the script finishes execution. If you run this script, you should see `true, true, false, false` displayed, confirming that the directory and file existed and then were successfully deleted.

Another useful file-management function is `rename()`, which changes the name of a file or directory. For example, you could rename *oldfile.txt* to *newfile.txt* with this statement:

```
rename('oldfile.txt', 'newfile.txt');
```

You need to be careful with this function, testing that the old file or directory exists first. It's also important to be mindful about the new file or directory. If you're renaming a file and another file already exists with that name, it will be overwritten with no error or warning, which could be problematic if you need the contents of that overwritten file. If you're renaming a directory and the new directory already exists, a warning will be generated, which is also not ideal, since it's best to avoid warnings. If you're renaming a file into a different directory, you also should ensure that the new directory exists and, if appropriate, is writable (which is required by Windows). See <https://www.php.net/manual/en/function.rename.php> for more about this function.

Reading a File into an Array

PHP's built-in `file()` function reads the contents of a file into an array rather than a single string, with one array element for each line in the file. This is useful when you want to perform an action for each line (such as displaying the line's contents alongside its line number, as in the following

example), or when each line represents one item in a set of data to be processed, such as the data in a comma-separated values (CSV) file. [Listing 9-12](#) shows a main script demonstrating the `file()` function.

```
<?php
$file = __DIR__ . '/data.txt';

$lines = file($file);

foreach ($lines as $key => $line) {
    print "[$key]$line";
}
```

Listing 9-12: A main.php script to loop through and print each line of a text file

We pass the file information (in the `$file` variable) to the `file()` function, which reads the contents of *data.txt* line by line into an array called `$lines`. Then we use a `foreach` loop to print each element of the array (a line from the file) individually, along with its numeric key. If *data.txt* contains the three-line haiku from [Listing 9-1](#), the output should look as follows:

```
[0]what is with this code?
[1]oh my, looks like I wrote it
[2]what was I thinking?
```

You can pass optional flags as a second argument to the `file()` function to, for example, exclude the newline character at the end of each line (`FILE_IGNORE_NEW_LINES`) or completely ignore empty lines in the file (`FILE_SKIP_EMPTY_LINES`).

Using Lower-Level File Functions

The `file_get_contents()` and `file_put_contents()` functions take care of all the steps of working with a file for you, such as opening the file, accessing its contents, and closing the file again. In most situations, those functions are all you need. Sometimes, however, you may need to work

with files at a lower level, perhaps processing them one line, or even one character, at a time. In those cases, you might need to explicitly manage the various file-access steps in your code through a series of separate, lower-level function calls.

PHP's lower-level file functions require you to work with a *filesystem pointer* (or just *file pointer*), a reference to a location in the file's data. Internally, PHP treats a file as a *bytestream* (a resource object that can be read from and written to in a linear fashion), and the file pointer provides access to that stream. You obtain a file pointer by calling `fopen()` with a path to the file you want to access. You also have to pass in a string specifying *how* you want to interact with the file; for example, files can be opened only for reading, only for writing, for both reading and writing, and so on. [Table 9-1](#) shows the strings for specifying some common `fopen()` modes.

Table 9-1: Common `fopen()` Modes

Mode string	Description	Position of file pointer	Outcome if file doesn't exist
'r'	Read only	Beginning of file	Warning
'r+'	Read and write (overwrite)	Beginning of file	Warning
'w'	Write only (overwrite)	Beginning of file (and <i>truncate</i> the file by removing any existing content)	Attempt to create a file
'a'	Write only (append)	End of file	Attempt to create a file

The typical sequence of actions when working with a file is as follows:

1. Open a file in the appropriate mode and get a file pointer.
2. Change the location of the file pointer in the file if necessary.
3. Read or write at the location of the file pointer.
4. Repeat steps 2 and 3 as required.
5. Close the file pointer.

[Listing 9-13](#) demonstrates this process. This script achieves the same results as [Listing 9-2](#) (reading the contents of a file to a string) by using the lower-level `fopen()`, `fread()`, and `fclose()` functions.

```
<?php
$file = __DIR__ . '/data.txt';

$fileHandle = fopen($file, 'r');
$filesizeBytes = filesize($file);
$text = fread($fileHandle, $filesizeBytes);
fclose($fileHandle);

print $text;
```

Listing 9-13: Using lower-level functions to read a file

First, we use `fopen()` to open *data.txt*, using the string `'r'` to specify read-only mode. The function returns a file pointer located at the beginning of the file, which we store in the `$fileHandle` variable. Next, we call `filesize()` to look up the size of the file (in bytes). We then call the `fread()` function, passing it the file pointer and the size of the file (`$filesizeBytes`) to read the entire contents of the file into the `$text` variable. If we wanted to read only part of the file, we could specify a different number of bytes as the second argument to the `fread()` function. (We'd also want to specify a different number of bytes if the file pointer were located somewhere other than the beginning of the file.) To finish up, we close the file by passing the file pointer to the `fclose()` function. Closing the file enables it to be used by other system processes and protects it from being corrupted if any errors occur in the script currently being executed.

This example illustrates some of the most common low-level file functions, but PHP has many others. For example, `fgets()` reads one line of a file (up to the next newline) from the current file-pointer location, and `fgetc()` reads just one character from the current file-pointer location. The `feof()` function takes in a file pointer and returns `true` or `false` based on whether the pointer is at the end of the file. This is useful for loops such as the following:

```
while (!feof($fileResource)) {  
    // Do something at current file pointer position  
}
```

Here we use the NOT operator (!) to negate the result of `feof()`, so the loop will keep repeating until the pointer gets to the end of the file. Inside this kind of loop, we might read a line from the file with `fgets()`, read the next character with `fgetc()`, or read a fixed number of bytes with `fread()`. Logic in the loop would then process the data (if successfully read), and if we reach the end of the file while reading, the loop would terminate.

Some functions are just for working with and changing the file pointer. For example, `rewind()` moves the file pointer back to the beginning of the file, and `ftell()` returns the current location of the pointer, specified as the number of bytes from the start of the file. The `fseek()` function moves the file pointer to a given position in the file specified relative to its current position, the beginning or the end of the file.

Processing Multiple Files

Let's combine a lot of what we've discussed so far in this chapter in a more sophisticated example that programmatically extracts data from multiple files and collects it all in a new summary file. We'll attempt to gather the names and game scores of three players, each in a separate file (*joe.txt*, *matt.txt*, and *sinead.txt*), reformat the data, and write it to a single output file called *total.txt*. [Listings 9-14](#) through [9-16](#) show the three raw data files we want to process.

```
Joe  
O'Brien  
  
55
```

Listing 9-14: joe.txt

```
Matthew  
  
Smith
```

Listing 9-15: matt.txt

```
Sinead  
Murphy  
  
101
```

Listing 9-16: sinead.txt

Notice that the content in each data file is a little messy, with randomly located blank lines: [Listing 9-15](#) ends with a blank line, and [Listing 9-16](#) starts and ends with two blank lines. That said, each data file has the same sequence of content: a line containing the player's first name, a line with their last name, and a line with their integer score.

In the output file, we want to consolidate all the data about each player onto a single line, as well as display the total of all three players' scores. [Listing 9-17](#) shows how the resulting *total.txt* file should appear.

```
Player = Joe O'Brien / Score = 55  
Player = Matthew Smith / Score = 99  
Player = Sinead Murphy / Score = 101  
total of all scores = 255
```

Listing 9-17: The consolidated total.txt file we want to create

To achieve this final result, we'll need to handle each part of the data files differently, so we can't simply load a whole file into a string with `file_get_contents()`. It will be better to use `file()` to read in each file as an array of individual lines.

When working with multiple files, PHP's oddly named `glob()` function is a powerful tool. It returns an array of file and directory paths that match a given pattern. This is particularly helpful for identifying and then looping through all the data files in a given location. For example, the following

statement provides an array of paths to all *.txt* files in the */data* subfolder relative to the location of the executing script:

```
$files = glob(__DIR__ . '/data/*.txt')
```

The *** is a wildcard representing any number of characters, so */data/*.txt* will match any filename with a *.txt* extension in the given folder. That's exactly what we'll need to gather the player data files in this example.

Start a new project and create a */data* subfolder containing the text files *joe.txt*, *matt.txt*, and *sinead.txt* shown previously in [Listings 9-14](#) through [9-16](#). Then, in the main project folder, create a *main.php* script with the contents of [Listing 9-18](#).

```
<?php
$dir = __DIR__ . '/data/';
$fileNamePattern = '*.txt';
$files = glob($dir . $fileNamePattern); ❶

$outputFile = __DIR__ . '/total.txt';
touch($outputFile);
unlink($outputFile);

$total = 0;
foreach ($files as $file) { ❷
    $lines = file($file, FILE_IGNORE_NEW_LINES | FILE_SKIP_EMPTY_LINES);
    $firstName = $lines[0];
    $lastName = $lines[1];
    $scoreString = $lines[2];
    $score = intval($scoreString);

    $outputFileHandle = fopen($outputFile, 'a');
    fwrite($outputFileHandle, "Player = $firstName $lastName
/ Score = $score\n"); ❸
    fclose($outputFileHandle);
```

```

        $total += $score;
    }

    $outputFileHandle = fopen($outputFile, 'a');
    fwrite($outputFileHandle, "total of all scores = $total");
    fclose($outputFileHandle);

    print file_get_contents($outputFile); ❹

```

Listing 9-18: A script processing multiple files

We first assign the path to the */data* subfolder from the location of the executing script to the `$dir` variable, and the filename pattern string `'*.txt'` to `$fileNamePattern`, using the `*` wildcard to represent any *.txt* file. We then call `glob()` to get an array of all the files in `$dir` matching the pattern in `$fileNamePattern`, storing the result in the `$files` variable ❶. Thanks to `glob()`, we know that all the files in the `$files` array exist, so we can avoid the ordeal of checking whether they exist before trying to read them.

Next, we assign a path to *total.txt* to the `$outputFile` variable. This file may or may not exist already, but we want a fresh output file each time we run the script. We therefore `touch()` the file, which creates it if it doesn't exist already, and then use `unlink()` to delete the file. Now we can be sure that we're writing to an empty file when it comes time to gather the data into *total.txt*.

After initializing the `$total` variable to 0, we use a `foreach` loop ❷ to iterate over the filepaths in the `$files` array, storing each path in a temporary `$file` variable. For each file, we use `file()` to read the contents into an array called `$lines`. Calling the function with the `FILE_IGNORE_NEW_LINES` and `FILE_SKIP_EMPTY_LINES` flags ensures that end-of-line characters will be ignored and that empty lines will be excluded from the resulting array. Knowing what we know about each data file, this means that `$lines` should be a three-element array: the first element is the player's first name, the second element is their last name, and the third element is their score (represented as a string). We read these values from the array into separate `$firstName`, `$lastName`, and `$scoreString`

variables and use the built-in `intval()` function to convert the score from a string to an integer.

Still within the `foreach` loop, we call `fopen()` to get a file pointer to the output file (*total.txt*) in write-append mode (specified with the `'a'` mode string), meaning the pointer will be located at the end of the file. The first time through the loop, *total.txt* won't exist, so `fopen()` will create the file. We then use `fwrite()` to append a string to the output file, summarizing the player's name and score and ending with a newline character (`\n`) ❸. We close the output file with `fclose()` and add the current player's score to the `$total` variable.

Finally, after the `foreach` loop has completed, we once again access the output file in write-append mode and append a final string including the value of `$total`. Then, to make sure this has all worked, we call `file_get_contents()` to read the output file into a string and print the result ❹. Notice that we call the function directly from the `print` statement, instead of storing the string in a variable first.

If you run the *main.php* script, you should get the *total.txt* file shown previously in [Listing 9-17](#). In fact, you can run this script as many times as you want and the result will always be the same, since any existing *total.txt* file is deleted with the combination of the `touch()` and `unlink()` functions.

Strictly speaking, our *main.php* script isn't the most efficient way to code the desired logic. We don't need to open and close the output file each time during the `foreach` loop; we could open it just once before the loop and then close it once after appending the total score. However, opening it each time through the loop illustrates the value of write-append mode, which places the file pointer at the end of the file. This way, any new content written to the file is added after any existing content.

JSON and Other File Types

PHP can work with more than *.txt* files. For example, it can also work with JavaScript Object Notation (JSON) and other text-based data formats. For JSON data, the built-in `json_encode()` function can turn a PHP array into a JSON string, and the `json_decode()` function does the opposite. This type of conversion is particularly smooth since JSON data, like PHP arrays,

revolves around key/value pairs. [Listing 9-19](#) shows these functions in action.

```
<?php
$filePath = __DIR__ . '/data.json';

$data = [
    'name' => 'matt',
    'office' => 'E-042',
    'phone' => '086-111-2323',
];

$jsonString = json_encode($data);

file_put_contents($filePath, $jsonString);

$jsonStringFromFile = file_get_contents($filePath);
print $jsonStringFromFile;

$jsonArrayFromFile = json_decode($jsonStringFromFile, true);
print "\n";
var_dump($jsonArrayFromFile);
```

Listing 9-19: A script to convert an array to JSON, and vice versa

We store a path to *data.json* in the `$filePath` variable. Then we declare a `$data` array that maps the values 'matt', 'E-042', and '086-111-2323' to the keys 'name', 'office', and 'phone', respectively. Next, we use the `json_encode()` function to convert the array to a JSON-formatted string, storing the result in the `$jsonString` variable. We then use `file_put_contents()` to write the JSON string to the *data.json* file, just as we would use it to write to a *.txt* file.

The rest of the script goes through the same process in reverse. We use `file_get_contents()` to read the JSON data from the file into the `$jsonStringFromFile` variable, which we print out. The variable contains a JSON string, but we use `json_decode()` to convert the string into a PHP array, which we display using `var_dump()`. We need to provide `true` as a second argument to the `json_decode()` function, or the result will be a type

of object rather than an array. Here's the output of running this script at the terminal:

```
{"name":"matt","office":"E 042","phone":"086 111 2323"}
array(3) {
    ["name"]=>
    string(4) "matt"
    ["office"]=>
    string(5) "E-042"
    ["phone"]=>
    string(12) "086 111 2323"
}
```

The first line shows the JSON string that we wrote into and read back out of the *data.json* file. The string consists of a JSON object, delimited by curly brackets, containing three key/value pairs separated by commas. The keys are set off from their corresponding values by colons. The rest of the output shows the contents of `$jsonArrayFromFile`, the array created by decoding the JSON data. Notice the direct correlation between the key/value pairs in the JSON object and the key/value pairs in the PHP array.

For YAML Ain't Markup Language (YAML) text data files, PHP provides several functions. For example, `yaml_parse()` and `yaml_emit()` are similar to `json_decode()` and `json_encode()` but for converting between YAML strings and PHP arrays. PHP also has direct file-to-string and string-to-file YAML functions: `yaml_parse_file()` and `yaml_emit_file()`.

For CSV files, PHP has the direct file-to-string and string-to-file functions `fgetcsv()` and `fputcsv()`. The `str_getcsv()` function takes a string in CSV format and converts it to an array. However, the function has some flaws. It doesn't escape newline characters, for example, so it can't cope with typical CSV files from spreadsheets like Google Sheets or Microsoft Excel. Perhaps because of this nonstandard treatment of CSV data, PHP doesn't have a reciprocal function to create a CSV-encoded string from an array.

Working with eXtensible Markup Language (XML) is a little more complex. PHP represents XML data with objects, so you need to be

confident with the basics of object-oriented programming to use functions such as `simplexml_load_file()` and classes such as `SimpleXMLElement`. However, PHP provides several powerful ways to traverse and manipulate XML data once you know how to use these features of the language. We'll discuss object-oriented PHP in [Part V](#).

Summary

In this chapter, we worked with basic PHP functions like `file_get_contents()` and `file_put_contents()` for reading and writing data to and from external files. We also discussed the `file()` function, which reads the lines of a file into separate array elements, and low-level functions like `fread()` and `fwrite()` that let you traverse a file by using a pointer. We explored how to ensure that a file or directory exists (or doesn't exist) before interacting with it, and how to use `glob()` to get a reference to all the files that match a certain criterion. Although we mostly worked with `.txt` files, we also touched on some PHP functions for interacting with JSON, YAML, CSV, and XML data formats.

Exercises

1. Find a limerick online or write your own. Here's one I found:

```
A magazine writer named Bing  
Could make copy from most anything  
But the copy he wrote  
of a ten-dollar note  
Was so good he now lives in Sing Sing
```

Write a script that declares an array; each element of the array is a line from the limerick. Then write those lines to a text file named *limerick.txt*.

2. Find a sample JSON file online that's accessible through a URL (for example, at <https://jsonplaceholder.typicode.com>). Write a script that reads the JSON string from the URL, converts it to an array, and then uses `var_dump()` to display the array.

3. Add a new data file for a game player and their high score in the *data* folder to be processed by the script in [Listing 9-18](#). Run the main script, and you should see another line added to the output file and the new score added to the total.

PART III

PROGRAMMING WEB APPLICATIONS

10

CLIENT/SERVER COMMUNICATION AND WEB DEVELOPMENT BASICS



As a language of the internet, PHP is closely connected to the communications between web *clients* and web *servers*. In this chapter, we'll look at how clients and servers work, and we'll examine the messages that pass between them. We'll also see how to efficiently embed PHP statements in static HTML code to construct a full HTML text file that a web browser client can understand and render onscreen as a web page. Finally, we'll discuss how a typical PHP web application is structured, including a first look at the model-view-controller (MVC) architecture.

Whether you realize it or not, you probably use clients and servers every day. When you check your email or social media accounts, you're using a client application to communicate with a server to request updates. These kinds of apps are continually making requests to servers; for

example, your email app requests from servers such as Google Gmail or Apple iCloud in order to download any new email and update the messages and folders on your phone to mirror any changes.

You can run a web server application in two places: locally on your own computer or publicly on an internet-accessible computer. As a PHP programmer, you'll do a lot of your software development locally on your own machine. Then, when you think a project is ready, you'll test it on a public server, and finally publish the website live when all testing is complete.

The HTTP Request-Response Cycle

At the heart of web-based client/server communications is the *http request-response cycle*. At a high level, a client sends a *request* to the server, and the server returns a *response* to the client. The response itself may be an error code, or it could be a message whose body is text, an image file, a binary executable, or other content. [Figure 10-1](#) illustrates a simple request-response cycle.

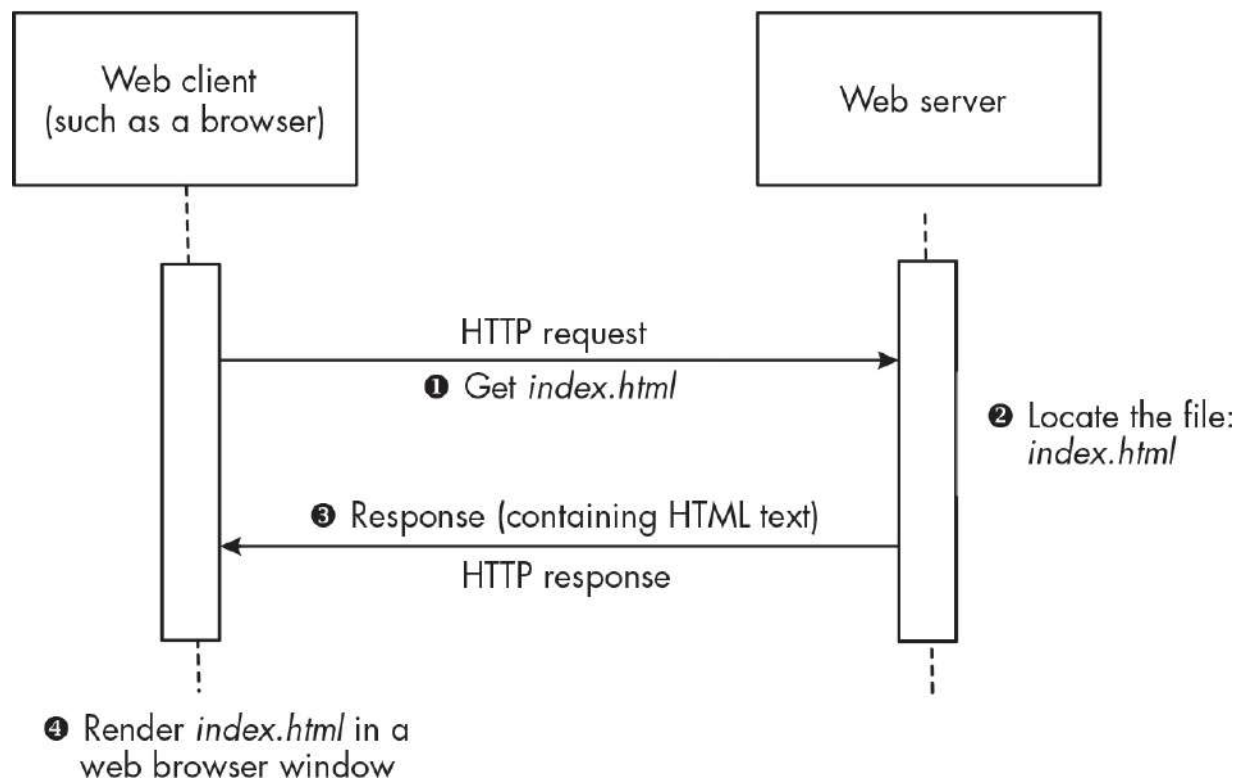


Figure 10-1: A simple HTTP request-response cycle

The client, a web browser, sends a request asking for the *index.xhtml* file ❶. The server receives and decodes the request, then searches for and successfully finds the requested resource (file) ❷. The server then creates and returns a response, whose body is the HTML text of *index.xhtml* ❸. Finally, the web browser reads through the received HTML and displays the web page contents nicely to the user in the browser window ❹.

Clients can send different types of requests. The two most common request methods are `GET` and `POST`. The HTTP `GET` method is simpler and, when using a web browser client, displays much of what's being sent in the web browser URL address bar. For example, if you use the Google search engine to search for the phrase *cheese cake*, you'll see those words appear at the end of the URL when you send the query to Google:

https://www.google.com/search?q=cheese+cake. In fact, anytime you type a URL into the web browser address bar and hit `ENTER`, you're sending a `GET` request.

The `POST` method, on the other hand, can hide much of what's being sent in the body of the request message. Therefore, it's often used for more private website operations.

In addition to `GET` and `POST`, the original HTTP 1.0 defined a third method, `HEAD`. It asks for a response with no body, just the headers, which contain general information about the response. Since the introduction of HTTP 1.1, five other methods are permitted (`OPTIONS`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`). These aren't needed for the level of web development in this book, although they can be useful for sophisticated web applications.

Response Status Codes

At the beginning of every HTTP response returned by the server is a three-digit HTTP *status code* that tells the client the status of the server's attempt to process and fulfill the request. All HTTP-compliant servers must use a set of standard codes, and on top of that, custom codes are used by different servers. The most common codes are `200 OK` to indicate that a request has been successfully fulfilled and `404 Not Found` to indicate that the server was unable to find the requested resource.

The first digit of the code indicates the general status of the server's interpretation and processing of the request. Here's a summary of what the

first-digit prefixes signify:

1nn (information) The request headers were received and understood, and further processing is needed. In other words, “So far so good, but not finished yet.” These status codes are fairly uncommon. They’re informational and used when the server needs to communicate some information, but not a full response, back to the client.

2nn (success) The request was received, understood, and accepted (for example, 200 OK).

3nn (redirection) The request was understood, but the client must take further action, such as choosing from options (300 Multiple Choices) or following a new URL if the resource has permanently moved (301 Moved Permanently).

4nn (client error) Either the request is invalid (such as 400 Bad Request), or the server can’t fulfill the request because of client error (such as 404 Not Found or 403 Forbidden).

5nn (server error) The server has experienced an error or is unable to complete the request for other reasons. Examples include 500 Server Error and 502 Service Unavailable.

You can learn more about HTTP and its status codes at Todd Fredrich’s free online REST API tutorial: <https://www.restapitutorial.com>.

An Example GET Request

Let’s look at a simple example of the request-response cycle by examining what happens behind the scenes when we visit the No Starch Press website. First, you need to display the browser request-response inspection tools. In Google Chrome, these tools are usually accessible as a menu item named Developer Tools. Once the developer tools are open, you’ll see a window like that at the bottom of [Figure 10-2](#).



Figure 10-2: A `GET` request to the No Starch Press home page

Click the **Network** tab, and you're ready to record and examine the HTTP request-response cycle. Type **nostarch.com** in the browser URL address bar. When you press `ENTER`, you should see the home page appear. Find the Name column on the left of the developer window, locate the first

file, which should be *nostarch.com*, and click it. Click **Headers** to see the HTTP headers summary, shown in [Figure 10-3](#).

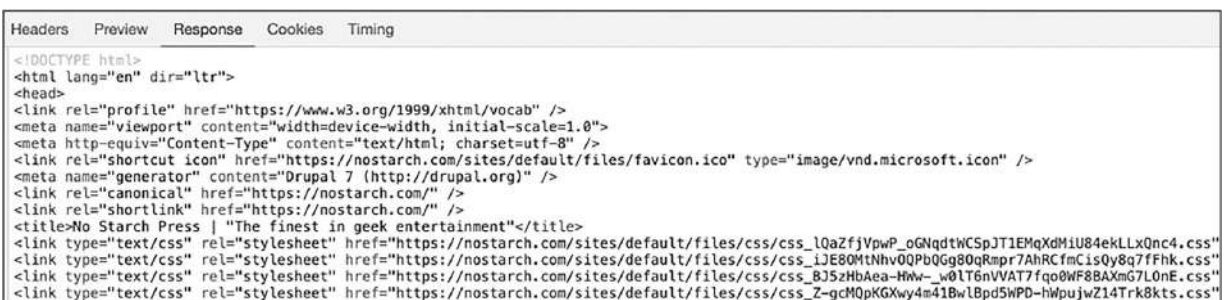
This summary indicates that the HTTP request is for the URL <https://nostarch.com> and that the request method is `GET` (since we just entered a URL in the address bar). The most important part of the HTTP response header is the success status code of `200`.

Scroll farther down the HTTP headers contents and you'll see full details of both the HTTP request and HTTP response headers. Under the Request Headers section, you can see the list of file types that the web client is willing to accept, such as HTML, XML, images, and so on. You can also see which human language the content is available in (for example, `EN` for English). Correspondingly, the response headers indicate the actual content type in the body of the response, such as `text/html`, the date the file was last modified, and so on.

NOTE

The majority of modern websites now use HyperText Transfer Protocol Secure (HTTPS), which enables the client and server to exchange certificates allowing HTTP messages to be securely encrypted. This is why the No Starch Press URL begins with `https://`. HTTPS is built into many PHP web servers, so we won't go into it at this point.

Now click the **Response** tab to see the content of the response's body, shown in [Figure 10-3](#). This is the HTML text that the web browser receives and then renders to make an attractive-looking graphical web page for you to see and interact with.

The image shows a screenshot of a web browser's developer tools, specifically the 'Response' tab. The tab is selected, and the content area displays the raw HTML of the response. The HTML starts with the DOCTYPE declaration and includes various meta tags and links. The title of the page is 'No Starch Press | The finest in geek entertainment'. The content is partially obscured by a large, semi-transparent watermark that reads 'nostarch.com'.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
<link rel="profile" href="https://www.w3.org/1999/xhtml/vocab" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="shortcut icon" href="https://nostarch.com/sites/default/files/favicon.ico" type="image/vnd.microsoft.icon" />
<meta name="generator" content="Drupal 7 (http://drupal.org)" />
<link rel="canonical" href="https://nostarch.com/" />
<link rel="shortlink" href="https://nostarch.com/" />
<title>No Starch Press | "The finest in geek entertainment"</title>
<link type="text/css" rel="stylesheet" href="https://nostarch.com/sites/default/files/css/css_l0aZfjVpWp_oGNqdtWCSpJT1EMqXdM1U84ekLLx0nc4.css" />
<link type="text/css" rel="stylesheet" href="https://nostarch.com/sites/default/files/css/css_iJE80MtNhv00PbQg80qRmpr7AhRCfmCis0y8q7fFhk.css" />
<link type="text/css" rel="stylesheet" href="https://nostarch.com/sites/default/files/css/css_BJ5zHbAea-HWw-_w0LT6nVVAT7fq00WF8BAXmG7L0nE.css" />
<link type="text/css" rel="stylesheet" href="https://nostarch.com/sites/default/files/css/css_Z-gcMQpKGXwy4m41Bw1Bpd5WPD-hWpujwZ14Trk8kts.css" />
```

Figure 10-3: HTML text content in the HTTP response body

At the bottom of the HTML code, you'll see a list of CSS links. When processing the received HTML, the web client (browser) looks for any additional content files needed for the web page, like CSS stylesheets, image files, and JavaScript files. The browser quickly (we rarely notice this with modern network speeds) makes additional HTTP requests to the server for each of these files, and as the corresponding HTTP responses arrive, the browser renders the web page. These extra files received from the web server can be seen in the Name column in [Figure 10-2](#), beneath the original request to *nostarch.com*. They have names like *css_lQaZ* and so on.

It's important to underscore that not every HTTP request has to be initiated by a human user entering a URL, clicking a link, or submitting a form. The web browser can, behind the scenes (asynchronously), make additional requests for required resources such as images, CSS files, and JavaScript files. These additional requests may be to the same web server that delivered the HTML the browser is processing, or to other web servers (perhaps to download a free Google font, for example, or the Bootstrap CSS and JavaScript).

NOTE

JavaScript code can also make additional HTTP requests, such as retrieving data from remote websites. This is known as asynchronous JavaScript and XML (AJAX), although many types of data files may be retrieved, such as JSON and plaintext, so such HTTP requests aren't limited to retrieving only XML data. This topic is beyond the scope of this PHP book.

How Servers Operate

We've discussed at a high level how clients and servers communicate through HTTP requests and responses. Now let's take a closer look at how web servers function. We'll also begin to see how PHP can play a role in the server's operations.

Simple Web Servers for File Retrieval

The task of a simple web server is to listen for requests for resources and, when a request is received, to identify the resource requested and return

either a message containing the resource or an error message if it can't be found. A simple web server is basically a file server that's able to understand HTTP requests and send HTTP responses. [Figure 10-4](#) illustrates a simple web server.

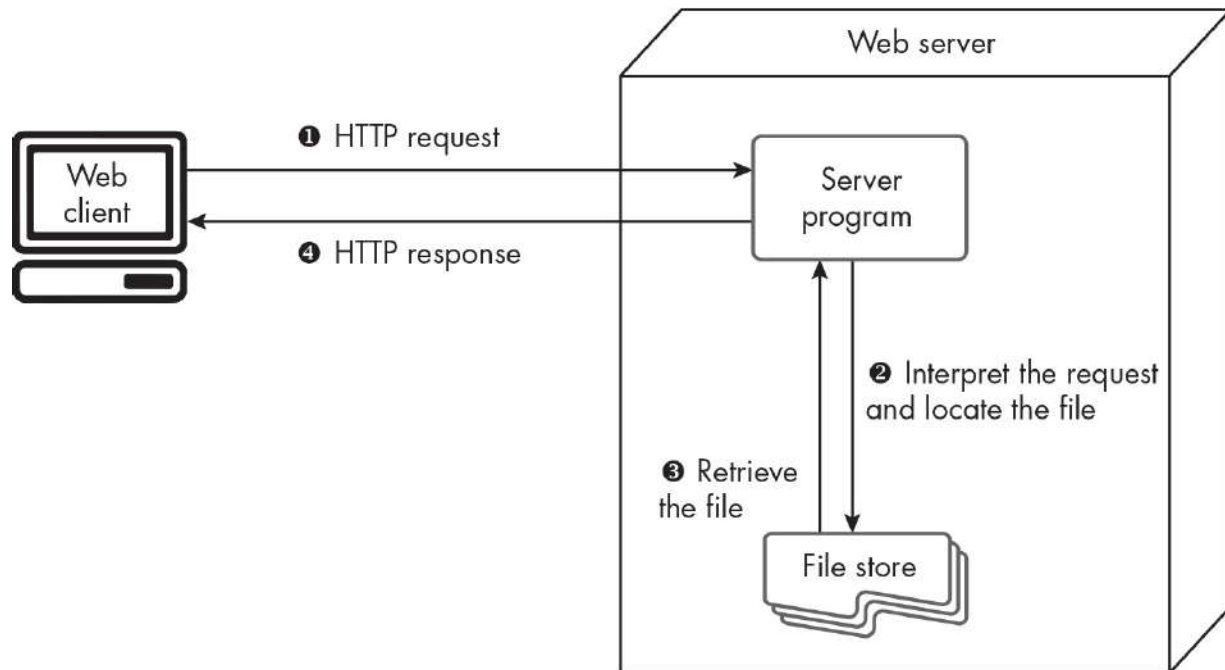


Figure 10-4: A simple web server communication with a web client

Typically, the client sends a `GET` request ❶, requesting a file such as *index.xhtml*, *style.css*, or *logo.png*. The server receives and interprets the request, then searches for the requested resource (file) ❷. If the file can't be found, the server creates and returns a `404 Not Found` error. If the file is found, the server retrieves its contents ❸. Finally, the server creates and returns a response to the client ❹. The response body is the content of the requested file, and its header includes the `200 OK` status code.

A good analogy for this process is that a simple web server functions like a librarian in a library: the librarian goes off to locate a requested book and returns with either the book or a message saying the book can't be located.

Simple web servers are sufficient for hypertext or hypermedia browsing of an unchanging set of HTML pages, such as a set of frequently

asked questions (FAQ) and answer paragraphs or reference materials that rarely need to be updated, like a user manual. Simple web servers are *stateless*, meaning the same request will always get the same file returned. Different clients will also get the same file returned. This is often termed *static* content to indicate that it's unchanging. Returning to the librarian analogy, you wouldn't expect a librarian to change the content of a book as they're retrieving it.

We can summarize the behavior of simple stateless web servers as follows:

- Never changes
- The same regardless of whether the user has visited before
- The same for every user

Most web activity is more interactive than simply clicking links to specific static documents. The majority of modern web projects require *dynamic interactivity*, in which the system responds differently according to user inputs. Dynamic interactivity encompasses tasks like processing web forms, managing shopping carts, customizing content based on recent browsing history, and more. Most PHP web applications are *dynamic* web servers, which we'll explore next.

Dynamic Web Servers for Processing Data

For a web-based system to be interactive beyond static resource retrieval, further technologies are required beyond basic content markup and hypertext linking. These capabilities include the following:

- Support for user input methods, like inputting text, clicking buttons, and choosing from menus
- Short code scripts that can process and respond in different ways to different user inputs
- Methods for the browser to send user inputs or data to the server programs that will process the data and generate interactive responses

Dynamic servers with capabilities such as these handle many typical modern internet activities, like entering keywords into a search engine and being presented with a tailor-made page of prioritized links, logging into your personal email system and retrieving your own email in your inbox,

and browsing catalogs of products online and making a purchase using a credit card.

In this book, we're most interested in dynamic web servers that understand and can run PHP scripts. [Figure 10-5](#) illustrates client communication with one such dynamic web server.

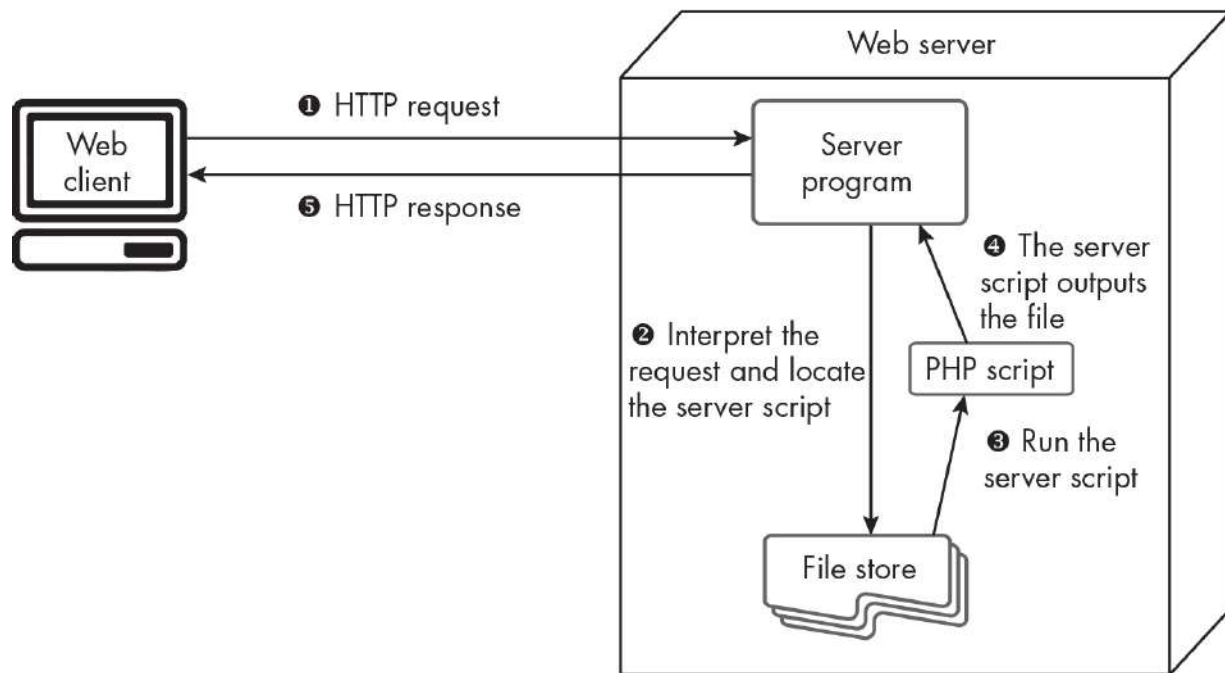


Figure 10-5: A dynamic web server communication with a web client

In this model, the client sends an HTTP request to the server ❶. Then the server program interprets the request and identifies which PHP server script should be executed ❷. The script is executed ❸ and generates output, such as HTML text. Running a PHP script can also trigger other actions on the web server, such as communication with a database, which is something we'll explore in [Part VI](#). Next, the web server application receives the output ❹. Finally, the output is packaged up in the body of an HTTP response message and returned with appropriate headers to the client that originally made the request ❺.

The Routing Process

Routing is the process the web server uses in deciding what to do to respond appropriately to the HTTP request it has received; the server examines the request and determines what action it believes the client is requesting, such as asking for a file, trying to log in with username and password data included in the request, deleting an item from a database, and so on. In the simplest scenario, the request contains a valid path for a specific resource file, like */images/logo.jpg* or */styles/homepage.css*. In this case, the web server acts like a file server and returns an HTTP response message containing the contents of the file with appropriate header information.

If a valid path to a publicly available *.php* file is requested, such as */about.php*, that PHP script will be interpreted and executed to build the HTTP response that's returned to the client. If no specific file is requested, almost all web servers have *default routing* defined, which will often route to a home page file, usually named *index*. Simple static web servers will look for *index.xhtml* to return as the default home page, whereas PHP web servers will usually look first for *index.php* and perhaps then look for *index.xhtml* if no default PHP file is found. If no file is requested and no index file is found, the server will return a 404 Not Found response.

Sophisticated PHP web applications will use logic encoded inside the default *index.php* script to examine the contents and pattern of the URL path requested and from there decide how to respond to the request. An *index.php* file that uses logic like this to manage the complexity of a many-featured website is known as a *front controller*.

Here are some examples of the types of URLs that web browsers use to make requests to web servers, with explanations of what they mean:

tudublin.ie No path is indicated beyond the domain name, so the web server will execute the default home page script (*index.php* if it's a PHP web server). The TU Dublin home page HTML content is returned to the client.

bbc.com/travel/columns/discovery The path contains text separated by forward slashes, so the home page script executes with logic to search the site's database for today's content relating to the main topic *travel* and the subtopic *discovery*.

nostarch.com/sites/all/themes/nostarch/logo.png The path includes a static resource file, so the web server locates and returns the contents of the *logo.png* image file.

google.com/search?q=cheese+cake The path contains text indicating a search after the forward slash (*search*) and then the search text (*cheese cake*) in a variable (*q*) assigned after a question mark character (*?*). So the Google home page script executes with logic to search for web pages relating to *cheese cake*. In [Chapter 11](#), you'll learn all about passing data through variables in URLs like this one.

In [Chapter 13](#), we'll look at how to write PHP front-controller logic to perform routing decisions such as the ones summarized here.

Templating

Almost all PHP applications are designed to run websites. For most HTTP requests, the content of the response is some sort of text, like an HTML, JavaScript, or CSS file, or perhaps data encoded as JSON or XML. PHP consequently was designed to facilitate outputting text (with, for example, the `print` and `echo` commands).

Further, as was hinted in [Chapter 1](#), the language also makes it easy to mix prewritten text such as HTML with text created on the fly by executing PHP code. This feature is what makes PHP a *templating language*: it can insert dynamically generated values into static templates of HTML or other text. PHP-driven websites benefit from this sort of dynamic output, which may result from database interactions or communication between various data sources like Google Maps, weather APIs, and so on.

In the previous chapters, we've been writing pure PHP programs, which are scripts that contain only PHP code. Once we start using PHP as a templating language, mixing PHP statements with other template text (often HTML) becomes more common. This allows us to write the unchanging HTML for web pages as just HTML; any parts that need to change dynamically can be output from the logic we write in PHP statements. Conveniently, the HTML in many website pages contains much of the same content, such as a header, a navigation bar (which might change only by highlighting the particular page being visited), and page layout HTML code

(for example, a hierarchy of `div`, `header`, and `footer` elements). All this nonchanging, static content is perfect for PHP templating.

It's theoretically possible to make a pure PHP script output HTML by writing lots of `print` statements, but this approach results in code that's long and hard to read. Take a look at [Listing 10-1](#), which outputs HTML by using pure PHP `print` statements.

```
<?php
print '<!doctype html>';
print '<head><title>home</title></head>';
print '<body>';
print '<p>Welcome to My Great Website<br>';
❶ print 'today is ' . date('F d, Y');
print '<p>';
print '</body></html>';
```

Listing 10-1: Outputting HTML through `print` statements

The only real PHP logic we're using here is calling the `date()` function to get the current date as a string in the form *Month day, year* (for example, *January 1, 2025*) ❶. All other lines are `print` statements that output unchanging HTML, and these `print` statements aren't necessary. We can make the code more compact and readable by using PHP only where it's needed, inserting it into an HTML template. That's what we do in [Listing 10-2](#), where the unchanging HTML is written just as it will appear in the final HTML text file to be sent to the client.

```
<!doctype html>
<head><title>home</title></head><body>
<p>Welcome to My Great Website<br>
today is
❶ <?php
    print date('F d, Y');
❷ ?>
</p>
</body></html>
```

Listing 10-2: Mixing HTML template text with a PHP code block

We use opening ❶ and closing ❷ PHP tags to surround just the `print` statement where we call the `date()` function, since this is the only place where PHP code is needed to dynamically generate content. Meanwhile, we've written everything else as regular HTML; no `prints`, quotes, or semicolons are needed. [Figure 10-6](#) shows how PHP sees and processes the script content.

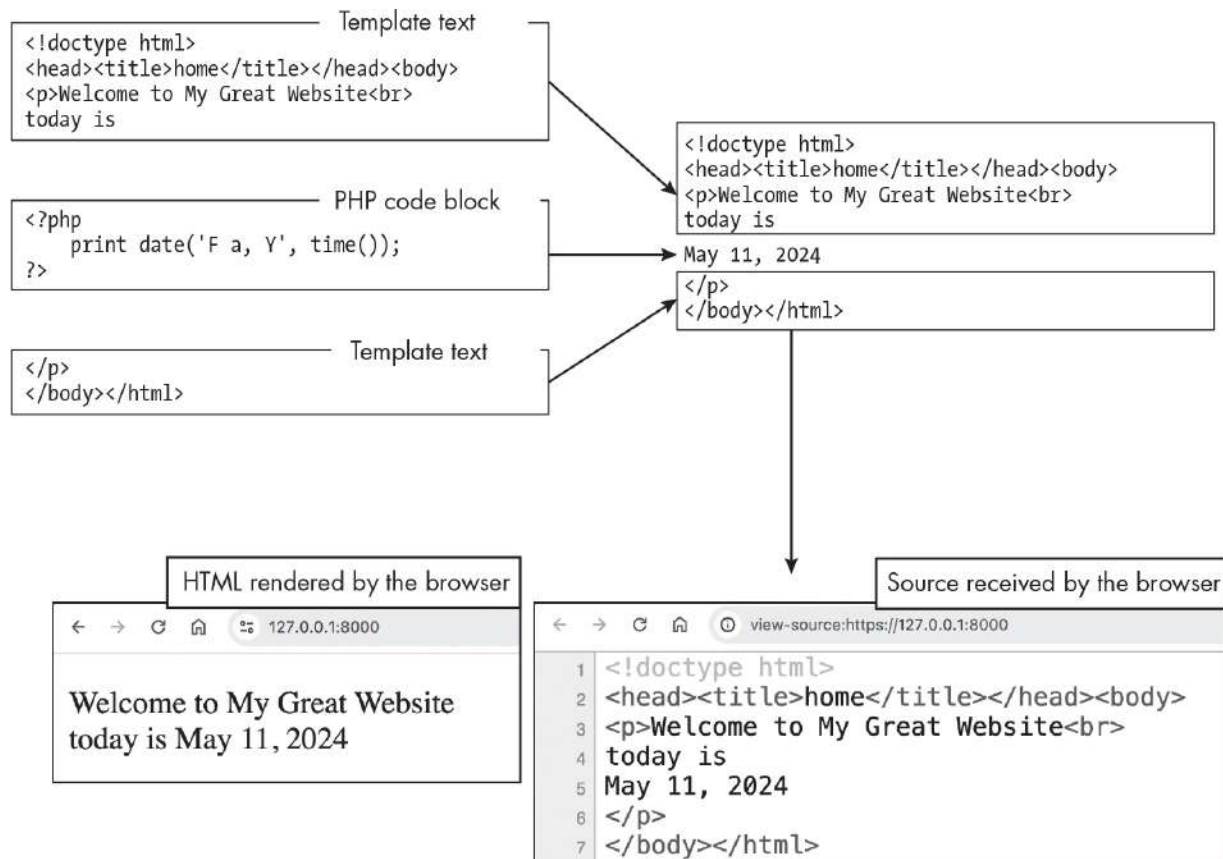


Figure 10-6: How PHP processes mixed template text and dynamic code

First, a block of template text needs to be copied verbatim into the text output. Next, a block of PHP code (between `<?php` and `?>`) needs to be interpreted and executed before the result is added to the script's text output. Finally, another block of template text needs to be copied verbatim to the output text. The temporary store for the text being output by the multiple parts of a PHP script is referred to as the *output buffer*.

Imagine for a moment that the script in [Listing 10-2](#) is part of an HTTP request from a web browser. When all PHP execution in the script is completed, the text in the output buffer will be wrapped up into an HTTP response by adding headers and then sent back to the browser web client. The browser will then render (draw) the web page for the user to see, interpreting the HTML it received in the body text of the HTTP response, resulting in the simple page shown at the bottom of [Figure 10-6](#).

PHP Tags

As you've just seen, when you embed PHP code in template text, it's important to use both the opening `<?php` tag and the closing `?>` tag to delimit the code. By contrast, when writing PHP scripts that contain *just* code without any template text, the script should start with an opening `<?php` tag, but you shouldn't include the closing `?>` tag at the end of the file.

You leave off the closing tag for two reasons. First, you don't need it, since the code has no template text to be separated from the PHP statements. Second, if you did include the closing PHP tag, any (unintentional and invisible) whitespace that occurs after the closing tag, including spaces, tabs, or newline characters, will be interpreted as template text and could prematurely begin creating the output buffer.

Short Echo Tags

We've so far focused on PHP's main `<?php` tag, but the language also provides a *short echo tag*, denoted with the `<?=` symbol, that further simplifies templating. This tag allows you to avoid writing lengthy commands when all you want to do is output the result of an expression as text. This might be to display the contents of a variable, or the result of a complex calculation or series of string concatenations. For example, instead of writing something like `<?php print $someVariable; ?>` to output the value of `$someVariable`, you can simply write `<?= $someVariable ?>` with the short echo tag.

The short echo tag calls for less typing since it omits `print` (or `echo`) and doesn't require an ending semicolon. Also, any experienced PHP programmer who encounters the short echo tag can immediately recognize that the only logic is to output a string. Overall, the key advantage of the short echo tag is that it doesn't distract the reader (or writer) with

extraneous PHP code-block syntax when a script mostly contains HTML template text. The dynamically generated PHP code values blend in better with the surrounding HTML, as [Listing 10-3](#) illustrates.

```
<?php
❶ $dateString = date('F d, Y', time());
?>
<!doctype html><head><title>home</title></head><body>
<p>Welcome to My Great Website<br>
❷ Today is <?= $dateString ?>
</p>
</body></html>
```

Listing 10-3: Simplifying code with the PHP short echo tag

In a full PHP code block surrounded by ordinary PHP tags, we create a `$dateString` variable containing our formatted date string ❶. This frees us up to simply write `<?= $dateString ?>`, using the short echo tag at the spot in the template where we want the string to be output ❷. There’s no need for a `print` statement or semicolon.

The Model-View-Controller Architecture

Almost all large-scale web applications delegate different responsibilities to different system components. Most do this by implementing some form of the *model-view-controller (MVC)* architecture. This is a software design pattern that distinguishes between the data underlying the software (the *model*), the way that data is displayed to the user (the *view*), and the decisions about what data to display when (the *controller*).

We’ve already touched on aspects of the MVC architecture in this chapter. We’ve noted how PHP applications can make routing decisions based on incoming HTTP requests (a controller task) and how we can use PHP for templating by injecting dynamically generated values into static HTML text (a view task). Now let’s fill in a few more gaps to see how the MVC pattern fits into the request-response cycle. [Figure 10-7](#) illustrates a typical interpretation of the MVC architecture for a web application.

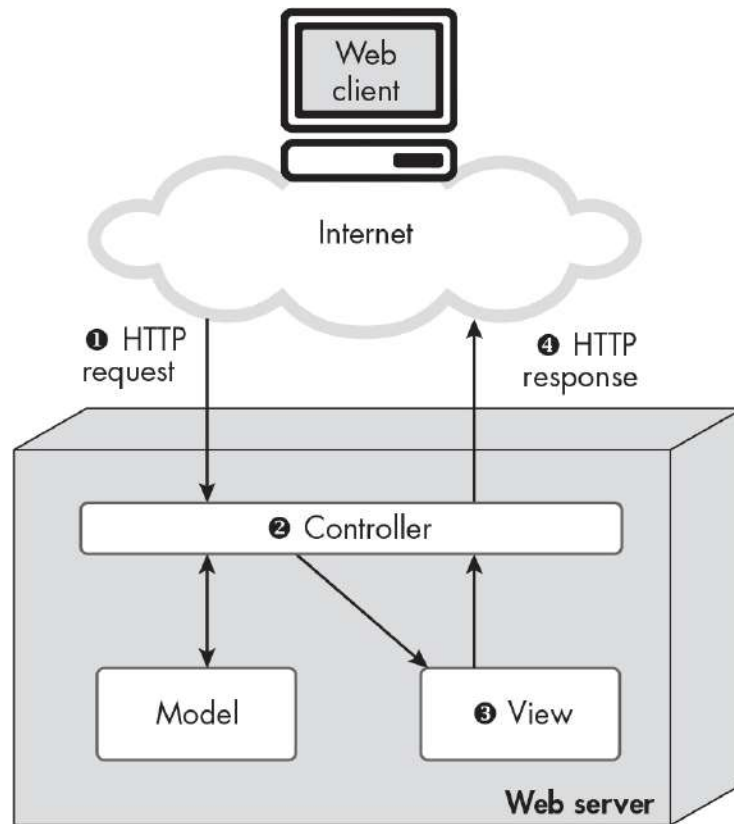


Figure 10-7: The MVC architecture common for web applications

First, the web client sends an HTTP request ①. Then the controller (the main application logic) interprets the request and decides what to do ②. This may involve checking any stored security credentials and other data (such as shopping-cart contents) and deciding the appropriate actions to take in response to the received request. Often the controller needs to read the contents of a data store, such as a database system, file store, or even an API running on another server. This data is the model component of the MVC pattern. If the received request contains data submitted from a form, the controller may need to update or delete some of the model data.

Then the controller invokes the view component ③, such as a template file, to create the contents of the response to be returned to the user. If appropriate, the controller passes along data collected from the model when it invokes the view component. Finally, the controller sends the response that it has created back to the web client (adding any appropriate headers, response codes, and the like) ④.

We'll revisit the MVC pattern throughout this book as we delve further into structuring PHP web applications. As mentioned, in [Chapter 13](#), we'll look at how to create a front-controller script to manage the controller portion of the architecture. In [Chapter 21](#), we'll introduce the Twig library, which simplifies templating for the view portion of the architecture. Finally, in [Part VI](#), starting with [Chapter 27](#), we'll investigate how to integrate a PHP application with a database to handle the model portion of the architecture.

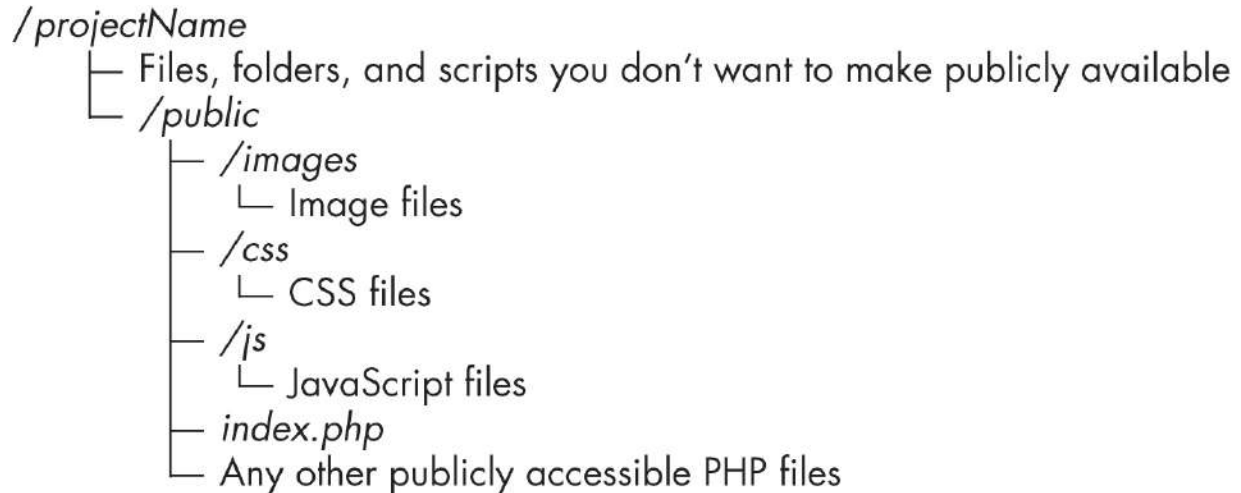
Structuring a PHP Web Development Project

As we discussed in [Chapter 1](#), the PHP engine comes with a built-in web server for testing purposes that you can run at the command line by using the `php -S localhost:8000` command. By default, this command makes every file and folder that lives in the directory that the command line is currently pointing to publicly available through the web server. For example, if your command line was navigated to the root of your main hard disk (such as `C:\` on a Windows computer) and you executed the PHP web server command, you'd be making everything on the hard disk available to be requested! This probably isn't a good idea from a security point of view.

Even within a specific PHP project folder, you may have files or other content that you wouldn't want to publish publicly, such as code containing username and password credentials for data access or scripts that should be accessed only by authorized users. Therefore, it's customary (and highly recommended) to create a *public* folder within the overall folder for any PHP web development project. This *public* folder (and its subfolders, if any) should contain only those files that are to be made publicly accessible via the web server, including any images, sound files, video files, CSS stylesheets, JavaScript text files, and the like that are needed for the website. Any PHP scripts that are to be executed in direct response to incoming HTTP requests from web clients should also be located in the *public* folder, while other content that shouldn't be publicly accessible should be located elsewhere in the project's directory structure.

The usual way to organize a secure web application is to have just one PHP script named *index.php* in the project's *public* folder. This script (the front controller we'll discuss further in [Chapter 13](#)) then decides which

other nonpublic scripts should be executed based on the properties of the incoming HTTP requests and other stored data. A typical PHP project folder therefore looks as follows:



Generally, it's best to do any command line work from the root folder of a project rather than from the *public* folder. Since this practice is so common, the built-in PHP web server offers the `-t` command line option for specifying a subfolder from which to serve web pages. With your command line interface navigated to the root project directory, you can therefore enter the following command to serve only files in the *public* folder via port 8000:

```
php -S localhost:8000 -t public
```

Let's test these two ways to run the built-in PHP web server: with and without the `-t` option. First, create a new empty folder named [chapter 10](#), and in this folder create an *index.php* file containing the code shown in [Listing 10-4](#).

```
<?php
❶ $total = 2 + 2;
?>
<!doctype html><html><head><title>Home page</title></head>
<body>
```

```
❷ <?= "total = $total" ?>
</body></html>
```

Listing 10-4: A simple index.php file

This script contains only two PHP statements: within full PHP tags, we set the `$total` variable to the result of evaluating the mathematical expression `2 + 2` ❶, and with the short echo tag we output the contents of this variable ❷. To make sure this script works, navigate your command line interface to the *chapter10* folder (use the `cd` command to change directories if you aren't there already), and then run the built-in PHP web server at port 8000 without specifying a folder to serve:

```
% php -S localhost:8000
```

Open a web browser to *localhost:8000* and you should see a web page showing the result of the PHP output statement: `total = 4`.

Now let's see why publishing the entire contents of a project folder is a bad idea. In your *chapter10* folder, also create a text file called *password.txt* containing the text `password=mysecret`. Then visit *localhost:8000/password.txt* in your web browser to see that this text file is also publicly accessible from the web server, just like the *index.php* script (see [Figure 10-8](#)).

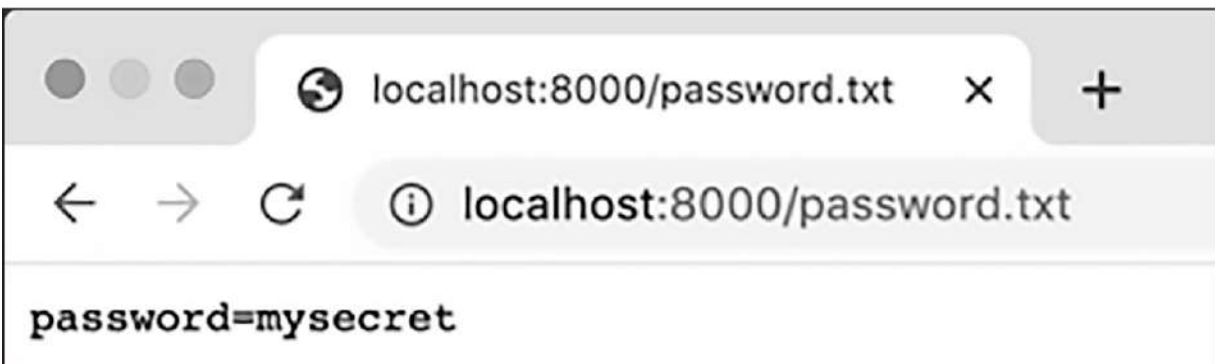


Figure 10-8: The web server publishing a secret password

Let's make this file more secure by creating a subfolder named *public* and moving the *index.php* script into this subfolder, while keeping *password.txt* in the main *chapter10* folder. Once you've made this change, press CTRL-C to kill the old web server process and then run the web server again, this time restricting it to only the content of the *public* subfolder:

```
% php -S localhost:8000 -t public
```

Try visiting both *localhost:8000* and *localhost:8000/password.txt* in your browser again. You should still be able to see the index page since it's located in the *public* folder, but you should get a 404 error when you try to access the *password.txt* file since it isn't in the *public* folder. We'll follow this structure of using a *public* folder to isolate just the resources that should be publicly accessible throughout the book.

INSTALLING A DIFFERENT WEB SERVER

In most cases, the built-in PHP web server is fine for development purposes. (Personally, I don't have any other server on my PHP development laptop.) That said, for a given project or client, you might know that the final published website will run on a specific web server such as Apache HTTP Server or nginx. If this is the case, installing the same server application on your local development computer makes sense so you can identify server-specific issues as early as possible.

Perhaps the most common way to install a web server on a local computer is to install an "all-in-one" *AMP stack*. This acronym refers to *Apache-MySQL-PHP*, although usually other applications and languages are also available, such as Python and the nginx web server. A good option, available in both free and paid versions, is MAMP, which was originally for macOS but is now also available for Windows. Once installed, MAMP offers a choice of either Apache or nginx.

See [Appendix A](#) for other suggestions on AMP stack installations.

Summary

In this chapter, we explored concepts fundamental to PHP web development. We considered the HTTP messages that form the basis for web client/server communications and began to discuss the concept of routing, which is how web servers evaluate the content of the HTTP request's path and decide what files to return or which server scripts to

execute. We also looked at PHP as a templating language that enables us to mix dynamic PHP statements with unchanging template text. We saw a tidy way to mix PHP output with HTML template text using the short echo tag.

We had our first look at the MVC architecture, a powerful way of dividing and organizing the tasks and data that drive a web application. Finally, we looked at the typical structure of a PHP web development project. In particular, we discussed the need for a *public* subfolder containing any resources that should be publicly accessible; any files or scripts that shouldn't be publicly accessible must be located outside this subfolder.

Exercises

1. Open the developer tools for your web browser and visit a favorite website. Examine the headers of your HTTP `GET` request and the body of the HTTP response message that's returned to the browser.
2. With the developer tools for your web browser open, visit a web page that offers a form. Complete the form and, when you submit it, view the HTTP request body. You should see the name or value variables that were sent to the web server via the `POST` HTTP method.
3. Write a "pure" PHP script, all in a single PHP block of code, to do the following:
 - a. Define a PHP `$pageTitle` variable containing the string `'Home Page'`.
 - b. Output `<!doctype html><html><head><title>`.
 - c. Output the value inside the `$pageTitle` variable.
 - d. Output `</title></head>`.
4. Rewrite your answer for Exercise 3, using template text instead of PHP code where possible. Use complete code blocks with `<?php` and `?>` tags for the PHP code.
5. Rewrite your answer for Exercise 4 to use the short echo tag to output the value inside the `$pageTitle` variable.

11

CREATING AND PROCESSING WEB FORMS



After simple, clickable links, web forms are perhaps the most common way people interact with websites. In this chapter, we'll look at how web clients can submit form data to server scripts, and we'll create a range of web forms that send data. We'll also practice writing server-side PHP scripts to extract and process the incoming form data. You'll learn to handle data from a range of web-form elements, sent with both `GET` and `POST` HTTP requests.

A *web form* is simply a portion of a web page that allows the user to enter data and then communicates that user input to server applications. Examples of web form interactions include creating a Facebook post, booking flights or entertainment tickets, and entering login information. As you'll see, each form on a web page is defined between starting and ending HTML `<form>` tags. The form data might be text input by the user, or it might come from mechanisms like radio buttons, selections lists, or

checkboxes. We'll discuss how to work with all these types of input in this chapter.

Basic Client/Server Communication for Web Forms

Behind the typical web form lies a sequence of four messages between a web client (such as the user's browser) and a web server, whereby the form is requested, received, submitted, and processed. [Figure 11-1](#) summarizes these messages.

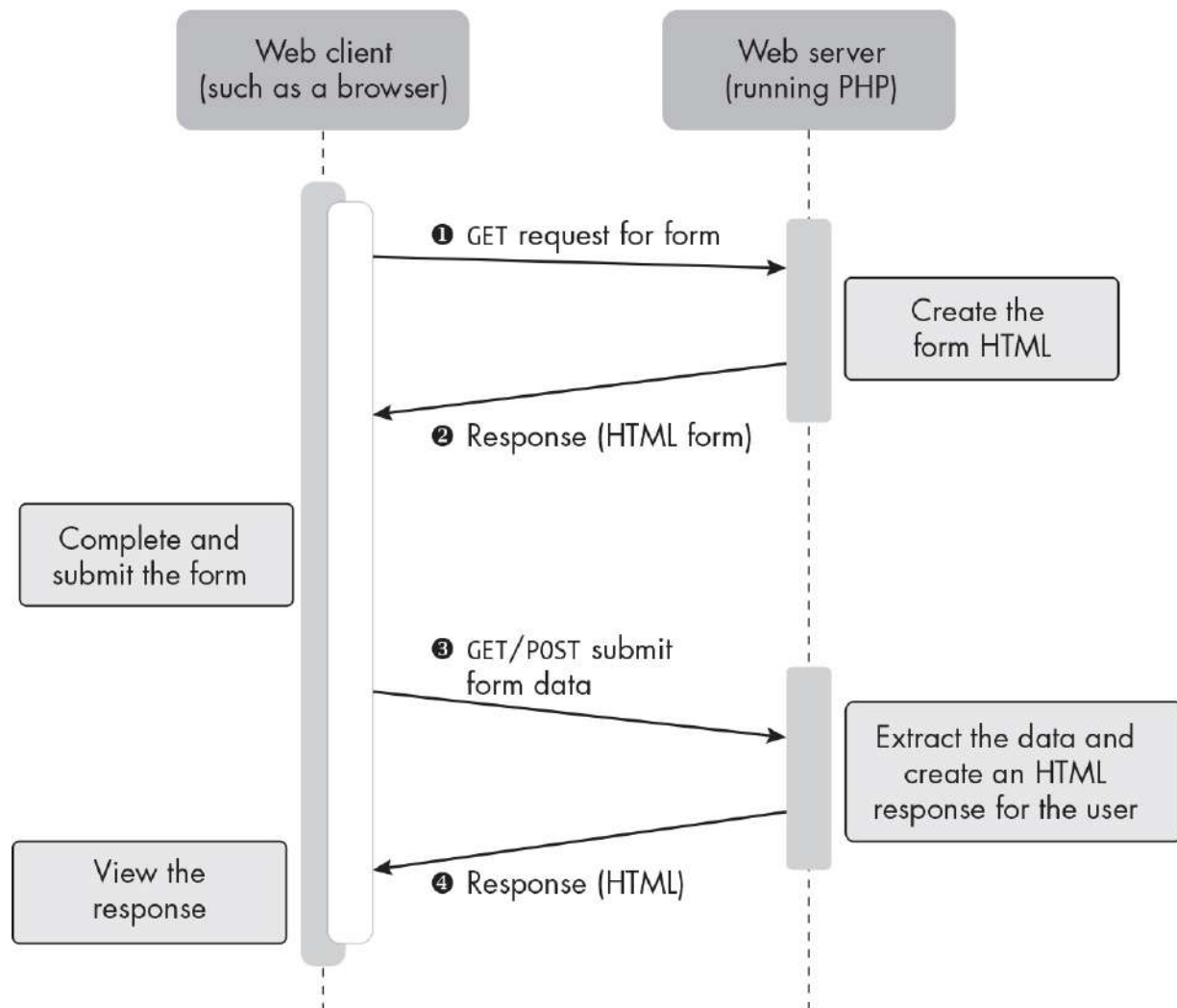


Figure 11-1: The typical exchange of messages for displaying and processing a web form

First, the web browser client requests the HTML of the form from the server ❶. The user triggers this request through an action such as clicking a link or a button. Next, the server retrieves and, in some cases, customizes the HTML for the form and sends it back to the client ❷. Once the user has entered data and submitted the form, the form data is sent back to the server ❸. Finally, the server processes the received data, constructs an appropriate message for the user, and sends that message back ❹. This final message may be a simple confirmation of received data or an error message if an issue occurs, or it could be the original form with messages highlighting missing required data.

GET vs. POST Requests

As mentioned in [Chapter 10](#), the two most common types of HTTP request sent from a client to a server are `GET` and `POST`. When you create an HTML form, you can send the data to the server with either type of request, since both `GET` and `POST` can send variables from the browser client to the web server as part of the request. In almost all cases, the data variables sent from web forms to the server are simple name/value pairs, such as

`username=matt` or `q=chocolate`.

The request type you use comes down to the purposes of the form and how you want the form data to be sent. As you'll see, the `GET` method makes the submission variables visible in the URL, while the `POST` method can hide the variables in the body of the HTTP request.

Sending Data Visibly with GET

An HTTP `GET` request is primarily for retrieving data or a web page from a server. While you can send data along with the request to help with this retrieval, a `GET` request should never result in changes to content stored on the server (such as modifications to values in a database).

With a `GET` request, any variables the server needs to complete the request, including values submitted through web forms, are added to the end of the URL of the request, after a question mark character (`?`). This part of the URL after the question mark is known as a *query string*, and it will be visible in your browser's address bar. The variables are encoded as name/value pairs in the form `name=value`, such as `username=matt`. For

example, when you perform a search using Google or Bing, the terms you enter into the search engine's web form are assigned to the variable `q`, added to a URL query string, and sent using a `GET` request.

Say you use Google to search the phrase *cheese cake*. When you view the search results, you should see something like <https://www.google.com/search?q=cheese+cake> in the address bar. The single letter `q` represents your search query and is paired with the value you entered into Google's web form. This indicates your query was passed to Google's servers through a `GET` request.

Special rules define the characters allowed in a URL, and the variables sent via the HTTP `GET` method must follow these rules too. As a result, special characters and spaces can't be represented verbatim in a query string but must instead be encoded as other symbols. For example, each space is replaced with either `%20` or a plus sign (`+`), which is why the Google search query string reads `q=cheese+cake` rather than `q=cheese cake`. When two or more variables are being encoded (for example, from separate fields in a form), the name/value pairs are separated by ampersand (`&`) characters, as in `?firstname=matt&lastname=smith`. The web browser will look after this sort of encoding automatically, but knowing about it is handy since it explains why you'll often see cryptic, percent-encoded characters when sending form data with the `GET` method.

One common use of the `GET` method is to create a URL that's easily bookmarked, perhaps to share with someone else via email or text message. The Google *cheese cake* query is one example:

<https://www.google.com/search?q=cheese+cake>. Another example could be a Google Maps search, such as this one for Dublin, Ireland:

<https://www.google.com/maps?q=dublin+ireland>. The variables in the URL can come from values entered into a web form, as is the case with a Google search, or they can be hardcoded by explicitly adding the question mark and the desired name/value pairs to the end of a URL.

[Figure 11-2](#) shows an example of the latter, where clicking the COMP H2029 -FT link initiates a `GET` request that includes the name/value pair `id=1499` in the query string. This `id` value doesn't come from user input but rather was hardcoded into the logic of the website.

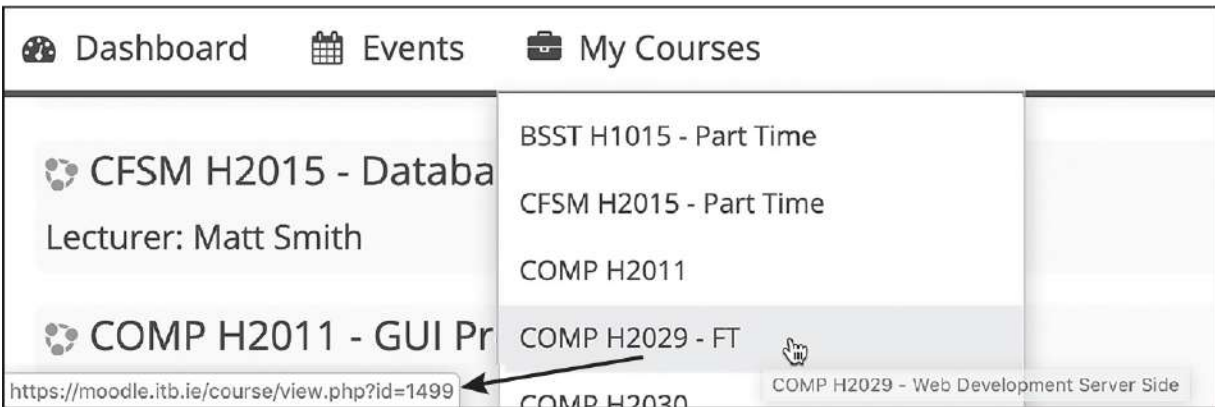


Figure 11-2: A link with a hardcoded value to be sent via the `GET` request method

A web server doesn't care how a `GET` request is created. Whether the query-string variables come from form submissions or were hardcoded, the name/value pairs can be extracted by a server-side script for processing.

Sending Data Invisibly with POST

HTTP `POST` requests send data primarily for the purposes of creating or modifying a resource on a server. With the `POST` method, you can send variables in the body of the HTTP message, meaning they won't be seen in the resulting URL. For any confidential data like usernames and passwords, you should use `POST` requests so that people looking at the screen can't see the data values being sent to the server. In fact, most web forms send their data by using the `POST` method. [Figure 11-3](#) illustrates a `POST` method login form.

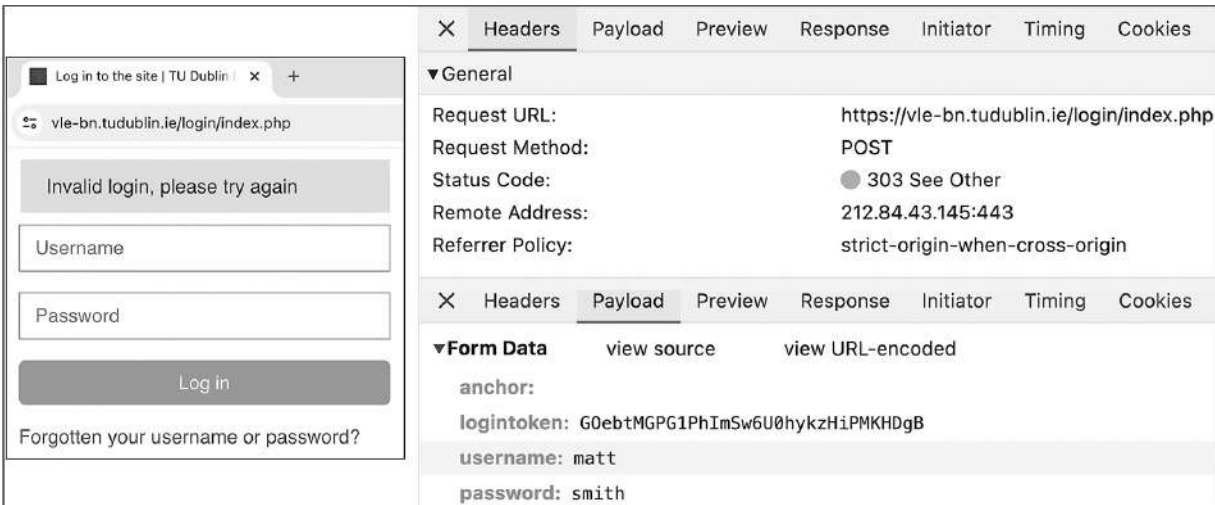


Figure 11-3: *POST* method variables in the request body

In this example, I’ve tried to log in to a site with a username of `matt` and a password of `smith`. The browser’s HTTP message inspection tool reveals that the username and password values were sent in the body of the HTTP *POST* request. These values therefore don’t appear as part of the URL in the address bar.

NOTE

*A *POST* request can send data directly in the query string, like a *GET* request, as well as in the request body. We’ll explore *POST* requests that do both in “Sending Noneditable Data Along with Form Variables” on [page 206](#).*

A Simple Example

To more clearly understand how the *GET* and *POST* methods send data differently, let’s build a simple site with a web form consisting of a single text box where the user can enter their name (see [Figure 11-4](#)). We’ll try passing data from the form by using both HTTP methods. As you’ll see, we can choose which method to use whenever we create an HTML `<form>` element.

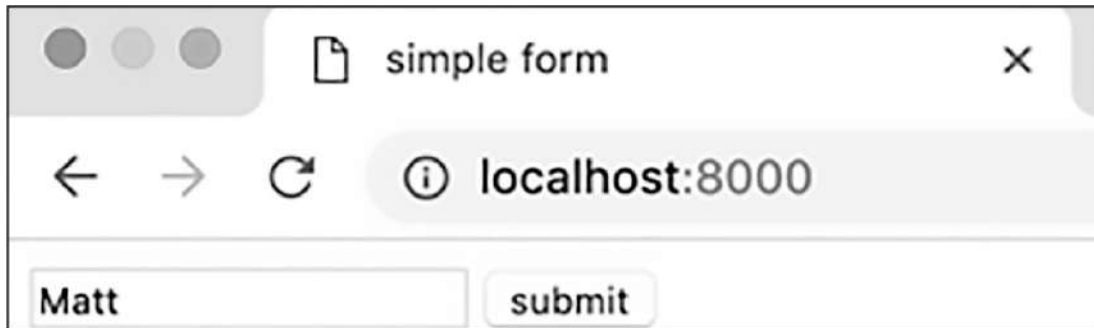


Figure 11-4: A simple web form displayed in the browser

Our project will consist of a *public* folder containing two PHP script files, *index.php* and *process.php*. Here, *index.php* is the default home-page script that will display the form, and *process.php* will receive the name submitted by the user from the index page and generate a *Hello <name>* message in response.

Creating a Form with the GET Method

We'll start with the `GET` version of our simple web form. Create a new project with a *public* folder, and inside that folder create a new PHP script file named *index.php*. Enter the code shown in [Listing 11-1](#).

```
<!doctype html><html><head><title>simple form</title></head>
<body>
  <form method="GET" action="process.php">
    <input name="firstName">
    <input type="submit">
  </form>
</body>
</html>
```

Listing 11-1: The HTML code for a simple web form using the `GET` method

The file consists entirely of HTML template text, including a `<form>` element that defines the web form. We use the element's `method` attribute to declare that the form data should be submitted with the `GET` HTTP method, and the `action` attribute to specify that the HTTP request and its data should be sent to the *process.php* server script. Note that the `GET` or `POST`

values of the `method` HTML form attribute are case insensitive, so we could also write `method="get"`.

Within the form, we create an `<input>` element and give it a `name` of `firstName`. We also create a second `<input>` element with a `type` of `submit` to add a Submit button to the form. Since we don't specify the `type` of the `firstName` input, HTML 5 automatically defines the default form input type to be a text box. Text boxes are displayed to the user as rectangular input boxes. If you wanted to explicitly declare the type of input, you could do so via `<input type="text" name="firstName">`. You could go further and set the character width of the text box and other specs by using the various optional attributes for each type of HTML form input.

Since our form inputs only a single value, we don't need to bother displaying a text label to the user. However, when several input controls are present, you should precede each with a prompt so that the user knows which text box (or radio button or other input type) relates to which value. For example, if we wanted the user to input an age, we might write the template text `Age:` and then the form input, like so:

```
Age: <input name="age">
```

Modern HTML good practice would also require us to add an `id` attribute to the age input with `<input name="age" id="age">` and a `<label>` element around the template text with `<label for="age">Age:</label>`. This allows the user to click either the label or the text box to make `age` the active form input.

Processing the GET Request

We've created an HTML web form, but our work is only half done; we also need to write the *process.php* script to handle the data submitted through the form. All we need to know when writing processing scripts for simple forms is the name of the variable the script is to receive and whether that variable was submitted through the query string (as with the `GET` method) or in the request body (as with the `POST` method).

In this case, the script should attempt to find a value for the `firstName` variable in the query string received from the `GET` request and then output

HTML to present to the user a greeting featuring that name. Add *process.php* to the *public* folder of your project and enter the code in [Listing 11-2](#).

```
<?php
//----- (1) LOGIC -----
❶ $firstName = filter_input(INPUT_GET, 'firstName');
?>

<!-- (2) HTML template output -->
<!doctype html> <html><head><title>process</title></head><bo
dy>
❷ Hello <?= $firstName ?>
</body></html>
```

Listing 11-2: A process.php server script to respond to the web form

Notice that we use two kinds of comments in this script, since it mixes two languages: a PHP comment starting with `//` and an HTML `<!--` comment. Inside the initial PHP code block, we call the `filter_input()` function to read data from the incoming HTTP request, storing the result in the `$firstName` variable ❶. The `INPUT_GET` argument specifies that we want to read data embedded directly in the URL query string, and the `'firstName'` argument identifies the specific HTML form variable we're looking for. Form input variable names are case sensitive, so it's important to carefully match the variable names defined in HTML forms when calling the `filter_input()` function. If we passed `'firstname'` rather than `'firstName'` as a function argument, for example, the script wouldn't work. When passing the value of a form variable along to a PHP variable, as we're doing here, it's generally good practice to give the PHP variable the same name as the corresponding form variable.

Next, we declare the HTML that should be sent in response to the form submission. This includes the template text `Hello` followed by the value of the `$firstName` variable inside PHP short echo tags ❷.

NOTE

The `INPUT_GET` argument to `filter_input()` is somewhat misleadingly named. Its purpose is to retrieve data from the URL query string, regardless of whether that data was sent via the `GET` method (where all variables are part of the query string) or via the `POST` method (where data can be either in the query string or in the request body). Therefore, when you see `GET` while working on form-processing code in PHP, interpret this as query-string variables and don't necessarily assume they came from a `GET` request.

Testing the Form

Now that we've created the web form in `index.php` and written the `process.php` script to respond to it, let's test our work. Launch the PHP web server at the command line by using the `php -S localhost:8000 -t public` command, as discussed in [Chapter 10](#); then open a browser tab to `localhost:8000`. The form we've created should be displayed by default since the file is named `index.php`. You'll see something like the form in [Figure 11-4](#): a text box with a Submit button.

Enter your name into the form, then click **Submit**. When you do, the text entered should be sent as part of an HTTP `GET` request from the browser to the PHP server. The `GET` request triggers the server to execute the `process.php` script, as declared in the `action` attribute of the HTML `<form>` element in `index.php`. This script extracts the submitted value and injects it into its HTML template text, which is then added to the text buffer that becomes the body of the HTTP response message the server sends back to the requesting client (the web browser). You should see something like [Figure 11-5](#) as a result.

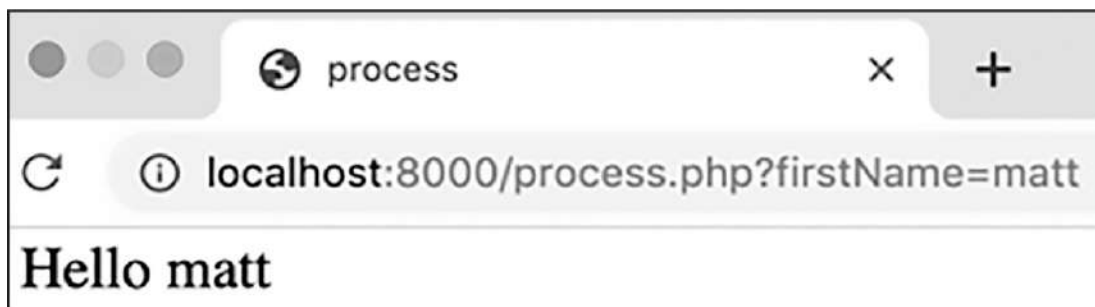


Figure 11-5: The result of the `process.php` script, confirming the data received through the form

You see your name not only in the greeting displayed on the page (such as *Hello matt* in the figure) but also at the end of the URL in the browser's address bar, because the form is submitted via a `GET` request. For example, the URL I get when I submit `matt` through the form is *localhost:8000/process.php?firstName=matt*. This indicates the `GET` request is trying to access the *process.php* script and pass it a `firstName` variable with a value of `matt`. Notice the question mark separating the query string from the rest of the URL.

Switching to the POST Method

Let's modify our project to send the form data with the `POST` method rather than `GET` and see what difference it makes. Only a few minor changes are needed. First, update the *index.php* script as shown in [Listing 11-3](#).

```
<!doctype html><html><head><title>simple form</title></head>
<body>
<form method="POST" action="process.php">
    <input name="firstName">
    <input type="submit">
</form>
</body>
</html>
```

*Listing 11-3: Switching from `GET` to `POST` in *index.php**

We now declare the `method` of the `<form>` element to be `POST`. That's it: no further changes are required to the *index* script to ensure that the web browser uses the `POST` method instead of `GET` to submit the form data. Next, update *process.php* as shown in [Listing 11-4](#).

```
<?php
//----- (1) LOGIC -----
$firstName = filter_input(INPUT_POST, 'firstName');
?>

<!-- (2) HTML template output -->
<!doctype html> <html><head><title>process</title></head><bo
dy>
```

```
Hello <?= $firstName ?>
</body></html>
```

Listing 11-4: Switching from `GET` to `POST` in `process.php`

This is another simple change: all we have to do is pass `INPUT_POST` rather than `INPUT_GET` as an argument to `filter_input()` to tell the function to look for a variable submitted in the body of the request with `POST`.

Try running the web server and submitting your name through the form again. You should still see the same *Hello <name>* greeting as before. If you look a little closer, however, you'll see some key differences, as shown in [Figure 11-6](#).

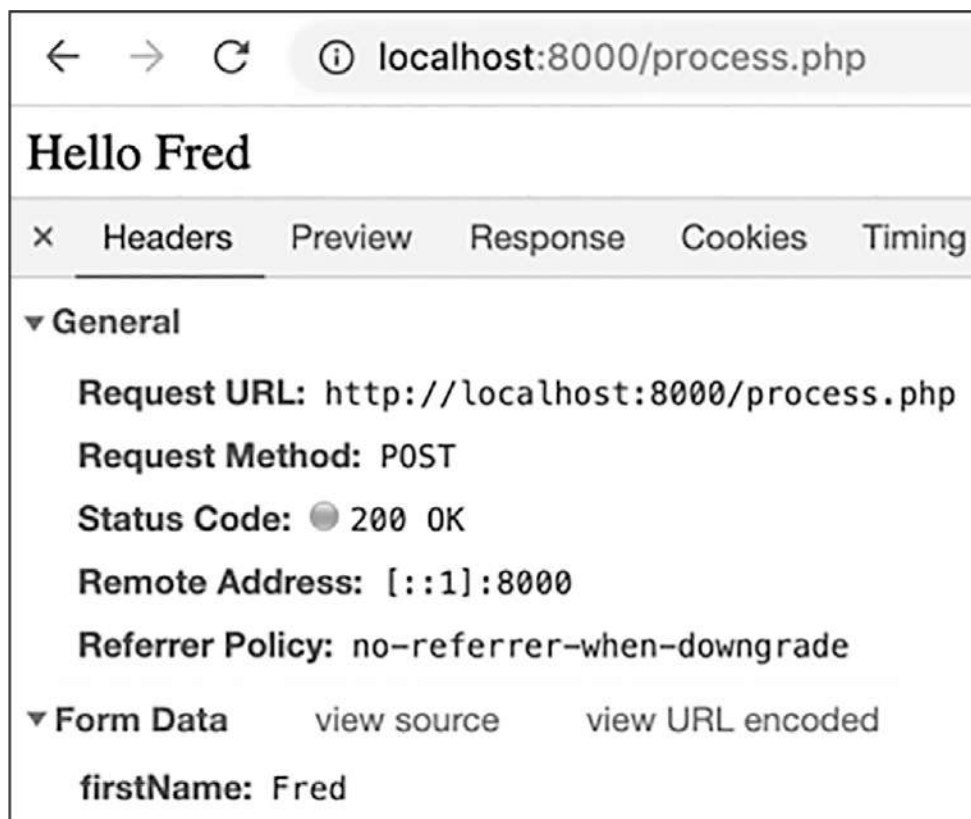


Figure 11-6: Viewing the HTTP `POST` request and the `firstName` variable with the browser developer tools

First, the URL in the browser's address bar should read just *localhost:8000/process.php*. Because the `firstName` form variable is now being sent in the request body with `POST`, it no longer appears in the URL query string for all to see. You can verify that the variable is still being transmitted by viewing the request with the browser's developer tools. In this example, I submitted the name `Fred` through the form, and you can see in [Figure 11-6](#) that, indeed, the form data variable `firstName=Fred` is shown in the body of the `POST` request.

The `filter_input()` Function

Our simple web form project illustrated how to receive incoming form data by using PHP's `filter_input()` function. This function makes it easy to extract values submitted with the `GET` and `POST` methods. Writing form-processing scripts wasn't always so simple, however; in older versions of PHP, the usual way to extract data received from the user was to access one or both of the built-in `$_GET` and `$_POST` *superglobal arrays*.

The `$_GET` array contains key/value pairs representing all the variables received as part of the URL query string, while the `$_POST` array contains key/value pairs representing all the variables received via the `POST` HTTP method. For instance, submitting the first name `Matt` through the `GET` request version of our simple web form would produce a `$_GET` array containing `['firstName' => 'Matt']` and an empty `$_POST` array.

NOTE

The `$_GET` and `$_POST` arrays are examples of PHP's superglobals. These are arrays that always exist and that can be accessed from anywhere in your PHP code (that is, from any scope), including inside functions and class methods.

Even in modern PHP programming, extracting form data from these two superglobal arrays is still theoretically possible. But PHP version 5.2 introduced the `filter_input()` function as a much better way of accessing submitted data. To illustrate the improvement, let's look at what it takes to work with these superglobal arrays.

In [Chapter 7](#), you learned that trying to access a nonexistent key in an array will trigger a warning, which you can avoid by using the `isset()` function to verify that an array key exists before accessing its value. This kind of test makes scripts more robust and error-proof, and it's especially important when working directly with the `$_GET` and `$_POST` arrays. Unfortunately, such a test also adds extra code to a script. For example, [Listing 11-5](#) illustrates how to safely retrieve a `$firstName` variable from the `$_GET` array.

```
<?php
if (isset($_GET['firstName'])) {
    $firstName = $_GET['firstName'];
    // Now use filters / apply sanitization/validation to ex
    tracted value
} else {
    $firstName = NULL;
}
```

Listing 11-5: Testing an array key with `isset()` before attempting to extract a value from `$_GET`

We use `isset()` in an `if...else` statement to check whether the `'firstName'` key exists in the `$_GET` array (indicating that a `firstName` form variable was submitted through the incoming query string). If the key exists, we pass on its value to the `$firstName` variable. Otherwise, we set `$firstName` to `NULL`. This `if...else` statement saves us from getting a warning if we naively access a nonexistent value in the array.

Listing 11-5 will work fine, but it represents such a common series of actions in PHP form-processing code that `filter_input()` was introduced to encapsulate it. Our entire `if...else` statement can therefore be replaced with a single statement:

```
$firstName = filter_input(INPUT_GET, 'firstName')
```

The `filter_input()` function automatically checks whether the desired variable exists before trying to access it and typically returns `NULL` if it

doesn't. This spares us from writing clunky conditional tests like the one in [Listing 11-5](#).

An additional advantage of the `filter_input()` function is that it can use filters to ignore and remove unwanted and potentially dangerous content from the received form data. This helps prevent security vulnerabilities such as cross-site scripting attacks. For example, to filter out (discard) any nonalphabetic characters from the user input, we could add a third argument of `FILTER_SANITIZE_SPECIAL_CHARS` to the `filter_input()` call:

```
$firstName = filter_input(INPUT_GET, 'firstName',  
FILTER_SANITIZE_SPECIAL_CHARS);
```

DATA FILTERING IN PHP

PHP has two main types of filtering: sanitization and validation. *Sanitization* removes characters that aren't permitted and that may potentially be a security threat, such as the less-than (<) and greater-than (>) symbols for tags around JavaScript code. You can also sanitize characters when you display them back to the user; for example, you might display the Unicode symbol for a less-than sign when it's something the user might reasonably have entered in a blog about programming, in order to prevent user-entered special characters from becoming executable code in the output HTML document. By contrast, *validation* tests that a received value is acceptable.

To illustrate the difference between the two processes, say someone submits the character sequence `3.14.15` through a form. This sequence is sanitized for floats, since it contains only digits and decimal-point characters. It isn't a valid floating-point number, however, since it contains two decimal points, whereas valid numbers can contain only one.

Sanitization filters take in a value and a filter type and return a value with any illegal characters removed (and in some cases, replaced). If all characters are illegal, or `NULL` is given, then an empty string or `0` is returned (a value is always returned when using the `filter_input()` function). Validation filters generally take a value and a filter type and return `true` or `false`, or sometimes `NULL`, depending on whether the value is valid for the given filter.

The sanitization filter `FILTER_SANITIZE_SPECIAL_CHARS` replaces HTML special characters (such as < and &) with HTML entities (such as `<` and `&`), so a received string and later output to a web page won't mess up the HTML grammar with extra special characters.

Learn more about the data filters from the PHP documentation at <https://www.php.net/manual/en/intro.filter.php>.

Other Ways to Send Data

Taking user input through a web form isn't the only way to send data via an HTTP request. In this section, we'll consider other techniques for transmitting data to a server. We'll look at how to embed noneditable data into a query string for submission along with user-entered form data, how to send data about the form's Submit button itself, and how to add query-string variables to a regular hyperlink, separate from any web form. Along the way, you'll also see how to process a mixture of query string and `POST` variables, and how to harness PHP arrays and loops to generate query-string variables programmatically.

Sending Noneditable Data Along with Form Variables

Often you'll want a web form to send extra data that the user can't edit. Perhaps the most common example occurs when the user, maybe an employee, is editing details for an item in a database. The item, which might be a record about a product or a customer, already has an assigned ID that should be included with the form data, but the ID itself should never be changed via the form. For these cases, you can send the noneditable values as query-string variables at the end of the URL in the form's `action` attribute (for example, `action="/process.php?id=1022"`).

To illustrate, let's create a new web form for submitting information about movies. Start a new project containing a *public* folder and create an *index.php* script within it. Then enter the HTML code in [Listing 11-6](#) to create the web form.

```
<!doctype html><html><head><title>Movie Form 1</title></head>
<body>
<h1>Edit movie</h1>
❶ <form method="POST" action="/process.php?id=1022">
    ❷ <label for="title">Title: </label><input name="title" id
    ="title">
        <br>
```



```
    ❸ <label for="price">Price: </label><input name="price" id
    ="price">
        <br>
        <input type="submit">
</form>
</body>
</html>
```

Listing 11-6: The HTML code for a movie form in index.php

We declare a form using the `POST` method ❶. (In a more realistic scenario, this form would likely result in changes to a record in a database, so `POST` rather than `GET` is the appropriate method here.) For the form's `action` attribute, we specify that the script to process the form is *process.php*, and we also send a URL query-string variable named `id` with a hardcoded value of `1022`. When the form is submitted, this extra name/value pair will be visible in the resulting URL itself (much like data sent with the `GET` method). Meanwhile, the form will also send two variables with user-entered values in the body of the `POST` request: `title` ❷ and `price` ❸.

Processing Mixed Query-String and POST Variables

Now let's write the *process.php* script to receive and extract data from this movie form. Unlike our earlier form-processing script, this one needs to extract multiple variables from the incoming `POST` request, including the `id` variable sent through the query string and the `title` and `price` variables embedded in the request body. Add *process.php* to your project's *public* folder and enter the code in [Listing 11-7](#).

```
<?php
//----- (1) LOGIC -----
$id = filter_input(INPUT_GET, 'id');
$title = filter_input(INPUT_POST, 'title');
$price = filter_input(INPUT_POST, 'price');
?>

<!-- (2) HTML template output -->
<!doctype html> <html><head><title>process</title></head><bo
```

```

dy>
id = <?= $id ?>
<br>title = <?= $title ?>
<br>price = <?= $price ?>
</body></html>

```

Listing 11-7: The PHP server script to process the movie form

We use the `filter_input()` function with the `INPUT_GET` argument to read the `id` query-string variable into the corresponding PHP variable, `$id`. (Remember, `INPUT_GET` simply means we're reading data from the query string, even if that data was sent with the `POST` method rather than `GET`. The actual method of the HTTP request makes little difference from the server-side script's perspective.) Then we use `filter_input()` twice more with `INPUT_POST` to read the two values from the request body into the `$title` and `$price` variables. After some basic HTML page tags, we output each variable's name and value by using PHP short echo tags, separating them with HTML `<br>` line breaks.

[Figure 11-7](#) shows how the *process.php* script handles the incoming form data.



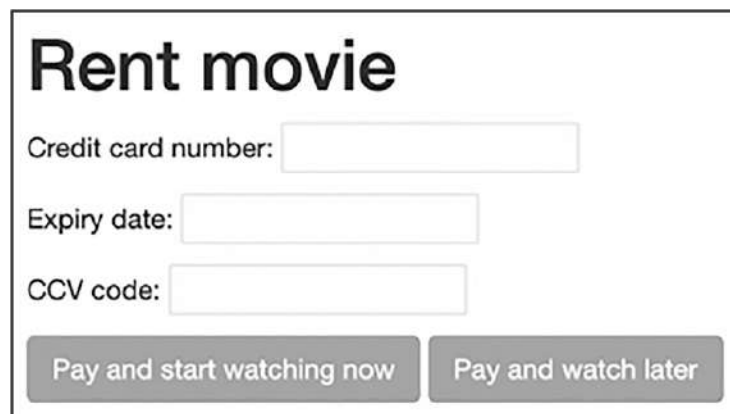
Figure 11-7: An HTTP request sending query-string and `POST` variables

In this example, I filled in `The Lost World` in the title field of the form and `9.99` in the `price` field. The output in the browser shows these values echoed back, along with the `id` value of `1022` that we hardcoded into the query string. You should also see the `id` variable in the URL in the browser's address bar, and if you view the request with the browser developer tools, you should see `id` listed as a query-string parameter, while `title` and `price` are listed as form data variables in the request body.

Offering Multiple Submit Buttons

Another way to send data through a form is to give a `name` attribute to the form's Submit button. This is especially useful when you want a form to feature multiple Submit buttons so the user can choose how to process the data in a form.

For example, a customer renting an online movie might want to pay for it and immediately start watching it, or they might want to pay for it but start watching later. Each option could be triggered by a different Submit button, as illustrated in [Figure 11-8](#). The server-side script can then detect the name of the button the user clicked and respond accordingly.



Rent movie

Credit card number:

Expiry date:

CCV code:

Figure 11-8: Two Submit buttons for the same form

Let's design such a form with multiple Submit buttons. Create a new project containing a *public* folder with a PHP script file named *index.php*, and enter the code in [Listing 11-8](#).

```
<!doctype html><html><head><title>Movie Rent Form 1</title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
css/bootstrap.min.css"
> ❶
</head>
<body class="container">
<h1>Rent movie</h1>
<form method="POST" action="process.php?movieId=80441"> ❷
  <p><label for="number">Credit card number:</label>
  <input name="number" id="number"></p>
  <p><label for="date">
  Expiry date:</label>
  <input name="date" id="date"></p>
  <p><label for="ccv">
  CCV code:</label>
  <input name="ccv" id="ccv"></p>
  <p>
  <input type="submit" name="watchNow" ❸
    value="Pay and start watching now" class="btn btn-su
ccess">
  <input type="submit" name="watchLater" ❹
    value="Pay and watch later" class="btn btn-success">
  </p>
</form>
</body>
</html>
```

Listing 11-8: The HTML code for a form with two Submit buttons

First, we read in the Bootstrap CSS stylesheet ❶. This allows us to style the `submit`-type inputs to look like nice green buttons by using `class="btn btn-success"` without having to write any CSS code ourselves. Then we set up a form using the `POST` method since this form submits data that would likely lead to changes on the server (processing the payment and recording the movie as being rented by the user) ❷. Notice that we've hardcoded a `movieId` variable into the query string through the form's `action` attribute, much as in the previous movie form example.

We give the form input fields for the user's credit card information, and then we define two Submit buttons, one with a `name` attribute of `watchNow` ③ and the other with a `name` attribute of `watchLater` ④. These buttons also have `value` attributes to define the text that will appear in each button. Thanks to these buttons' `name` attributes, when one of them is clicked, its `name` and `value` will be sent as a key/value pair in the body of the `POST` request along with the other form data. For example, if the user clicks the `watchNow` button, a `watchNow=Pay and start watching now` will be sent with the request. The value portion is of little significance, but the server-side script can check for a key of `watchNow` among the form data to determine which Submit button was clicked. [Listing 11-9](#) shows a *process.php* file that does just that.

```
<?php
if (filter_has_var(INPUT_POST, 'watchNow')) {
    print 'you clicked the button to <b>Watch Now</b>';
} else {
    print 'you clicked the button to <b>Watch Later</b>';
}
```

Listing 11-9: Detecting which Submit button was clicked in process.php

Since the form has only two Submit buttons, we use an `if...else` statement to test whether one of them (`watchNow`) was clicked; if not, we can safely assume that the other was clicked instead. (If we had three or more buttons, we could use `elseif` statements or a `switch` statement to detect the correct button.) In theory, we could call the `filter_input()` function as usual, extracting the value of the `watchNow` variable and checking that its value isn't `NULL` to determine whether that's the button that was used. Since we aren't interested in the value of `watchNow`, but rather in whether such a variable even exists in the incoming request, we instead use PHP's `filter_has_var()` function to set the `if...else` statement's condition. This function takes two input parameters, the source of the variable (usually `INPUT_GET` or `INPUT_POST`) and the name of the variable, and returns `true` or `false` based on whether that named value is found.

[Figure 11-9](#) shows a sample submission through our movie rental web form.

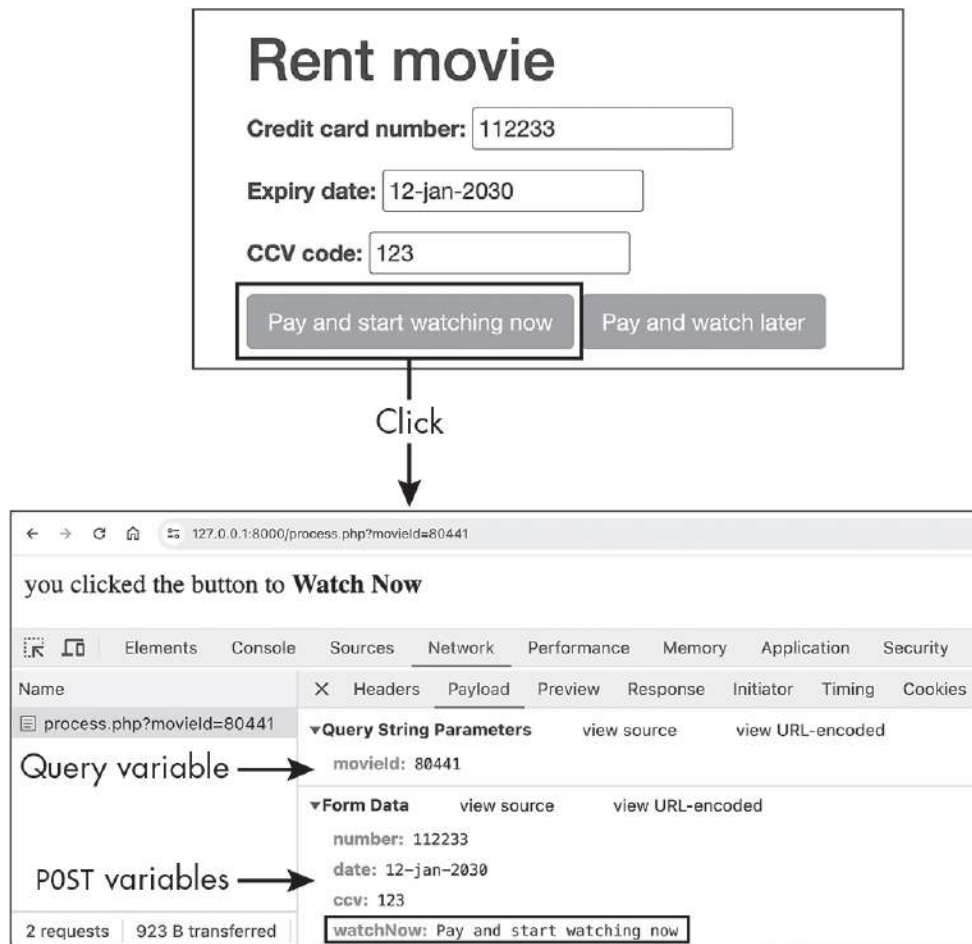


Figure 11-9: Finding a Submit button name among the `POST` variables in the request body

In this example, I've used the `watchNow` button to submit the form data. The message on the resulting page confirms that the *process.php* script detected this button. Further, a look at the request with the browser developer tools shows `watchNow` listed with the other `POST` variables.

Encoding Data in Hyperlinks

In addition to sending data through a web form, we can send data to a server by adding name/value pairs to the query string at the end of the URL in an HTML hyperlink. This data will be sent with the `GET` method, since whenever you click a link in a web page, your browser is making an HTTP

GET request using the URL of the clicked link. An HTML hyperlink is represented with the anchor (<a>) element; the link's URL is set via the element's href attribute.

We'll explore this additional way to send data via the GET method through a typical example of a link that shows details about an item in an online shopping cart. [Figure 11-10](#) shows the button-styled links we want to create.



Figure 11-10: The Details hyperlinks styled as buttons in a shopping cart

Each item in the cart has a hyperlink (styled as a button) alongside it to show details about that item. The links might be to URLs such as `/show.php?id=102`. This URL would request the PHP script `show.php` via the GET method, while passing the product's ID (in this case, 102) through the query string `id` variable. The GET method is more appropriate than POST in this case since the intended result is simply to display data (so no content is being changed on the server).

Hardcoding the Links

Let's create the page shown in [Figure 11-10](#). For simplicity, we'll begin by hardcoding the product IDs into the hyperlinks. Create a new project containing a *public* folder and add an *index.php* script to that folder. Then enter the code in [Listing 11-10](#).

```
<!doctype html><html><head><title>Basket Form 1</title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
css/bootstrap.min.css"
>
</head>
```

```

<body class="container">
<h1>Your shopping cart</h1>

<div class="row">
  <div class="col-lg-3 text-end py-2">
    Hammer $ 19.99
    <a href="/show.php?id=102" class="btn btn-primary">D
etails</a> ❶
  </div>
</div>

<div class="row">
  <div class="col-lg-3 text-end py-2">
    Bucket of nails $ 9.99
    <a href="/show.php?id=511" class="btn btn-primary">D
etails</a> ❷
  </div>
</div>
</body>
</html>

```

Listing 11-10: An index.php file with data embedded in the Details hyperlinks

As in the previous example, we read in the Bootstrap CSS stylesheet as a shortcut for styling the page. In the page's HTML, we create two links with the text `Details`, styled as blue buttons using the Bootstrap CSS class `btn btn-primary`: one for a hammer ❶ and one for a bucket of nails ❷. Each link is to the PHP script `show.php`, with the ID of the product encoded into the URL in the fashion `?id=102`. Clicking one of these links will send the appropriate `id` variable via a `GET` request, as shown in [Figure 11-11](#).

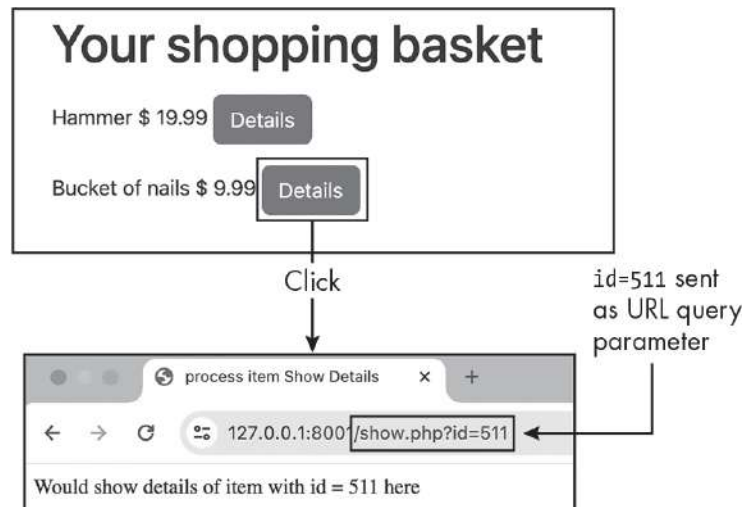


Figure 11-11: The result of clicking one of the Details links

We won't worry about writing the *show.php* script, but notice that clicking the Details link for the bucket of nails initiates a `GET` request with an `id` value of 511 embedded in the query string.

NOTE

In a more realistic scenario, we wouldn't hardcode `id` values into links like this. Instead, we'd loop through an array representing the products in the user's shopping cart and programmatically insert each product's ID into its corresponding Details link. We'll look at how to do this next.

Generating the Links Programmatically

Most of the content presented in web pages is dynamically generated at runtime based on values in the website's database. Therefore, instead of hardcoding product ID values into links like */show.php?id=102*, links are usually created programmatically with PHP statements looping through a collection of data representing the items in the user's shopping cart. Each time through the loop, the product's ID is looked up and dynamically inserted into a hyperlink. The product's description and price are similarly dynamically inserted into generic HTML template text.

Let's update our shopping cart page to try this approach. We'll use an array to represent the shopping cart as a whole; each item in the array will itself be an array representing a particular cart item, with three values for

the ID, description, and price of the product. Modify the *index.php* file as shown in [Listing 11-11](#).

```
<?php
// Set up data array
$items = [
    ['id' => 102, 'description' => 'Hammer', 'price' => 9.9
9],
    ['id' => 511, 'description' => 'Bucket of nails', 'pric
e' => 19.99],
];
?>

<!doctype html><html><head><title>Cart From Array</title>
<link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
css/bootstrap.min.css"
>
</head>
<body class="container">
<h1>Your shopping cart</h1>
<?php foreach ($items as $item): ?> ❶
    <div class="col-lg-3 text-end py-2">
        <?= $item['description'] ?> $ <?= $item['price'] ?>
        <a href="/show.php?id=<?= $item['id'] ?>" class="btn
btn-primary">Details</a> ❷
    </div>
<?php endforeach; ?> ❸
</body>
</html>
```

Listing 11-11: Using a PHP loop to create the Details links for the shopping cart items

Before any HTML, we use a PHP code block to declare an `$items` array containing information about our two products. (Of course, we’re still hardcoding the information, albeit within an array; in a more realistic scenario, we’d retrieve the product information from a database, as we’ll discuss in [Part VI](#)). Then we begin the HTML template text. Under the `Your shopping cart` heading, we use another PHP code block to begin a

`foreach` loop in which the `$item` variable will represent the current element of the `$items` array ❶. We use alternative loop syntax with a colon (`:`) to set up the start of the loop and `endforeach` ❸ to close it. See [Chapter 6](#) to review this alternative loop syntax which makes it easier to combine PHP with HTML.

Inside the loop, we use a mix of HTML and PHP short echo tags within a `<div>` to insert the current product's `'description'`, `'price'`, and `'id'` values into the template text. In this way, we dynamically create a `<div>` element for each product, including a Bootstrap-styled Details link. Notice in particular that we embed the product's ID into the `href` property of the `<a>` element ❷, which will result in hyperlinks like `/show.php?id=102`, just as before. Overall, the page should look exactly the same as [Figure 11-10](#).

Other Form Input Types

Single text and numeric form inputs (`text`, `password`, `textarea`, and so on) are all sent in the HTTP request as a name/value pair, but other types of form data aren't quite so simple. It's important to understand how the browser chooses the variable names and values for these other form elements so that you can write server scripts to correctly retrieve and validate the incoming data. In this section, we'll discuss how to work with other common form elements like radio buttons, checkboxes, and single- and multiple-selection lists.

Radio Buttons

Radio buttons are a set of two or more form input choices offered to the user, of which only one value can be chosen. Radio button inputs are declared in groups that share the same `name` attribute, with each having a unique `value` attribute to distinguish which input was selected. In this way, a set of radio buttons forms a group of mutually exclusive choices for the value assigned to the shared `name` attribute. Except in rare situations where it's acceptable for no option to be selected, one of the radio buttons should be automatically checked so the user is offered a default choice. This way, a value will definitely be sent in the HTTP request.

Let's write the HTML for a form that uses radio buttons to present a choice between two Irish counties, as shown in [Figure 11-12](#). This figure

also illustrates the output when the form is processed by *process.php*.

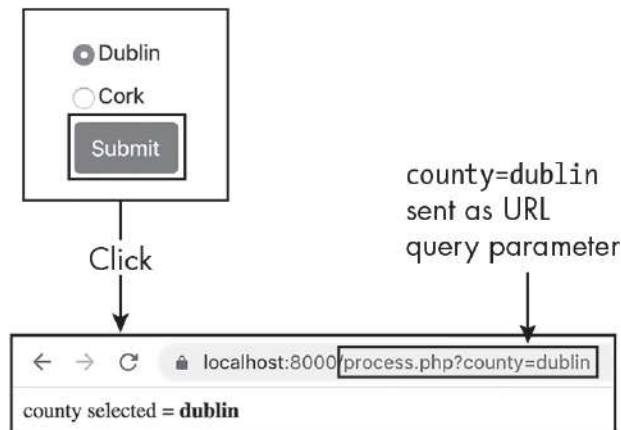


Figure 11-12: Radio buttons submitting values via a query-string parameter

Create a new project containing a *public* folder with an *index.php* script, then enter the code shown in [Listing 11-12](#).

```
<!doctype html><html><head><title> Radio Buttons </title>
<link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
css/bootstrap.min.css"
>
</head>
<body class="container">
<form method="GET" action="process.php">
    <br>
    <label>
        <input type="radio" name="county" value="dublin" checked> ❶
        Dublin
    </label>

    <br>
    <label>
        <input type="radio" name="county" value="cork"> ❷
        Cork
    </label>
```

```
<br>
<input type="submit" class="btn btn-primary">
</form>
</body>
</html>
```

Listing 11-12: An HTML form illustrating radio buttons

We create a form using the `GET` method with an `action` attribute to request the `process.php` script. Within the form, we declare two radio buttons (`<input>` elements of type `"radio"`), both with `"county"` as a `name` attribute. One has a `value` of `"dublin"` ❶, and the other a `value` of `"cork"` ❷. We use the `checked` attribute to set the Dublin option as the default. All the user sees for a radio button is a small circular input, so it's important to add a text or image prompt immediately next to each radio button, and to use `<label>` elements so that the user can click either the text or the button.

NOTE

The labels show each county starting with a capital letter (such as `Dublin`), while the corresponding values start with a lowercase letter (`dublin`). Personally, I always use lower camel case for the values of radio buttons, checkboxes, and other inputs. Having a consistent naming convention like this makes it easier to write the form-processing logic without having to look back at the form code itself and reduces the number of mistakes you're likely to make.

Because this form uses the `GET` method, we'll see either `?county=dublin` or `?county=cork` in the URL when the form is submitted. In other words, the `name` attribute of the radio button group serves as the key for a query-string variable, and the `value` attribute of the selected button serves as the variable's value. We would therefore use `filter_input(INPUT_GET, 'county')` to extract the value submitted by the user through the button group.

Checkboxes

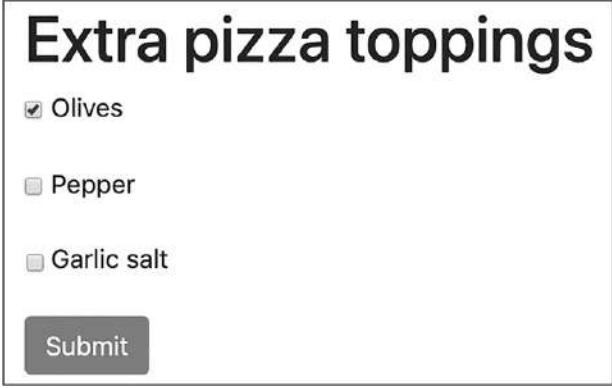
Checkboxes offer Boolean (`true/false`) choices to the user. They appear in the browser as small squares that can be checked or unchecked. You might use checkboxes in a form, for example, to allow the user to select toppings when ordering a pizza. Unlike radio buttons, checkboxes aren't mutually exclusive; the user can check as many of the boxes as they want. As with radio buttons, adding text or an image prompt immediately next to each checkbox lets the user know what they're selecting.

The checkboxes in a form can be treated individually or processed collectively as an array. We'll look at both approaches.

Treated Individually

When checkboxes are treated individually, each one should have a unique `name` attribute. You can define a `value` attribute for each checkbox as well, but it isn't really necessary for processing the form. A checkbox will send its name/value pair with the HTTP request only if it has been selected, so on the receiving end, it's sufficient to simply test for the box's name with the `filter_has_var()` function, regardless of the box's value. If a checkbox is selected and no value was defined, the default value of `on` will be submitted with the form data.

To see how this works, let's use checkboxes to create a form for pizza toppings. [Figure 11-13](#) shows how the form should look.



The image shows a web form with a title "Extra pizza toppings" in a large, bold, black font. Below the title are three checkboxes, each followed by a label: "Olives" (checked), "Pepper" (unchecked), and "Garlic salt" (unchecked). At the bottom of the form is a gray "Submit" button.

Figure 11-13: A form with checkboxes

Start a new project with a *public/index.php* file containing the code in [Listing 11-13](#) to design the pizza toppings form.

```
<!doctype html><html><head><title> Checkboxes </title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/css/bootstrap.min.css"
>
</head>
<body class="container">
<h1>Extra pizza toppings</h1>
<form method="GET" action="process.php">
  <p><label><input type="checkbox" name="olives"> Olives</
label></p>
  <p><label><input type="checkbox" name="pepper"> Pepper</
label></p>
  <p><label><input type="checkbox" name="garlic"> Garlic s
alt</label></p>
  <p><input type="submit" class="btn btn-primary"></p>
</form>
</body>
</html>
```

Listing 11-13: An HTML form with checkboxes

Inside a form declared with the `GET` method, we create checkboxes for the olives, pepper, and garlic salt options, as well as a Submit button. Each checkbox has a unique `name` attribute and is immediately followed by a text label to indicate which option the checkbox represents. Because the checkboxes don't have explicitly declared `value` attributes, when the user submits the form, each checkbox that has been selected will add a variable to the query string in the form `<name>=on`. For example, if only “Olives” is selected, the form will trigger a `GET` request with the URL *http://localhost:8000/process.php?olives=on*.

Listing 11-14 illustrates the sort of logic we might use in the *process.php* script to detect one of the checkboxes.

```
<?php
$olivesSelected = filter_has_var(INPUT_GET, 'olives');
var_dump($olivesSelected);
```

Listing 11-14: Detecting an individual checkbox

We call `filter_has_var()` with `INPUT_GET` and `'olives'` to detect whether an `olives` variable was sent with the `GET` request, indicating this box was checked. We store the resulting `true/false` value in `$olivesSelected`, which for simplicity we pass to `var_dump()`. In a more realistic scenario, we might use the Boolean to set up conditional logic. We could use a similar `filter_has_var()` test for the other checkboxes, as long as each is uniquely named.

Treated as an Array

Sometimes it's more practical to treat two or more checkboxes as related by giving them all the same `name` attribute ending in square brackets. For example, the checkboxes in the pizza toppings form could all be given a name of `toppings[]`. This way, all the toppings that are selected will be grouped into an array and sent in the HTTP request under the same `toppings` variable name. When taking this approach, it's vital to ensure that each checkbox has a unique `value` attribute so the individual boxes can still be distinguished from one another.

Listing 11-15 shows how to rewrite the pizza toppings form to send all selected checkboxes as values in a single array.

```
<!doctype html><html><head><title>Checkboxes array</title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
>
</head>
<body class="container">
<h1>Extra pizza toppings</h1>
<form method="GET" action="process.php">
  <p><label>
    <input type="checkbox" name="toppings[]" value="oliv
```



```

es"> Olives
    </label></p>
    <p><label>
        <input type="checkbox" name="toppings[]" value="pepp
er"> Pepper
    </label></p>
    <p><label>
        <input type="checkbox" name="toppings[]" value="garl
ic"> Garlic salt
    </label></p>
    <p><input type="submit" class="btn btn-primary"></p>
</form>
</body>
</html>

```

Listing 11-15: Grouping a form's checkboxes into an array

As before, we declare checkboxes for the three topping options. However, this time we assign `toppings[]` as the name for each checkbox so they will be grouped into a `toppings` array. We also add a `value` attribute to each checkbox, indicating the type of topping to be added to the array if selected. [Figure 11-14](#) shows how multiple values for `toppings[]` appear in the query string when the form is submitted with all three boxes checked.

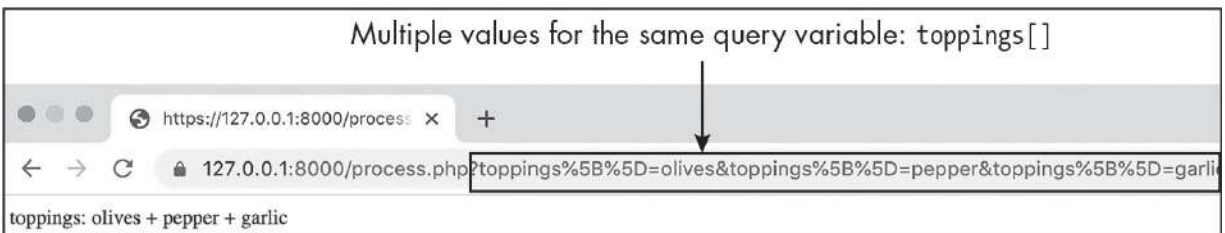


Figure 11-14: Checkbox values being submitted as values for a single array variable

To successfully process this form, we'll have to ensure that the *process.php* script is set up to expect an array of checkbox values. The array may be empty if no checkboxes were selected, or it could contain one or more values. [Listing 11-16](#) uses an `if...else` statement to handle both

cases, so it should work regardless of how many pizza toppings are selected.

```
<?php
❶ $toppings
    = filter_input(INPUT_GET, 'toppings', options: FILTER_
    REQUIRE_ARRAY);

if (empty($toppings)) {
    ❷ print 'no extra toppings selected';
} else {
    ❸ $toppingsString = implode('+', $toppings);
    print "toppings: $toppingsString";
}
```

Listing 11-16: Processing an array of checkbox values

Now that the values of the checkboxes matter, we need to use `filter_input()` instead of `filter_has_var()` ❶. As usual, we use the function's first two arguments to retrieve a `toppings` variable from the query string, but this time we also use a named argument of `options:` `FILTER_REQUIRE_ARRAY` to specify that we're looking for an array of values under the `toppings` variable name. We store the resulting array in the `$toppings` PHP variable.

We need to use a named argument here because `$options` is the `filter_input()` function's fourth parameter. We're skipping over the third parameter, `$filter`, leaving it with its default value of `FILTER_DEFAULT`. See [Chapter 5](#) for a review of named and optional parameters.

We next use an `if...else` statement to check whether the `$toppings` array is empty. If so, we print a message accordingly ❷. Otherwise, we use the built-in `implode()` array function to collapse the array of toppings into a single string, with the topping names separated by plus signs ❸. Look back at the bottom of [Figure 11-14](#) to see the resulting string of toppings when all three boxes are checked.

Single-Selection Lists

A *single-selection list* displays as a drop-down menu enabling the user to choose one option. This list is created with an HTML `<select>` element containing `<option>` elements representing the possible choices. The `<select>` element has a name that will be assigned the value of the chosen `<option>` element when the form is submitted, resulting in a straightforward name/value pair on the receiving end. [Figure 11-15](#) shows a form offering a simple single-selection list of flowers.

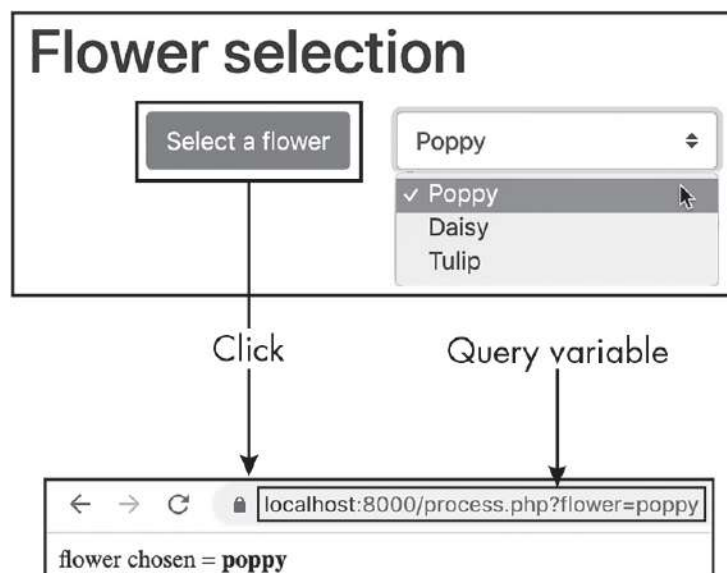


Figure 11-15: A single-selection list

Let's create this flower form. Start a new project with a `public/index.php` file and enter the contents of [Listing 11-17](#).

```
<!doctype html><html><head><title>Single Selection list</title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
>
</head>
<body class="container">
<h1>Flower selection</h1>
```

```

<form method="GET" action="process.php">
<div class="row">
  <div class="col-lg-3 text-end py-2">
    <input type="submit" value="Select a flower" class
="btn btn-primary">
  </div>
  <div class="col-lg-3 text-end py-2">
    <select name="flower" class="custom-select"> ❶
      <option value="poppy">Poppy</option>
      <option value="daisy">Daisy</option>
      <option value="tulip">Tulip</option>
    </select>
  </div>
</div>
</form>
</body>
</html>

```

Listing 11-17: An HTML form with a single-selection list

We declare a form with the `GET` method containing a Submit button alongside a selection list with the name `flower` ❶. The `<select>` element holds three `<option>` elements with `value` attributes of `poppy`, `daisy`, and `tulip`. When the form is submitted, the selected flower will be sent in the HTTP request as a query parameter named `flower`. In [Figure 11-15](#), for example, *flower=poppy* appears in the query string when the poppy is selected.

We can process the choice submitted through a single-selection list by using `filter_input()` and the name of the `select` element. In the case of our flower form, for example, we can use `filter_input(INPUT_GET, 'flower')` to extract the user’s chosen flower.

Multiple-Selection Lists

A multiple-selection list allows the user to choose more than one option from a menu. This list is created by adding the `multiple` attribute to the HTML `<select>` element. [Figure 11-16](#) shows a multiple-selection version of our flower form.



Figure 11-16: A multiple-selection list

Whereas a single-selection list results in a simple name/value pair, a multiple-selection list should be handled similarly to the array approach we used for checkboxes: the list should have a `name` attribute ending with square brackets, as in `flowers[]`. This way, we'll get an array containing the zero, one, or more values selected from the list's options. [Listing 11-18](#) shows how to modify the `index.php` file for our flower form to turn it into a multiple-selection list.

```
<!doctype html><html><head><title>Multiple Selection list</t
itle>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/css/bootstrap.min.css"
>
</head>
<body class="container">
<h1>Flower selection</h1>
<form method="GET" action="process.php">
<div class="row">
  <div class="col-lg-3 text-end py-2">
    <input type="submit" value="Select some flowers" cla
ss="btn btn-primary">
  </div>
  <div class="col-lg-3 text-end py-2">
    <select name="flowers[]" ❶
      class="custom-select"
      size="3"
      multiple> ❷
      <option value="poppy">Poppy</option>
      <option value="daisy">Daisy</option>
      <option value="tulip">Tulip</option>
```

```
        </select>
--snip--
```

Listing 11-18: An HTML form illustrating a multiple-selection list as an array of values

We modify the selection list to have an array name of `flowers[]` ❶ and specify that it should be displayed in a box that fits three rows. We include the `multiple` attribute to indicate that the user may select more than one item from the list (this attribute doesn't need to be assigned a value) ❷. When the form is submitted, any selected flowers will be sent in the HTTP request as an array named `flowers`.

We can process this form by using similar logic to that of our script processing an array of checkboxes from [Listing 11-16](#). We use an `if...else` statement to check whether the incoming array is empty, as shown in [Listing 11-19](#).

```
<?php
$flowers = filter_input(INPUT_GET, 'flowers', options: FILTER_REQUIRE_ARRAY);

if (empty($flowers)) {
    print 'no flowers selected';
} else {
    $stoppingsString = implode(' + ', $flowers);
    print "flowers: $stoppingsString";
}
```

Listing 11-19: Processing the array of values from a multiple-selection list

We extract the `$flowers` array from the incoming `GET` request, once again using the extra `options: FILTER_REQUIRE_ARRAY` argument to `filter_input()` to specify that we're looking for an array. Then, if the array isn't empty, we use `implode()` to merge the received flowers into a string separated by plus signs.

Summary

In this chapter, we explored how to create web forms that submit data to server scripts, and we discussed the difference between submitting those forms with the HTTP `GET` and `POST` methods. We also covered how to write server-side scripts to identify and extract the submitted form data. We focused on how to use PHP's `filter_input()` function to extract the values of variables from an incoming HTTP request, as well as how to use `filter_has_var()` when all we need to know is whether a variable exists at all. We applied this approach to a variety of form input types, including text inputs, radio buttons, checkboxes, and selection lists.

Exercises

1. Create a form for a costume-rental company that enables the user to submit the name of the superhero costume they want. Make the form submit data via the HTTP `GET` method to a PHP script called *costumeSearch.php*, and use `hero` for the name of the text value entered by the user. Visit the form through your web server and submit the value **superman**. You should see *costumeSearch.php?hero=superman* in the resulting URL.
2. Create a basic login form using the HTTP `POST` method, with `<input>` elements for a username (using the default `text` type) and a password (use the `password` type). Use your browser's developer tools to inspect the form data when it's submitted.
3. Create a form to input the user's age and a script to process the received age. The processing script should return an HTML page containing a message stating how old the user will be on their next birthday.
4. Create a form with radio buttons asking the user whether they would be willing to pay more for an environmentally friendly car. If they say yes, return an HTML page containing a message recommending they buy an electric car. If they say no, recommend a gas car.
5. Create a form offering the user several options for a new car, each represented with a uniquely named checkbox. For example, offer options such as metallic paint, fog lights, and a reversing camera. Write

a script that processes the form data and returns an HTML page containing a message confirming each selected option.

6. Duplicate your answer for Exercise 5 about car options, but change the checkboxes to send the data as part of an array variable named `extras[]`. Update your form-processing code to handle the array. Think about whether you find the array approach easier or harder than processing the checkboxes individually.

12

VALIDATING FORM DATA



Not all data received through web forms will be valid; users may make mistakes or miss required values, or a number of other things could go wrong. In this chapter, we'll explore ways to validate the received data, and we'll design some typical decision logic to take appropriate action depending on the values received (and any that are missing).

It's important to recognize that HTML forms submit values only as text strings, regardless of the input type. One of the very reasons PHP isn't type-sensitive is to make it easy to treat a string containing digits as a number without explicitly type-casting or including data-type conversion statements. This makes it that much more important to carefully write (and test) the validation rules for processing received form data.

Writing Custom Validation Logic

In [Chapter 11](#), you learned to use simple built-in filters such as `FILTER_SANITIZE_SPECIAL_CHARS` in conjunction with `filter_input()` to sanitize incoming form data. However, real-world data often has its own special

validation criteria that goes beyond these built-in filters. Once you've retrieved data from the submitted form request, you may therefore need to write custom validation logic to make sure the data is as it should be.

Let's illustrate how this works with a simple example: a product details form asking the user to enter a product code and a price (see [Figure 12-1](#)). We'll assume that the product code must have a minimum of three characters and that the price must be a number (either an integer or a decimal value).



The image shows a form titled "Product details" in a large, bold, black font. Below the title, there are two input fields. The first field is labeled "Product code:" and the second field is labeled "Price:". Both labels are in a standard black font. The input fields are simple rectangular boxes with thin black borders.

Figure 12-1: A product details form in need of validation logic

As we explored in [Chapter 11](#), we can write a simple script confirming receipt of the form data with a few lines of PHP. [Listing 12-1](#) shows how.

```
<?php
$productCode = filter_input(INPUT_GET, 'productCode');
$price = filter_input(INPUT_GET, 'price');
?>
<h1>Data received:</h1>
<p>Product Code: <?= $productCode ?></p>
<p>Price: <?= $price ?></p>
```

Listing 12-1: Confirming the received product code and price

Here we use the `filter_input()` function to extract the `productCode` and `price` variables from the received URL-encoded form data and display them back to the user by using PHP short echo tags. This will work if the user has submitted values as expected, but it's never a good idea to assume the data that has arrived from the user is all present and correct.

We should expand the script to validate the data before confirming it to the user. We'll try to validate the price and product code data in the

following ways:

- If the product code is missing or empty, display the error message `missing product code`.
- If the product code is less than three characters in length, display the error message `product code too few characters`.
- If the price isn't a numeric value, display the error message `price was not a number`.
- If no validation errors occur, display the message `input data was error free`.

As you'll see, PHP has built-in functions to help with these sorts of validation checks.

Managing Multiple Validation Errors

One common approach to managing custom validation logic with multiple potential errors is to use an array for the errors. Start with an empty array, then use a series of `if...else` statements to add an error message to the array each time a new validation error is detected. If the array is empty after all the validation, it means the data was found to be error free. If it isn't empty, you can loop through the array to display all the error messages to the user. [Listing 12-2](#) uses this approach to implement the validation logic for our product details form.

```
<?php
$productCode = filter_input(INPUT_GET, 'productCode');
$price = filter_input(INPUT_GET, 'price');
```

- ```
❶ $errors = [];
❷ if (empty($productCode)) {
 $errors[] = 'missing product code';
} elseif (strlen($productCode) < 3) {
 $errors[] = 'product code too few characters';
}

❸ if (!is_numeric($price)) {
 $errors[] = 'price was not a number';
```

```

 }

❷ if (isset($_POST['productCode'])) {
 // validate product code
 if (empty($productCode)) {
 $errors[] = 'missing product code';
 } elseif (strlen($productCode) < 3) {
 $errors[] = 'product code too few characters';
 }

 // validate price
 if (!is_numeric($_POST['price'])) {
 $errors[] = 'price was not a number';
 }

❸ if (count($errors) > 0) {
 // display errors
 print 'Data validation errors:';
 foreach ($errors as $error) {
 print " $error ";
 }
 print "";
} else {
 print 'input data was error free';
}

```

---

*Listing 12-2: Implementing custom validation logic with an array of error messages*

After extracting the product code and price from the incoming form data as before, we create an empty array in the `$errors` variable ❶. Then we use an `if...elseif` structure ❷ to validate the `$productCode` variable. The `if` branch adds the message 'missing product code' to the `$errors` array if `$productCode` is empty. (The `empty()` function will return `true` if `$productCode` wasn't found in the received form data or if it contained an empty string.) The `elseif` branch, which is visited only if `$productCode` isn't empty, adds the message 'product code too few characters' to the array if the `strlen()` function finds the product code to contain fewer than three characters.

Next, we use a separate `if` statement to validate the `$price` variable ❸. We pass the variable to `is_numeric()`, which returns `true` if the received string can be interpreted as an integer or a float. If not, we add the message 'price was not a number' to the array.

Finally, we test whether the `$errors` array contains any errors ❹. If it does, we use a `foreach` loop to display each error to the user as an item in an HTML ordered list. Otherwise, if the array is empty, we display the message 'input data was error free'.

This example has illustrated a range of typical validations. You'll often find yourself checking for the following:

**Any required data that's missing** You can test for this with the `empty()` function.

**Too few characters for text data** You might need to ensure that the input meets minimum-length rules for usernames or passwords, for example. You can test for this with the `strlen()` function.

**Non-numeric values** You might want to know if the value received is neither a valid integer nor a decimal number. You can test for this with the `is_numeric()` function.

While we were able to rely on built-in PHP functions for our validation checks, we needed custom logic to string them together and match them to the particular requirements of our product details form.

## ***Testing for a Valid Zero Value***

Because of PHP's type insensitivity, it's easy to write code that incorrectly treats a numeric value of 0 as missing or `false`. This happens because in PHP all the following (among other values) are considered `false`:

- The Boolean `false` itself
- The integers 0 and -0
- The floats 0.0 and -0.0
- The empty string and the string "0"
- The special `NULL` type (including unset variables)

To illustrate the problem, say you want to test whether the user left a form field blank. You'd typically use the `empty()` function, but if the user happened to enter the digit 0 in the form field, the test `empty('0')` will return `true`, just like the test `empty('')` for a truly empty form field (remember that all form values arrive as strings in the HTTP request, even if they are numeric characters). If you want the 0 to be a valid entry for the field (for example, if the field is to record someone's score on a test), you'll need to write code that distinguishes between a valid 0 entry and an unacceptable empty string. The solution is to use the triple-equal-sign identical operator (`===`), which tests whether its operands have the same value *and* data type, as discussed in [Chapter 2](#).

---

## NOTE

*The PHP documentation describes type comparisons at <https://www.php.net/manual/en/types.comparisons.php>, comparing the results of the `==` and `===` operators, as well as showing the results for functions such as `gettype()`, `empty()`, and `isset()` for potentially confusing values. I recommend you bookmark this page for reference.*

Listing 12-3 assumes a form has been submitted with the `GET` method sending a variable named `score`. The code uses an `if...elseif` statement to differentiate between no value and a `0` value.

---

```
$score = filter_input(INPUT_GET, 'score');
if ($score === '0') {
 print "score was the string '0'";
} elseif (empty($score)) {
 print 'score was empty (but not zero)';
}
```

---

*Listing 12-3: Distinguishing between the number 0 and an empty or `NULL` value*

We first use the `===` operator to test whether `$score` holds the exact string `'0'`. If not, we use the `empty()` function to check whether an empty string was received.

If testing for a valid `0` is a task you'll need to perform often, it would be useful to encapsulate the necessary logic in a function that returns `false` when given the string `'0'` or the result of the `empty()` function otherwise, as shown in [Listing 12-4](#). Create a `zeroFunction.php` file containing this code, since we'll make use of it later in this chapter.

---

```
<?php
function isEmptyNonZeroString($s): bool
{
 if ($s === '0') return false;
 return empty($s);
}
```

---

*Listing 12-4: A function that tests for an empty string but does not consider the digit 0 as empty*

Keep in mind that if a numeric value of 0 is an acceptable form input, you may need to do more than simply compare the incoming string with the string literal '0'. This is because other strings may also evaluate to 0, such as '0.0', '0.00', and so on. To be thorough, you need to test the underlying numeric value that the incoming string represents. Fortunately, PHP provides the `intval()` function, which can take in a string and return the numeric integer value of its contents. Normal PHP type juggling will take place, and so any valid numeric characters at the beginning of the string will be used to determine the string's integer value. As discussed in [Chapter 2](#), any non-numeric content in a string is ignored during type juggling.

[Table 12-1](#) lists examples of strings and their `intval()` evaluations. Notice that the non-numeric remainders of strings are simply ignored (such as '5abc' evaluating to 5), as are any decimal components of numeric strings.

**Table 12-1:** Example `intval()` Calls

| Function call               | Return value                       |
|-----------------------------|------------------------------------|
| <code>intval('0')</code>    | 0                                  |
| <code>intval('00')</code>   | 0                                  |
| <code>intval('0.00')</code> | 0                                  |
| <code>intval('0.99')</code> | 0 (decimal component ignored)      |
| <code>intval('0005')</code> | 5 (leading zeros ignored)          |
| <code>intval('5abc')</code> | 5 (everything from 'a' on ignored) |

At the end of the day, validation should match whatever data the form owner specifies is acceptable. It is, of course, always a good idea to offer the user a confirmation screen with a chance to correct values after any validation and string-to-number evaluation, to make sure the form's validation logic matches the user's intent.

## Displaying and Validating Forms in a Single Postback Script

A frequently used strategy for validating web forms is to use a single script, known as a *postback*, both to display the form and to perform validation. Combining these two actions ensures that the form can be redisplayed to the user in the event of any validation errors, with the added validation error messages included. A postback also makes it easy to implement a *sticky form*, a form that's redisplayed with the previously submitted values prepopulated in the appropriate fields, saving the user from having to enter the data again. (The term *sticky* refers to the values remaining, or sticking, in the form after submission.)

Using a single script to both display and validate has two drawbacks. First, the script can get very long, especially for complex forms with lots of validation logic. Second, the complexity of a script performing multiple tasks is high, making the code potentially harder to understand, more error-prone, and harder to update or maintain at a later date. In [Chapter 13](#), we'll address these disadvantages as we look at strategies to separate logic from display code.

For now, though, we'll focus on creating single scripts for sticky postback forms. We'll start with a form that requires only simple validation logic, and then we'll revisit the array-based approach to handling multiple validation errors discussed earlier in the chapter.

### **Simple Validation Logic**

Let's create a simple postback script for a sticky form where the user enters their name and receives a *Hello <name>* greeting in response. We'll arbitrarily say that the submitted name must have at least three characters to be valid. Our postback script needs to handle three possible situations that can occur after the server receives a new HTTP request. The first situation is the initial form request (via the `GET` HTTP method), when an empty text box and Submit button are presented to the user, as in [Figure 12-2](#).





*Figure 12-2: The first presentation of the sticky postback form*

The second situation is a form submission with invalid data (via the `POST` HTTP method). In this case, the form should be redisplayed to the user with an error message about the invalid submission. Since the form is sticky, the text box should be populated with whatever text the user submitted last, as shown in [Figure 12-3](#). In this example, a single letter `a` was submitted.



*Figure 12-3: Redisplaying the sticky postback form with a message about the invalid data*

The third and final situation is a form submission with valid data (via the `POST` HTTP method). In this case, a confirmation message should be displayed to the user, as shown in [Figure 12-4](#).

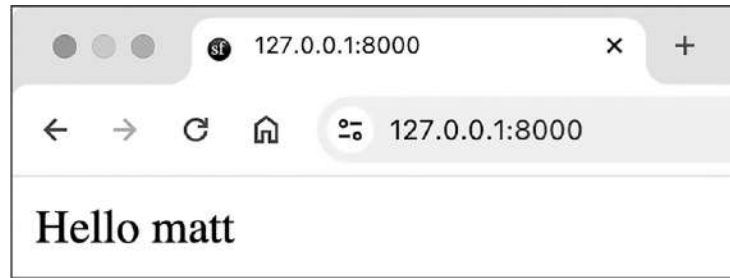


Figure 12-4: Displaying a confirmation message after valid data is submitted

The logic for our postback PHP script can be visualized as a flow chart with two decisions ([Figure 12-5](#)).

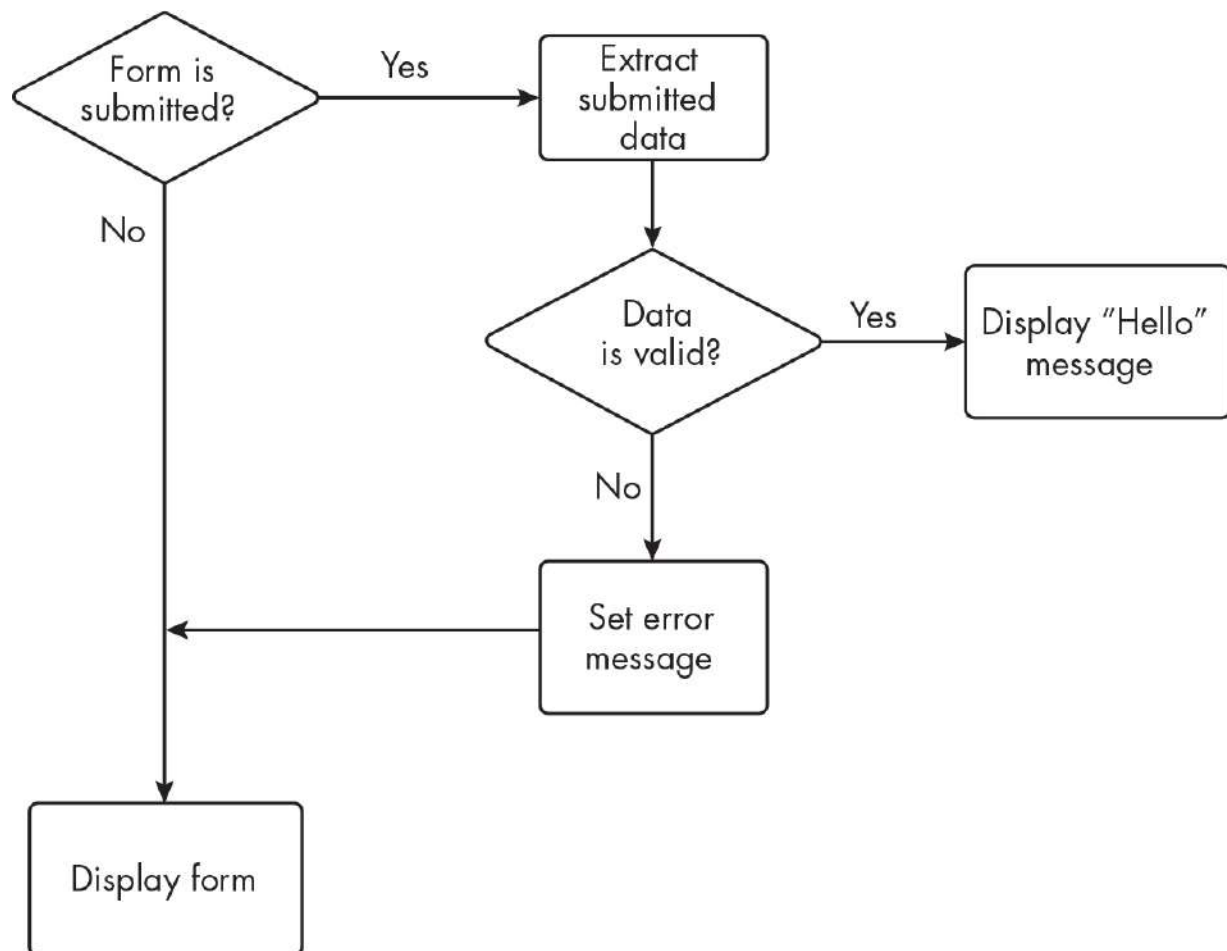


Figure 12-5: A decision flowchart for the postback sticky form

We'll write the postback script in a *public/index.php* file in four stages: setting default values, validating the submitted data, displaying a confirmation for valid data, and displaying the form.

## Setting Default Values

First, we need to set default values for the two key postback variables `$isSubmitted` and `$isValid`. The first of these Boolean flags records whether the form has been submitted, as determined by whether the incoming request uses the `POST` method (indicating a form submission) or the `GET` method (indicating the initial request for the form). The second flag signals whether the data received is valid. A third variable will set the default value of the `$firstName` variable for our logic. [Listing 12-5](#) shows the code.

---

```
<?php
$isSubmitted = ($_SERVER['REQUEST_METHOD'] === 'POST');
$isValid = true;
$firstName = '';
```

---

### *Listing 12-5: Setting default values*

We set the `$isSubmitted` Boolean flag to `true` or `false` depending on the value of the `'REQUEST_METHOD'` key in the `$_SERVER` array. This superglobal array holds information from the web server about the current executing script, the request being processed, and the like. If the value is the string `'POST'`, then `$isSubmitted` will be `true`; otherwise, it will be `false`.

We next set the `$isValid` Boolean flag to `true` by default; we're therefore assuming the received data is valid unless later logic determines otherwise. Finally, we set `$firstName` to an empty string to ensure that, whatever happens, we have a value for this variable that can be inserted into the form text box when the form is presented to the user.

## NOTE

Learn more about the `$_SERVER` array in the PHP documentation at <https://www.php.net/manual/en/reserved.variables.server.php>.

## Validating the Submitted Data

The second block of code in our postback script, shown in [Listing 12-6](#), attempts to validate the submitted form data after the form has been successfully submitted.

---

```
if ($isSubmitted) {
 ❶ $firstName = filter_input(INPUT_POST, 'firstName');

 ❷ if (strlen($firstName) < 3) {
 $isValid = false;
 $errorMessage = 'invalid - name must contain at least 3 letters';
 }
}
```

---

*Listing 12-6: Validating the number of characters in `$firstName`*

We perform the validation inside an `if` statement whose body will execute only if the `$isSubmitted` Boolean flag is `true`. Therefore, we'll skip this code when the form is requested for the first time. To validate the data, we use `filter_input()` to retrieve the value of `firstName` from the submitted `POST` variables, overwriting the default, empty-string value of `$firstName` in the process ❶. Then we use `strlen()` to check whether the received string contains less than three characters ❷. If so, we change the `$isValid` Boolean flag to `false` and assign an error message to the `$errorMessage` variable.

## Displaying a Confirmation for Valid Data

At this point, we have values in the `$isSubmitted` and `$isValid` flags that we can use to decide what to present to the user. Our third block of code, shown in [Listing 12-7](#), uses these flags in a single `if` statement for the situation in which the form has been submitted with valid data.

---

```
if ($isSubmitted && $isValid) {
 print "Hello $firstName";
 die(); // End script processing here
```

---

```
}
?>
```

---

#### *Listing 12-7: Responding to valid data*

If both flags are `true`, we display a "Hello `$firstName`" confirmation message to the user. Then we terminate the script with the `die()` function.

### Displaying the Form

If the `if` statement in [Listing 12-7](#) fails, either the form is being requested for the first time (`$isSubmitted` is `false`) or the submitted data is invalid (`$isValid` is `false`). In either case, the result is the same: we need to display the form to the user. [Listing 12-8](#) shows the necessary mix of HTML and PHP code.

---

```
<!doctype html>
<html><head>
 <title>Tiny sticky postback form</title>
 ❶ <style>.error {background-color: pink; padding: 1rem;}</
 style>
</head>
<body>
 ❷ <form method="POST">
 ❸ <?php if ($isSubmitted && !$isValid): ?>
 <div class="error"><?= $errorMessage ?></div>
 <?php endif; ?>

 ❹ <input name="firstName" value="<?= $firstName ?>">
 <input type="submit">
</form>
</body></html>
```

---

#### *Listing 12-8: Displaying the form to the user*

In the HTML head, we define a CSS `error` class for any error message we need to display, with a pink background and some padding ❶. In the page's body, we declare a `<form>` element with the `POST` method ❷. Notice that we don't include an `action` attribute setting the PHP script that will

process the form; when no action is specified, the form submission request is sent by default to the same URL that displayed the form. This default is perfect for a postback form like this one.

## NOTE

*In HTML 4, a form had to specify an `action` attribute, but in HTML 5, the attribute defaults to an empty string, which results in the form submitting to the same URL as led to the form's display.*

Within the `<form>` element, we use PHP's alternative `if` statement syntax to display a `<div>` element with the content of the `$errorMessage` variable if the form is submitted but the data isn't valid ❸. We style the `<div>` with our `error` CSS class. We then display the form text input box and set its `value` attribute to the contents of the PHP `$firstName` variable ❹. If the form is being displayed for the first time, this will be the default empty string we declared in [Listing 12-5](#), but if the form is being redisplayed after an invalid input, `$firstName` will hold the user's previous submission. This mechanism is what makes the form sticky: the submitted value sticks in the text box when submitted and redisplayed, saving the user from typing values again.

## Array-Based Validation Logic

Let's now join the two key concepts from this chapter and implement our earlier product details form with its array-based approach to error messages as a postback sticky form. [Figure 12-6](#) shows an example of the error messages to be output when the submitted data is missing values or violates any validation rules.

• missing product code  
• price was not a number

## Product details

Product code:

Price:

*Figure 12-6: The sticky postback product form with errors*

Even with an array-based approach to data validation, our postback script will follow the same basic steps as before: setting default values, validating the data, displaying a confirmation for valid input, and displaying the form. [Listing 12-9](#) tackles the first step.

---

```
<?php
require 'zeroFunction.php'; // Read in our function

$isSubmitted = ($_SERVER['REQUEST_METHOD'] === 'POST');
$productCode = '';
$price = '';
$errors = [];
```

---

*Listing 12-9: Setting default values*

As before, we set the `$isSubmitted` Boolean flag to `true` or `false` depending on whether `$_SERVER['REQUEST_METHOD']` contains `'POST'`. We then set the `$productCode` and `$price` variables to empty strings to ensure that we can refer to them safely for default values in the form inputs. Finally, we set `$errors` to an empty array. We'll add to this array only if

any validation errors are found; an empty array will indicate that the data is valid.

Listing 12-10 attempts to validate the submitted form data.

---

```
if ($isSubmitted) {
 $productCode = filter_input(INPUT_POST, 'productCode');
 $price = filter_input(INPUT_POST, 'price');

 ❶ if (empty($productCode)) {
 $errors[] = 'missing product code';
 }

 ❷ elseif (strlen($productCode) < 3) {
 $errors[] = 'product code too few characters';
 }

 ❸ if (isAnEmptyNonZeroString($price)) {
 $errors[] = 'missing price';
 } elseif (!is_numeric($price)) {
 $errors[] = 'price was not a number';
 }
}
```

---

*Listing 12-10: Validating the submitted form data*

Once again, all our validation occurs inside an `if` statement executed only if the `$isSubmitted` flag is `true`. We first retrieve the `productCode` and `price` values from the submitted `POST` variables. Then we test whether `$productCode` is empty ❶ and add an error message to the `$errors` array if it is. Otherwise, we test whether `$productCode` is less than three characters long ❷, again adding an error message to the array if not. For our last validation check, we add another error message if `$price` is empty or isn't numeric ❸.

Next, [Listing 12-11](#) shows the code to confirm a valid form submission.

---

```
$isValid = empty($errors);
if ($isSubmitted && $isValid) {
```



```
 print 'input data was error free';
 die(); // End script processing here
}
?>
```

---

#### *Listing 12-11: Confirming a valid submission*

First, we test whether the `$errors` array is empty and set the `$isValid` Boolean flag accordingly. Then, if both the `$isSubmitted` and `$isValid` Boolean flags are `true`, we display a confirmation message to the user and terminate the script with the `die()` function. If either flag is `false`, we need to display the form to the user, with any errors if appropriate. [Listing 12-12](#) shows the code.

---

```
<!doctype html>
<html><head>
 <title>Two-field postback form</title>
 <style>.error {background-color: pink; padding: 1rem;}</
style>
</head>
<body>
 ❶ <?php if ($isSubmitted && !$isValid): ?>
 <div class="error">

 ❷ <?php foreach ($errors as $error): ?>
 <?= $error ?>
 <?php endforeach; ?>

 </div>
 <?php endif; ?>

 <h1>Product details</h1>
 <form method="POST">
 ❸ Product code: <input name="productCode" value="<?=
$productCode ?>">
 ❹ <p>Price: <input name="price" value="<?= $price ?>">
</p>

 <p><input type="submit"></p>
```

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

---

*Listing 12-12: Displaying the form with any error messages*

Initially, this code is similar to our earlier sticky form template from [Listing 12-8](#), up to the `if` statement checking whether the form was submitted but the data isn't valid ❶. From there, we use a PHP `foreach` loop ❷ to print each message in the `$errors` array as a separate list item inside the `<div>` styled with our `error` CSS class. Later, we make the `<form>` element sticky by prefilling the product code and price fields with the values from the `$productCode` ❸ and `$price` ❹ variables, which will be either the user's previous submissions or the default empty strings.

As you can see, the same basic structure that worked for our simpler form also worked for this more complex postback sticky form script, and the strategy of building up and displaying an array of error messages fits well into this structure. However, the complete PHP postback script is nearly 60 lines long and performs enough actions as to make the single script complex.

In the next chapter, we'll explore strategies to keep the benefits of the postback approach for form processing (such as displaying validation messages with the form and using sticky form values to save the user from retyping) while breaking up the tasks of form display, validation, error message display, and confirmation logic into simpler scripts. In the process, we'll develop a basic web application architecture that can scale up to meet the requirements of complex websites, forms, and validation rules.

## Summary

In this chapter, we covered strategies for validating submitted form data, including using an array to flexibly handle situations where zero, one, or several validation errors must be addressed. We highlighted the special care that must be taken when a 0 is submitted as a valid entry in a form. Finally, we looked at the postback technique for displaying and validating forms with a single script, and we implemented sticky forms that conveniently prefill with the user's previous entries. As a PHP programmer, you may have to understand and maintain a range of website programming styles,

and the postback approach in this chapter is a common one you'll likely come across in other programmers' code, even if it's not an approach you use often when writing your own.

## Exercises

1. Create a sticky form using the HTTP `POST` method that prompts for an integer age and redisplay the form populated with the submitted value each time.
2. Improve your answer to Exercise 1 so that an error message is displayed if a non-numeric age is entered.
3. Improve your answer to Exercise 2 so that after a valid (numeric) submission is received, a confirmation message displays, stating the user's age after their next birthday.
4. Create a sticky form using the HTTP `POST` method that prompts for an email address. If the address is valid, a confirmation message is displayed; if it's not valid, the form is redisplayed with an error message stating that the email address is invalid. Think about the requirements for a valid email address.

# 13

## ORGANIZING A WEB APPLICATION



In this chapter, we'll progressively explore a structured approach to dividing a web application's responsibilities among multiple scripts while also developing a project folder architecture that gives every script a clearly defined home. You'll be introduced to the *front-controller* design pattern, which requires the sending of every request to the web server through a single script that decides how to respond, then delegates the responsibility for generating that response to other server scripts.

[Chapters 11](#) and [12](#) introduced differing approaches to designing a web application. One is to write multiple scripts for each web form that separately handle displaying, validating, and confirming the form; another is to create a single postback script that does all those tasks. Both approaches have advantages and disadvantages, but neither is scalable for web applications that may grow in size and complexity over time. You wouldn't want to have tens, hundreds, or even thousands of PHP scripts

sitting in your application's *public* folder, as might happen with the first approach, for reasons of maintenance as well as security (malicious users may attempt to execute public scripts out of sequence to bypass validation checks, for example). But neither would you want an individual script to become overly complex, as might happen with the second approach.

The architecture we'll discuss in this chapter addresses these problems by focusing all decision-making logic on the single front-controller script while using other scripts or functions to carry out all other tasks. The front controller acts much like the receptionist in a large office building, whom every guest must visit; it identifies and interprets each request (who the visitor is and what they're asking for), and directs it to the appropriate server script (a room in the building) that satisfies the request. With this architecture, no one script is overly complicated, and while an application can grow to include many scripts, they're organized in such a way that the application is easy to maintain and extend.

## Front Controllers and the MVC Architecture

[Chapter 10](#) introduced the model-view-controller (MVC) architecture, a design pattern that divides a web application into three main components. The *model* represents the data underlying the application, typically stored in a database; the *view* uses templating to determine how the data is displayed to the user; and the *controller* makes decisions about what data to display when. The front-controller scripts we'll explore in this chapter are the core piece of the controller portion of the MVC architecture, as they take in every HTTP request to the application and determine how to respond.

[Figure 13-1](#) illustrates how a front controller fits into the MVC pattern.

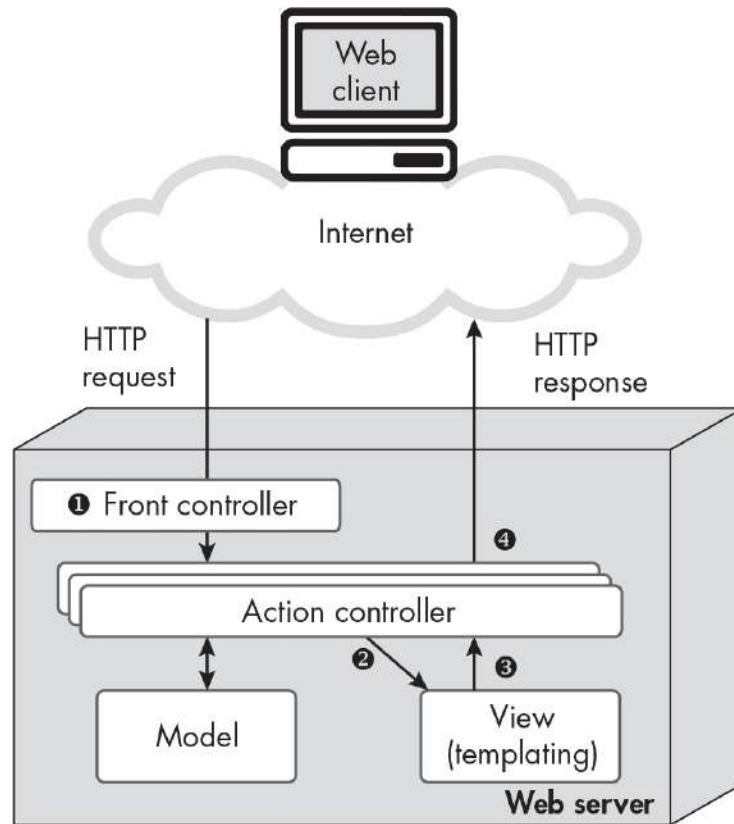


Figure 13-1: The MVC architecture, featuring a front controller

The *front controller* ❶ handles every HTTP request received from web clients, deciding which *action controller* should be invoked. These action controllers are more specialized scripts or functions, also part of the controller component of the MVC architecture, that carry out particular tasks for the application, such as checking stored security credentials or manipulating data like shopping-cart contents. An action controller may need to read the contents of a data store, such as a database system (more about this in [Part VI](#)). The action controller then invokes the view (templating) component ❷ to create the contents of the response to be returned to the user, passing along the data from the model if appropriate. The action controller then receives the output from the view component ❸ and adds any appropriate headers or response codes. Finally, the response is sent back to the web client ❹.

## Separating Display and Logic Files

We'll revisit the simple *Hello <name>* form developed in the preceding two chapters to start the process of separating the display code from the decision-making logic in an application. See “A Simple Example” on [page 199](#) to review how we initially created and processed that form with separate scripts, and “Simple Validation Logic” on [page 231](#) for how we validated and displayed the form with a single postback script. In this version, the application will have three possible responses it may return:

- The blank form if the HTTP request used the `GET` method
- A *Hello <name>* confirmation message if a valid name was received
- An error message if an invalid name (less than three characters long) was received

The decision-making logic and possible outputs are modeled in the flowchart in [Figure 13-2](#).

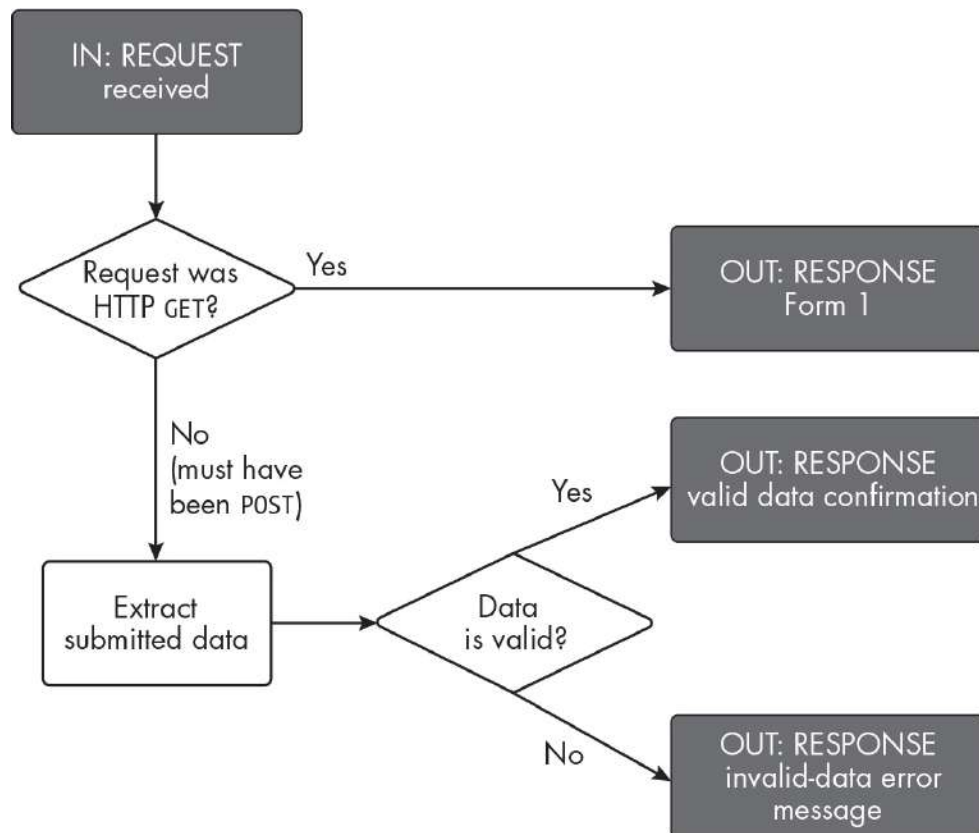


Figure 13-2: Modeling the logic and outputs of the *Hello <name>* application

We'll create a separate file for each of the three possible outputs. Meanwhile, every request will be routed to the front controller in the *index.php* file. The logic in the front controller will decide what to do (that is, which of the other three files to draw upon to generate the output to be returned to the user). The file structure for the project will therefore look as follows when we've finished:

```
/public
├── index.php
├── displayError.php
├── displayForm.php
└── displayHello.php
```

Notice that all the files are located in the project's *public* folder.

## Creating the Front Controller

We'll first write the *index.php* front-controller script. All requests for this project will be for *index.php*, some with the `GET` method, some with the `POST` method, some containing valid submitted data, and some containing invalid submitted data. Depending on the type of HTTP method (`GET` or `POST`) and whether the submitted data is valid, this script decides which of the three display pages to return as the response. [Listing 13-1](#) shows the code.

---

```
<?php
$isGetMethod = ($_SERVER['REQUEST_METHOD'] === 'GET');

if ($isGetMethod) {
 ❶ require_once 'displayForm.php';
} else {
 $firstName = filter_input(INPUT_POST, 'firstName');

 if (strlen($firstName) < 3) {
 $errorMessage = 'invalid - name must contain at least
t letters';
 ❷ require_once 'displayError.php';
 } else {
 ❸ require_once 'displayHello.php';
 }
}
```



```
}
}
```

---

*Listing 13-1: The front-controller script in index.php deciding which page to display*

First, we test whether the request received was via the `GET` method and set the `$isGetMethod` Boolean flag accordingly. Then we pass this flag to an `if...else` statement. If `true`, we display the form by requiring the file *displayForm.php* ❶. Otherwise, we must be processing a `POST` request containing the form submission, so we use the logic in the `else` branch (the rest of the script) to read and validate the data. For that, we use `filter_input()` to extract a `$firstName` value from the `POST` variables. Then we test the validity of the received name. If it's too short, we set `$errorMessage` to a suitable message, then read in the *displayError.php* file to show the error page ❷. If the name is valid, we read in the *displayHello.php* file to show the *Hello <name>* confirmation message ❸.

## **Writing the Display Scripts**

Next, we'll create the three display scripts, starting with the web form itself. Enter the contents of [Listing 13-2](#) into a file named *displayForm.php* in the *public* folder.

---

```
<!doctype html><html><head><title>Tiny Form</title></head>
<body>
 <form method="POST">
 <input name="firstName">
 <input type="submit">
 </form>
</body></html>
```

---

*Listing 13-2: The displayForm.php script to display the form to the user*

This script consists entirely of HTML template text, namely a `POST`-method form with a `name` input and a Submit button. Because we don't give the form an `action` attribute, the form will be submitted to the same URL as the request that displayed the form. In this case, that's our *index.php* front-controller script.

Next, [Listing 13-3](#) creates the page confirming a valid submission. Copy the listing into a file called *displayHello.php*.

---

```
<!doctype html><html>
<head><title>hello</title></head><body>
 Hello <?= $firstName ?>
</body></html>
```

---

*Listing 13-3: The displayHello.php script to confirm that a valid name has been received*

Within the HTML template text, we use the PHP short echo tag to output the value of the `$firstName` variable. Notice that this variable hasn't been declared in *displayHello.php* itself, but that's okay. This file isn't written as a stand-alone script but rather is intended to be required from another script: *index.php*. As long as we've set the `$firstName` variable in *index.php* before reading in this script (which we have), the variable will be accessible within *displayHello.php* as well.

Finally, we'll write the third display page that outputs an error message. Create *displayError.php* and enter the contents of [Listing 13-4](#).

---

```
<!doctype html><html><head>
 <title>error</title>
 <style>.error {background-color: pink; padding: 1rem;}</
style>
</head><body>
<div class="error">
 <p>Sorry, there was a problem with your form submission
</p>
 <p><?= $errorMessage ?></p>
</div>
</body></html>
```

---

*Listing 13-4: The displayError.php script to show an error message to the user*

We again use a PHP short echo tag to output the value of a variable (in this case, `$errorMessage`). Once again, we need to ensure that this variable has been assigned a value in *index.php* before the front-controller script references the *displayError.php* display script.

We now have four PHP scripts in our project. This may seem like a lot, but each script has the advantage of having a single core responsibility. We have three scripts to display the application's three possible pages, along with an *index.php* front-controller script that pulls together the main program logic.

## Moving Website Logic into Functions

We can further organize our project by taking each task or decision in the *index.php* front controller and encapsulating it in its own function. While the current level of complexity of *index.php* is suitable for this small project, multipage websites with multiple forms would become much too complex for a single script to handle. We'll create a PHP file named *controllerFunctions.php* to store the functions, thereby streamlining the front-controller script.

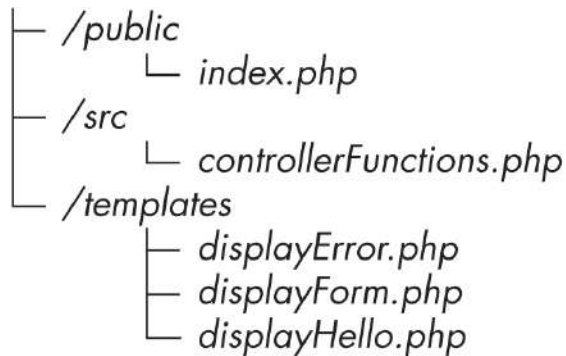
Also, at present all the files are in the project's *public* folder, meaning any of them can be directly requested and served to a user. For example, you could enter *localhost:8000/displayHello.php* in your browser's address bar and the display script would be executed without the `$firstName` variable first having been assigned a value, since the form-processing logic in *index.php* wouldn't have been executed. This would trigger a notice-level error flagging the undefined variable.

This file structure would be messy and unmanageable for a larger project, with potentially hundreds of directly accessible files in the *public* folder. In addition, we're exposing PHP scripts that the user shouldn't ever be able to directly request. The solution, as first discussed in [Chapter 10](#), is to use folders to separate public scripts (generally the *index.php* front controller) from those that shouldn't be directly accessible to the public. We'll implement this change as well.

## Designing a Secure Folder Structure

To make the application more secure, move the three output scripts (*displayHello.php*, *displayError.php*, and *displayForm.php*) from *public* to a new folder named *templates*. Likewise, we don't want users to be able to directly request our new *controllerFunctions.php* file. This file will contain only function declarations, so there's no reason for it to be published in the

*public* folder. Instead, we'll put *controllerFunctions.php* in its own folder, which by convention we'll name *src* (short for *source code*). The file structure for our refactored project will therefore look as follows:



Using separate folders like this not only makes the application more secure but also keeps it better organized, with the scripts grouped into folders based on their purpose.

## ***Simplifying the Front-Controller Script***

Now let's simplify our front-controller script by replacing some of the logic with function calls. The main task the front controller needs to do is check whether the received request used the `GET` method (in which case the form should be displayed) or the `POST` method (in which case the form should be processed).

We'll update the script to focus just on this decision and leave the remaining details to separate functions (which we'll write next). [Listing 13-5](#) shows how to modify the *index.php* file.

---

```
<?php
require_once '../src/controllerFunctions.php';

$isGetMethod = ($_SERVER['REQUEST_METHOD'] === 'GET');
if ($isGetMethod) {
 displayForm();
} else {
 processForm();
}
```

---

*Listing 13-5: The simplified front-controller script in index.php*

We read in the definitions for all the functions we'll be using from *src/controllerFunctions.php*. The two dots (..) at the start of the filepath signify the parent directory relative to the current script's directory (that is, the overall project folder that contains both *public* and *src*). As before, we next set the `$isGetMethod` Boolean flag and use it to control an `if...else` statement. This time, however, the `if...else` statement simply invokes the `displayForm()` function if the server receives a GET request or invokes the `processForm()` function if it is a POST request.

## Writing the Functions

All that remains is to declare the functions handling the more granular front-controller logic in the new *src/controllerFunctions.php* file. First, let's write the `displayForm()` function, as in [Listing 13-6](#).

---

```
<?php
function displayForm(): void
{
 require_once '../templates/displayForm.php';
}
```

---

*Listing 13-6: A function to display the form*

This function simply reads in and executes the contents of the *templates/displayForm.php* file by using `require_once`.

Next, let's write the `processForm()` function that will extract the name from the POST variables and decide what to do depending on whether it's valid, as in [Listing 13-7](#).

---

```
function processForm(): void
{
 $firstName = filter_input(INPUT_POST, 'firstName');

 if (strlen($firstName) < 3) {
 displayErrorMessage();
 } else {
 displayHello($firstName);
 }
}
```

---

```
}
}
```

---

*Listing 13-7: A function to process the form*

This function reads the `$firstName` variable from the incoming request and tests whether it's less than three characters long. We invoke another custom function depending on the result: either `displayErrorMessage()` or `displayHello()`. Notice that the latter takes the `$firstName` variable as an argument; you'll learn more about this shortly. We'll define these two functions in [Listing 13-8](#).

---

```
function displayHello($firstName) : void
{
 require_once '../templates/displayHello.php';
}

function displayErrorMessage(): void
{
 $errorMessage = 'invalid - name must contain at least 3
letters';
 require_once '../templates/displayError.php';
}
```

---

*Listing 13-8: Functions to display the Hello <name> greeting or an error message*

The `displayHello()` function receives the valid first-name string, then reads in and displays the *displayHello.php* page. The `displayErrorMessage()` function assigns an error message string to the `$errorMessage` variable, then reads in and displays the *displayError.php* page.

The `displayHello()` function has a `$firstName` parameter because of the scoping of variables in PHP. When a function reads in and executes a PHP script file by using a `require_once` statement, any variables expected by that script must be either parameters received by the function or variables created within the function. In this case, since the *displayHello.php* script needs access to `$firstName`, a value for that variable must be passed to the `displayHello()` function. If `$firstName`

weren't a function parameter, no variable of that name would be available to output in the *displayHello.php* script, and we would see the error displayed in [Figure 13-2](#).

Overall, the functionality of our application is just the same as before, but it has a much better architecture that's easier to maintain, more secure, and more scalable in the event the project requirements expand in the future. Our only public script is the front controller in *public/index.php*, and every HTTP request to the server goes through this script. The front controller determines whether the user wants the form displayed (via a `GET` request) or has submitted the form for processing (a `POST` request). The remaining controller logic is encapsulated in a collection of functions declared in the *src/controllerFunctions.php* file. Meanwhile, the three responses that can be returned to the client are defined in scripts in the *templates* folder: *displayForm.php*, *displayHello.php*, and *displayError.php*.

## Generalizing the Front-Controller Structure

You've seen how much simpler scripts and functions become once you start breaking an application into individual responsibilities. Let's now test the architecture we've developed on a new, more complex website project with several pages. You'll see how a front-controller script can cope with several `GET` requests as well as a `POST` form submission and its validation logic.

[Figure 13-3](#) shows the three-page site we'll create, including a home page, a contact details page, and an inquiry form. The *MGW* name is a placeholder, standing for *My Great Website*.

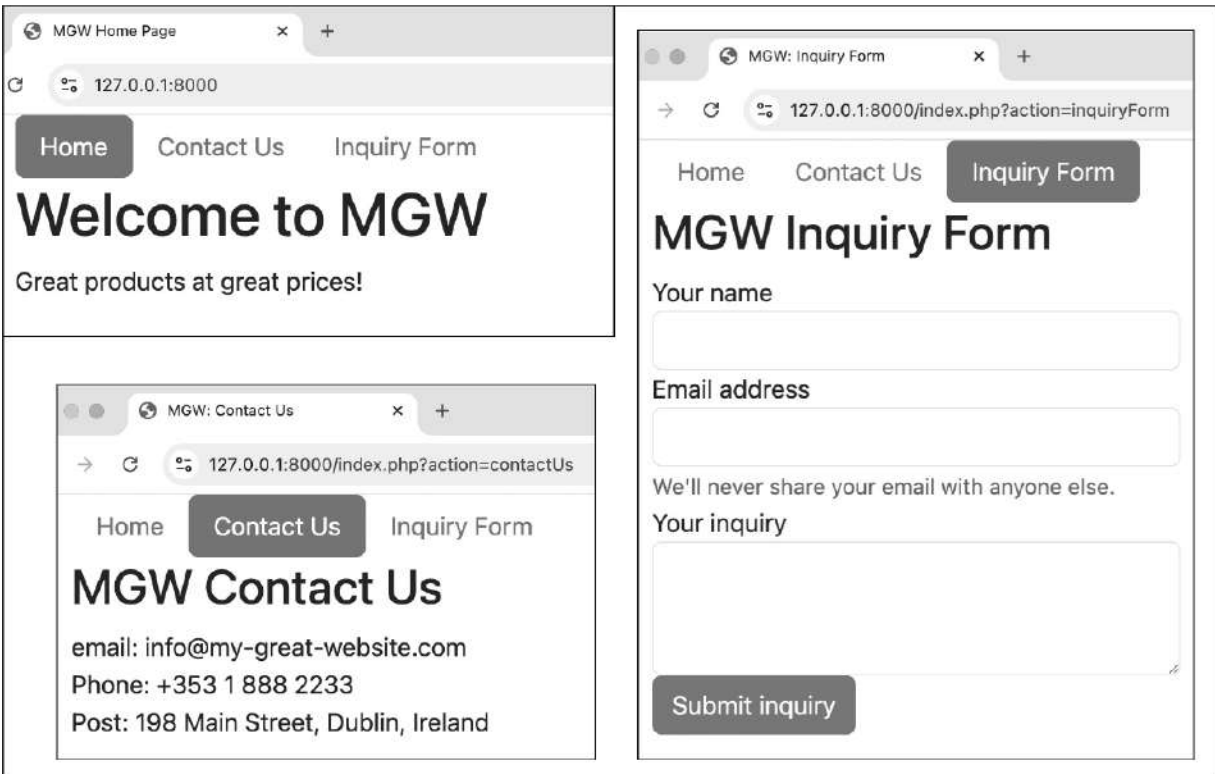


Figure 13-3: A three-page web application displayed in the browser

Each page has a navigation bar at the top with the current page highlighted, followed by a level 1 heading and some page content. To give the pages the nice styling shown in the figure, we'll link to the prewritten Bootstrap CSS stylesheet, as we did in [Chapter 11](#).

## ***Distinguishing Between Requested Pages***

Each of the three pages in our application will be displayed in response to a `GET` request, so we need a way to determine which page is being requested. This is a bit different from our *Hello <name>* application, where we knew a `GET` request could only be for displaying the web form. One simple and common solution is to add an `action` query-string variable to every request, regardless of whether the `GET` or `POST` method is used, and give the variable a value indicating the desired action or page. The request URLs will therefore look something like *index.php?action=contactUs* to display the contact details page and *index.php?action=inquiryForm* to display the inquiry form page.



As we build our new site, we'll be sure to present links to the user that will pass such an `action` variable to the server with each `GET` request. We'll also set an `action` query-string variable to `processForm` when the inquiry form is being submitted with the `POST` method. All the requests will go to the *index.php* front-controller script, which simply has to detect the value of the `action` variable to determine what to do. If no value for `action` is received, or if the value isn't recognized, the home page will be displayed by default.

Notice that the different values for the `action` query-string variable can also be seen in the URL address bars in [Figure 13-3](#). The home page, which is also the website root, is simply *localhost:8000*; the Contact Us page ends with *index.php?action=contact*; and the Enquiry Form page ends with *index.php?action=enquiryForm*.

## ***Building a Multipage Application***

We'll now build a three-page web application like the one illustrated in [Figure 13-3](#). Create a new folder for this project, and create the *public*, *src*, and *templates* subfolders inside it. We'll follow the same directory structure used for the simpler application earlier in the chapter. We'll start by writing the application's front-controller script.

### **Creating the Front Controller**

We'll structure our *public/index.php* script around a straightforward PHP `switch` statement (see [Chapter 4](#)) based on the value of the `action` query-string variable. [Listing 13-9](#) shows the script. It includes calls to some functions we'll write in a moment.

---

```
<?php
require_once '../src/controllerFunctions.php';

$action = filter_input(INPUT_GET, 'action');

switch ($action) {
 case 'contactUs':
 displayContactUs();
 break;
```

```
 case 'inquiryForm':
 displayInquiryForm();
 break;

 case 'processForm':
 processForm();
 break;

 default:
 displayHomePage();
 }
}
```

---

*Listing 13-9: The front-controller logic in index.php for a three-page website*

The front controller reads in the function declarations in *controllerFunctions.php*, then extracts the value of the `action` query-string variable and passes it to a `switch` statement. The statement has cases for displaying each of the pages (including the default case of displaying the home page), as well as a case for processing the submitted inquiry form. Each case simply calls a custom function that will handle the actual work of carrying out the desired action, such as `displayContactDetails()` to show the contact details page.

Designing the front controller around a `switch` statement makes it quite simple to expand the application if necessary. To add another page to the website, all we'd have to do is insert another case into the `switch` statement, write a corresponding function in *controllerFunctions.php*, and add a template for displaying the page to the *templates* folder. For more complex websites, we could break things up even further by having different collections of functions for different types of website actions. For example, manager actions could be in a file named *managerFunctions.php*, and so on.

## Writing the Display Functions

Now we'll write simple functions to display the three basic pages. Create *src/controllerFunctions.php* containing the code in [Listing 13-10](#).

---

```
<?php
function displayHomePage(): void
```

```

{
 require_once '../templates/homePage.php';
}

function displayContactUs(): void
{
 require_once '../templates/contactDetails.php';
}

function displayInquiryForm(): void
{
 require_once '../templates/inquiryForm.php';
}

```

---

*Listing 13-10: The functions to display the three basic pages*

Each of the functions uses a `require_once` statement to read in and display the contents of the corresponding template file. (We'll make those template files next.) For example, the `displayHomePage()` function reads in and displays the *templates/homepage.php* script.

## Creating the Page Templates

We'll now create the templates for the application's various pages, starting with the project's home page in *templates/homePage.php*. Create the file as shown in [Listing 13-11](#).

---

```

<!doctype html><html><head><title>MGW Home Page</title>
<link rel="stylesheet"
 href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
>
</head><body class="container">
<ul class="nav nav-pills"> ❶
 <li class="nav-item"><a class="nav-link active" href
= "/">Home ❷
 <li class="nav-item"><a class="nav-link"
 href="/index.php?action=contact">Contact Us
 <li class="nav-item"><a class="nav-link"
 href="/index.php?action=inquiryForm">Inquiry Form

```

```
>

<h1>Welcome to MGW</h1>
<p>Great products at great prices!</p>
</body></html>
```

---

**Listing 13-11: The homePage.php template**

After linking to the Bootstrap stylesheet, we create a navigation bar at the top of the page by using an unordered list with the Bootstrap `nav` and `nav-pills` styles ❶. Each list item, styled with the `nav-item` class, contains a link to one of the three pages. Each link is styled with the `nav-link` class. The link for the currently displayed page (home) is also styled with the `active` class so it will be highlighted ❷. Notice that we include an `action` query-string variable in the hyperlinks for the Contact Us and Inquiry Form pages. The hyperlink for the home page is simply a forward slash (/). We could have used `/index.php` instead, but since the `index.php` script loads by default, this isn't necessary.

We can now copy our home-page script and alter its code to create the contact details page in `templates/contactUs.php`, as shown in [Listing 13-12](#).

---

```
<!doctype html><html><head><title>MGW: Contact Us</title>
<link rel="stylesheet"
 href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
>
</head><body class="container">
<ul class="nav nav-pills">
 <li class="nav-item">Home
 <li class="nav-item"><a class="nav-link active"
 href="/index.php?action=contact">Contact Us
 <li class="nav-item"><a class="nav-link"
 href="/index.php?action=inquiryForm">Inquiry Form

<h1>MGW Contact Details</h1>
```

```
<p>email: info@my-great-website.com

 Phone: +353 1 888 2233

 Post: 198 Main Street, Dublin, Ireland
</p>
</body></html>
```

---

*Listing 13-12: The contactUs.php template*

We've kept the boilerplate HTML at the start of the file the same, except for changing the page title and styling the Contact Us page link in the navigation bar as `active` instead of the home-page link. Then we've filled in some contact information for the body of the page.

Next, we'll write the code for the inquiry form. Once again, start by copying one of your other templates into a new *templates/inquiryForm.php* file. Then change the page title and set the Inquiry Form link in the navigation bar to `active`. Use the HTML code in [Listing 13-13](#) to create the inquiry form itself.

---

```
--snip--
<ul class="nav nav-pills">
 <li class="nav-item">Home
 <li class="nav-item"><a class="nav-link"
 href="/index.php?action=contact">Contact Us
 <li class="nav-item"><a class="nav-link active" ❶
 href="/index.php?action=inquiryForm">Inquiry Form

<h1>MGW Sales Inquiry</h1>
<form method="POST" action="/index.php?action=processForm">
❷
<div class="form-group">
 <label for="customerName">Your name</label>
 <input class="form-control" name="customerName" id="customerName">
</div>
```

```

<div class="form-group">
 <label for="customerEmail">Email address</label>
 <input type="email" class="form-control" name="customerE
mail"
 id="customerEmail">
 <small class="form-text text-muted">
 We'll never share your email with anyone else.
 </small>
</div>

<div class="form-group">
 <label for="inquiry">Your inquiry</label>
 <textarea class="form-control" name="inquiry" id="inquir
y" rows="3"></textarea>
</div>

<input type="submit" value="Submit inquiry" class="btn btn-s
uccess">
</form>

```

---

*Listing 13-13: The inquiryForm.php template*

We make the Submit an Inquiry link active in the navigation bar ❶. We declare the form with the `POST` method and define its action as `/index.php?action=processForm` ❷, so when the form is submitted, it will pass an action query-string variable with the value `processForm`. The rest of the HTML adds form inputs for a customer name, an email address, and the text of an inquiry (as a `textarea` input), plus a Submit button, all decorated with more Bootstrap CSS classes.

## Confirming Receipt of Valid Form Data

If the user enters valid data (we'll just test for non-empty form fields for this project), we'll display a confirmation page stating that the inquiry has been received and echoing the submitted values back to the user. One simple way to do this is to present a similar form prefilled with the submitted values, with each field set to `disabled` so the user can't edit the information and attempt to resubmit it.

[Figure 13-4](#) shows an inquiry form complete with valid values (left) and the corresponding read-only confirmation form page (right). Users will understand that the page is read-only because the boxes are grayed out, there isn't a Submit button, and nothing happens when they click the mouse cursor in the fields.

Figure 13-4: Comparing the form submission and confirmation pages

Enter the content in [Listing 13-14](#) into a new template file called *templates/confirmData.php* to create the valid form confirmation. Much of the code can be copied from the inquiry form script *inquiryForm.php*. You'll also need to add our standard HTML head and navigation bar at the beginning of the file.

---

```
--snip--
<h1>Inquiry received</h1>
<p>Thank you for your inquiry - we aim to respond within 2 w
orking days.</p>
<p>Data received:

<div class="form-group">
 <label for="customerName">Your name</label>
 <input class="form-control" name="customerName" value="
```

```

<?= $customerName ?>"
 id="customerName" disabled> ❶
</div>

<div class="form-group">
 <label for="customerEmail">Email address</label>
 <input type="email" class="form-control" name="customerEmail"
 id="customerEmail" value="<?= $customerEmail ?>" disabled >
 <small class="form-text text-muted">
 We'll never share your email with anyone else.
 </small>
</div>

<div class="form-group">
 <label for="inquiry">Your inquiry</label>
 <textarea class="form-control" name="inquiry" rows="3" id="inquiry"
 disabled> <?= $inquiry ?></textarea>
</div>

```

---

*Listing 13-14: The confirmData.php template*

We use the PHP short echo tag to insert the value of the `$customerName` variable into the `customerName` input field, setting the field to `disabled` so it can't be edited ❶. We use the same mechanism to show the received `$customerEmail` and `$inquiry` values. We don't need to add a `<form>` element or a Submit button to this template page; we're just echoing the data back to the user, so this isn't an interactive form.

## Processing the Submitted Data

Let's now write the `processForm()` function to receive the form data submitted with the `POST` request and decide what to do depending on whether the data is valid. Add the function shown in [Listing 13-15](#) to the end of the `src/controllerFunctions.php` file (although it doesn't actually matter in what order functions are declared in files like this).



---

```

--snip--
function processForm(): void
{
 ❶ $customerName = filter_input(INPUT_POST, 'customerName',
 FILTER_SANITIZE_SPECIAL_CHA
RS);
 $customerEmail = filter_input(INPUT_POST, 'customerEmail',
 FILTER_SANITIZE_EMAIL);
 $inquiry = filter_input(INPUT_POST, 'inquiry',
 FILTER_SANITIZE_SPECIAL_CHARS);

 ❷ $errors = [];
 if (empty($customerName)) {
 $errors[] = 'missing customer name';
 }

 if (empty($customerEmail)) {
 $errors[] = 'missing or invalid customer email';
 }

 if (empty($inquiry)) {
 $errors[] = 'missing inquiry message';
 }

 ❸ if (sizeof($errors) > 0) {
 require_once '../templates/displayError.php';
 } else {
 confirmData($customerName, $customerEmail, $inquir
y);
 }
}

```

---

*Listing 13-15: A function to process the form data*

First, we use three `filter_input()` calls ❶ to attempt to extract the received `POST` form data for the three expected values. Notice that for the two strings (name and inquiry), we use a `FILTER_SANITIZE_SPECIAL_CHAR` filter as the third function argument, and we use `FILTER_SANITIZE_EMAIL`

for the customer email input. These filters remove any unsafe characters submitted through the form, protecting against cross-site scripting (XSS) attacks, where JavaScript code is submitted through forms to attempt to gain entry to website data.

We next implement the array-based approach to data validation discussed in [Chapter 12](#). We declare `$errors` as an empty array ❷, then use three `if` statements to test whether any of the received form variables are empty, adding an appropriate error message to the array if necessary. Finally, we test the size of the `$errors` array ❸, and if any errors have been found, we display the file *templates/displayError.php* (which we'll create next). Otherwise, if the submitted data is all valid, we invoke the `confirmData()` function, passing `$customerName`, `$customerEmail`, and `$inquiry` as arguments. [Listing 13-16](#) shows the code for this function. Enter the code at the end of *src/controllerFunctions.php*.

---

```
function confirmData($customerName, $customerEmail, $inquiry): void
{
 require_once '../templates/confirmData.php';
}
```

---

*Listing 13-16: A function confirming valid data to the user*

This function reads in and executes the *templates/confirmData.php* file. As before, the function must take in the variables needed within the template script as parameters.

## Creating the Error Page

As a final step, we need to create the template file for a page displaying any error messages to the user. [Figure 13-5](#) shows this error page when all three validation checks have failed.

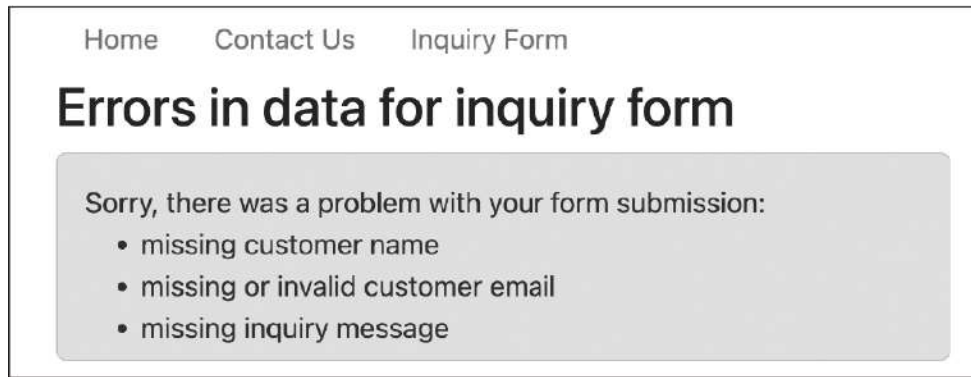


Figure 13-5: The error page

Create a new file called *templates/displayError.php* containing the error output script shown in [Listing 13-17](#). Once again, you'll have to also add the standard HTML head and navigation bar code to the start of the script.

---

```
--snip--
<h1>Errors in data for inquiry form</h1>
❶ <div class="alert alert-danger" role="alert">
 Sorry, there was a problem with your form submission:

 ❷ <?php foreach ($errors as $error): ?>
 ❸ <?= $error ?>
 <?php endforeach; ?>

</div>
```

---

Listing 13-17: A displayError.php template

We set the CSS style of the page's main `<div>` to the Bootstrap classes "alert alert-danger" ❶, a typical website error style with a pink background. Then we use a `foreach` loop with alternative syntax to cycle through each message in the `$errors` array ❷ and display it as a list item ❸.

## Summary

In this chapter, we've worked toward a well-organized and extensible web application architecture, which we've applied to a multipage website featuring both static pages and a web form. The architecture hinges on a single *public/index.php* front-controller script, which at its core is a straightforward `switch` statement deciding what to do based on the `action` query-string variable sent with each incoming HTTP request. We've delegated other, more granular logic to custom functions, which we've safely declared outside the *public* folder in *src/controllerFunctions.php*. Meanwhile, all the scripts that output HTML content to the user are located in the *templates* folder. These scripts harness Bootstrap CSS to achieve professional-looking formatting with minimal code.

Although this project has many more pages and features than the previous form-processing projects from the last few chapters, none of our scripts or functions are complex or long-winded, demonstrating that the architecture we've adopted is suitable for more sophisticated websites, while still being readily understandable and easy to maintain.

## Exercises

1. Create a website containing four simple page templates (no forms): a home page (*home.php*), an about page (*about.php*), a contact details page (*contact.php*), and a customer recommendations page (*recommendations.php*). These pages should be accessed from the index page through the value of the `action` query-string parameter, like this:

---

```
index.php?action=home
index.php?action=about
index.php?action=contact
index.php?action=recommendations
```

---

Structure your project by using the front-controller architecture demonstrated in this chapter and the functions declared in *src/controllerFunctions.php*. You should have the following file structure:



2. Using the front-controller architecture from this chapter, create a new website with a home-page template (*home.php*), containing a link to a staff login page. The staff login page template (*loginStaff.php*) should display a standard username/password login form. The values for a successful staff login are the username `author` and the password `words`. If correct values are entered, display a staff login success page (*successStaff.php*), but if the received username and password don't match the correct ones, display a login error page (*loginError.php*). The error page should contain a link back to the home page.

The website pages should be accessed from the index page through the value of the `action` query-string parameter, like this:

---

```
index.php?action=home
index.php?action=loginStaff
```

---

3. Copy and extend your solution for Exercise 2, adding a second login page for clients (*loginClient.php*), with a link to this page also offered on the home page. Values for the client login are the username `customer` and the password `paying`. If correct client login values are received, display a client login success page (*successClient.php*), but if the values are invalid, display the login error page (*loginError.php*).

The website pages should be accessed from the index page through the value of the `action` query-string parameter, like this:

---

```
index.php?action=home
index.php?action=loginStaff
```

---

`index.php?action=loginClient`

---

4. Copy and extend your solution for Exercise 3, placing the staff and client login forms into the home page. The home page should contain welcome text and two login forms, one for staff and one for clients.
5. Copy and extend your solution for Exercise 4, now offering a single login form on the home page, but two Submit buttons: one for staff and one for clients.

Hint: See [Chapter 11](#) to review how to detect which of the two Submit buttons has been clicked.

# **PART IV**

## **STORING USER DATA WITH BROWSER SESSIONS**

# 14

## WORKING WITH SESSIONS



This chapter introduces browser sessions, which allow a web client and server to remember user information over time. When browsing an online store, for example, you expect to be able to add items into a shopping cart and for those items to be remembered a few minutes later, or even across browser tabs. Similarly, if you enter a username and password to access a web-based email system, you expect your successful login to be remembered as you click through pages to display email, draft messages, and so on. Sessions make this kind of memory possible.

This chapter discusses how to work with browser sessions in PHP, including storing and retrieving values, and resetting or destroying sessions entirely. We'll develop a general pattern for writing code that uses sessions, which will be applicable to most situations, such as shopping carts and login authentication.



## A Web Browser Session

A *browser session* is a temporary information exchange between a web client, such as a browser or phone app, and a web server. A session begins at a certain point in time and will terminate at a later point in time. Sessions often begin when a user directs their web browser to a new website; the browser and server agree on a unique session ID, and this ID will be used in the subsequent HTTP requests and responses exchanged between the client and server to indicate that they are all part of the same session. [Figure 14-1](#) illustrates a web client making repeated requests by continuing to use the session ID created after its first request.

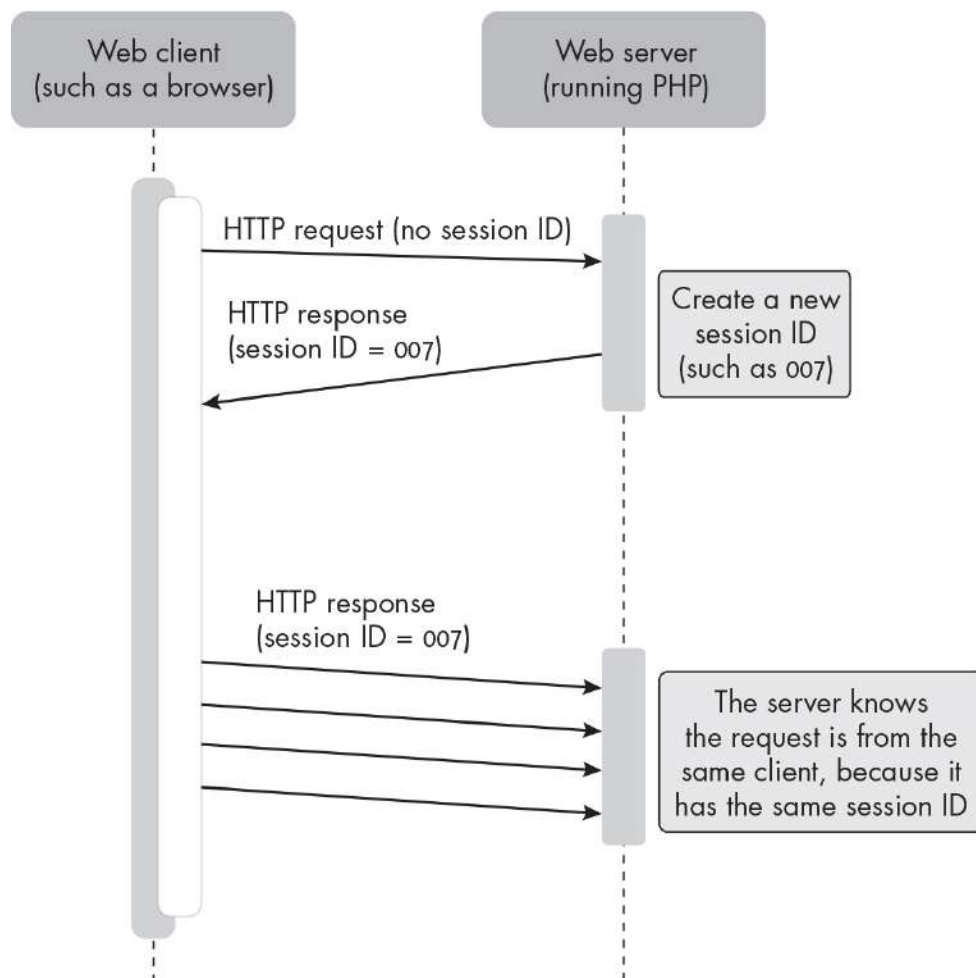


Figure 14-1: Repeated requests from a web client, each including the same session ID

You can find the session IDs behind real-world web interactions by using your browser's developer tools to examine HTTP requests. For example, [Figure 14-2](#) shows the Amazon UK website agreeing on a session ID with my web browser.

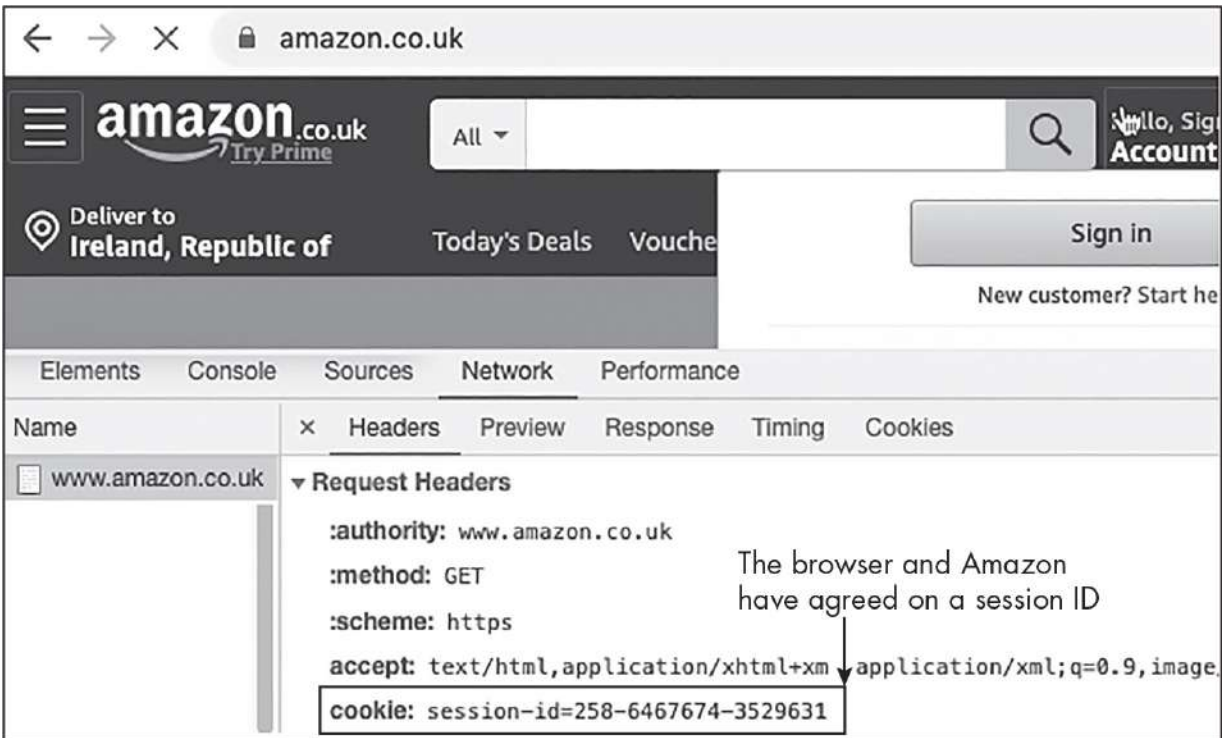


Figure 14-2: The Chrome browser tools showing a session ID from Amazon

Both the server and the client need to keep a record of the agreed-upon session ID, since this unique token must be included in each HTTP request. This way, when the server receives an incoming request, it can immediately tell which session it belongs to out of the potentially thousands of sessions the server might be tracking. The server also uses the session ID to store and manage the data for each session, such as shopping carts, successful logins, and so on. PHP web servers automatically create these session IDs, and the PHP language provides several functions for working with sessions. Web clients usually use an HTTP cookie to temporarily store session IDs.

Sessions can be ended in several ways, depending on the web server settings, the PHP code, and sometimes additional JavaScript code running on the web clients. Sessions are closed when the user quits the browser

application. Sessions can also be terminated by PHP server code, such as when a user chooses to log out of their account. Some websites have JavaScript running in the web browser to detect when the user closes or navigates away from the website browsing tab, at which point the JavaScript sends a message to the server requesting that the session be ended.

Sessions might also time out; the server can set a time limit that starts with the latest client request so that if no new request is received within the designated time period, the server will automatically terminate the session. Time-outs help keep sites secure: if a user walks away from their computer, the session can time out and prevent a nonauthorized person from continuing the authorized session. (Even with time-outs, though, logging out or quitting the browser *before* walking away from your computer is always a good idea.)

#### LOCALLY STORED SESSION COOKIES

An *HTTP cookie*, often just called a *cookie*, is a small piece of data stored on the system that runs the client (for example, a laptop or mobile phone running a web browser). Cookies can store temporary data such as the session ID, shopping cart items, open pages or tabs, and authentication status (whether a user is logged in).

Cookies can also be used to store data for longer periods so certain information, such as usernames, can be automatically prefilled into websites that are visited frequently. Currently, almost all websites use cookies to track users' behavior across multiple sites and for long periods of time in order to deliver targeted advertising. This has sparked concerns about privacy that are being addressed through various laws around the world (especially in Europe), often requiring users to be informed of and to give consent to the use of tracking cookies.

## The `session_start()` and `session_id()` Functions

PHP provides the `session_start()` function, which starts a new session if none currently exists, or renews an existing session if a valid session ID is included in the received HTTP request. When renewing an existing session, the function restarts the time-out timer. Although you rarely need to know the unique session ID when writing PHP scripts, the language does provide a function to retrieve it: `session_id()`.

Listing 14-1 shows a two-statement PHP script that first calls `session_start()` and then prints the value returned by `session_id()`.

---

```
<?php
session_start();
print session_id();
```

---

*Listing 14-1: Starting (or restarting) a session and printing its ID*

If you run this script, the output will be a long string of letters and numbers similar to `d98rqmn9amvtf3cqbpifv95bdd`. This is the unique session ID generated by `session_start()` and retrieved by `session_id()`.

## NOTE

*An alternative to using the `session_start()` function in your PHP code is to enable automatic session starting through a configuration setting for the PHP engine (`session.auto_start = 1` in the `php.ini` file) or for the web server (`php_value session.auto_start 1` in `.htaccess` for Apache web servers). However, when learning to use sessions or if your web hosting makes configuration changes difficult, the best approach is to use the `session_start()` function, as illustrated throughout this chapter.*

## The `$_SESSION` Superglobal Array

You don't usually need to reference a specific session ID in your PHP code to work with sessions. Instead, you primarily work with session data through the built-in `$_SESSION` array. This is another of PHP's *superglobals*, like `$_GET` and `$_POST` that we met in [Chapter 11](#).

The `$_SESSION` array holds data related to the current session using string keys. This array is automatically provided by the PHP engine when an HTTP request with a session ID is received from a client. It's there so PHP web programmers have a variable for storing any values that need to be remembered for the current client's session from one request to another.

One way to understand this is to consider that a typical web server might be maintaining tens, hundreds, or thousands of `$_SESSION` arrays, one for each session with each of the clients currently communicating with the

server. (Think of the thousands of people using eBay or Amazon at any given time.) When the server executes a PHP script for a particular client request that has been received (containing a unique session ID), the PHP engine retrieves data stored on the server associated with that session ID and puts it in the `$_SESSION` array for that copy of the script to work with. (Many copies of the script may be being executed at any point in time, one for each of the clients using the website.) This process allows that copy of the script to remember any values from previous client/server interactions during the session.

To see how this all works, let's write a script that attempts to both store and retrieve a value from the `$_SESSION` array. One common use of sessions is to store login authentication tokens, so we'll work with the username of the currently logged-in user as an example. [Listing 14-2](#) shows the code.

---

```
<?php
session_start();
$message = '(no username found in session)';

if (isset($_SESSION['username'])) {
 $message = 'username value in session = ' . $_SESSION['u
ername'];
}

$_SESSION['username'] = 'matt';

print $message;
```

---

*Listing 14-2: Attempting to retrieve, then store, a value in the `$_SESSION` array*

After (re)starting the session with `session_start()`, we store the default string value (no username found in session) in the `$message` variable. Then we use the `isset()` function to test whether any value can be found in the `$_SESSION` array under the 'username' key. If a value is found, we update `$message` with a new string including that value. Next, we store the value 'matt' into `$_SESSION['username']`. This will overwrite any existing value in the `$_SESSION` array for the 'username' key. Finally, we

print out whatever string is stored in `$message`. [Figure 14-3](#) shows the result of visiting this web page twice in a row.

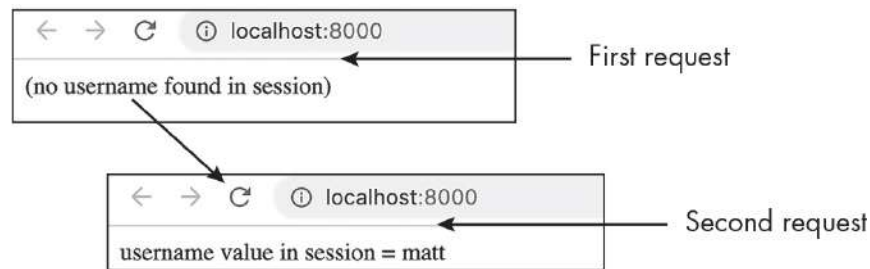


Figure 14-3: Retrieving the username from the session on the second request

The first time the page is visited, no value is found in the session for 'username' at the time the `if` statement is executed, so the default message is displayed. The second time the page is visited, however, the value 'matt' that was stored to the `$_SESSION` array the first time through the script is retrieved and displayed back. In this way, the session allows us to remember a value from one execution of the script to the next.

## Updating a Stored Value

One benefit of the `$_SESSION` array is that its values can be updated as needed. For example, if you were using the session to keep track of a user's shopping cart, you'd need to make updates each time the user adds or removes an item. We'll explore that exact scenario in [Chapter 15](#), but for now we'll consider a simpler example of updating a value in the `$_SESSION` array: a hit counter that stores and displays the number of HTTP requests made to a website. When personal websites first became popular, having such a hit counter was common.

A caveat here: in reality, sessions aren't an appropriate mechanism for storing data from different website visitors or for storing values for time periods of more than seconds or minutes. As we've discussed, a separate set of data is stored for each user's session, so a session-based hit counter can count only the number of website visits made by the *same user*. Also, sessions are terminated when the user quits the browser or the session times out, so visiting the site later in the day (or on another day) will mean the session-based hit counter will restart at 1, having "forgotten" the previous

visits. Still, a session-based hit counter is a helpful project for introducing some of the core logic involved in session storage operations.

[Figure 14-4](#) illustrates the counter we're aiming to create. The first time the page is visited, the counter is 1. Then, with each page refresh, the previous total is remembered and incremented by one.

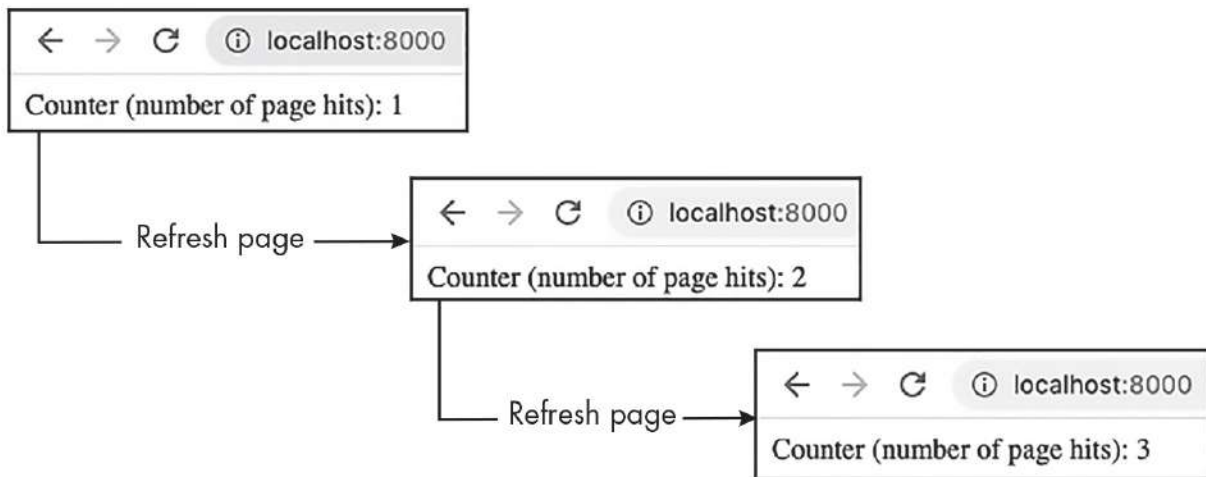


Figure 14-4: A hit counter incrementing after each page refresh

Listing 14-3 shows the *public/index.php* script needed to create the session-based hit counter.

```
<link rel="icon" href="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAQAAAC1AAAAQAAAACAAQAAAQAAIAAAACQd1PeAAAADe1EQVQI12P4//8/AAX+Av7czFnnAAAAE1FTkSuQmC">
```

```
<?php
session_start();
```

- ```
❶ $pageHits = 0;
❷ if (isset($_SESSION['counter'])) {
    ❸ $pageHits = $_SESSION['counter'];
}

❹ $pageHits++;
```



```
❺ $_SESSION['counter'] = $pageHits;  
❻ print "<p>Counter (number of page hits): $pageHits</p>";
```

Listing 14-3: Using a session variable to simulate a website hit counter

The first part of the script is HTML for a dummy favicon. Since modern browsers will send an extra request for a favicon image if one isn't defined in the HTML received, adding this `<link>` element at the beginning of the script keeps the browser happy and prevents it from making twice as many requests, which would make the hit counter confusing.

This script hinges on the typical logic of first testing whether any value exists in the session before attempting to retrieve and update it. We use a local PHP variable called `$pageHits` to represent the number of page hits when the script is executed, while we use the `'counter'` key to store the running total in the `$_SESSION` array. (The distinct names help avoid any confusion between these two values.)

After starting the session, we set `$pageHits` to a default value of 0 to represent the case when there's no existing value stored in the session ❶. Next, we test whether any value can be found in the `$_SESSION` array under the `'counter'` key ❷. If a value is found, we retrieve it from the array and store it in the `$pageHits` variable, overwriting the default value ❸.

At this point, whether or not a value is found in the `$_SESSION` array, we know we have an appropriate value in the `$pageHits` variable: either 0 or the running total of hits up to but not including the current page visit. In either case, we add 1 to `$pageHits` to account for the current visit to the page ❹. Then we store the updated value of `$pageHits` into the `$_SESSION` array under the `'counter'` key, either overwriting the key's existing value or creating it if this is the first page visit ❺. Finally, we output a message stating the number of times the page has been visited ❻.

The flowchart in [Figure 14-5](#) illustrates the general logic behind our hit-counter script. You can correlate this flowchart with points ❶ through ❻ in [Listing 14-3](#).

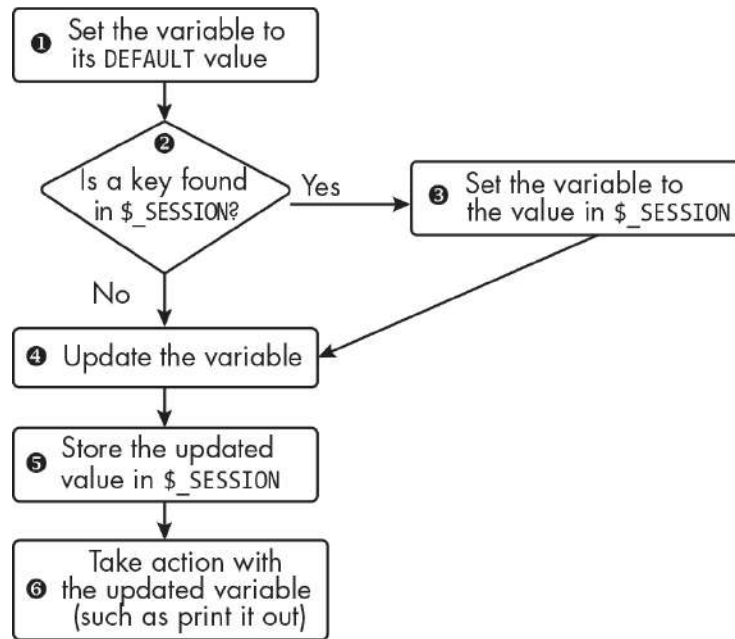


Figure 14-5: How to update (or set) a session variable

The logic in the flowchart generalizes to just about any work you may need to do with session values. First, you set a variable to a default value in the local script. Then you check whether a previously stored value can be found in the `$_SESSION` array and use that to overwrite the default if appropriate. Next, you update the local variable and store the updated value back in the `$_SESSION` array. Usually, you'll also want to do something with the updated variable. This approach works whether it's the beginning of the session (meaning nothing is stored in the `$_SESSION` array) or the code is being executed upon the second, third, or *n*th request during the session (meaning a value is stored in the `$_SESSION` array from a previous run of the script).

Unsetting a Value

At times you'll want to remove a particular value stored in the session. As we discussed in [Chapter 8](#), you can delete a value from an array by using the `unset()` function. This is different from setting an array element to something like `NULL`, an empty string, or `0`, since unsetting an element removes *any* value associated with the string key. Using our hit-counter example, we would remove any session value associated with the

'counter' key by calling `unset($_SESSION['counter'])`. We might do this, for example, if the page had a Reset button that cleared the hit counter.

Let's implement such a Reset button now, as well as add a link to revisit the hit-counter page (and therefore increment the counter). [Figure 14-6](#) shows the page we'll try to create.

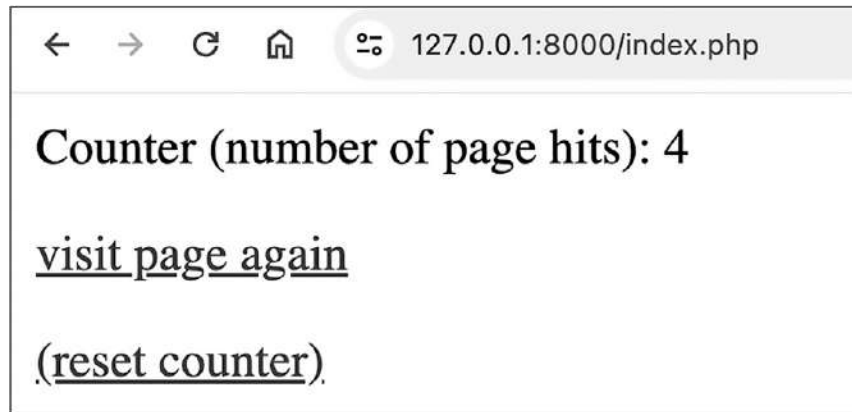


Figure 14-6: The hit-counter page with revisit and reset links

To add this functionality, update your *index.php* script to match [Listing 14-4](#).

```
<link rel="icon" href="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABC  
AIAAACQd1PeAAAADElEQVQI12P4//8/AAX+Av7czFnnAAAAElFTkSuQmC  
C">
```

```
<?php  
session_start();  
  
$action = filter_input(INPUT_GET, 'action');  
if ('reset' === $action) {  
    ❶ unset($_SESSION['counter']);  
}
```

```
$pageHits = 0;  
❷ if (isset($_SESSION['counter'])) {  
    $pageHits = $_SESSION['counter'];
```

```

    }

    $pageHits++;

    $_SESSION['counter'] = $pageHits;
    ?>

<p>Counter (number of page hits): <?= $pageHits ?>
<p><a href="/index.php">visit page again</a>
❸ <p><a href="/index.php?action=reset">(reset counter)</a>

```

Listing 14-4: Adding a reset link to the hit counter

After (re)starting the session, we retrieve and test the value of the 'action' query-string variable. If its value is 'reset', we unset the 'counter' element in the `$_SESSION` array ❶. Then the script proceeds as before. In the event that the user has clicked the Reset button and the 'counter' element was unset, it will be as if this element never existed, so the `isset()` test ❷ will fail and the 'counter' element will end up with a fresh value of 1 (after the default value of 0 is incremented).

At the end of the file, we add two links. The first is simply to revisit *index.php* (and so increment the counter). The second link is also to *index.php* but includes an 'action' query-string variable with a value of 'reset', which will trigger the script to reset the counter ❸.

Destroying the Session and Emptying the Session Array

Sometimes you might want to destroy the entire session and so invalidate the session ID and delete all stored session data. The deletion may be a security requirement, for example, since destroying a session should result in the server session data being *immediately* destroyed rather than waiting for a garbage-collection process (such as after a session time-out). That said, completely destroying a session is generally not recommended, since it may interfere with ongoing concurrent requests, such as asynchronous JavaScript. If all you want to do is clear the `$_SESSION` array, you can do so without entirely killing the session: use `unset($_SESSION)` or `$_SESSION = []` to turn `$_SESSION` into an empty array.

If you *do* need to completely destroy a session, take these steps:

1. (Re)start the session with `session_start()`.
2. Set the `$_SESSION` array to an empty array.
3. If using cookies, invalidate (time out) the session cookie.
4. Destroy the PHP session by executing the `session_destroy()` function.

See the PHP documentation at

<https://www.php.net/manual/en/function.session-destroy.php> for more information about this process.

Next, let's add a link for killing the session to our hit-counter page. [Figure 14-7](#) shows the page with the added link, which passes the `action=kill` query-string variable when the user wants to completely destroy the session.

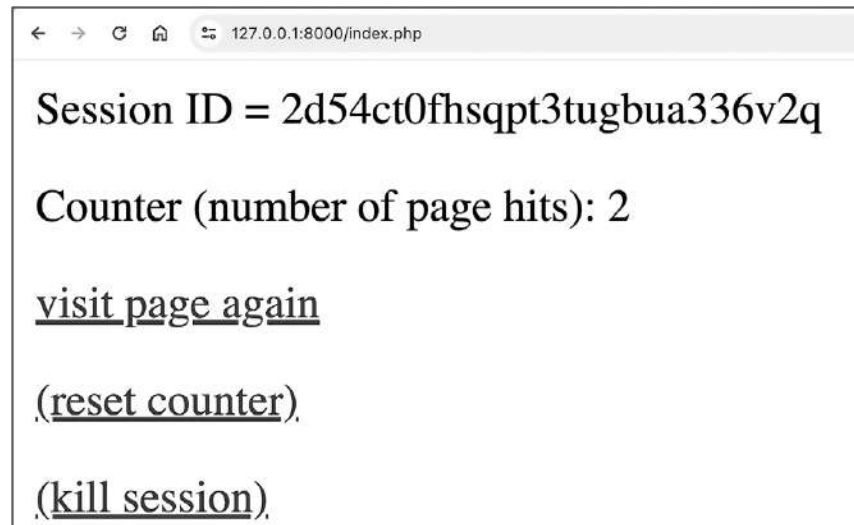


Figure 14-7: The hit-counter page with a new link to kill the session

To keep our *index.php* file from getting too complex, we'll encapsulate the code to kill the session in a separate function. [Listing 14-5](#) shows the code for this `killSession()` function; it implements steps 2 through 4 of the session-killing process outlined previously (step 1 happens at the beginning of the *index.php* file). Add a *src/usefulFunctions.php* file to your hit-counter project and enter the code in the listing.

```
function killSession() {  
    $_SESSION = [];
```

```

        if (ini_get("session.use_cookies")) {
            $params = session_get_cookie_params();
            setcookie(session_name(), '', time() - 42000,
                $params["path"], $params["domain"],
                $params["secure"], $params["httponly"]
            );
        }

        session_destroy();
    }
}

```

Listing 14-5: A function for killing a session

The function starts by setting `$_SESSION` to an empty array (step 2) and ends by calling `session_destroy()` (step 4). In between, the `if` statement implements step 3 of the session-killing process: invalidating the session cookie. For this, we check whether cookies are in use, then change the cookie with the current session name to an empty string, also setting an expiring time that's in the past (`time() - 42000`), effectively deleting the cookie.

With the `killSession()` function declared, update the *public/index.php* script as shown in [Listing 14-6](#) in order to offer a kill-session link to the hit-counter page.

```

<link rel="icon" href="data:image/png;base64,iVBORw0KGgoAAAA
NSUhEUgAAAAEAAAABC
AIAAACQdlPeAAAADElEQVQI12P4//8/AAX+Av7czFnnAAAAAE1FTkSuQmC
C">

```

```

<?php
require_once __DIR__ . '/../src/usefulFunctions.php';

session_start();

$action = filter_input(INPUT_GET, 'action');

```

```

❶ switch ($action) {

```

```

        case 'reset':
            unset($_SESSION['counter']);
            break;

        case 'kill':
            killSession();
            break;
    }

    $pageHits = 0;
    if (isset($_SESSION['counter'])) {
        $pageHits = $_SESSION['counter'];
    }

    $pageHits++;

    $_SESSION['counter'] = $pageHits;

    ?>

```

```

❷ <p>Session ID = <?= session_id() ?>
    <p>Counter (number of page hits): <?= $pageHits ?>
    <p><a href="/index.php">visit page again</a>
    <p><a href="/index.php?action=reset">(reset counter)</a>
    <p><a href="/index.php?action=kill">(kill session)</a>

```

Listing 14-6: Adding a kill-session link to the hit-counter page

First, we read in the declaration of the `killSession()` function from its source file. Then, since we need to check for multiple values of the 'action' query-string variable, we use a `switch` statement ❶ to decide how to process the incoming HTTP request. If the action is 'reset', we `unset` the 'counter' key of the `$_SESSION` array as before, or if the action is 'kill', we invoke `killSession()`. In the HTML at the end of the script, we add a kill-session link that passes the `action=kill` query-string variable to *index.php*. We also add a line displaying the current session ID to prove that the session is indeed being destroyed ❷; if you click the kill-session

link, this field should come up blank, in addition to the hit counter resetting to 1.

Summary

This chapter introduced you to sessions, which provide a mechanism for a web server to remember information about a user across multiple HTTP requests. You learned how to start a session with `session_start()`, how to store and update values in the `$_SESSION` superglobal array, and how to clear values from this array or destroy a session entirely. We outlined a basic pattern for working with session data, whereby you first set a default value, then overwrite this default with a value from the `$_SESSION` array (if one exists) before updating the value and storing it back in the array.

Exercises

1. Visit a website where you think sessions are being used, such as an e-commerce website with a shopping cart feature or a site with a login page. Use your browser developer tools to find the session ID that has been agreed upon by the server and client and is being stored as a cookie on your client device.
2. Write a PHP script that looks in the `$_SESSION` array for a value with the key `'guess'`. If it isn't found, store 0 for this key and display a message to the user stating no previous value was found. If a value *is* found in the session, add a random number from 1 to 10 to that value. Store the result back in the `$_SESSION` array and display it to the user.
3. Write a script to display a form that has a text box in which the user can enter a number, along with two Submit buttons. One Submit button should take the value from the text box and store it in the session. The second button should simply display the current value stored in the session, or a message stating no value was found in the session, as appropriate.

15

IMPLEMENTING A SHOPPING CART



When browsing an online store, you expect to add items to a shopping cart and for them to be remembered until you're ready to check out and pay. This is an extremely common requirement, so in this chapter we'll focus on building an application with a shopping cart. In the process, you'll learn how to work with sessions in a more sophisticated way, storing and updating whole arrays within the `$_SESSION` superglobal. We'll also continue our efforts to encapsulate the core logic for user actions into separate functions coordinated by a front-controller script, yielding a well-organized, easy-to-maintain application structure.

A *shopping cart* is basically a way to record which products and what quantities a user has selected. Our website therefore needs to show a list of products as well as enable the user to add items to their shopping cart and view its contents. We should also offer a way to edit the shopping cart,

allowing the user to change quantities, remove items, or empty the cart altogether. As an example, [Figure 15-1](#) shows an Amazon shopping cart with a few items.

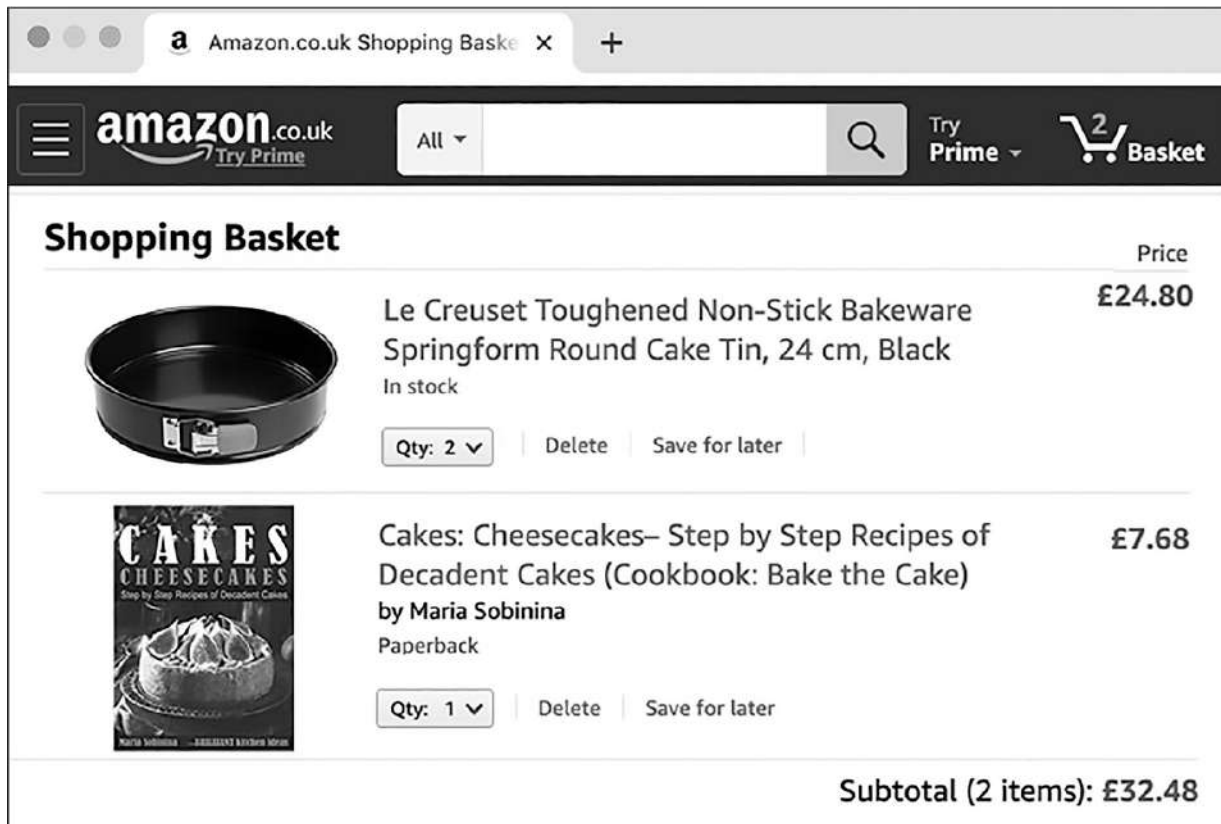
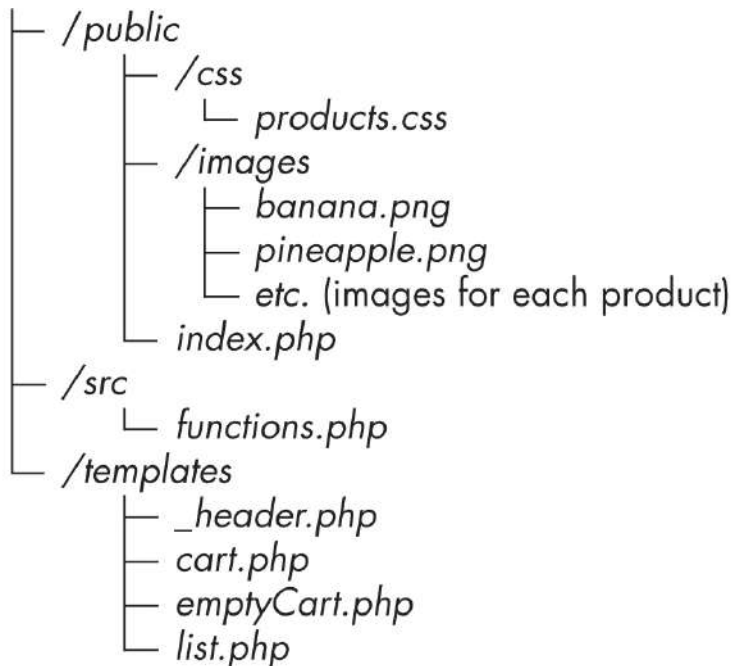


Figure 15-1: An e-commerce shopping cart

Notice that the cart displays the cost of the individual items as well as a total cost of the entire order. The Amazon shopping cart page also offers ways to change the quantity of each item and to delete items from the cart. We'll emulate all these features in our own shopping cart.

The Shopping Cart File Structure

Our shopping cart web application will span several files, including an *index.php* front-controller script, a *functions.php* file declaring various useful functions, a collection of template scripts, and other supporting files. The complete structure for the project will ultimately be as follows:



For now, start a new PHP project, create the necessary folders, and copy the image files (*banana.png*, *pineapple.png*, and so on) from the book's provided resource files into the *public/images* folder. With that, we're ready to start building the application. The book resource files and exercise solutions can be found at <https://github.com/dr-matt-smith/php-crash-course>.

Defining the Product List

Before we create the shopping cart itself, we'll begin by building a list of products for the user to choose from. We'll use an array to store the products available in our online shop. (Normally, product information would be stored in a database instead, but we won't be covering these until [Part VI](#).) In addition to the product array, we'll use a second array stored in the session to keep track of the contents of each user's shopping cart. You learned to loop through an array of products to automatically generate links featuring the product IDs in [Chapter 11](#) (see Listing 11-11 on [page 214](#)). We'll do something similar here.

[Figure 15-2](#) shows the product list page we'll create. As usual, we can use a little Bootstrap to help produce a professional-looking page with minimal CSS.

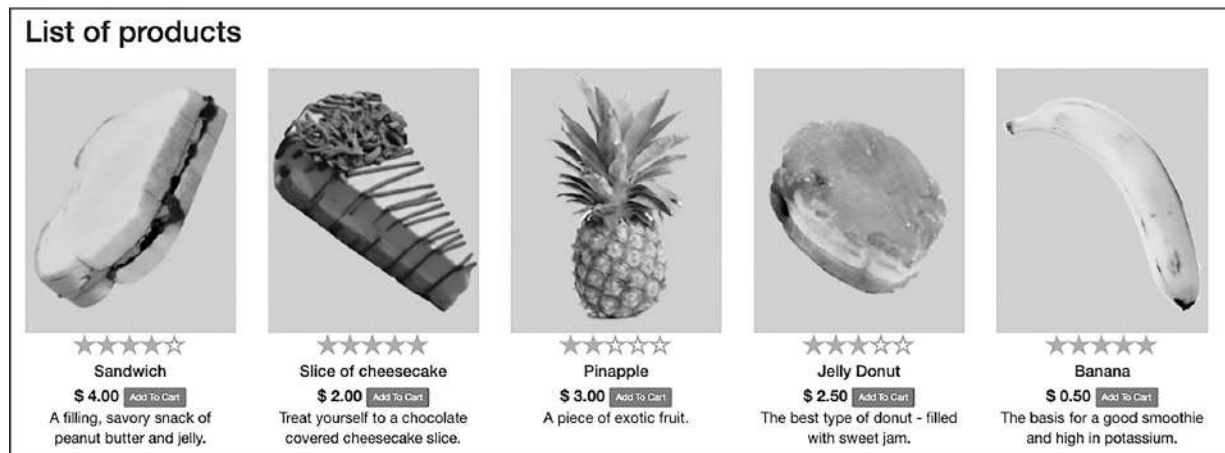


Figure 15-2: A page showing the list of products

We'll have five products on our site. [Table 15-1](#) lists the data values we need to store for each product.

Table 15-1: Product Details

Attribute	Value of attribute per product				
ID	010	025	005	021	002
Name	Sandwich	Slice of cheesecake	Pineapple	Jelly donut	Banana
Description	A filling, savory snack of peanut butter and jelly.	Treat yourself to a chocolate-covered cheesecake slice.	A piece of exotic fruit.	The best type of donut — filled with sweet jam.	The basis for a good smoothie and high in potassium.
Price	1.00	2.00	3.00	2.50	0.50
Stars	4	5	2	3	5
Image	<i>peanut_butter.png</i>	<i>chocolate_cheese_cake.png</i>	<i>pineapple.png</i>	<i>jellydonut.png</i>	<i>banana.png</i>

Notice that each product has a unique ID, name, description, price, star rating (from 1 to 5), and associated image filename. To implement all of this, we'll create a `$products` array that contains an element for each product, using the product IDs as keys. Each product in the array is itself an

array with keys for each attribute, such as 'name', 'description', 'price', and so on.

Creating the Products Array

Listing 15-1 shows how to declare each product array element to store its data values, creating elements for the first two products. Enter this code into *public/index.php*.

```
<?php
$products = [];

$products['010'] = [
    'name' => 'Sandwich',
    'description' =>
        'A filling, savory snack of peanut butter and jelly.',
    'price' => 1.00,
    'stars' => 4,
    'image' => 'peanut_butter.png'
];

$products['025'] = [
    'name' => 'Slice of cheesecake',
    'description' =>
        'Treat yourself to a chocolate-covered cheesecake slice.',
    'price' => 2.00,
    'stars' => 5,
    'image' => 'chocolate_cheese_cake.png'
];
--snip--
```

Listing 15-1: Declaring the first two products in index.php

First, we create a new, empty array called `$products`. Then we use `$products['010']` to append a new element to the end of `$products`, indexed with the key '010'. This new element is set to an array containing the properties for the peanut butter and jelly sandwich. We then append another element to `$products` with the key '025', containing an array with the properties for the cheesecake slice.

Using these first two products as a model, add the code declaring array elements for the remaining three products from [Table 15-1](#). Be sure to use strings containing the products' IDs as array keys.

Adding CSS

We'll need a little CSS to style the product list page, so create the *public/css/products.css* file containing the contents of [Listing 15-2](#).

```
.glyphicon-star {  
  ❶ color: goldenrod;  
    font-size: 150%;  
}  
  
.glyphicon-star-empty {  
  ❷ color: darkgray;  
    font-size: 150%;  
}  
  
❸ .product img {  
    width: 100%;  
}  
  
❹ .price {  
    font-size: 1.5rem;  
    font-weight: bold;  
}
```

Listing 15-2: The CSS to style the product list in `css/products.css`

We first style the stars that represent the product's ratings. We use Bootstrap Glyphicon stars. Filled stars will be gold ❶, and disabled stars will be gray ❷. Then we set the product images to fill (100%) the available horizontal spaces (for dynamically flexed blocks) ❸. I've used an image editor to make all images the same size for a consistent, professional look and feel. We also set the prices to be larger than normal text (1.5) and bold ❹. This will apply to prices both on the product list page and in the shopping cart itself.

Displaying the Star Ratings

Next, we'll write a PHP function that will return a string containing the HTML and CSS classes for a given number of stars. Create and complete the PHP script *src/functions.php*, based on the contents of [Listing 15-3](#).

```
<?php
function starsHtml($stars): string
{
    $s = '';
    switch ($stars) {
        case 0:
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            break;

        case 1:
            $s .= '<span class="glyphicon glyphicon-star"></s
pan>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            $s .= '<span class="glyphicon glyphicon-star-empty
y"></span>';
            break;

        case 2:
            $s .= '<span class="glyphicon glyphicon-star"></s
pan>';
```

```

        $s .= '<span class="glyphicon glyphicon-star"></span>';
        $s .= '<span class="glyphicon glyphicon-star-empty"></span>';
        $s .= '<span class="glyphicon glyphicon-star-empty"></span>';
        $s .= '<span class="glyphicon glyphicon-star-empty"></span>';
        break;
--snip--
        // Fill in the rest up to case 5.
    }

    return $s;
}

```

Listing 15-3: A function to output gold and gray stars

To keep our code simple, we've created a reusable function to return the HTML needed to display a given number of filled gold stars (set with the function's `$stars` parameter) while displaying the remaining (disabled) stars as gray. We use a `switch` statement to handle each possible star rating from 0 to 5. (I've shown cases 0 through 2; you can fill in the remaining cases.) We display each Bootstrap Glyphicon star character by using an HTML `<span>` element, with the CSS class `glyphicon glyphicon-star` for a filled gold star or `glyphicon glyphicon-star-empty` for a gray star. We could also implement this `starsHtml()` function by using some kind of loop, but the `switch` statement is more straightforward to follow.

Creating the Template Script

We now need to write the template script for the product list page. It will loop through the `$products` array, decorating the data for each product with the appropriate HTML. (For this to work, the `$products` array must already have been created in the calling script before the template script itself is executed; we've done this in [Listing 15-1](#).) Create `templates/list.php` and enter the contents of [Listing 15-4](#).

```
<!doctype html>
<html>
<head>
    <title>Shopping site: Product List</title>
    <link rel="stylesheet" href="/css/products.css">
    <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/
dist/css/bootstrap.min.css">
    <link rel="stylesheet"
        href="https://netdna.bootstrapcdn.com/bootstrap/3.
0.0/css/bootstrap-glyphicons.css">
</head>

<body class="container">

<h1>List of Products</h1>

<div class="row">

<?php
foreach ($products as $id => $product): ❶
    $price = number_format($product['price'], 2); ❷
?>
    <div class="product col-md-2 text-center">
        " ❸
        <?= starsHtml($product['stars']) ?> ❹
        <h1 class="fs-5"><?= $product['name'] ?></h1>
        <div class="price">
            $ <?= $price ?>
            <form method="post" action="/?action=addToCart&id
= <?= $id ?>"
                style="display: inline"> ❺
                <button class="btn btn-primary btn-sm">Add To
Cart</button>
            </form>
        </div>
    <div>
        <?= $product['description'] ?> ❻
```



```

        </div>
    </div>
<?php endforeach; ?>
</div>
</body>
</html>

```

Listing 15-4: The list.php template

We use three `<link>` elements to load the Bootstrap CSS styles and glyphs, plus our custom *products.css* stylesheet in the *css* folder. Then we declare a PHP `foreach` loop to iterate through the `$products` array, one product at a time ❶. On each iteration, the loop extracts the array containing details about the current product (with elements indexed by 'name', 'price', 'stars', and 'image'), and names that array `$product`. The `foreach` syntax `$products as $id => $product` also means that the key for the current array element (a string containing the current product ID) will be available in the `$id` variable. This will become important because our shopping cart logic needs to know the ID of each product so it can display and modify the cart contents.

Inside the loop, we create a `$price` variable containing the value of the 'price' element of the current product array, formatted to two decimal places with the `number_format()` function ❷. Next, we create an HTML `<img>` element that uses the image filename stored in `$product['image']` to create the path to the appropriate image file in the public *images* folder (for example, *images/banana.png*) ❸. We then pass the integer number of stars (`$product['stars']`) to the `starsHtml()` function so the appropriate HTML for the five gold and gray stars will be output ❹.

Still in the `foreach` loop, we set the name of the product as a level 1 heading styled with the `fs-5` Bootstrap CSS class for smaller text. We then create a `<div>` styled with the `price` CSS class (defined in our *css/products.css* file). This `<div>` displays the price of the current product, along with a form containing a button labeled `Add to Cart`. Once we've added shopping functionality, clicking this button will add the ID of the current product to the shopping cart via a `POST` HTTP request. The form passes two query-string parameters: `action` with a value of `addToCart` and

`id` with a value of the current product ID ❸. Finally, we display the product’s text description in its own `<div>` styled with the `fs-6` Bootstrap CSS class for small text ❹, before closing the `foreach` loop.

Updating the Index Script

To complete our product list, we need to add `require_once` statements to the *index.php* script so it can access code from other parts of the project. Update *public/index.php* as shown in [Listing 15-5](#).

```
<?php
require_once __DIR__ . '/../src/functions.php';

$products = [];
$products['010'] = [
    'name' => 'Sandwich',
    --snip--

require_once __DIR__ . '/../templates/list.php';
```

Listing 15-5: Reading the function declarations and template script into index.php

We use `require_once` to read in the function declaration file, which gives the page template access to the `starsHtml()` function. Then, at the end of the script, we read in the template script to display the product list page. Because we declare the `$products` array in between, the template will have access to this too. At this point, if you run the PHP web server (`php -S localhost:8000 -t public`) and visit the home page in a web browser, you should see the list of products, as shown earlier in [Figure 15-2](#).

Designing the Shopping Cart

Now let’s create a shopping cart display page, so users will have a way to view their cart. In “Implementing Cart-Manipulation Functions” on [page 293](#), we’ll implement the logic for adding to, deleting from, and changing the quantities of the shopping cart contents. [Figure 15-3](#) shows the shopping cart page we’ll create.

localhost:8000/?action=cart

Shopping Cart

Image	Item	Price	Quantity	Subtotal	Action
	Sandwich A filling, savory snack of peanut butter and jelly.	\$ 1.00	- 2 +	\$ 2.00	✕ Remove
	Pineapple A piece of exotic fruit.	\$ 3.00	- 4 +	\$ 12.00	✕ Remove
				\$ 14.00	Total

Figure 15-3: The shopping cart page

We'll need to display a row for each product in the cart, along with the unit price, the quantity in the cart, and the subtotal for each product. Each item also needs + and - buttons to incrementally change the quantity of the product by plus or minus 1, and a red `Remove` button to completely remove that product from the cart. We'll have a final row beneath the products displaying the total cost of all items in the cart.

Creating the Front Controller

Since the shopping cart will be our second display page, we need to introduce front-controller logic to `public/index.php` to select the appropriate template to display. We're going to offer two navigation links at the top of each page: List of Products (`href="/"`) and Shopping Cart (`href="/?action=cart"`). The front controller can therefore check for an `action` query-string variable with a value of `cart` to determine which page the user is requesting. Delete the `require_once` statement currently at the end of `public/index.php` and replace it with the code in [Listing 15-6](#).

```
--snip--
// Default is product list page
$page = 'list.php';
```

```
// Try to find "action=cart" in query-string variables
$action = filter_input(INPUT_GET, 'action');
if ('cart' == $action){
    // If found, change template file to be displayed
    $page = 'cart.php';
}

// Read in and execute the $page template
❶ require_once __DIR__ . '/../templates/$page';
```

Listing 15-6: Deciding which template to display at the end of index.php

This code creates a `$page` variable whose value is the name of a template file. We incorporate whatever value `$page` has into the `require_once` statement at the end of the script to display the appropriate template ❶. We have just two templates to choose from: the product list (*list.php*) and the shopping cart (*cart.php*, which we’ll write shortly). By default, we first set `$page` to the product list template. Then we retrieve a value for the `action` variable from the query-string parameters, and if it’s found to be `cart`, we change `$page` to the shopping cart template.

In “Writing the `switch` Statement” on [page 297](#), we’ll expand on the front-controller logic to account for all the actions users can take on the shopping cart page. For now, though, [Listing 15-6](#) has all the front-controller logic we need in order to view the cart.

Managing the Product and Cart Arrays

We’ll represent the contents of the shopping cart by using a `$cartItems` array whose keys are product IDs and whose values are the quantities of those products in the cart. We don’t need to store additional product information, such as prices, in this array, since we can use the product IDs to retrieve the other details from the source of our product data, the `$products` array.

Ultimately, we’ll be reading the contents of the shopping cart array from the session, but for now, we’ll hardcode an array of cart items for testing purposes. This is a common approach when developing a new feature: you hardcode initial data so you can write scripts to work with data for the new

feature, and then once that's all working, you make the source of the data dynamic (for example, coming from the session or a database).

To keep our *index.php* script from getting too complicated, we'll write a separate function that returns the array of items currently in the shopping cart. Add the `getShoppingCart()` function shown in [Listing 15-7](#) to the end of the *src/functions.php* file.

```
function getShoppingCart(): array
{
    $cartItems = [];
    $cartItems['010'] = 2; // 2 sandwiches
    $cartItems['005'] = 4; // 4 pineapples

    return $cartItems;
}
```

Listing 15-7: The `getShoppingCart()` function

The `getShoppingCart()` function creates and returns the contents of the `$cartItems` array. Each element in the array has a product ID string that acts as its key (in this hardcoded example, '010' and '005' for the sandwich and pineapple products, respectively). The value of each element is the quantity of that product in the cart (two sandwiches and four pineapples).

While we're at it, let's also move all the code declaring the `$products` array from *index.php* into a `getAllProducts()` function. Again, this will help keep the index script from becoming too complex. Copy the code you created in [Listing 15-1](#) from *public/index.php* and paste it at the end of the *src/functions.php* script, as shown in [Listing 15-8](#). To embed the code within a function, you'll also need to add the lines shown in black.

```
function getAllProducts(): array
{
    $products = [];
    $products['010'] = [
        'name' => 'Sandwich',
        'description' =>
            'A filling, savory snack of peanut butter and jell
y.',
```

```
        'price' => 1.00,
        'stars' => 4,
        'image' => 'peanut_butter.png'];

$products['025'] = [
    'name' => 'Slice of cheesecake',
    'description' =>
ce.',
    'price' => 2.00,
    'stars' => 5,
    'image' => 'chocolate_cheese_cake.png'];

$products['005'] = [
    'name' => 'Pineapple',
    'description' =>
    'A piece of exotic fruit.',
    'price' => 3.00,
    'stars' => 2,
    'image' => 'pineapple.png'];

$products['021'] = [
    'name' => 'Jelly donut',
    'description' =>
    'The best type of donut - filled with sweet jam.',
    'price' => 4.50,
    'stars' => 3,
    'image' => 'jellydonut.png'];

$products['002'] = [
    'name' => 'Banana',
    'description' =>
m.',
    'price' => 0.50,
    'stars' => 5,
    'image' => 'banana.png'];
return $products;
}
```

Listing 15-8: The `getAllProducts()` function

This function builds up the `$products` array and then returns it, allowing the array to be used in *public/index.php*.

Streamlining the Index Script

We can now update our *index.php* script to make use of the new `getAllProducts()` and `getShoppingCart()` functions. If you haven't already, delete the code building the `$products` array from the *index.php* script. Then update the file by adding the statements shown in [Listing 15-9](#).

```
<?php
require_once __DIR__ . '/../src/functions.php';

$products = getAllProducts();
$cartItems = getShoppingCart();

// Choose page to display
$page = 'list.php';
$action = filter_input(INPUT_GET, 'action');
if ('cart' == $action) {
    // If found, change template file to be displayed
    $page = 'cart.php';
}
require_once __DIR__ . '/../templates/$page';
```

*Listing 15-9: The simplified *index.php* script*

We now use the `getAllProducts()` and `getShoppingCart()` functions declared in *src/functions.php* to create the `$products` and `$cartItems` arrays. The arrays are therefore available to whichever template script is invoked in the final `require_once` statement.

Creating a Header Template

When we write a display template for the shopping cart, much of its page header content will be the same as that of the product list page. To simplify both the cart and product display templates, we'll put all the common HTML content into a separate template file named *templates/_header.php*.

This name follows the common convention of using an underscore to prefix the name of a *partial template* (a file that renders only part of a page and is shared by several other templates). This convention enables you to quickly identify partial templates within a folder, so you can ignore them when you're looking for a particular full template file.

Copy the code from *templates/list.php*, paste it into a new *templates/_header.php* file, and update the code as shown in [Listing 15-10](#).

```
<!doctype HTML>
<html>
<head>
    <title>Shopping site: <?= $pageTitle ?></title> ❶

    <link rel="stylesheet" href="/css/products.css">
    <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/
dist/css/bootstrap.min.css">
    <link rel="stylesheet"
        href="https://netdna.bootstrapcdn.com/bootstrap/3.
0.0/css/bootstrap-glyphicons.css">
</head>

<body class="container ">

    <nav>
        <ul>
            <li>
                <a href="/">
                    List of Products
                </a>
            </li>
            <li>
                <a href="/?action=cart">
                    Shopping Cart
                </a>
            </li>
        </ul>
    </nav>
```



```
<h1><?= $pageTitle ?></h1> ❷
```

Listing 15-10: The common page header content in `_header.php`

The header template uses a `$pageTitle` variable, which will need to be defined in each script that requires this header template. The variable appears twice: in the required HTML title in the `<head>` element ❶ and as a level 1 HTML heading in the body of every page ❷. In between, we add a simple navigation list with two links, one to show the product list and one to display the shopping cart.

We can now remove all that common header content from our product list page (*templates/list.php*). [Listing 15-11](#) shows the updated, simplified contents of the template.

```
<?php
    $pageTitle = 'List of products';

    require_once '_header.php';
?>

<div class="row">

<?php
foreach ($products as $id => $product):
    $price = number_format($product['price'], 2);
?>

    <div class="product col text-center">
--snip--
```

Listing 15-11: The simplified `list.php` template

We use a PHP code block to set a value for `$pageTitle` and read in the *_header.php* template to create the page header. The remainder of the file is just as before: a single `<div>` styled as a Bootstrap `row`, and a `foreach` loop to add column `divs` for each product.

Creating the Cart Display Template

We're now ready to write the template for displaying the items in the shopping cart. This script will loop through and display each product in the `$cartItems` array, calculate a subtotal for each product, and find the overall total for the complete shopping cart. The script is quite long, so we'll discuss it in sections.

The first part of the script accesses the common HTML page header code and sets up headings for each column in the cart display. Create a new *templates/cart.php* file and enter the code in [Listing 15-12](#).

```
<?php
    $total = 0;
    $pageTitle = 'Shopping Cart';

    require_once '_header.php';
?>

❶ <div class="row">
    <div class="col-2 fw-bold text-center">
        Image
    </div>

    <div class="col-4 fw-bold">
        Item
    </div>

    <div class="col fw-bold text-right">
        Price
    </div>

    <div class="col-3 fw-bold text-center">
        Quantity
    </div>

    <div class="col fw-bold text-right">
        Subtotal
    </div>
```

```
<div class="col fw-bold">
    Action
</div>
</div>
```

Listing 15-12: The first part of the cart.php template: setting up the cart display

First, we zero the `$total` variable; this will eventually hold the grand total of all items in the shopping cart. Then we set the `$pageTitle` variable and read in the header template, just as we did on the product list page. We then create a row of column headers **❶** identifying the `Image`, `Item`, `Price`, `Quantity`, `Subtotal`, and `Action` columns. We use the Bootstrap 12-column layout, making the image (`col-2`), item (`col-4`), and quantity (`col-3`) columns wider than the price, subtotal, and action columns (which default to 1/12 page-width columns).

The workhorse section of the shopping cart display script is a loop through the `$cartItems` array. For each product in the array, we'll display the product's image, name, description, unit price, quantity, and subtotal. Within the loop, we'll also create form buttons for changing the quantity and removing the product completely from the shopping cart. [Listing 15-13](#) shows the main PHP code block at the beginning of the loop.

```
<?php
foreach ($cartItems as $id => $quantity):
    ❶ $product = $products[$id];
      $price = $product['price'];
    ❷ $subtotal = $quantity * $price;

      // Update total
    ❸ $total += $subtotal;

      // Format prices to 2 d.p.
      $price = number_format($price, 2);
      $subtotal = number_format($subtotal, 2);
?>
```

Listing 15-13: The second part of the cart.php template: the item loop

The loop will continue in [Listing 15-14](#) with all the HTML column `<div>` elements needed to display each cart item.

Within a PHP code block, we start a `foreach` loop through the `$cartItems` array, setting `$id` to equal the current array element key and `$quantity` to equal the current element value. Since `$id` also corresponds to a key in the `$products` array, we use it to retrieve all product details for the current shopping cart item, storing them in the `$product` variable ❶. We then extract the price of the product into `$price` and multiply it by `$quantity` to get the subtotal for the cart item ❷. Next, we add the subtotal for the current cart item to the grand total (`$total`) ❸. This grand total will accumulate over the course of the loop. Finally, we format both `$price` and `$subtotal` to be numbers with two decimal places, since they represent currency values.

Listing 15-14 shows the rest of the `foreach` loop, where we lay out the HTML needed to display each cart item.

```
<div class="row border-top"> ❶

    <div class="col-2 product text-center"> ❷
        ">
    </div>

    <div class="col-4"> ❸
        <h1><?= $product['name'] ?></h1>
        <div>
            <?= $product['description'] ?>
        </div>
    </div>

    <div class="col price text-end align-self-center"> ❹
        $ <?= $price ?>
    </div>

    <div class="col-3 text-center align-self-center"> ❺
        <form action="/?action=changeCartQuantity&id=<?= $id
        ?>" method="post">
```

```

        <button type="submit" name="changeDirection" valu
e="reduce"
        class="btn btn-primary btn-sm">
        <span class="glyphicon glyphicon-minus"></spa
n>

        </button>
        <?= $quantity ?>
        <button type="submit" name="changeDirection" valu
e="increase"
        class="btn btn-primary btn-sm">
        <span class="glyphicon glyphicon-plus"></span
>

        </button>
    </form>
</div>

<div class="col price text-end align-self-center"> ❹
    $ <?= $subtotal ?>
</div>

<div class="col align-self-center"> ❺
    <form action="/?action=removeFromCart&id=<?= $id ?>"
method="post">
        <button class="btn btn-danger btn-sm">
            <span class="glyphicon glyphicon-remove"></sp
an>

            Remove
        </button>
    </form>
</div>

</div>
<?php endforeach; ?> ❻

```

Listing 15-14: The third part of the cart.php template: the item <div> elements

We start a new <div>, which will be a row for the current shopping cart item in the `foreach` loop ❶. Then we display a <div> containing an HTML image tag for the current product's image, getting the image filename from

the value of `$product['image']` ❷. The image is centered and also styled with the `product` CSS class (from our *products.css*) so that the image is sized to fit the `<div>` (`size: 100%`). For cart product display, we have a different (smaller) image with the same name, so the URL for the image references the *cart* subfolder of the *images* folder: `/images/cart/<?=$product['image'] ?>`.

We next create a `<div>` displaying the product name (`$product['name']`) as a level 1 heading, along with the text description of the product (`$product['description']`) ❸. Then we display the two-decimal-place value stored in `$price` in a `<div>` styled with the Bootstrap `align-self-center` class to vertically center the content in the row, with text aligned to the right (`text-end`) and with the `price` CSS class (bold and larger text) from our *products.css* stylesheet ❹.

The next `<div>` displays the quantity for the current cart item ❺. This is actually an HTML form, with two buttons (a minus and a plus) and the `$quantity` variable displayed. The form uses the `POST` method (since we're changing content on the server) and passes two query-string parameters: `action=changeCartQuantity` and the ID of the product (`id=<?=$id ?>`). Later in the chapter, we'll add more logic to our *index.php* script to recognize this new value of the `action` parameter and process a change of quantity. For the buttons, we use a minus-sign Glyphicon symbol (Bootstrap CSS class `glyphicon-minus`) with the name `changeDirection` and the value `reduce`, and a plus-sign Glyphicon, also with the name `changeDirection` but with the value `increase`.

Next, we display the value of `$subtotal`. Once again we format its `<div>` with our `price` CSS class because it is a currency value, and we align the text to the right (`text-end`) ❻. Then we output another vertically centered `<div>` with text aligned to the right (`text-end`) that presents a button for the user to remove the item completely from the shopping cart ❼. This is another form submitting with the `POST` method. It sends two query-string variables: `action=removeFromCart` and the ID of the current item (`id=<?=$id ?>`). The form offers the user a button consisting of a cross Glyphicon followed by the word `Remove`. Again, later in the chapter, we'll

add more logic to our *index.php* script to handle removing an item from the shopping cart. And with that, we end our `foreach` loop ❸.

Listing 15-15 shows the final part of the *templates/cart.php* script. This section of the code displays the grand total.

```
<div class="row border-top">
  <div class="col-11 price text-end">
    <?php
      $total = number_format($total, 2);
    ?>
    $ <?= $total ?>
  </div>

  <div class="col fw-bold ">
    Total
  </div>
</div>
</body>
</html>
```

Listing 15-15: The final part of the cart.php template: showing the total

This code outputs a final row, displaying two `<div>` elements. One gives the cart total (`$total`), formatted to two decimal places. The other outputs just the word `Total`. The `div` to display the total is made wide enough to cover all the missing columns to its left by using the Bootstrap class `col-11`, and since the value is a currency, it's again styled with the `price` CSS class defined in the *products.css* stylesheet file.

We've now achieved a two-page website. One page displays a list of products from data stored in the `$products` array. The second page displays the contents of a shopping cart by using the `$cartItems` array.

Interacting with the Session

Up to this point, we've used a hardcoded function to always return the same shopping cart contents. Now we'll modify the application to work with a dynamic, interactive shopping cart stored in the `$_SESSION` array so that the

cart contents can be remembered over the course of the browser session. In the process, we'll implement functions to modify the contents of the cart.

Updating the Cart-Retrieval Function

To work with dynamic shopping cart data, we must first update the `getShoppingCart()` function to retrieve the shopping cart contents from the session. As we explored in [Chapter 14](#), before we attempt to retrieve a value from the session, we should set a default value in case nothing is found. Our default case will be an empty shopping cart array.

Listing 15-16 shows the updated `getShoppingCart()` code. This function replaces the previous one in *src/functions.php*.

```
function getShoppingCart(): array
{
    // Default is empty shopping cart array
    $cartItems = [];

    if (isset($_SESSION['cart'])) {
        $cartItems = $_SESSION['cart'];
    }

    return $cartItems;
}
```

Listing 15-16: A function to retrieve the cart array from the session

We set `$cartItems` to the default value of an empty array. Then we use `isset()` to test whether a value can be found in the `$_SESSION` array for the 'cart' key. If an element exists for this key, its value is copied into the `$cartItems` variable. Finally, the function returns the contents of `$cartItems`.

Implementing Cart-Manipulation Functions

Next, we'll implement functions to manipulate the contents of the cart. First, to add an item to the shopping cart stored in the `$_SESSION` array, we need a function that will add a new element to the array with the product ID as the key and a quantity of 1 as the value. If `$_SESSION` doesn't contain a current

shopping cart array, this same function should create a new one containing one element. We'll achieve this by adding a new function to *src/functions.php*, as shown in [Listing 15-17](#).

```
function addItemToCart($id): void
{
    $cartItems = getShoppingCart();
    $cartItems[$id] = 1;

    $_SESSION['cart'] = $cartItems;
}
```

Listing 15-17: A function to add a new product to the cart

The function takes an `$id` parameter representing the ID of the item being added to the cart. We first store the current shopping cart array in the `$cartItems` variable by calling our `getShoppingCart()` function. Then we add a new element to `$cartItems`, with the key of the `$id` parameter and a value of 1 (the quantity of this product in the cart). Finally, we store the updated `$cartItems` array in `$_SESSION`, overwriting the previous `$_SESSION['cart']` array if one existed.

Removing a product completely from the shopping cart is similar to adding an item. We'll add a new `removeItemFromCart()` function to *src/functions.php*, as shown in [Listing 15-18](#).

```
function removeItemFromCart($id): void
{
    $cartItems = getShoppingCart();
    unset($cartItems[$id]);
    $_SESSION['cart'] = $cartItems;
}
```

Listing 15-18: A function to remove a product from the cart

Once again, the function takes in an `$id` parameter and starts by getting the array of items in the shopping cart. Then we use `unset()` to remove the element with the specified ID from the `$cartItems` array. As before, we

finish by storing the updated array in the session, overwriting the previous array.

Next, we'll write a `getQuantity()` function to look up the current quantity of a given product in the shopping cart array. This function, shown in [Listing 15-19](#), will help us write other functions for increasing and decreasing the quantity of a product in the cart.

```
function getQuantity($id, $cart): int
{
    if (isset($cart[$id])) {
        return $cart[$id];
    }

    // If $id not found, then return zero
    return 0;
}
```

Listing 15-19: A function to check the quantity of a product

This function takes in the desired product ID and the shopping cart array as parameters. We use `isset()` to test whether an element can be found in the cart for the given ID. If it's found, we return the value for that array element, representing the current quantity for that item. If no element is found indexed by `$id`, the item isn't in the cart, so we return a quantity of 0.

To increase the quantity of a product in the cart by 1, we need to retrieve the existing quantity, add 1 to it, and save the updated array back to the session. We'll encode this logic in an `increaseCartQuantity()` function, as shown in [Listing 15-20](#).

```
function increaseCartQuantity($id): void
{
    $cartItems = getShoppingCart();
    $quantity = getQuantity($id, $cartItems);
    $newQuantity = $quantity + 1;
    $cartItems[$id] = $newQuantity;
}
```

```
    $_SESSION['cart'] = $cartItems;
}
```

Listing 15-20: A function to increase the quantity of a cart item

After retrieving the `$cartItems` array, we use our `getQuantity()` function to determine the quantity of the product with the given `$id`. Next, we add 1 to this quantity and assign the new quantity to the `$id` key in the `$cartItems` array. Then we store the updated array in `$_SESSION['cart']`, overwriting the previous array.

Decreasing the quantity by 1 is a little more complicated than increasing it, since this may reduce the quantity to 0, in which case we should remove the product completely from the shopping cart. We need to retrieve the existing quantity, subtract 1 from it, test whether the quantity is now 0, update the cart array appropriately, and save the updated array back to the session. We'll encode this logic in a `reduceCartQuantity()` function, as shown in [Listing 15-21](#).

```
function reduceCartQuantity($id): void
{
    $cartItems = getShoppingCart();
    $quantity = getQuantity($id, $cartItems);
    $newQuantity = $quantity - 1;

    if ($newQuantity < 1) {
        unset($cartItems[$id]);
    } else {
        $cartItems[$id] = $newQuantity;
    }

    $_SESSION['cart'] = $cartItems;
}
```

Listing 15-21: A function to reduce the quantity of a cart item in file `src/functions.php`

We retrieve the shopping cart from the session and look up the quantity of the product with the given `$id`, just as we did in the previous function. Then we subtract 1 from this quantity. Next, we test whether the reduced

quantity is less than 1 (that is, 0), and if so, we remove the entire element from the shopping cart by using `unset()`. Otherwise, we store the reduced quantity in the shopping cart. Finally, we store the updated `$cartItems` array in the session, overwriting the previous values.

NOTE

When easy to do so, I always recommend using strong tests, such as less-than or greater-than, as we do for [Listing 15-21](#)'s `if` statement, rather than testing for equality with a value like 0.

Creating the Empty Cart Template

Now that we're making the shopping cart dynamic, we need to account for the possibility that the user will try to view their cart when it's empty. We'll create a separate empty cart template to display when this happens. [Figure 15-4](#) shows how it should look.

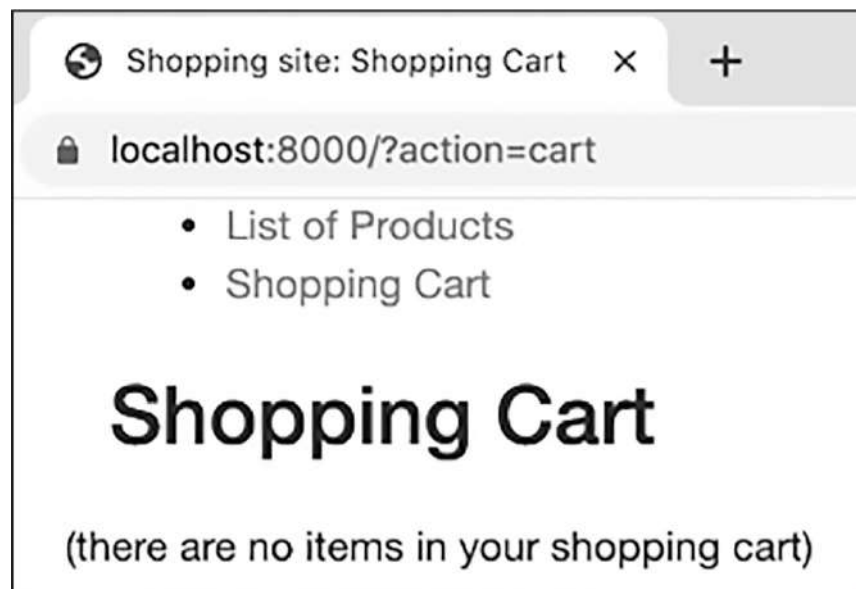


Figure 15-4: The empty shopping cart page

Listing 15-22 shows the code for the empty cart template. Enter this code in a new file called *templates/emptyCart.php*.

```
<?php
$pageTitle = 'Shopping Cart';

require_once '_header.php';
?>

<div class="row">
    (there are no items in your shopping cart)
</div>

</body>
</html>
```

Listing 15-22: The emptyCart.php template

This template is very short since, like our other display templates, it uses all the common content in *_header.php* (after first setting the required `$pageTitle` variable). Besides the header, it contains a single row of text informing the user that the cart is empty.

Finalizing the Front Controller

Our website now has three possible pages the user can see: the list of products, the empty cart, and the cart containing items. The application needs to identify and perform a range of actions, such as adding items to the cart and changing quantities. To finalize our shopping cart application, we therefore need to expand the front-controller script in *index.php* to choose from all these pages and actions.

Adding Display Functions

Although we'll be expanding the front controller, we still want to keep it as simple and concise as possible. Therefore, we'll declare two final helper functions: one to display the list of products and one to display the shopping cart (either empty or full). Add the code shown in [Listing 15-23](#) to the *src/functions.php* file.

```
function displayProducts(): void
{
```

```

    $products = getAllProducts();
    require_once __DIR__ . '/../templates/list.php';
}

function displayCart(): void
{
    $products = getAllProducts();
    $cartItems = getShoppingCart();

    ❶ if (!empty($cartItems)) {
        require_once __DIR__ . '/../templates/cart.php';
    } else {
        require_once __DIR__ . '/../templates/emptyCart.php';
    }
}

```

Listing 15-23: Functions to display the products and the shopping cart

First, we declare the `displayProducts()` function. It invokes `getAllProducts()` to obtain an array of all the products, then reads in and executes the *list.php* template. Then we declare the `displayCart()` function. It retrieves the `$products` and `$cartItems` arrays, then tests whether `$cartItems` is empty ❶. Depending on the result, the function displays the appropriate template, either *cart.php* or *emptyCart.php*.

Writing the switch Statement

We're ready to bring everything together with updated front-controller code. We'll follow the pattern we discussed in [Chapter 13](#) of using a `switch` statement to detect the required action and respond appropriately. Our front controller needs to handle six queries from the user:

- Display all products
- Display the shopping cart
- Add a product to the shopping cart (given a product ID)
- Remove a product from the shopping cart (given a product ID)
- Increase the quantity of a product by 1 (given a product ID)
- Decrease the quantity of a product by 1 (given a product ID), and remove it if the quantity is now 0

Replace the existing contents of *public/index.php* with the code in [Listing 15-24](#). This is a well-organized script, since all the complex logic has been relegated to separate functions. The front-controller script simply focuses on retrieving query-string and `POST` values, deciding which function to call, and choosing whether to display the product list or shopping cart after executing the appropriate function.

```
<?php
session_start();

require_once __DIR__ . '/../src/functions.php';

// Try to find "action" in query-string variables
$action = filter_input(INPUT_GET, 'action');
switch ($action){
    ❶ case 'cart':
        displayCart();
        break;

    ❷ case 'addToCart':
        $id = filter_input(INPUT_GET, 'id');
        addItemToCart($id);
        displayCart();
        break;

    ❸ case 'removeFromCart':
        $id = filter_input(INPUT_GET, 'id');
        removeItemFromCart($id);
        displayCart();
        break;

    ❹ case 'changeCartQuantity':
        $id = filter_input(INPUT_GET, 'id');
        $changeDirection = filter_input(INPUT_POST, 'changeDi
rection');

        if ($changeDirection == 'increase') {
            increaseCartQuantity($id);
```

```

        } else {
            reduceCartQuantity($id);
        }

        displayCart();
        break;

    ❸ default:
        displayProducts();
    }
}

```

Listing 15-24: The final version of the front-controller logic in index.php

We use PHP's `session_start()` function to start a new session if none currently exists or to renew the existing session. Calling this function is one of the first tasks you must do when processing a request to the server involving data stored in the `$_SESSION` array. Next, after reading in the function declarations from *src/functions.php*, we try to find a value for the `$action` variable from the query-string parameters and use it to begin a `switch()` statement.

If the value of `$action` is 'cart' ❶, we display the shopping cart by using the `displayCart()` function. If the value is 'addToCart' ❷ or 'removeFromCart' ❸, we retrieve the value of the `id` query-string variable, pass it to the appropriate function to add or remove a cart item, and then display the shopping cart. If `$action` is 'changeCartQuantity' ❹, we retrieve the product ID from the query string and get the value of `changeDirection` from the `POST` variables. If the latter is 'increase', we call `increaseCartQuantity()`; otherwise, we call `decreaseCartQuantity()`. In either case, we then display the shopping cart. Finally, if no value of the `action` query-string variable is found, we use the default case to display the list of products ❺.

Summary

In this chapter, we explored a real-world application of working with sessions: an interactive shopping cart. Our work with the shopping cart stored in the `$_SESSION` array followed the same approach as in the

preceding chapter: we set a default, attempt to read a value from the `$_SESSION` array, do something with the value, then save the updated data back into the session. The core logic for the project in this chapter revolves around two arrays, a product list and a shopping cart, both of which are keyed with product IDs. While in this chapter our list of products is fixed in an array, it would be straightforward to refactor this code to read the list of products from a database, which is how most real-world e-commerce websites operate.

By identifying the operations we wanted to apply to our shopping cart array and then encoding this logic as individual functions in *src/functions.php*, our final *public/index.php* front controller was simple to write. The listings for *templates/list.php* and *templates/cart.php* are relatively long, but this is mostly because we incorporated some Bootstrap CSS to make the list of products and shopping cart look more professional. In fact, the display templates would have been even longer had we not used a *templates/_header.php* file to store information common to all three pages of the site.

Many software systems are designed around the relationship between *data structures* (the way information is stored) and *algorithms* (the programming of the application logic). In this case, we structured both our product details and our shopping cart arrays around unique product IDs. This decision made creating the site logic and display templates straightforward.

We also saw that not all information about a session needs to be stored to the session. For example, the shopping cart subtotals and grand total can be calculated dynamically each time the cart page is loaded; they don't need to be stored. Likewise, we don't store product details within the shopping cart array, just the product ID, since we can retrieve all the other attributes of each product in the cart from the array of product items. To sum up, we created two data structures (the arrays for product details and shopping cart contents) and then designed algorithms (our front-controller logic and display templates) to efficiently interact with them.

Exercises

1. Add a new 'category' attribute for the products in the shopping cart application, with possible values 'savory' (the sandwich), 'sweet' (the cheesecake and the donut), or 'fruit' (the pineapple and banana). You'll need to add this element to each product in the `getAllProducts()` function, and add a new `<div>` in the *list.php* template to display it.
2. Add a button labeled `Empty Cart` to the shopping cart display page that results in a `POST` request sending the query-string parameter `action=emptyCart`. Then add an `emptyShoppingCart()` function to the *src/functions.php* file and a new case to the front controller's `switch` statement to invoke this function and clear the shopping cart's contents.

Hint: Since nothing else is being stored in the session for this project, you could either use the `killSession()` function approach from [Chapter 14](#) or simply replace the existing contents of `$_SESSION['cart']` with an empty array.
3. Our code doesn't have any validation checking for missing or invalid data. For example, when the `action` query-string variable is `addToCart`, we would have a problem if the ID is missing or invalid or if no product exists in the `$products` array matching the received ID. Add some simple validation so that if any problem arises with the ID when one is required, the product list page is displayed and no change is made to the shopping cart.

16

AUTHENTICATION AND AUTHORIZATION



Many websites implement security measures to safeguard private content or sensitive data. In this chapter, we'll use PHP sessions to develop an application with one such measure, a login form. In the process, you'll learn how to implement two related security concepts: authentication and authorization.

Authentication determines the identity of the person using the computer system—that is, *who* is the user? Our application will harness the username- and-password login method of authentication to identify the user. Meanwhile, *authorization* determines whether the user is permitted to access a particular part of the computer system (*what* is the user permitted to do?). Our application will use data stored in a PHP session, combined with access control logic, to authorize certain aspects of the web application that a user can access.

A Simple Login Form

At its heart, a login page is an HTML form usually consisting of a text field for the unique user identifier (such as a username or email address), a

password text field, and a Submit button. That's it! The difference between a regular text field and a password text field is that for the latter, the browser displays a placeholder symbol like an asterisk (*) for each character typed so that the actual password isn't displayed onscreen for a snooper to read. [Figure 16-1](#) shows a bare-bones login form.



Figure 16-1: A simple login form

The form has the three elements we've described: a Username field, a Password field, and a Submit button. [Listing 16-1](#) shows the HTML code needed to display this login form.

```
<!doctype html>
<html>
<head>
<title>login form</title>
</head>
<body>

    <form action="/?action=login" ❶ method="post">
        <label>Username:<input name="username"></label>
        <label>Password:<input name="password" ❷ type="password"><label>
            <input type="submit">
        </form>

</body>
</html>
```

Listing 16-1: The code for a basic login form

This code creates our labeled input boxes for a username and password, along with a `submit` input button. We specify `type="password"` for the Password field so that the input will display as placeholder characters ❷. The login form submits via the `POST` HTTP method ❶. Almost all login forms use the `POST` method so that the user's password won't be displayed as a query-string variable in the browser address bar, as would happen with the `GET` method. A second reason to use `POST` is that we don't want the login data to be cached. Instead, we want the server to process each username and password at the time the login form is submitted.

Caching occurs when a computer system or application, such as a web browser, stores copies of files locally (on the desktop, laptop, or phone) in order to retrieve them faster the next time they're requested. Although this works well for website logos and unchanging page content like home pages, you usually wouldn't want a web browser to store a local copy of submitted forms, such as login forms.

Web browser applications often cache web pages requested with the `GET` HTTP method, but they don't cache the content of web pages received after a `POST` HTTP request. Remember, `GET` requests simply retrieve information (without changing content on the server), so there is no problem with caching such requests. However, `POST` requests often involve form data submission (including login forms), and such requests can result in changes to the server contents such as deleting or changing database contents, so it would be dangerous, and perhaps insecure, to cache and repeat such `POST` requests.

Creating a Site with a Login Form

Now that you know how to create a basic login form, let's build a website that includes a functional, professional-looking login page. We'll secure one of the pages of the website by requiring users to log in to view it. The website will have the following pages:

- A home page
- A Contact Us page
- A login form
- An error message page

- A Secure Banking page (with Swiss bank account details!)

While we're happy for any user (whether logged in or not) to see the home page, Contact Us page, and login page, we need to authenticate users via the form on the login page in order to allow only authorized users to view the secured Swiss bank account page.

All the pages of the website will have the same structure and look. For example, [Figure 16-2](#) shows the home page.

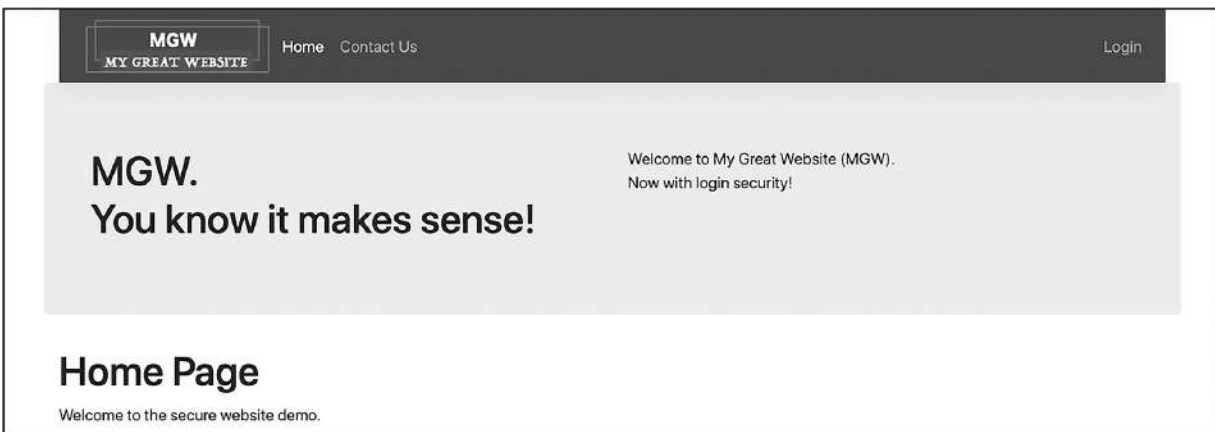


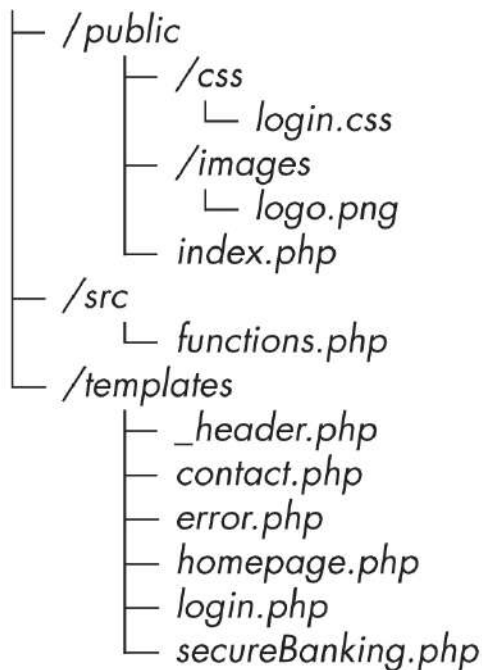
Figure 16-2: The home page of our website

Every page will have a Bootstrap-styled header featuring a custom logo and a navigation bar. Where appropriate, the navigation link relating to the page currently being displayed will be highlighted in white, while the other links will be gray. Below the navigation bar is a banner with a website tagline on the left and a greeting on the right. The bottom part of each page contains the individual page content (in this case, the heading and text telling users that this is the home page).

To build the website, we'll first create the individual pages. Then we'll create the login form and login-processing logic, and add the code to authorize only successfully logged-in users to view the secured Swiss bank account page.

Defining the File Structure

Create a folder for the project. Inside, it will have the following structure:



Creating the Shared Page Content

Now we'll create a `_header.php` file defining the header content shared by all the page templates. Using a shared header file will give a consistent look and feel to the site and avoid unnecessary code duplication. Additionally, if we ever want to change the site style or navigation bar contents, we'll need to change only the contents of this one header file.

Add the `_header.php` file to the `templates` subfolder. The file will contain quite a few lines of code (mostly Bootstrap classes and HTML `<div>` elements), so we'll look at it in three parts, starting with [Listing 16-2](#).

```
<?php
$homeLink = $homeLink ?? '';
$contactLink = $contactLink ?? '';
$loginLink = $loginLink ?? '';

$pageTitle = $pageTitle ?? '';
?>
<!doctype HTML>
<html>
<head>
    <meta name="viewport" content="width=device-width">
    <title>Secure site: <?= $pageTitle ?></title> ❶
```

```
<link rel="stylesheet" href="/css/login.css">
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.
3/dist/css/bootstrap.min.css"
    >
</head>
<body class="container">
```

Listing 16-2: The first part of _header.php

We start with a PHP code block declaring three variables to help control the display of the navigation links at the top of the page: `$homeLink`, `$contactLink`, and `$loginLink`, corresponding to the Home, Contact Us, and Login page links. Later, when we write the individual templates for each of these pages, we'll add code setting that page's variable to the 'active' string, which corresponds to a CSS style selector that will make the page's navigation link appear white. Here in the `_header` template, we use the null-coalescing operator `??` (see [Chapter 4](#)) to set all three variables to empty strings if they don't already have a value. An empty string will make the navigation link appear gray.

Thus, when we load the login page, for example, `$loginLink` will be set to 'active', and the Login navigation link will appear white, while `$homeLink` and `$contactLink`, not having any prior value, will be set to empty strings and their links will appear gray. This use of PHP variables to fill in CSS style values is an effective way to highlight the current page in a navigation bar and gray out the others.

NOTE

If you aren't confident using the null-coalescing operator, you can always write an `if` statement using the `isset()` function to provide the same functionality, such as `if (!isset($homeLink)) $homeLink = ''`.

We next use another null-coalescing operator to set the `$pageTitle` variable to an empty string in case it hasn't been set. Then we use the value of `$pageTitle` to form an HTML `<title>` element for the page ❶. This way, each function that includes our `_header.php` file can define a value for the PHP variable `$pageTitle`, giving each page a meaningful title that most

browsers will use for the bookmark text. We then read in the Bootstrap stylesheet as well as our own CSS stylesheet file from `/public/css/login.css`. This stylesheet, which we'll create later, will have a few styles for the login page. The final line in this listing starts a `<body>` element, styled with the Bootstrap `container` class.

The code for our header template continues in [Listing 16-3](#).

```
❶ <header class="navbar navbar-expand navbar-dark d-flex mb-3
    bg-primary">

    <ul class="navbar-nav p-2">
        <li class="nav-item">
            ❷ <a class="nav-link <?= $homeLink ?>" href="/">
                Home
            </a>
        </li>

        <li class="nav-item">
            ❸ <a class="nav-link <?= $contactLink ?>" href="/?
action=contact">
                Contact Us
            </a>
        </li>
    </ul>

    <ul class="navbar-nav ms-auto p-2">
        <li class="nav-item">
            ❹ <a class="nav-link <?= $loginLink ?>" href="/?ac
tion=login">
                Login
            </a>
        </li>
    </ul>
</header>
```

Listing 16-3: The second part of `_header.php`

We declare a `header` element that will contain the logo image and navigation links ❶. Within it, we declare the navigation bar link for the home page, styling this link with `class="nav-link <?= $homeLink ?>"` ❷. Here's where we continue implementing the navigation link styling mechanism we set in motion at the start of [Listing 16-2](#). The link will be styled as a Bootstrap navigation link (`nav-link`), but also as `active` (highlighted in white) if we've set the `$homeLink` variable to `'active'`. Otherwise, if `$homeLink` is an empty string, the navigation bar link won't be highlighted in white as the active page link. We style the Contact Us ❸ and Login ❹ links in a similar way, again making them active only if their corresponding link variable (`$contactLink` or `$loginLink`) contains the string `'active'`.

Listing 16-4 is the final part of our common page-header code.

```
<div class="row bg-light p-5">
  <div class="col">
    <h1>MGW. <br>You know it makes sense!</h1>
  </div>

  <div class="col">
    <p>
      Welcome to My Great Website (MGW).
      <br>
      Now with login security!
    </p>
  </div>
</div>
```

Listing 16-4: *The third part of _header.php*

Here we declare a Bootstrap row `<div>` with the standard content for every page on the website. This `<div>` is styled with a light gray background and some padding. It contains two `<div>` elements styled as columns, one with the website tagline and the other with a greeting touting the site's login feature.

Designing the Page Templates

Next, we'll create the templates for the home, Contact Us, and Secure Banking pages. With much of the work being done by the common page-header template, the template scripts for these three pages are straightforward. [Listing 16-5](#) shows our Home page template script. Save this script in the *templates* subfolder as *homepage.php*.

```
<?php
$pageTitle = 'Home Page';
$homeLink = 'active';
require_once '_header.php';
?>
```

❶ `<h1><?= $pageTitle ?></h1>`

```
<p>
Welcome to the secure website demo.
</p>
</body>
</html>
```

Listing 16-5: The homepage.php template

We first assign the `$pageTitle` variable a value, heading off the null-coalescing operator in [Listing 16-2](#). Additionally, since we want the Home link highlighted in the navigation bar, we assign the string `'active'` to the `$homeLink` variable. Then we read in and execute the `_header.php` template. Next, we display the value in `$pageTitle` as a level 1 heading in the body of the HTML page ❶. This is followed by a paragraph of page content, then tags to close the `<body>` and `<html>` elements of the page.

Listing 16-6 shows the code for the Contact Us page in *templates/contact.php*.

```
<?php
❶ $pageTitle = 'Contact Us';
❷ $contactLink = 'active';
require_once '_header.php';
```

```

?>

<h1><?= $pageTitle ?></h1>

❸ <p>
    Contact us as follows:
</p>

<dl>
    <dt>Email</dt>
    <dd>enquiries@securitydemo.com</dd>

    <dt>Phone</dt>
    <dd>+123 22-333-4444</dd>

    <dt>Address</dt>
    <dd>1 Main Street,<br>Newtown,<br>Ireland</dd>
</dl>

</body>
</html>

```

Listing 16-6: The contact.php template

The Contact Us template is similar to the Home page template, differing only in the value of `$pageTitle` ❶, the variable set to 'active' to highlight the Contact Us navigation link ❷, and the page content paragraph and definition list details ❸.

Next, we'll create the Secure Banking page, which is shown in [Figure 16-3](#). We'll add authorization logic later so that only logged-in users can view this page.

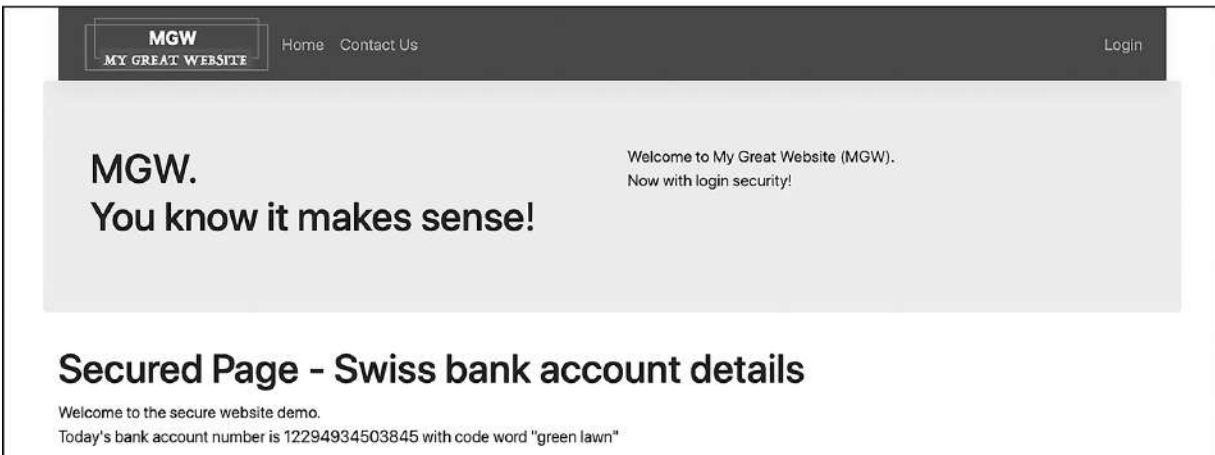


Figure 16-3: The Secure Banking page of our website

Listing 16-7 creates the Secure Banking page. Save this code in *templates/secureBanking.php*.

```
<?php
❶ $pageTitle = 'Secure Banking - Swiss bank account details';
require_once '_header.php';
?>

<h1><?= $pageTitle ?></h1>

❷ <p>
    Welcome to the secure website demo.
    <br>
    Today's bank account number is 12294934503845 with code
    word "green lawn"
</p>
</body>
</html>
```

Listing 16-7: The secureBanking.php template

Once again, this template is similar to those for the home page and Contact Us page. It differs only in the value of `$pageTitle` ❶ and the page content paragraph ❷. Since we don't currently link to this page in the navigation bar, we don't bother setting a variable to 'active'.

Developing the Login Form

Now we'll create a login form for our website ([Figure 16-4](#)). Though we'll use some extra HTML and CSS to make the form look more professional, at its core it'll be the same as the basic login form we created at the start of the chapter, with a Username field, a Password field, and a Log In submit button.

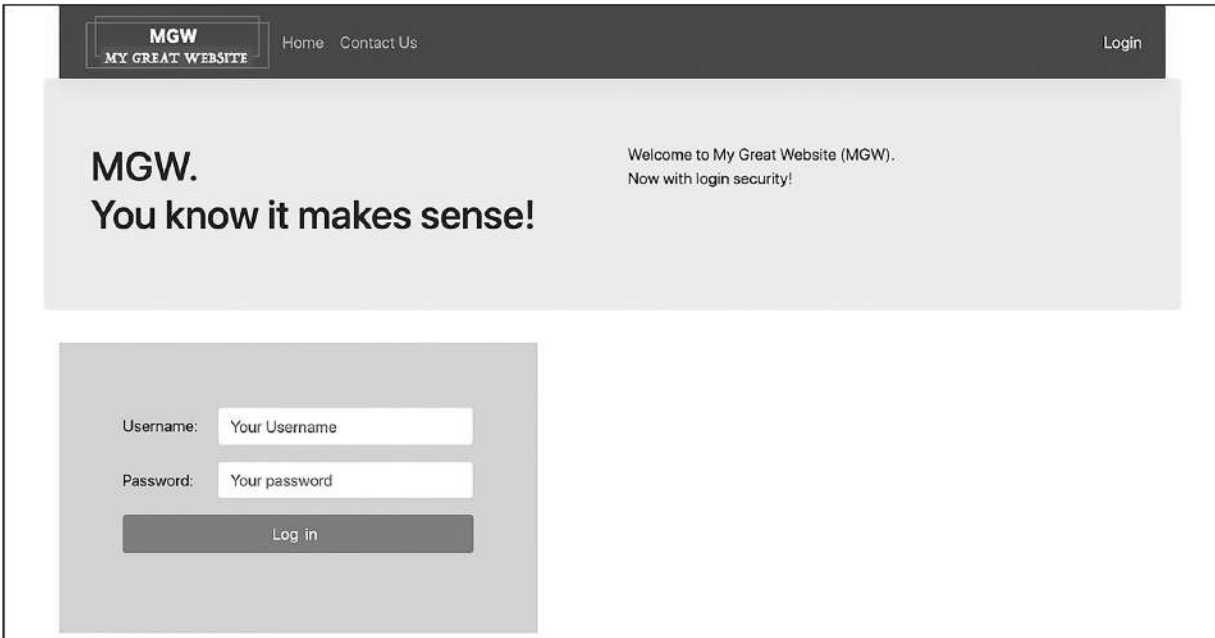


Figure 16-4: The login page of our website

Listing 16-8 shows the code used to create the login form, saved in *templates/login.php*.

```
<?php
$pageTitle = 'Login';
$loginLink = 'active';
require_once '_header.php';
?>

<div class="formLogin">

    ❶ <form action="/?action=login" method="post">
```

```

    ❷ <div class="form-group row m-3">
        <label for="username" class="col-form-label col-
sm-3">
            Username:
        </label>
        <div class="col">
            <input name="username" id="username"
placeholder="Your username" class="form-
control"
            >
        </div>
    </div>

    ❸ <div class="form-group row m-3">
        <label for="password" class="col-form-label col-
sm-3">
            Password:
        </label>
        <div class="col">
            <input name="password" id="password" type="p
assword"
placeholder="Your password" class="form-
control"
            >
        </div>
    </div>

    ❹ <div class="form-group">
        <input type="submit" class="btn btn-primary w-10
0"
value="Log in" class="form-control"
        >
    </div>
</form>

</div>
</body>
</html>

```

Listing 16-8: The login.php template

The script starts much like our other templates: we assign a value to `$pageTitle`, set `$loginLink` to the string 'active' so the Login link will be highlighted in the navigation bar, and read in and execute the `_header.php` template. Then we define a `<div>` to encapsulate the login form, styled with a custom `formLogin` CSS class (which we'll create shortly). The login form itself is declared with the `POST` method and the `/?action=login` action ❶.

We'll use this same `action` value (`login`) to both request the display of the login form and process the submitted form data, distinguishing between the requests by their HTTP method: `GET` will request the form be displayed, and `POST` will request processing of submitted login form data by the web application. We'll implement this logic later in the chapter.

Our form is structured as three Bootstrap rows for the Username ❷, Password ❸, and Log In ❹ inputs, each represented with a `<div>` element. The Username and Password rows contain `<label>` and `<input>` elements. Notice that the inputs have a `placeholder` attribute whose value will appear as faint gray filler text, and that we specify `type="password"` as an attribute of the Password input box to obscure the password while it's being typed in.

To finish up the login page, we'll create the CSS stylesheet `public/css/login.css`, shown in [Listing 16-9](#). It adds custom styling to the login form. Recall that the common `_header.php` template reads in this stylesheet for every page.

```
.formLogin {
    background-color: lightgray;
    padding: 4rem;
    max-width: 30rem;
}
```

Listing 16-9: The CSS code in login.css

The stylesheet defines the `formLogin` class referenced in [Listing 16-8](#). This style sets the form background to light gray, adds padding, and sets a maximum width of 30 characters.

Writing the Front Controller

As usual, we'll create a single front controller through which every request to our web application must arrive. Create *public/index.php* containing the code in [Listing 16-10](#).

```
<?php
require_once __DIR__ . '/../src/functions.php';

$action = filter_input(INPUT_GET, 'action');

switch ($action) {
    ❶ case 'contact':
        contact();
        break;

    ❷ case 'login':
        $isSubmitted = ($_SERVER['REQUEST_METHOD'] === 'POST');

        if ($isSubmitted) {
            // POST method so process submitted login data
            processLogin();
        } else {
            // GET method to display login form
            loginForm();
        }
        break;

    ❸ default:
        home();
}
```

Listing 16-10: The index.php front-controller script

The script follows the usual pattern of reading in the function-declaration file, extracting the value of the `action` query-string parameter (if found in the request), and passing it to a `switch` statement that decides what to do. If the value is `contact` ❶, we invoke `contact()`, which reads in the template to display the Contact Us page. If the value is `'login'` ❷,

we test whether the HTTP request used the `POST` method, indicating the user has submitted username and password values through the login form, and invoke the `processLogin()` function if so. Otherwise, we invoke the `loginForm()` function to display the login page. Finally, the default case ❸ displays the home page by invoking the `home()` function.

Implementing the Logic Functions

Next, we need to create the functions for implementing the logic of the website, saved in *src/functions.php*. Five of the functions are straightforward: they simply display the four main pages of the site (home page, Contact Us, login page, Secure Banking), plus an error message page. We'll look at these functions first, shown in [Listing 16-11](#).

```
<?php
function home(): void
{
    require_once __DIR__ . '/../templates/homepage.php';
}

function contact(): void
{
    require_once __DIR__ . '/../templates/contact.php';
}

function loginForm(): void
{
    require_once __DIR__ . '/../templates/login.php';
}

function secureBanking(): void
{
    require_once __DIR__ . '/../templates/secureBanking.php';
}

function showError($message): void
{
```

```
        require_once __DIR__ . '/../templates/error.php';
    }
```

Listing 16-11: The display functions in functions.php

The first four functions all perform the same task: they use a `require_once` statement to read in and display one of the template scripts. Next, the `showError()` function expects a `$message` string as a parameter. It too uses a `require_once` statement to read in and display one of the template scripts. In this case, since `$message` is a parameter, it has scope when the *error.php* template is read in and executed, so the template can display the contents of the string inside `$message`. (We'll create the *error.php* template shortly.)

The second part of the *functions.php* script, shown in [Listing 16-12](#), declares three functions for processing submitted usernames and passwords from the login form.

```
❶ function getUsers(): array
{
    $users = [];
    $users['matt'] = 'smith';
    $users['james'] = 'bond';
    $users['jane'] = 'doe';

    return $users;
}

❷ function processLogin(): void
{
    $username = filter_input(INPUT_POST, 'username');
    $password = filter_input(INPUT_POST, 'password');

    ❸ if (validLoginCredentials($username, $password)) {
        secureBanking();
    } else {
        showError('invalid login credentials - try again');
    }
}
```

```

❷ function validLoginCredentials($username, $password): bool
{
    $users = getUsers();

    if (isset($users[$username])) {
        $storedPassword = $users[$username];
        if ($password == $storedPassword) {
            return true;
        }
    }

    // If get here, no matching username/password
    ❸ return false;
}

```

Listing 16-12: The second part of functions.php

In this part of the script, we declare the `getUsers()` function ❶, which returns an array called `$users` whose keys are usernames and whose values are passwords. This is the list of users who can be authenticated through our website's login system (by providing a valid username and its corresponding password). Although we're using an array here, a real-world website would usually get username and password data from a database, and the passwords would be hashed for security reasons. We'll look at how to do this in [Chapter 30](#).

Next, we define the `processLogin()` function ❷. In it, we use `filter_input()` to attempt to retrieve the username and password submitted via the login form, storing the values in the `$username` and `$password` variables. Then we pass these values to the `validLoginCredentials()` function ❸. If the function returns `true`, we've successfully authenticated the user, since they were able to provide a matching username-password pair. Therefore, we display the secure bank page to the user by invoking the `secureBanking()` function. Otherwise, if `validLoginCredentials()` returns `false`, we invoke the `showError()` function to display the error page, passing an error message stating the login credentials are invalid.

Notice that the error message doesn't tell the user whether the problem is with the username or password. This follows the common security practice of *minimum information disclosure*. We shouldn't inform the user (or hacker-bot or whatever is trying to log in) when they've found a valid username. Armed with that information, an attacker could repeatedly use the valid username with different passwords in an attempt to gain access to the system, which would be easier than needing to guess the username *and* the password each time.

The final function is `validLoginCredentials()` ④, which expects two parameters, `$username` and `$password`. This is where we perform the all-important task of authenticating the user attempting to log in. We first retrieve the array of passwords indexed by the username from `getUsers()`, storing the array in the `$users` variable.

Then we test whether an element can be found in `$users` with the key `$username`. If no such key is found (`isset($users[$username])` is `false`), we exit the `if` statement and the function will return `false` ⑤, indicating the submitted username and password aren't valid. However, if `$username` can be found in `$users`, its corresponding value is stored in the `$storedPassword` variable. Then we test whether the password received from the login form (`$password`) matches the retrieved password from the array (`$storedPassword`). If the two passwords match, we have valid credentials, so we return `true`. Otherwise, the script will drop out of the `if` statement and return `false`.

Creating the Error Page Template

Now we'll create the template for the error page ([Figure 16-5](#)).

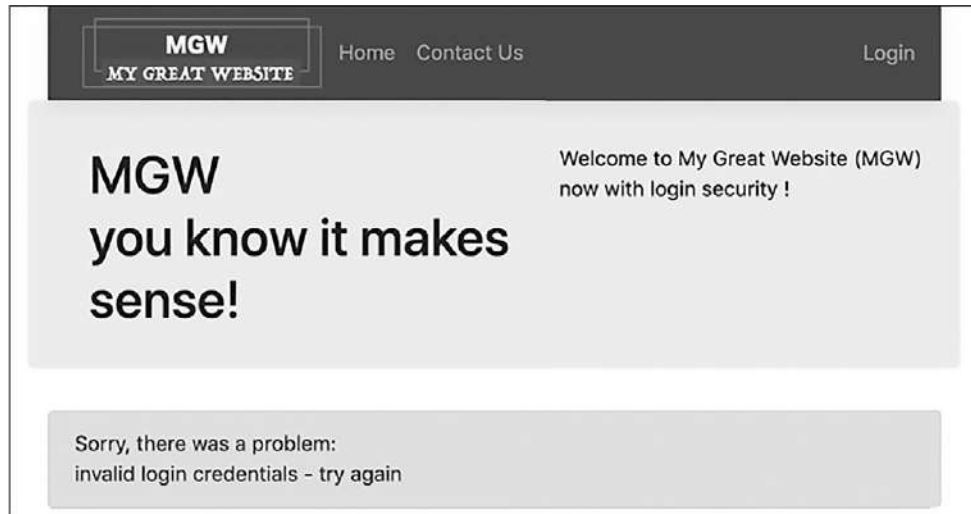


Figure 16-5: The error message page after invalid login credentials

This template, saved in *templates/error.php*, is similar to the other page templates we've created, as shown in [Listing 16-13](#).

```
<?php
$pageTitle = 'Error page';
require_once '_header.php';
?>

<div class="alert alert-danger" role="alert">
    Sorry, there was a problem:
    <p>
        ❶ <?= $message ?>
    </p>
</div>
</body>
</html>
```

Listing 16-13: The error.php template

We set the value of `$pageTitle` to 'Error page', then read in and execute the common *_header.php* template. In a pink Bootstrap alert-styled `<div>`, we output the string inside the `$message` variable ❶. All scripts that include this error page template should have first assigned a string to this variable (as we did, for example, in [Listing 16-12](#) when we called

`showError()` with the string 'invalid login credentials - try again').

Storing Login Data with Sessions

While our website at present allows a user to authenticate through the login form and visit the Secure Banking page, the site doesn't remember the successful login credentials. Once the user clicks away from the bank details page, they'll have to return to the login form and resubmit their credentials to view it again. To make the site more user-friendly, we can use PHP sessions to remember successful logins.

If all logged-in users should have the same level of access, we can simply store the username to the session after a successful login, as we'll do in this section. If different users have roles that come with different levels of authorization (for example, sales, supervisor, manager, administrator), we could store both the username and the corresponding role in the session. Then we would write logic so logged-in users can access only pages appropriate to their role. We'll talk through this second approach in Exercise 3 at the end of the chapter.

Let's add some code to save login data to the session. We'll also add a link to the Secure Banking page for the navigation bar, but we'll let the user visit that page only if they've logged in. Otherwise, we'll display an authentication error message.

Updating the Front Controller

We first need to edit our *index.php* front controller to handle navigation to the Secure Banking details page. Since we now plan to use sessions to remember login data, we also need to (re)start a PHP session at the beginning of the front-controller script. [Listing 16-14](#) shows the updated script, with new code highlighted.

```
<?php
session_start();
require_once __DIR__ . '/../src/functions.php';

$action = filter_input(INPUT_GET, 'action');
```

```

switch ($action) {
    case 'contact':
        contact();
        break;

    case 'login':
        $isSubmitted = ($_SERVER['REQUEST_METHOD'] === 'POST');

        if ($isSubmitted) {
            // POST method so process submitted login data
            processLogin();
        } else {
            // GET method to display login form
            loginForm();
        }
        break;

    ❶ case 'secured':
        if (isLoggedIn()) {
            secureBanking();
        } else {
            showError('invalid login credentials - try again');
        }
        break;

    default:
        home();
}

```

Listing 16-14: The updated index.php front-controller script

At the start of the script, we (re)start a session. Then we add a new case to the `switch` statement for when the value of `$action` is 'secured' ❶. In this case, we call the `isLoggedIn()` function, which we'll write shortly. If it returns `true`, we invoke `secureBanking()` to display the Secure Banking page. Otherwise, we display the error page with the message 'invalid login credentials - try again'.

Writing the Login Function

Now we need to write a new `isLoggedIn()` function to check whether a username is stored in the `$_SESSION` array, indicating a user has successfully logged in. We also need to update our `processLogin()` function so that when valid login credentials are processed, we store the username in `$_SESSION`. First, add `isLoggedIn()` to the end *src/functions.php*, as shown in [Listing 16-15](#).

```
function isLoggedIn(): bool
{
    if (isset($_SESSION['username'])) {
        return true;
    } else {
        return false;
    }
}
```

Listing 16-15: The `isLoggedIn()` function

The function uses a simple `if...else` statement based on whether a value can be found in the `$_SESSION` array for the string key `'username'`. If so, we return `true`; if not, we return `false`. Notice that we don't need to test the actual value stored in the session under the `'username'` key. We simply test whether *any* value is stored for this key. We don't care what the username is of the user who's logged in, as long as they've successfully logged in.

Now edit the `processLogin()` function in *src/functions.php* as shown in [Listing 16-16](#) to store the username in the session after a successful login.

```
function processLogin(): void
{
    $username = filter_input(INPUT_POST, 'username');
    $password = filter_input(INPUT_POST, 'password');

    if (validLoginCredentials($username, $password)) {
        $_SESSION['username'] = $username;
    }
}
```

```

        secureBanking();
    } else {
        showError('invalid login credentials - try again');
    }
}

```

Listing 16-16: Updating the `processLogin()` function

In the `if` branch of the function's conditional logic, we store the submitted username in the `$_SESSION` array under the `'username'` key. This way, the test in `isLoggedIn()` will pass after a successful login.

Updating the Header Template

Let's now edit the common `templates/_header.php` file to add a navigation bar link to the secured bank page, along with its associated CSS style variable. We'll use an `if` statement so that this link will appear only while the user is logged in. We need to add this conditional `nav-item` after the navigation bar items for the home and Contact Us pages, as shown in [Listing 16-17](#).

```

<?php
$homeLink = $homeLink ?? '';
$contactLink = $contactLink ?? '';
$loginLink = $loginLink ?? '';
❶ $securedLink = $securedLink ?? '';

$pageTitle = $pageTitle ?? '';
?>

--snip--

<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link <?= $homeLink ?>" href="/">
            Home
        </a>
    </li>

```

```

        <li class="nav-item">
            <a class="nav-link <?= $contactLink ?>" href="/?action=contact">
                Contact Us
            </a>
        </li>

```

```

❷ <?php if (isLoggedIn()): ?>
    <li class="nav-item">
        ❸ <a class="nav-link <?= $securedLink ?>" href="/?action=secured">
            Secure banking
        </a>
    </li>
<?php endif; ?>
</ul>
--snip--

```

Listing 16-17: Adding a conditional navigation link for the Secure Banking page in _header.php

We use the null-coalescing operator to set the `$securedLink` variable to an empty string if it has no value already ❶. Then we add an `if` statement that uses our `isLoggedIn()` function to test whether the user is logged in ❷. If so, the navigation link in the body of the `if` statement will be displayed. The link adds an `action=secured` variable to the query string ❸. Notice also that the value of the `$securedLink` variable is part of the CSS class for this link. As with our other navigation links, if this variable contains the string `'active'`, the link will be highlighted.

Updating the Banking Page Template

Now that we've added a navigation link for the Secure Banking page, we need to update the *templates/secureBanking.php* script to set the `$securedLink` variable to `'active'`. This will highlight the page's navigation link when the page is being viewed. Update the template as shown in [Listing 16-18](#).

```
<?php
$pageTitle = 'Secure Banking- Swiss bank account details';
$securedLink = 'active';
require_once '_header.php';
?>
--snip--
```

Listing 16-18: Updating the secureBanking.php template

The only change we need to make here is to add the statement that sets the `$securedLink` variable before we read in the shared header template.

Offering a Logout Feature

If we offer the user a way to log in and have their login information remembered, we should also offer a way to log out. Logging out a user means setting the `$_SESSION` array to be empty so it no longer contains an element with the string key 'username'. To put this into practice, we need to add a new function, update the front controller, and create a logout link in the navigation bar.

Adding the Logout Function

First, let's write a `logout()` function in *src/functions.php* that clears the user's data from the session. Add the code shown in [Listing 16-19](#) to the end of the file.

```
function logout(): void
{
    $_SESSION = [];
    home();
}
```

Listing 16-19: The `logout()` function

We set `$_SESSION` to an empty array, erasing the stored username from the session. Then we invoke the `home()` function to display the home page to the user after they've logged out.

Updating the Front Controller

Now we need to add a new logout case to the `switch` statement in our *index.php* front controller. Update the file as shown in [Listing 16-20](#).

```
--snip--

    case 'secured':
        if (isLoggedIn()) {
            secureBanking();
        } else {
            showError('invalid login credentials - try again');
        }
        break;

    ❶ case 'logout':
        logout();
        break;

    default:
        home();
}
```

Listing 16-20: The `logout` case in `index.php`

We add a case that invokes that `logout()` function when the `$action` variable has the value `'logout'` ❶.

Displaying the Logout Link

Finally, we need to conditionally decide whether to offer the user a Login link or a Logout link, depending on whether the user is currently logged in. We therefore need to add an `if` statement to the common *templates/_header.php* file, as shown in [Listing 16-21](#).

```
--snip--
    <?php if (isLoggedIn()):?>
    <li class="nav-item">
        <a class="nav-link <?= $securedLink ?>" href="/?
```

```

        action=secured">
            Secure Banking
        </a>
    </li>
    <?php endif; ?>

</ul>

<ul class="navbar-nav ms-auto p-2">
    <li class="nav-item">

        ❶ <?php if (isLoggedIn()): ?>
            <a class="nav-link" href="/?action=logout">
                Logout
            </a>

        ❷ <?php else: ?>
            <a class="nav-link <?= $loginLink ?>" href="/?ac
tion=login">
                Login
            </a>
        <?php endif; ?>

    </li>
</ul>
</header>

```

Listing 16-21: The conditional Login/Logout navigation bar link in _header.php

Inside the declaration of an HTML list item with the `nav-item` class, we use an `if...else` statement to test the value returned by the `isLoggedIn()` function. If the user is logged in ❶, we display the `/?action=logout` link. Otherwise, if the user isn't logged in ❷, we display the `/?action=login` link as before.

[Figure 16-6](#) shows the navigation bar when the user has successfully logged in and is visiting the secured bank details page.



Figure 16-6: The navigation bar showing the Secure Banking and Logout links

Notice that the Logout link appears on the right instead of the Login link. Additionally, the Secure Banking link in the middle is highlighted, since that's the page the user is currently viewing.

Displaying the Logged-in Username

The final feature we'll add to our website is to display the username of the logged-in user in the navigation bar, above the Logout link. To do this, we need a function to return the username stored in the `$_SESSION` array. We'll also need to update the shared header template and add extra code to our CSS stylesheet.

Retrieving the Username

To look up the current user's username, add the function in [Listing 16-22](#) to the end of the `src/functions.php` file.

```
function usernameFromSession(): string
{
    if (isset($_SESSION['username']))
        return $_SESSION['username'];
    else
        return '';
}
```

Listing 16-22: The `usernameFromSession()` function

Here we define the `usernameFromSession()` function. Using `isset()`, we check whether a value can be found in the `$_SESSION` array under the 'username' key. If a value exists, it's returned. Otherwise, the function returns an empty string.

Updating the Navigation Bar

Listing 16-23 shows what we need to add to the navigation bar in the common *templates/_header.php* file to display the current username as well as the Logout link.

```
--snip--
<?php if (isLoggedIn()):?>
    <span class="username">
        You are logged in as:
        <strong><?= usernameFromSession() ?></strong>
    </span>
    <a class="nav-link" href="/?action=logout">
        Logout
    </a>
<?php else: ?>
--snip--
```

Listing 16-23: Displaying the username in _header.php

We declare an HTML `<span>` element, styled with the CSS `username` class (which we'll create next). This displays the text `You are logged in as:` followed by the value returned from the `usernameFromSession()` function. Since we should display this text only when the user is logged in, there will always be a stored username, so `usernameFromSession()` should never return an empty string.

Updating the CSS

Finally, we need to add a CSS rule for the `username` class to *public/css/login.css*, as shown in [Listing 16-24](#). This style rule colors the username text yellow (in contrast with the dark background of the navigation bar)..username {

```
    color: yellow;
}
```

Listing 16-24: The `username` CSS class in login.css

[Figure 16-7](#) shows how the username is displayed in the navigation bar as a result of this CSS declaration.

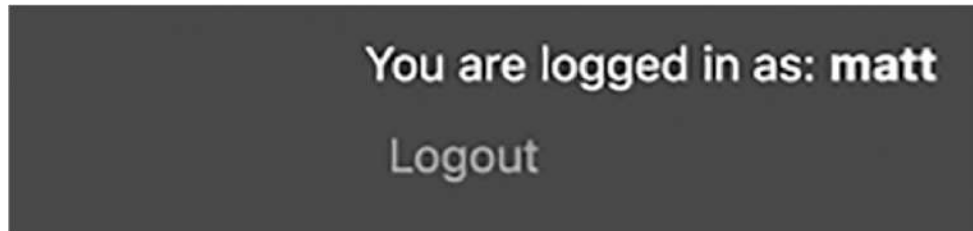


Figure 16-7: The username and Logout link in the navigation bar

The text showing the username appears above the Logout link. In this example, I've logged in with the username `matt`. This username was successfully stored in the `$_SESSION` array and then retrieved for display.

Summary

In this chapter, we created a front controller–driven website that uses the login form method of authenticating a user's identity. Although this is a small website with only a few pages, its basic architecture and approach to security mirror the way real-world, secure websites operate. We wrote functions to search for a match between submitted username and password credentials and a stored array of username and password pairs.

We stored details of a successfully authenticated user in a PHP session to remember when a user has logged in. Then we wrote program logic such as the `isLoggedIn()` function to allow our website to decide whether a user is authorized to view bank details. We used the same logic to decide whether to display a Login or a Logout link in the navigation bar.

Exercises

1. Add a second secured page to the website for this chapter that displays the solution to a math question (`answer = -2!`). In the navigation bar, add a link to the secured page that displays only when a user has successfully logged in.

Hint: You'll need to add a new case to the *index.php* front controller and a new function to display the page in *functions.php*.

2. Add two additional authorized users to the system, one with a username of `fred` and a password of `flintstone`, and the other with a username of `teddy` and a password of `cuddly`.
3. Try adding another layer of security to the website by having two user-authentication roles: `'USER'` and `'BANKER'`. Any logged-in user can view the math solution page, but only those with the `'BANKER'` role can view the bank details page. Add two more authorized banker user credentials to the system, one with a username of `banker1` and a password of `rich`, and the other with a username of `banker2` and a password of `veryrich`.

Hint: Try the following:

- a. Just as you have a `getUsers()` function, add a `getBankers()` function.
- b. Rename the `validLoginCredentials()` function to `validUSERLoginCredentials()`.
- c. Write a second version of this function as `validBANKERLoginCredentials()`.
- d. Change the logic in the `processLogin()` function to do the following: If a valid user logs in, store their username in the session and display the home page. If a valid banker logs in, store their username in the session, store their role in the session (`$_SESSION['role'] = 'BANKER'`), and display the home page.
- e. Add a new `getRoleFromSession()` function that returns the role found in the session. If a value is found for `$_SESSION['role']`, that string is returned; otherwise, an empty string is returned.
- f. Change the logic in the *index.php* front controller as follows: For the math solution, check whether a user is logged in. For the bank page, check whether the role of the logged-in user is `'BANKER'`. You could write something like `getRoleFromSession() == 'BANKER'`.

PART V

OBJECT-ORIENTED PHP

17

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING



So far in this book, we've been using PHP to write *procedural* code, a sequence of instructions executed in order. We're now going to shift our attention to a different way of using PHP: object-oriented programming (OOP). This chapter provides an overview of some important OOP concepts. Then the next several chapters will present in more depth how to harness OOP in your PHP projects.

The object-oriented style of programming revolves around *objects*, computer representations of real-world things, and *classes*, generalized models that define all the abilities and characteristics every object of a certain category should have. In an object-oriented computer system, objects send messages to each other, interpret those messages, and decide what instructions to execute in response, often creating a value to be returned to the sender.

The power of OOP lies in its capacity for abstraction: programmers can focus much of their attention on planning out a system of classes with features relating to the real-world task or problem the application is meant to solve, rather than always having to think about the code itself. For example, an online banking system might need classes like `Client`, `BankAccount`, and `Transaction`, and objects created from those classes would represent specific instances of clients, bank accounts, and transactions. The messages and operations to make changes to these objects might include functions like `withdrawCash($sum)`, `setNewOverdraft($limit)`, or `updateClientAddress($address)`. Similarly, an online computer game might need such classes as `Player`, `Level`, and `InventoryItem`, with messages and operations like `purchaseInventoryItem($itemID)` and `setPlayerName($name)`. A programmer can identify all these requirements and map out the necessary web of class relationships before writing a single line of code. Thanks to this planning and organization, the process of writing the code becomes much easier.

Ultimately, the programmer must declare each class, which does require writing code. The programmer will declare data variables and functions to carry out typical programming tasks such as performing numeric calculations, manipulating strings and arrays, and so on. However, the beauty of OOP is that once you've created a class, its structure is essentially hidden "under the hood." The rest of the coding process can focus on harnessing the objects' messages and functions, which closely relate to real-world concepts and tasks.

Classes and Objects

An object-oriented program is made up of PHP files that declare classes. A class can be thought of as the blueprint, or template, from which objects are created. Just like a blueprint of a car is just a drawing on paper, a PHP file declaring a class doesn't itself do anything. However, just as you can ask a factory to take the car blueprint and manufacture one or more physical cars, you can ask the PHP engine to use a class declaration to create objects based on that class.

Sometimes people refer to an object as an *instance* of a class, since each object is one specific manifestation of the general characteristics and behaviors defined by the class. You can treat the terms *object* and *instance* as synonyms: an object in the computer's memory, created from a class template, with a set of data values and the capability to respond to messages and execute functions. [Figure 17-1](#) illustrates the relationship between a class and the objects created from that class.

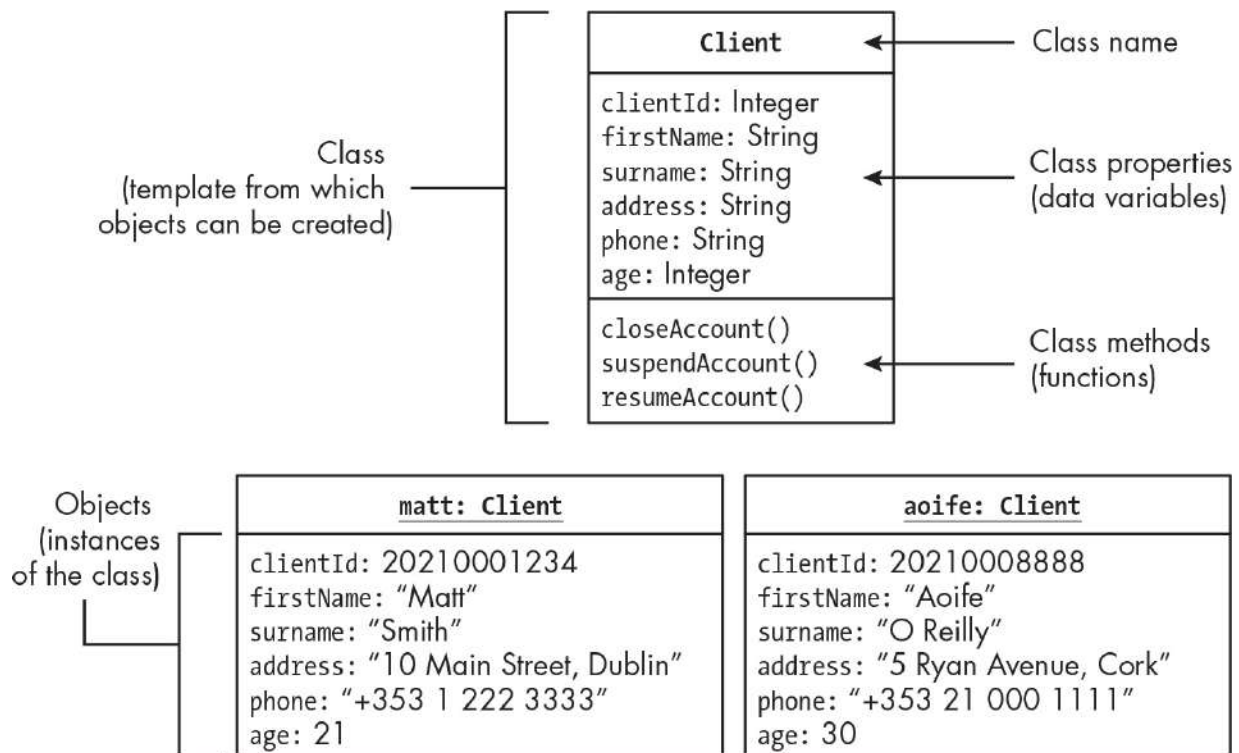


Figure 17-1: The `Client` class and two `Client` objects, `matt` and `aoife`

The class in the figure, `Client`, represents customers of a bank. You need to know three important aspects of a class: its name, its data variables, and its functions. In this example, our `Client` objects will have data variables for the client's ID number, name, and contact information. When variables are declared as part of a class, they're called *properties*. Likewise, our `Client` objects have several functions: you can close, suspend, or resume a customer's account. When functions are declared as part of a class, they're called *methods*. The various parts of a class are known

collectively as its *members*; the members of a class include all its properties (variables), methods (functions), and constants.

The bottom of [Figure 17-1](#) also shows two objects (or instances) created from the `Client` class, named `matt` and `aoife`. Each object has its own set of properties (for example, the `matt` object has a surname of `Smith` and the address `10 Main Street, Dublin`), and both objects have access to the methods defined in the `Client` class. In PHP programming, you can have a `$matt` variable that functions as a reference to the `Client` object of the same name, and you can send it a message to close Matt's account by writing `$matt->closeAccount()`. When the `$matt` object receives this message, it would execute its `closeAccount()` method.

WARNING

When you're writing object-oriented PHP code, make sure you don't confuse the `->` object operator (for objects and messages) with the `=>` operator, which is for key/value relationships in arrays.

Creating Relationships Between Objects

One of the powerful features of OOP is that you can build relationships between objects by linking a property of one object to another object. In some cases, you might relate objects of the same class. For example, if you have a `Person` class, you might link one `Person` object to another to demonstrate that one person is another person's parent. Other times, you might relate objects of different classes, such as to establish that a `Client` object is the owner of a `BankAccount` object, as shown in [Figure 17-2](#).

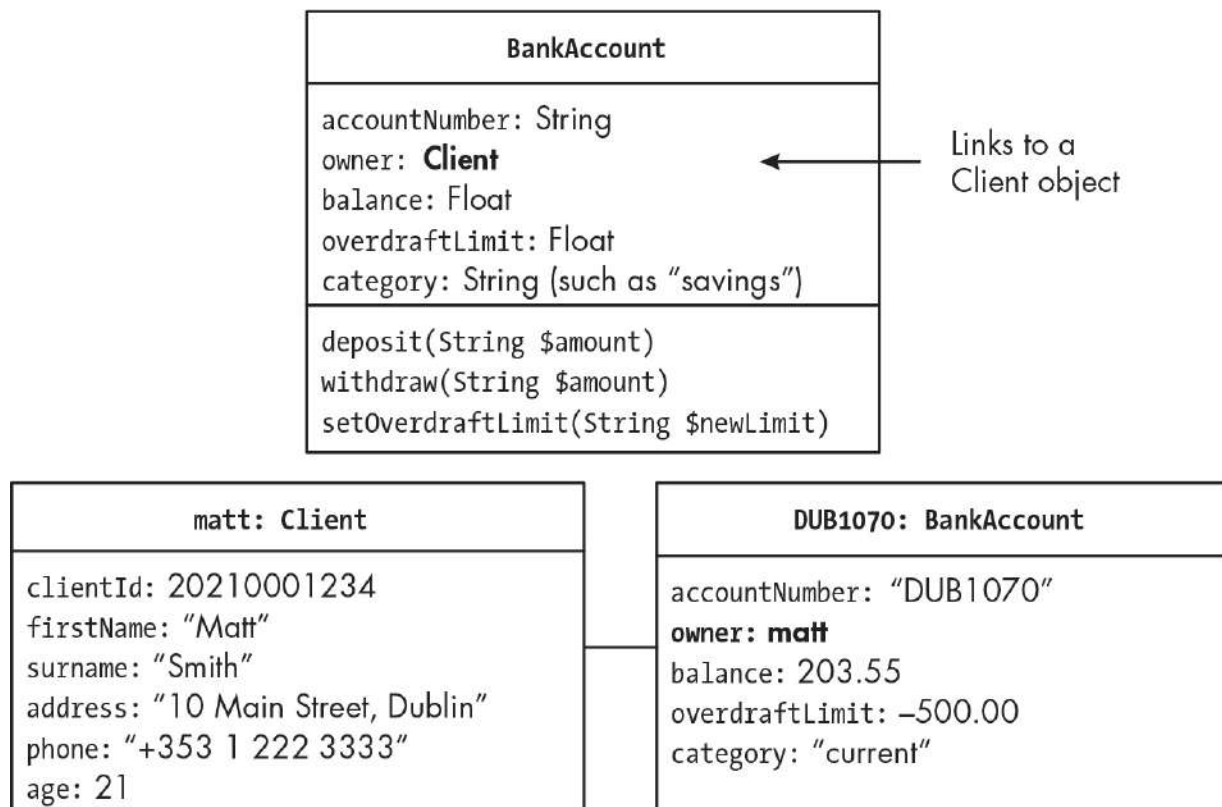


Figure 17-2: The `BankAccount` class declares that each `BankAccount` object is linked to a `Client` object.

The top of the figure shows the `BankAccount` class. Like the `Client` class we considered earlier, it includes data properties and methods that objects of this class can have: each `BankAccount` object has an account number, an owner, a balance, an overdraft limit, and a category, along with methods to deposit and withdraw money and set the overdraft limit.

The `owner` property is particularly significant: its value must be a reference to a `Client` object. The `owner` property thus creates a link between objects of the `BankAccount` and `Client` classes. For example, as you can see at the bottom of the figure, `DUB1070`, a `BankAccount` object, is linked to `matt`, one of the `Client` objects. The beauty of this mechanism is that for any `BankAccount` object we're working with, we can follow the link through the `owner` property to its related `Client` object and find out the name, address, and other details of the person who has the bank account.

Encapsulation and Information Hiding

A class organizes an object's data and the methods that can affect that data, gathering them in the same place. This principle, known as *encapsulation*, is central to OOP. Encapsulation helps keep projects organized; returning to the example in [Figure 17-1](#), it's logical that the methods for working with customer data are declared in the same file that also declares the data properties that should be stored about customers.

A danger arises, however, if all the data of an object can be directly changed by any part of the computer system that has access to that object. For example, we wouldn't want the `age` of a `Client` object to be set to 0 or a negative number! In fact, the bank might have a policy setting the minimum age of a client to, say, 16 years old. To avoid such unauthorized changes and ensure valid data, object-oriented languages, including PHP, provide ways to control access to an object's data.

The OOP feature of managing access to the data and methods of an object is known as *information hiding*. In PHP, you use the `public`, `private`, and `protected` keywords to declare different levels of access to the properties and methods of a class of objects. Continuing our example, we might prevent direct access to a `Client` object's `age` property by making it `private`. Then we might declare a `public setAge()` method that will update the age only if certain validation requirements are met, such as being an integer 16 or greater. We'll discuss how to use these features of object-oriented PHP in detail in the next few chapters.

Superclasses, Inheritance, and Overriding

You can assign properties and methods that are common among several classes to a *superclass*, a generalized class that other classes (called *subclasses*) can *inherit* characteristics from. For example, both staff and clients of a bank will share many common data properties, such as a name, address, and phone number. [Figure 17-3](#) shows the common properties and methods of the `Client` and `StaffMember` classes in bold. Some properties and methods are unique to each class, such as `clientId` for `Client` objects versus `staffId` for `StaffMember` objects.

Client	StaffMember
clientId: String firstName: String Surname: String address: String zipCode: String phoneNumber: String age: Integer customerType: String (such as "business")	staffId: String department: String jobTitle: String firstName: String Surname: String address: String zipCode: String phoneNumber: String status: String (such as "on leave")
setAddress(String) setPhoneNumber(String) setZipCode(String)	setAddress(String) setPhoneNumber(String) setZipCode(String) setStatus(String)

Figure 17-3: The `Client` and `StaffMember` classes have many duplicate members—very inefficient!

[Figure 17-4](#) illustrates how we can generalize the common properties and methods into a new superclass named `Person`, from which the `Client` and `StaffMember` classes both inherit. Only those properties and methods unique to a particular subclass are defined directly in the subclass itself. In PHP, we write something as simple as `class Client extends Person` to indicate that one class is to inherit from another.

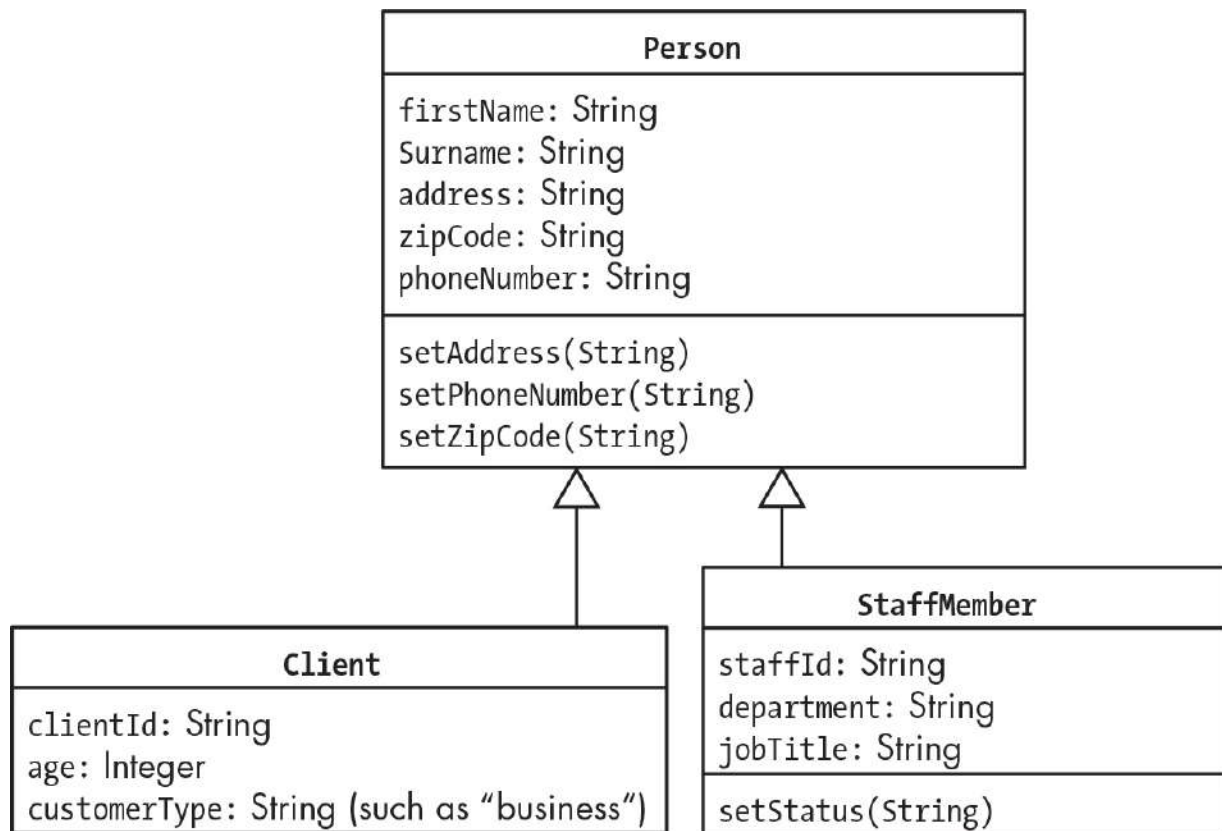


Figure 17-4: The generalized *Person* superclass eliminates duplication.

Superclasses and inheritance help you avoid duplicating code across several classes. For example, you wouldn't want to write code for operations like validating telephone numbers and addresses in multiple places; if something were to change (such as the 2014 introduction of Irish ZIP codes, called Eircodes!), you'd have to update several classes, and perhaps addresses and phone numbers would end up being treated differently in different parts of the system. Thanks to superclasses and inheritance, the code needs to be updated only once.

Often you'll want subclasses to inherit all the methods from their superclass, but this isn't always the case. Sometimes a class may need to have logic that's different from that of its superclass. For example, you might have a subclass of clients whose costs or taxes are calculated differently, or you might have products that require a special disclaimer to be displayed. In such circumstances, a subclass can *override* an inherited method; that is, you can create a method directly in the subclass that takes precedence over the method of the same name from the superclass. In PHP,

overriding a method is straightforward: if a subclass declaration implements a method matching one that would have been inherited from a superclass, then the subclass's method will be used.

The Flow of Control for Object-Oriented Systems

Each type of programming language has a *flow of control*, which indicates how a computer system starts running and how it decides what to do next, after it's started. As you've seen in the last several chapters, the flow of control for a procedural PHP web application usually is driven by a front controller in the *index.php* PHP script. When the web server receives an HTTP request, the statements in *index.php* are executed in sequence. In an object-oriented application, however, where many of the PHP files are devoted to declaring classes of objects, the flow of control may seem a bit more obscure. When do objects of those classes actually get created and start sending messages to one another?

Object-oriented PHP web applications still have an *index.php* script, though it looks a bit different from what we've seen previously. It typically creates the main application object, which serves as a front controller, and tells this object to process the received request and act appropriately.

[Listing 17-1](#) illustrates the kind of *index.php* scripts we'll write in the coming chapters.

```
<?php
require_once __DIR__ . 'path to class declaration file';

$app = new WebApplication();

$app->run();
```

Listing 17-1: The typical code for an object-oriented index.php front controller

First, we read in the declaration for the class (or classes) we'll be using. You'll learn a simple way to do this in [Chapter 18](#), and then in [Chapter 20](#) you'll learn to use the Composer PHP command line tool to load class declarations in a more general way.

Next, we create a new object of the `WebApplication` class, storing a reference to this new object in the `$app` variable. The `WebApplication` class will contain the logic for processing an HTTP request received from a web client, which is everything we previously would have put into the *index.php* script itself.

Then we send the message `run()` to the `WebApplication` object, which will result in the `run()` method that's declared in the `WebApplication` class being executed for the particular `$app` object. The code for the `run()` method will include statements to do things like extract an action from the URL-encoded variables and check the session for login status. The code may also invoke other methods or create new objects as appropriate to complete the action requested by the web client.

In summary, for a web application like this, the flow of control is sequential within the *index.php* file; the statements in that file are executed in order. As a result, an object is created, and the object is sent a message to start processing the HTTP request. From this point on, all the other logic for our web application will be in the methods of the `WebApplication` class, or other classes for which objects will be created as part of the execution of methods in the `WebApplication` class.

An Example Class Declaration

Let's now consider an example PHP class declaration. In [Listing 17-2](#), we declare a class named `Player`, such as might be part of an online game system. Don't worry too much about the specifics of the code; we'll cover how to write PHP classes in much more detail in later chapters. For now, this example simply offers a glimpse of the type of code that's to come.

```
<?php
class Player
{
    private string $name;
    ❶ private int $highScore = 0;

    ❷ public function setName(string $name)
    {
```

```

        $this->name = $name;
    }

    ❸ public function setHighScore(int $newScore)
    {
        if ($newScore > $this->highScore) {
            $this->highScore = $newScore;
        }
    }
}

```

Listing 17-2: The PHP code to declare a `Player` class

We use the `class` keyword to declare a class called `Player`, and we give the class two properties, `name` and `highScore`. Just as for variables in non-object-oriented PHP, you can assign a default value to a property. We do that here, setting the default value of `highScore` to 0 ❶ so that each new `Player` object will be created with an initial high score of 0. We next declare the `setName()` method ❷, which when invoked will take in a string parameter and store it in a `Player` object's `name` property. Then we declare the `setHighScore()` method ❸. It takes in a parameter (`$newScore`), and if this new score is higher than the stored high score for the object, then the new score is stored in the object's `highScore` property.

When a method is executed, it will be working on the properties of a particular object created from the class the method is defined in. In the method's definition, the special PHP keyword `$this` refers to the object on which the method will be invoked. Thus, in the definition for the `setName()` method, we use the `$this` keyword (as in `$this->name = $name;`) as a stand-in for whichever `Player` object is being assigned a name. For example, I might have an object `$player1` whose name I set to "Matt", and a second object `$player2` whose name I set to "Aoife". In both cases, the `setName()` method code ❷ would be invoked to assign the name, and in both cases `$this` would stand for the `Player` object whose name is being set: first `$player1` and then `$player2`.

Our class declaration includes examples of information hiding, in that the `name` and `highScore` properties are declared as `private`. They can't be

changed by code from outside the `Player` class. However, we've also declared the `setName()` and `setHighScore()` methods as `public`. These *setter* methods allow for outside communication with `Player` objects, but only through internal validation checks coded in the methods (such as checking that a new score exceeds the previous high score before overwriting the `highScore` property); these checks ensure that the object's data is never set to invalid or inconsistent values.

Notice that each method in [Listing 17-2](#) is short and simple. Once the architecture of an application has been created, writing code to declare each property and method of a class is often relatively straightforward. While the methods for a full web application will be longer than those shown in this example, one benefit of OOP is that it typically allows programmers to focus on writing one method of a class at a time, with each method having one clear responsibility.

With OOP, you don't have to think about all the ways the method could be used; you only need to make sure that what you're writing is a correct behavior for the class. For example, it doesn't matter if a `Player` object's name is being set for the first time at the start of the game, is being updated in the middle of the game because the player changed their mind, or is being changed automatically because the player was turned into a frog. The point is to write the `setName()` method so that it requires valid parameters and results in the correct changes to the `Player` object's properties. As such, the code will be easy to write and maintain.

Summary

In this chapter, we surveyed some of the important OOP concepts. You saw that classes are templates for creating objects and that classes allow you to store all the variables and functions an object needs in one place, a concept called *encapsulation*. You also saw that objects can link to each other through their properties and that objects of different subclasses can inherit shared properties and methods from superclasses. Finally, you had a first look at some of the PHP code that implements these concepts.

In the next few chapters, you'll learn how to declare classes, create objects, and send messages to objects to invoke their methods and change their data. You'll then begin to put all that knowledge together to create

well-organized object-oriented web applications building on the features we've explored in all the previous chapters. As you read ahead, don't forget that you already know many of the core requirements for writing object-oriented PHP programs, since OOP comes down to declaring variables and writing PHP statements in functions; it's just that the variables (properties) and functions (methods) are grouped (encapsulated) together in classes, and that the methods will be invoked in response to messages sent to objects of the classes.

Exercises

1. Choose a computer system, such as an online store, an application on your desktop or laptop, or an app on your phone or tablet. Think about some of the classes of objects that system might be using. Choose one class of object and write a list of the pieces of data it might store, as well as some of the methods it might need in order to work on that data.
2. Thinking again of the classes from Exercise 1, try to identify one data property that you would want to restrict access to, so that it could be changed only through a method that would log the change.
3. Consider a computer system for a library. Think of two classes of objects the computer system might use that share several data properties and methods (for example, different types of items the library offers for lending). Now generalize those common properties and methods into a suitably named superclass, and draw a class diagram like [Figure 17-4](#) that shows which properties and methods are inherited from the superclass and which are specific to each subclass.

18

DECLARING CLASSES AND CREATING OBJECTS



In this chapter, you'll learn how to define the structure of a class by using a class-declaration file, and you'll practice creating individual objects of that class. You'll see that classes with public properties let you directly change an object's data, while classes with private properties mean you can change an object's data only via its methods, some of which can perform validation. You'll also learn about PHP “magic” methods that make it easier to write object-oriented code.

Declaring a Class

A *class declaration* defines a class: it lays out the properties (variables) each object of that class will have, as well as the methods (functions) that can act upon those properties. A class declaration also establishes any relationship that class has with other classes (such as inheritance, which you'll learn about in [Chapter 19](#)).

Like function declarations, class declarations are stored in PHP files in the *src* directory of a project. For all the projects in this book, each class will be declared in its own file; if a project has five classes, it will have five class-declaration files, and so on.

NOTE

In this book, we won't explore the advanced topic of anonymous classes, which is one of the few cases where more than one class may be declared in a single file. You can learn more at

<https://www.php.net/manual/en/language.oop5.anonymous.php>.

By a well-established convention in OOP, both class names and class-declaration filenames always start with a capital letter. If the name includes multiple words, each word should start with a capital letter, with no spaces between the words. This is known as *upper camel case*, or sometimes *Pascal case*. Examples of valid class names include `Product`, `NetworkSupplier`, `DesktopComputer`, `ReferenceBook`, and `InventoryItem`.

Throughout this chapter, we'll work with a class called `Product` that can represent various items for sale through an e-commerce site. Let's declare it now. Create a new directory for a new project, and in it create a *src* directory. In this *src* directory, create a *Product.php* file and enter the contents of [Listing 18-1](#).

```
<?php
class Product
{
    public string $name;
    public float $price;
}
```

Listing 18-1: The `Product.php` file to declare the `Product` class

We start with the standard PHP beginning code tag, since we use PHP code to declare classes. Then we use the `class` keyword to state that we're declaring a new class named `Product`. After the class name, enclosed in curly brackets, we define any properties or methods that will be associated with objects of that class. In this example, we declare two properties for

each object of the `Product` class: `name`, which will be a string, and `price`, which will be a float. We declare both properties as `public`, meaning any part of our program with access to a `Product` object can read and change the values of its properties. We'll explore the implications of public properties later in the chapter.

If we want all objects to have a *default value* for a property, we can assign a property a value in the class declaration. For example, if our system set an initial `-1` price for every new `Product` object, we could have written `public float $price = -1`.

[Figure 18-1](#) shows a *Unified Modeling Language (UML)* class diagram visualizing the class we've just written. UML is a common tool for representing classes, objects, and their interactions through diagrams and text.

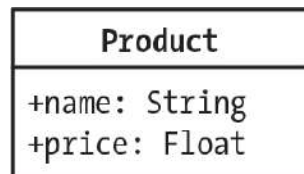


Figure 18-1: The `Product` class

The top row of the diagram indicates the class name (`Product`), and the second row lists the properties associated with that class, along with the data type expected for each property. The plus sign before each property name indicates that the properties have public visibility.

Creating an Object

You use the PHP keyword `new` to create an object of a class. The `new` keyword is followed by the name of the class for which an object is to be created, then a set of parentheses. Inside the parentheses, you may pass arguments for initialization, as we'll discuss in “Initializing Values with a Constructor Method” on [page 346](#). The general form of a statement to create an object is thus `new ClassName()`. Creating an object is also called *instantiation*, since the object is an *instance* of the class.

By writing the `new` keyword and a class name, you're asking PHP to create a new object of the named class. When an object is created in the computer's memory with the `new` keyword, the PHP engine automatically returns a reference to the new object. In most cases, you'll want to store the reference to that newly created object in a variable—for example, `$myObject = new ClassName()`. It's important to understand that with such statements, the variable `$myObject` doesn't actually contain the object itself, but rather a *reference to* the object. It's possible for several variables, or none, to refer to a particular object in memory.

Once you have a reference to an object, use the *object operator* (`->`) to access properties and methods of that object. For example, you could write `$myObject->description` to access the `description` property of the object referred to by `$myObject`. Likewise, you could invoke a `setDescription()` method of an object by writing something like `$myObject->setDescription('small carpet')`. The parentheses or lack thereof are important, since they tell the PHP engine (and people reading the code) whether a statement is attempting to access a property (no parentheses present) or a method (parentheses present).

With all this in mind, let's create an object. We'll write an *index.php* script to read in the *Product.php* class-declaration file, create a `Product` object, and set the values of its properties. [Figure 18-2](#) shows our goal: a `$product1` variable that stores a reference to a `Product` object whose properties have the values `'hammer'` and `9.99`.

<code>\$ product1: Product</code>
<code>name = "hammer"</code> <code>price = 9.99</code>

Figure 18-2: The `$product1` variable references an object of the `Product` class.

For simplicity, we'll start by creating a `Product` object and setting only its `name` property. To make sure our code is working, we'll also print the value of the object's `name` property to the project's home page. In your project directory, create a *public* folder, and in that folder create an *index.php* file containing the code in [Listing 18-2](#).

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product();
$product1->name = 'hammer';

print 'product 1 name = ' . $product1->name;
```

Listing 18-2: An index.php script to create and manipulate a `Product` object

We read in the declaration for the `Product` class, using the `__DIR__` magic constant to create a path from the location of this *index.php* file (in *public*) to the location of the *Product.php* file (in *src*). Then we use the `new` keyword to create a new object of the `Product` class. Since this class doesn’t require any initial values or options when creating objects, we don’t pass any arguments in the parentheses after the class name.

If you aren’t passing any arguments when you create a new object, PHP (unlike most object-oriented languages) allows you to omit the parentheses after the class name. Writing `new Product()` is the same as writing `new Product`. However, there are several good reasons to always include the parentheses, and so this is the style you’ll see throughout this book. Perhaps the most important reason to always use parentheses after the keyword `new` is to remind ourselves that a constructor method may be executed as the new object is created; we’ll look at such methods in “Initializing Values with a Constructor Method” on [page 346](#).

The `new Product()` expression creates the new object and returns a reference to it, which we store in the `$product1` variable. To reiterate, `$product1` doesn’t contain the object itself, nor does it contain a copy of the object. It contains just a *reference* to the object created in the computer system’s memory. In “Object Variables as References” on [page 351](#), we’ll have two variables refer to the same object to help illustrate this concept.

Next, we set the value of the object’s `name` property to the string `'hammer'`; we can do this because the property was declared as `public`. We use the object operator (`->`) after the `$product1` variable to refer to the `name` property of that object.

WARNING

Do not write a dollar sign after the `->` characters: `$product->name` is correct, and `$product->$name` is wrong. The PHP engine won't create a warning or error if you write the latter, but it will interpret the code as meaning there's a variable called `$name` whose value is the name of a property you want to access on the `$product` object. This is very different from accessing the value of the `$product` object's `name` property. If your code is behaving strangely, check for this possible programming mistake.

Finally, the script ends by printing a message featuring the value retrieved from the `$product` object's `name` property. If you run the web server and visit the project's home page, you should see this line of text displayed:

```
product 1 name = hammer
```

The `->` operator lets you manipulate any of an object's public properties by name. Let's update our script to set and display the price of the object as well as its name. Modify the `index.php` file as shown in [Listing 18-3](#).

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product();
$product1->name = 'hammer';

print 'product 1 name = ' . $product1->name;

$product1->price = 9.99;
print ", and price = {$product1->price}";
```

Listing 18-3: Setting and displaying the product price in index.php

We set the object's `price` property to `9.99` following the same format we used in [Listing 18-2](#) to set the `name` property. Then we display the value of the property. Notice that this time we use a double-quoted string to

combine the message and property value. This illustrates that in double-quoted strings, a reference to a public property of an object, such as `$product1->price`, will be parsed and the resulting value will be output, just as with a simple variable.

Visiting the home page again, you should see both the product name and price displayed:

```
product 1 name = hammer, and price = 9.99
```

We've now created an object of the `Product` class, and since the class's properties are public, we were able to set and retrieve the properties' values directly. In practice, however, most classes are written with private rather than public properties.

Private Properties with Public Accessor Methods

When properties are declared as `private`, they can't be accessed by code outside the class declaration itself. Instead, they're available through public *accessor methods*, functions that allow object property values to be retrieved (*getter methods*) or updated (*setter methods*). This mechanism of private properties with public accessor methods reduces the risk of invalid property values; when changes to a property must take place through a setter method, you can implement validation logic as part of the method (for example, preventing negative values or out-of-range values). Also, related properties or other objects might need to be updated together, such as the balance of one bank account being reduced by the same value that another account is increased. With setter methods and private properties, you can easily enforce such rules so that the data in the application stays correct and internally consistent.

The default visibility for class members is `public`, so if no access modifier is provided for a property, the PHP engine will automatically declare it as having public visibility. Even with this default behavior, it's still good practice to explicitly use the `public` access modifier in the class declaration when you want a class member to have public visibility. Otherwise, use the `private` access modifier to make the member private.

NOTE

In addition to `public` and `private`, a third access modifier, `protected`, can be used in conjunction with inheritance. We'll explore this topic in [Chapter 19](#).

For PHP and almost all OOP languages, a getter or setter method's name usually starts with the word *get* or *set*, followed by the property name the method affects, with its first letter capitalized. According to this convention, the getter method for the `name` property of our `Product` class should be `getName()`, and its setter method should be `setName()`. For the `price` property, the methods should be `getPrice()` and `setPrice()`. An exception to this convention is made when a property contains a Boolean `true/false` value. In this case, naming the getter method `isPropertyName` rather than `getPropertyName` is customary. For example, if the `Product` class had a `dangerousItem` property that contained `true` or `false`, its getter would be named `isDangerousItem()`.

A getter method usually returns a value of the same data type as the property it's paired with (although sometimes we have multiple getter methods for different representations of an object's property, such as methods to return both rounded-up integer and float values of a float property). A setter method usually takes in a parameter of the same type and stores its value in the property, perhaps conducting validation checks in the process. Usually, setter methods don't return any value and so are declared to return `void`.

Let's revise the declaration for the `Product` class, making its `name` and `price` properties private and adding four public accessor methods, two for each property. Update the `src/Product.php` file as shown in [Listing 18-4](#).

```
<?php
class Product
{
    private string $name;
    private float $price;

    ❶ public function getName(): string
    {
```



```

        return $this->name;
    }

    ❷ public function setName(string $name): void
    {
        ❸ $this->name = $name;
    }

    public function getPrice(): float
    {
        return $this->price;
    }

    public function setPrice(float $price): void
    {
        $this->price = $price;
    }
}

```

Listing 18-4: Modifying the `Product` class to use getter and setter methods

First, we change the declaration of the two properties to `private`. Then we declare `getName()`, the public getter method for the `name` property ❶. Methods in classes can use the special pseudo-variable `$this` to reference the calling object; that is, `$this` is a stand-in for the object whose properties and methods we’re working with. Our `getName()` method thus returns the value in the `name` property of whichever `Product` object the method is currently being called on. The method has a `string` return type, since the `name` property is a string.

We next declare `setName()`, the public setter method for the `name` property ❷. This method takes in a new `string` `name` value through the `$name` parameter and stores this value in the `name` property for the current object, again using `$this` to reference the object. This setter method returns `void`. The getter and setter methods for `price` follow the same pattern.

Notice in the body of `setName()` how PHP distinguishes between the `$name` parameter and the `name` property for the current object ❸. The former is prefixed by a dollar sign, while the latter is attached to `$this->`

and doesn't have a dollar sign to indicate it's a property of the current object. In other words, `$name` in the `setName()` method unambiguously refers to the value of the argument passed to the method, while the private `name` property of the object that has been sent the `setName()` message is unambiguously referred to by `$this->name`. The same goes for the `float $price` parameter of the `setPrice()` method versus the `price` property of the object the method is being called on.

When you write methods in a class-declaration file, you must always keep in mind that the same methods may be executed on zero, one, or thousands of objects, in response to objects receiving a message with the name of the method (and any required arguments). Although you may plan to create and use only one instance (object) of a class when you first write the declaration, a well-written class encapsulates the data (properties) and behavior (methods) for *any* object of that class. When you keep the general use in mind while programming, you can often use a class in other parts of the same project, or different projects altogether, with few or no changes required to the class declaration. Well-written class declarations lend themselves to reuse.

NOTE

While you can type out accessor methods yourself, many code editors, including PhpStorm, offer an automated feature to generate simple getter and setter methods for you. Autogenerating code is faster than typing it out by hand, and it ensures error-free scripts that follow PHP programming conventions.

Getting and Setting Private Properties

Since the two properties of any `Product` object are now declared `private`, we can't access them directly, such as by writing `$product1->name` or `$product1->price`. If you run the existing `index.php` script, you'll get a fatal error about not being able to access the private `name` property. Instead, we have to read and modify these private properties by using their public accessor methods. [Listing 18-5](#) shows how to update `index.php` to make use of these new methods.

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product();
$product1->setName('hammer');
$product1->setPrice(9.99);
print 'product 1 name = ' . $product1->getName();
print ", and price = {" . $product1->getPrice() . "}";
```

Listing 18-5: Using accessor methods in index.php

As in [Listing 18-3](#), we create the `$product1` object, set its properties, and print out those properties. This time, however, we rely entirely on accessor methods. We use setter methods to update the values of the object properties, such as `$product1->setName('hammer')`. Likewise, we use getter methods to retrieve values from the object, such as `$product1->getName()`. Thanks to these methods, the data in the `$product1` object is safely encapsulated but still accessible.

Screening for Invalid Data

One of the advantages of protecting the data properties of objects is that you can add validation logic to the setter methods to prevent invalid values from being stored in the properties. For example, most businesses probably wouldn't want to set a negative price for a product (although something might be a free gift, so we'll allow for a price of 0). We should therefore add an `if` statement to the `setPrice()` method that updates the stored price only if the new value is greater than or equal to 0. [Listing 18-6](#) shows how to update the method in `src/Product.php`.

```
--snip--
public function setPrice(float $price): void
{
    if ($price >= 0) {
        $this->price = $price;
    }
}
```

Listing 18-6: Adding validation logic to the `setPrice()` method of the `Product` class

In our validation logic, we confirm that the new `$price` argument is greater than or equal to 0 before setting the value of the object's `price` property. To make sure the validation check works, we can update our *index.php* script to attempt to set an invalid, negative price value. We should see that the invalid values aren't stored in the object. [Listing 18-7](#) adds extra statements to *index.php* for two tests of the validation logic.

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product();
$product1->setPrice(9.99);

print "(initial value) product 1 price = {"$product1->getPrice()}\\n";

$product1->setPrice(-0.5);
print '<br>(test 1) trying -0.5: ';
print "product 1 price = {"$product1->getPrice()}\\n";

$product1->setPrice(22);
print '<br>(test 2) trying 22: ';
print "product 1 price = {"$product1->getPrice()}\\n";
```

*Listing 18-7: Testing the setter validation logic in *index.php**

As before, we create a new `Product` object and set its `price` to 9.99. Then we try to set the price to an invalid negative value, and then a valid positive value that's different from its initial value, printing the product price each time. Here's the output of this script in the browser:

```
(initial value) product 1 price = 9.99
(test 1) trying -0.5: product 1 price = 9.99
(test 2) trying 22: product 1 price = 22
```

For test 1 (a negative price of `-0.5`), the stored price remains unchanged at `9.99`. For test 2 (a nonnegative value of `22`), the stored price is updated. Our validation logic has worked. In this example, we simply ignored the invalid value, but it's generally better to somehow indicate that there was a problem. One option is for setters to return a Boolean `false` value when no value was set. Another option is to throw an `Exception` object, as we'll explore in [Chapter 23](#).

Overriding Default Class Behavior with Magic Methods

PHP offers several *magic methods* that override default behavior of an object. For example, the `__construct()` magic method overrides the default way objects of a class are created, and the `__toString()` magic method overrides the way objects are handled in `print` statements and other contexts requiring a string. We'll explore each of these magic methods in this section.

Despite their name, magic methods are unrelated to PHP magic constants. Magic methods are a feature of object-oriented PHP, allowing the default behavior of an object to be changed. All magic methods have names beginning with double underscore characters (`__`); therefore, you should name methods with such a prefix only when declaring a magic method for a class. You can find a list of all the PHP magic methods at <https://www.php.net/manual/en/language.oop5.magic.php>.

Initializing Values with a Constructor Method

It's common to want to set some (or all) of an object's properties as soon as that object has been created. As shown in [Listing 18-5](#), you can do this by first creating an object and then having a sequence of statements invoking setter methods to set values for each property. However, initializing object properties immediately after creating an object is such a common requirement that PHP enables you to combine these actions into a single step by writing a magic method called a *constructor* as part of the class declaration.

Every class-declaration file either declares no constructor method (as you've seen so far in this chapter) or declares a single constructor magic method named `__construct()`. It's magic in the sense that it overrides the

default way of creating an object: creating it without setting any of its properties. The `__construct()` method takes in a series of parameters and assigns them as initial values of the newly created object's properties. Using a constructor method in an *index.php* file is as simple as providing the initial values as arguments in the parentheses after the class name:

`$myObject = new ClassName($value1, $value2)`, for example. Thanks to the use of the `new` keyword, PHP automatically links the arguments with the constructor, even though `__construct()` isn't called explicitly.

NOTE

PHP is quite unusual as an object-oriented language in that the constructor method doesn't have the same name as the class. In most other object-oriented languages, a `Product()` method in the `Product` class would be a constructor method, but in PHP, there's nothing special about a method that has the same name as the class in which it's declared.

Setting properties as part of the constructor method can save some code when it comes to creating new objects. For example, if we know we'll want to set the `name` and `price` properties upon creation of a `Product` object, we can add a constructor method to the `Product` class that takes in `$name` and `$price` arguments to set these properties automatically. That way, when we create our `$product1` object in *index.php*, we can replace these three statements

```
$product1 = new Product();  
$product1->setName('hammer');  
$product1->setPrice(9.99);
```

with just a single statement:

```
$product1 = new Product('hammer', 9.99);
```

Update *Product.php* as shown in [Listing 18-8](#) to add a constructor method that sets the `name` and `price` properties.

```
<?php
class Product
{
    private string $name;
    private float $price;

    public function __construct(string $name, float $price)
    {
        $this->setName($name);
        $this->setPrice($price);
    }

    public function getName()
}
--snip--
```

Listing 18-8: Adding a constructor method to the `Product` class

We declare a new `__construct()` method. It replaces the default no-parameter creation of an object via `new Product()` with a method requiring two parameters: the initial string name and float price values for the new `Product` object. Note that constructor methods don't specify any return type. Within the `__construct()` method definition, we call the `setName()` and `setPrice()` methods, which we've already defined elsewhere in the `Product` class declaration, feeding them the `$name` and `$price` parameters. This may not seem easier than calling those methods in the *index.php* script, but as you start creating more instances of the same object, setting properties through the constructor quickly becomes much more efficient. This approach also ensures that exactly the same validation is applied when values are set at the time of object construction as when values are changed at a later time with a direct call to a setter method.

NOTE

Many IDEs (such as PhpStorm) offer an interactive constructor method generator that enables you to add selected properties as parameters and have their values set by the generated constructor method code.

Listing 18-9 shows how to simplify *index.php* to take advantage of the new constructor method.

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product('hammer', 9.99);
print 'product 1 name = ' . $product1->getName();
print ", and price = {$product1->getPrice()}";
```

Listing 18-9: The simpler index.php script, using the constructor method

When we create the `$product1` object, we pass the desired initial values for the `name` and `price` properties as arguments for the constructor. As noted previously, this collapses three lines of code (creating the object and setting each of its two properties) into a single line.

CONSTRUCTOR PROPERTY PROMOTION

PHP 8 introduced *constructor property promotion*, which allows you to declare a class's properties and pass the initial values of those properties in the constructor method, both as a single step. This can make your class declaration files much more concise. This technique takes the place of separately declaring a private property and initializing it in the constructor method. Instead, you write just a constructor with a `private` argument and no body.

Without constructor property promotion, you'd have to set a class's `name` property through the constructor like this:

```
private string $name;

public function __construct(string $name)
{
    $this->name = $name;
}
```

With constructor property promotion, those lines become a single constructor declaration with no method body:

```
public function __construct(private string $name){}
```

Until you're very familiar with OOP, I recommend that you keep your class property declarations separate from your constructor method code. This separation helps you see at a glance which properties each object of the class will have, and if you want to, you can look at the code for the constructor to see which (if any) can be initialized when a new object is created. Constructor property promotion also isn't appropriate when new object values should be validated through calls to setter methods, as in [Listing 18-8](#).

Converting Objects to Strings

It's common to want to summarize the contents of an object as a string, sometimes to display details about the object, or sometimes for debugging and logging purposes. One common reason to convert objects to strings is to generate a list of objects for a web interface, such as a drop-down menu. [Figure 18-3](#) shows an example drop-down menu with a list of some of the courses I teach.



Figure 18-3: A list of courses summarized as strings

You can imagine that each of these courses is represented in PHP by a `Course` object, which has properties like `courseNumber` and `courseName`. To generate the drop-down menu, PHP converts each `Course` object to a string in the form `courseNumber - courseName`, such as `COMP H2029 - Web Development Server-Side`. These strings can then be fed into the HTML code for displaying the menu.

How does that conversion to a string happen? Most object-oriented languages, including PHP, offer a way to implement a special method to return a string when an object is used in an expression that requires a string (for example, something like `print $course1`, where `$course1` is a reference to a `Course` object). In PHP, this functionality comes from another magic method prefixed with two underscore characters: `__toString()`.

You don't *have* to implement a `__toString()` method for every class, but if you know you'll need a string summary of an object (such as for a drop-down HTML menu), or if you want to log details about objects to a report, then `__toString()` methods are useful. If a class has no `__toString()` method and you try to reference an object of that class in an expression requiring a string, you'll get a `could not be converted to string fatal error`. Let's see this happen by replacing the `print` statements at the end of our *index.php* script with `print $product1`. Update *index.php* to match [Listing 18-10](#).

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product('hammer', 9.99);
print $product1;
```

*Listing 18-10: Trying to output details of an object via `print` in *index.php**

We pass the expression `$product1` to a `print` statement. Because `print` statements expect a string expression and `$product1` isn't a string, PHP will try to convert it to one. Since the PHP engine can't convert an object reference to a string without a `__toString()` method, a fatal error occurs.

Let's now implement a `__toString()` method for our `Product` class, both to explore this common feature of OOP and to allow us to use the simplified *index.php* script in [Listing 18-10](#). [Listing 18-11](#) shows the new `__toString()` method added to the *src/Product.php* file.

```
<?php
class Product
```

```

{
    private string $name;
    private float $price;

    public function __construct(string $name, float $price)
    {
        $this->setName($name);
        $this->setPrice($price);
    }

    public function __toString(): string
    {
        ❶ return '(Product) name = ' . $this->name
            . ', and price = ' . $this->price;
    }
    public function getName(): void
--snip--

```

Listing 18-11: Adding a `__toString()` method to the `Product` class

We add a new `__toString()` method to the class. It contains a single statement that builds and returns a string summarizing the object property values. Note that we generalized the string message to start with `'(Product) '` rather than `'product 1 '` ❶. Since this is a method of a class and therefore will potentially be used by many objects, we shouldn't hardcode the name of the variable referring to a particular object into the general class declaration file.

Run the `index.php` script as it was updated in [Listing 18-10](#), and you should see that the `print $product1` statement works correctly, thanks to the new `__toString()` method.

Object Variables as References

As noted earlier, the `$product1` variable used throughout this chapter is a reference to a `Product` object in memory, not a `Product` object itself. One implication of this distinction is that more than one variable can reference the same object in memory. This can occur in lots of ways. For example, it would happen when you need to loop through a collection of objects and

perform actions on each. In this case, a temporary local variable would reference the current object being worked on, but the collection would also still have a separate reference to that object.

To see how object variables are just references to locations in memory, update *index.php* as shown in [Listing 18-12](#). In this code, we create `$variable2`, make it a reference to the same object as `$product1`, and change one of the object's properties through `$variable2`. As you'll see, this change impacts the object referenced by `$product1` as well, proving both variables are referencing the same object.

```
<?php
require_once __DIR__ . '/../src/Product.php';

$product1 = new Product('hammer', 5.00);
print $product1;
print '<br>';

❶ $variable2 = $product1;
print 'changing price via $variable2';
print '<br>';
$variable2->setPrice(20.00);
print $product1;
```

Listing 18-12: Updating index.php to illustrate how object variables are references

We make `$variable2` a reference to the same object as `$product1` ❶. Then we call `setPrice()` to the object that `$variable2` refers to, changing the value of the object's `price` property to `20.00`. We then print `$product1` a second time. Since `$product1` is a reference to an object, its `__toString()` method will be invoked. This produces the following output in the browser:

```
(Product) name = hammer, and price = 9.99
Changing price via $variable2
(Product) name = hammer, and price = 20
```

The object referenced by `$product1` has had its price changed to 20, even though we made the price change through `$variable2`. Therefore, the two variables must be referencing the same object.

Handling Missing Objects

Sometimes code is written in such a way that you expect a variable to refer to an object, but no such object is found. That variable would be `NULL`, so it's often important to include checks for `NULL` when you're writing object-oriented code.

Let's consider an example. Imagine you're writing code for a blog. To display a particular blog post, the code expects a valid ID of a blog post from an HTTP request, then uses that ID to retrieve data from a database and construct a `Blog` object. If no ID is found in the request, if the ID is invalid, or if the ID doesn't match any item in the database, then the application can't create a `Blog` object, and so the code would return `NULL` instead of an object reference.

To account for this situation, other code expecting to work with a `Blog` object would first test for `NULL` and then decide whether to deal with an invalid ID (say 0 or negative) or with a successfully retrieved `Blog` object. [Listing 18-13](#) shows an example method that might come from a database-driven blog website to illustrate this point.

```
<?php
--snip--

public function blogFromId (int $id): ?Blog
{
    ❶ if (is_numeric($id) && $id > 0) {
        return $this->blogRepository->find($id);
    }

    return NULL;
}

--snip--
```

Listing 18-13: Using a nullable return type

This `blogFromId()` method takes in a value for an `$id` and returns either a reference to a `Blog` object or `NULL`, using the nullable return type `? Blog`. (We could also have written this as union return type `Blog|NULL`.) The method tests whether `$id` is numeric and greater than 0 ❶. If so, it passes the valid `$id` to the `find()` method of the `blogRepository` property and returns the value from this method (either `NULL` or the `Blog` object found for this ID in the database). If the `$id` isn't valid, `NULL` is returned.

This example is making lots of assumptions, but the point is that the variable set to the result of calling the `blogFromId()` method will either have a reference to an object or be `NULL`. Code like this is quite common in OOP (as you'll see in [Part VI](#)), which is why you often test for a `NULL` value of a variable you expect to be a reference to an object, to identify whether any object is being referred to. This compares to working with non-object-oriented PHP variables, where `NULL` can mean, for example, that a variable hasn't been initialized or that no string value was received for a URL-encoded variable in an HTTP form submission.

Custom Methods and Virtual Attributes

You can write all sorts of custom methods for a class, beyond the standard getters and setters and the `__construct()` and `__toString()` magic methods. Remember, methods are simply functions attached to a class of objects, so *custom methods* are functions to implement logic and calculations relating to objects of the class. For example, our `Product` class might come with a method for calculating the total price of a product, including tax. The tax rate will be a float value, such as 0.5 (for 50 percent). Such a method would still be functioning as a getter, but instead of simply returning a stored property value, it would be dynamically calculating a value each time it's invoked.

To see how it works, we'll add a `getPriceIncludingTax()` method to our `Product` class declaration. The method will retrieve the tax rate and the pretax price of a product from the appropriate object properties, perform the necessary calculation, and return the total price with tax. For a tax rate of 0.1 (10 percent) and a price of 5.00, for instance, the method should return

1.1 * 5.00 = 5.50. To create the method, we also need to add a private `taxRate` property to the class, along with accessor methods for setting and getting the tax rate for a product.

Listing 18-14 shows an updated *Product.php* class-declaration file. In addition to adding the `taxRate` property, its accessors, and the custom method, we also modify the `__toString()` method to display the results of the tax calculation.

```
<?php
class Product
{
    private string $name;
    private float $price;
    ❶ private float $taxRate;

    public function __construct(string $name, float $price)
    {
        $this->setName($name);
        $this->setPrice($price);
    }

    public function __toString(): string
    {
        return '(Product) name = ' . $this->name
            . ', and price = ' . $this->price
            . ', and price with Tax = ' . $this->getPriceIncludingTax();
    }

    ❷ public function getTaxRate(): float
    {
        return $this->taxRate;
    }

    public function setTaxRate(float $taxRate): void
    {
        $this->taxRate = $taxRate;
    }
}
```

```

    ❸ public function getPriceIncludingTax(): float
    {
        return (1 + $this->taxRate) * $this->price;
    }

    public function getName()
--snip--

```

Listing 18-14: Adding the `taxRate` property and associated methods to the `Product` class

We declare the `taxRate` property ❶ along with its simple getter and setter methods ❷. Then we declare the `getPriceIncludingTax()` method ❸. It returns the price with the tax rate factored in.

As you can see, our `getPriceIncludingTax()` custom method is simply a function that performs a useful calculation for our class. In this case, it's essentially an extra getter method that provides a variation on one of the class's stored properties, `price`. In fact, it's quite common in OOP to see what amounts to multiple getter methods for the same property of an object: methods that return pre- and post-tax prices of a product, methods that return the same property with different levels of precision (rounded to the nearest whole number versus including up to two decimal places), methods that retrieve the same property converted to different currencies or units (dollars versus euros, feet versus meters), and so on.

In other cases, custom methods can act as *virtual attributes*: rather than provide a variation on an existing property, such methods perform calculations to arrive at a completely new piece of information. An example of a virtual attribute might be a method to calculate the age of a product. If products had a `dateReceived` property, the age of a product could be dynamically calculated as part of a `getProductAge()` method. The method would subtract `dateReceived` from the current date. In this case, the product's age isn't actually stored as a property of the object, but thanks to the `getProductAge()` method, the information is available as if it were a property.

Custom methods highlight some of the power of OOP: the person writing code that uses a public `getProductAge()` method of a `Product`

object doesn't need to worry about how that method is implemented. All that counts is that the method works. If the implementation of the method is changed (perhaps changing the data type of the `dateReceived` property from stored MySQL datetime values to Linux timestamps) but its behavior remains correct and unchanged, it makes no difference to the parts of the system that are sending messages to `Product` objects and using the values returned by those methods.

Summary

This chapter covered how to declare classes, how to read those declarations into an *index.php* file and use them to create objects, and how to invoke methods of objects to set and retrieve their property values. You saw how to protect an object's data properties by declaring them as `private`, and how to use getter and setter methods declared as `public` to manage access to the object's properties and perform validation where relevant. We also discussed how to perform common useful actions with PHP "magic" methods, such as creating new objects with some properties initialized via a constructor method and generating a string message representing an object's properties by declaring a `__toString()` method.

Exercises

1. Write a PHP class declaration for a `Cat` class, with public properties of `name`, `breed`, and `age`. Then write an *index.php* file to read in the class declaration and create a `Cat` object. Store a reference to the new object in a variable named `$cat1` and set its properties as follows:

```
name = 'Mr. Fluffy'
breed = 'long-haired mix'
age = 2
```

Finally, add statements to print the data values for each property of `$cat1`.

2. Write a PHP class declaration for a `Pet` class, with a private `name` property and public get and set accessor methods for this `name` variable. Then write an *index.php* file to read in the class declaration and create a `Pet` object referenced by a variable named `$pet1`. Use the setter to set

its name to 'Fifi', and add a statement to print the name stored in this object.

3. Add a constructor method to your `Pet` class so you can create new `Pet` objects with an initial value of the `name` variable by using a statement like this:

```
$pet1 = new Pet('Mr. Fluffy');
```

Update your *index.php* file to use this constructor method rather than setting the name with the setter method.

4. For the following properties and types, write their corresponding accessor (getter/setter) method names:

```
age // integer
houseNumber // integer
color // string
length // float
heavy // bool
```

19

INHERITANCE



This chapter covers perhaps the most powerful and important feature of OOP: *inheritance*. This is the capability of one or more classes, called *subclasses*, to automatically have all the same properties and methods as another class, called a *superclass*. Inheritance makes OOP more efficient: you have to define any general, shared members only once, in the superclass. Objects of the subclasses inherit those properties and methods, and they will also have whatever properties and methods are specific to the individual subclasses.

Associated with inheritance is the capability of subclasses to *override* inherited methods from their superclass when that general method isn't appropriate to the particular subclass. As we'll explore in this chapter, you can also have the best of both worlds (executing the inherited method *and* adding additional behavior through the subclass's method) by using the `parent` keyword. Additionally, we'll discuss the third kind of protection PHP offers for methods and properties: `protected`. You'll learn what

distinguishes protected members of a class from public and private members, and when to use each visibility designation.

Inheritance as Generalization

Inheritance simplifies code by identifying and generalizing properties and behaviors shared among classes of objects. Consider the properties identified for `Car` and `Boat` objects shown in [Figure 19-1](#).

Car	Boat
makeModel: String numPassengers: Integer topSpeed: Float bodyShape: String	makeModel: String numPassengers: Integer topSpeed: Float countryOfRegistration: String

Figure 19-1: The `Car` and `Boat` classes, with common properties shown in bold

The first three properties of each class are identical: both `Car` and `Boat` objects have a make and model, a number of passengers, and a top speed. To avoid redundancy, we can generalize these common properties (and any associated methods) into a superclass, which we'll name `Vehicle`. [Figure 19-2](#) shows this generalized `Vehicle` class with the three shared properties, as well as the simplified `Car` and `Boat` classes, each now with just one property of its own.

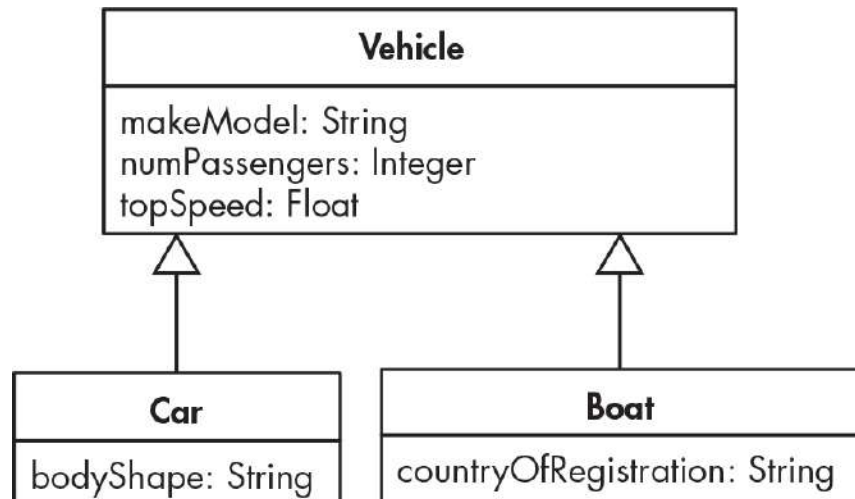


Figure 19-2: The *Vehicle* superclass and the *Car* and *Boat* subclasses

The arrows going from *Car* and *Boat* to *Vehicle* indicate that both *Car* and *Boat* inherit from *Vehicle*; that is, *Car* and *Boat* are subclasses of the generalized *Vehicle* superclass. Notice that the visibility of the properties (public versus private) hasn't been indicated in the diagram. For now, we'll implement the classes by using simple public properties. Later in the chapter, we'll refactor the classes by using the third visibility setting, *protected*.

Listing 19-1 implements the *Vehicle* class shown in [Figure 19-2](#). Start a new project and create *src/Vehicle.php* containing the code in this listing.

```
<?php
class Vehicle
{
    public string $makeModel;
    public int $numPassengers;
    public float $topSpeed;
}
```

Listing 19-1: Implementing the *Vehicle* superclass

We declare the *Vehicle* class with three public properties: *makeModel*, *numPassengers*, and *topSpeed*.

Listing 19-2 implements the `Car` subclass of `Vehicle`, which should be created in *src/Car.php*.

```
<?php
class Car extends Vehicle
{
    public string $bodyShape;
}
```

Listing 19-2: Implementing `Car` as a subclass of `Vehicle`

We declare that `Car` should be a subclass of `Vehicle` simply by adding the `extends` keyword followed by the superclass name at the start of the class declaration. Any `Car` objects we create will automatically inherit all the properties and methods from the `Vehicle` superclass, in addition to having the `bodyShape` property declared directly in the `Car` class.

Creating Objects from Subclasses

To verify that our example of inheritance works, we'll create a `Car` object and set values for its properties—both the one declared directly in `Car` and the ones inherited from `Vehicle`. Create *public/index.php* containing the code in [Listing 19-3](#).

```
<?php
require_once __DIR__ . '/../src/Vehicle.php';
require_once __DIR__ . '/../src/Car.php';

$car1 = new Car();

$car1->bodyShape = 'Sedan';
$car1->makeModel = 'Ford Mustang';
$car1->numPassengers = 5;
$car1->topSpeed = 150;

var_dump($car1);
```

*Listing 19-3: Creating a `Car` object in *index.php**

We first read in the files declaring the `Vehicle` and `Car` classes. It's imperative to read in declarations for *all* the classes the code uses, including inherited classes such as `Vehicle`, even though we won't be directly creating objects of this class. The order we read in the files is important too: for the `Car` class to be able to extend the `Vehicle` class, we must have read in and executed the declaration for the `Vehicle` class *before* reading in and declaring the `Car` class. Otherwise, we'll get an error about the `Vehicle` class not being found.

NOTE

Large projects may require tens, or even hundreds, of class declarations. In [Chapter 20](#), we'll look at a way to read in all the class declarations in the required sequence with a single statement, using a script called an autoloader.

Next, we create a new `Car` object and store a reference to it in the `$car1` variable. Then we set values for the `bodyShape`, `makeModel`, `numPassengers`, and `topSpeed` properties. The syntax is the same whether the property was declared for the subclass or inherited from the superclass: either way, we simply use `$car1->propertyName = value`. Finally, we use `var_dump()` to output structured information about the `$car1` variable and the object it refers to in memory. Here's the result of running the index script at the command line:

```
$ php public/index.php
object(Car)#1 (4) {
    ["bodyShape"]=>
    string(5) "Sedan"
    ["makeModel"]=>
    string(12) "Ford Mustang"
    ["numPassengers"]=>
    int(5)
    ["topSpeed"]=>
    float(150)
}
```

The `Car` object indeed has four properties, and we've successfully set all four, including the property declared in *Car.php* (`bodyShape`) and the three properties inherited from the `Vehicle` superclass.

Let's complete our implementation of the class hierarchy from [Figure 19-2](#) by declaring the `Boat` subclass. Create *src/Boat.php* and enter the code in [Listing 19-4](#).

```
<?php
class Boat extends Vehicle
{
    public string $countryOfRegistration;
}
```

Listing 19-4: Implementing the `Boat` class

We declare that `Boat` should inherit from `Vehicle`, once again using the `extends` keyword to establish the subclass/superclass relationship. The class has only one unique property, `countryOfRegistration`, but `Boat` objects will also inherit the three public properties of `makeModel`, `numPassengers`, and `topSpeed` from `Vehicle`.

Using Multiple Levels of Inheritance

Class hierarchies can involve many levels of inheritance: class `A` could have a subclass `B`, class `B` could have a subclass `C`, and so on. In this case, class `C` would inherit methods and properties from both class `B`, its immediate superclass, and from class `A`, the superclass of its superclass. [Figure 19-3](#) illustrates how we might use this mechanism to further extend our hierarchy of vehicular classes.

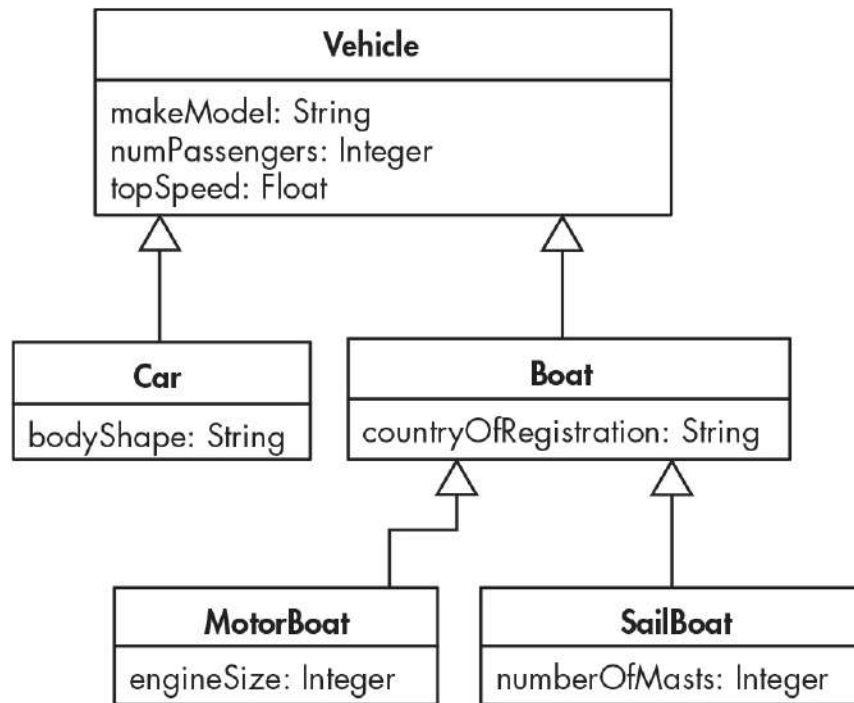


Figure 19-3: A three-level class hierarchy, with subclasses of subclasses

The class hierarchy features two subclasses of `Boat`: `MotorBoat` and `SailBoat`. All `MotorBoat` and `SailBoat` objects will inherit the properties and methods of `Boat`, their superclass. That includes the properties and methods that `Boat` inherits from `Vehicle`. Meanwhile, `MotorBoat` objects will also have a special property of their own, `engineSize`, while `SailBoat` will have a unique `numberOfMasts` property.

When it comes time to declare the `MotorBoat` class, we'd start by writing `class MotorBoat extends Boat`. We don't need to also mention `Vehicle` in the `MotorBoat` class declaration; the inheritance from `Vehicle` is already covered in the `src/Boat.php` class-declaration file.

Like most object-oriented languages, PHP permits each class to have only a single direct superclass. This safeguard prevents ambiguities as to where a property or method is being inherited from, but it also adds challenges when it comes to designing class hierarchies. For example, since cars and motorboats both have engines, it might make sense to create a `MotorizedVehicle` subclass of `Vehicle`. The `Car` class would naturally inherit from `MotorizedVehicle`, but what about `MotorBoat`? Should it inherit from `Boat` or `MotorizedVehicle`? It can inherit directly from only

one or the other. As this example illustrates, once class hierarchies get more complex, care must be taken in their design.

Protected Visibility

With the introduction of inheritance, a third visibility keyword, besides `public` and `private`, becomes relevant: `protected`. When a method or property is declared `protected`, it becomes accessible to any subclasses that inherit it, but it can't be accessed by code elsewhere in the project. For example, a `protected` property of the `Vehicle` superclass could be accessed within `Car.php`, since `Car` inherits from `Vehicle`. By contrast, that `protected` property couldn't be accessed within the general `index.php` file.

The `protected` visibility designation is more restrictive than `public`, since `public` properties and methods can be accessed from anywhere within the project code. On the other hand, `protected` is less restrictive than `private`. When a method or property is `private`, it's accessible only within the class itself; not even methods of subclasses can directly access `private` properties and methods from their superclass.

To illustrate the distinction between the three visibility keywords, we'll refactor the `Vehicle` class. To begin, we'll update the class to have `private` properties and `public` accessor methods, as we discussed in [Chapter 18](#). Then we'll add the `protected` designation into the mix. [Listing 19-5](#) shows the first changes to `src/Vehicle.php`.

```
<?php
class Vehicle
{
    private string $makeModel;
    private int $numPassengers;
    private float $topSpeed;

    ❶ public function getMakeModel(): string
    {
        return $this->makeModel;
    }

    public function setMakeModel(string $makeModel): void
```

```

        {
            $this->makeModel = $makeModel;
        }

        public function getNumPassengers(): int
        {
            return $this->numPassengers;
        }

        public function setNumPassengers(int $numPassengers): void
id
        {
            $this->numPassengers = $numPassengers;
        }

        public function getTopSpeed(): float
        {
            return $this->topSpeed;
        }

        public function setTopSpeed(float $topSpeed): void
        {
            $this->topSpeed = $topSpeed;
        }
    }
}

```

Listing 19-5: *Revising the `Vehicle` class to have private properties and public accessor methods*

We declare all three class properties as `private` rather than `public`. This means they can't be set directly. Instead, we've declared public getter and setter methods for each property ❶.

We now need to update the index script to use the setter methods to set values for the `Car` object. Modify *public/index.php* to match the code in [Listing 19-6](#).

```

<?php
require_once __DIR__ . ' ../src/Vehicle.php';
require_once __DIR__ . ' ../src/Car.php';

```

```
$car1 = new Car();
```

```
❶ $car1->bodyShape = 'Sedan';
```

```
$car1->setMakeModel('Ford Mustang');
```

```
$car1->setNumPassengers(5);
```

```
$car1->setTopSpeed(150);
```

```
var_dump($car1);
```

Listing 19-6: Updating index.php to set inherited properties via setter methods

We can still directly access the `bodyShape` property declared in the `Car` class ❶, since it's still set as `public`. However, we now need to use the setter methods to assign values to the three inherited properties, `makeModel`, `numPassengers`, and `topSpeed`, since these properties are now declared as `private` in the `Vehicle` superclass. The `Car` object inherits the accessor methods from `Vehicle`, just as it inherits the properties themselves. Thanks to the accessor methods, `var_dump()` will work as before.

Using public setter methods to update private properties within *index.php* makes sense; this is exactly what we did in [Chapter 18](#). But what if we want to use one of the properties inherited from `Vehicle` as part of a method declared in the `Car` class? Let's try this and see what happens. We'll add a new `getMakeModelShape()` method to `Car` that returns a string summary of some of the object's properties. Update *src/Car.php* to match the code in [Listing 19-7](#).

```
<?php
class Car extends Vehicle
{
    public string $bodyShape;

    public function getMakeModelShape(): string
    {
        return "$this->makeModel, $this->bodyShape";
    }
}
```

```
}  
}
```

Listing 19-7: Trying to access private superclass properties from a subclass

We declare the new `getMakeModelShape()` method in the `Car` class. This method attempts to insert the values of the `makeModel` and `bodyShape` properties into a string. To see whether this method works, we'll invoke it in our index script. Update *public/index.php* as shown in [Listing 19-8](#).

```
<?php  
require_once __DIR__ . '/../src/Vehicle.php';  
require_once __DIR__ . '/../src/Car.php';  
  
$car1 = new Car();  
  
$car1->bodyShape = 'Sedan';  
  
$car1->setMakeModel('Ford Mustang');  
$car1->setNumPassengers(5);  
$car1->setTopSpeed(150);  
  
print $car1->getMakeModelShape();
```

Listing 19-8: Testing the `getMakeModelShape()` method in index.php

We've replaced the `var_dump()` function call with a `print` statement invoking the `getMakeModelShape()` method of our `$car1` object reference. However, if you now run the index script, you'll get an `Undefined property` warning, as shown here:

```
$ php public/index.php
```

```
Warning: Undefined property: Car::$makeModel in /Users/matt/  
src/Car.php  
on line 10  
, Sedan
```

Statements in our `Car` class methods have no access to inherited private properties, so the PHP engine simply can't find a `makeModel` property when it executes `getMakeModelShape()`. Notice, however, that the warning message is followed by the text `, Sedan`. This indicates that the index script continued executing despite the issue. Indeed, after continuing from its undefined property warning, the PHP engine had no problem with the `$this->bodyShape` portion of the `getMakeModelShape()` method, since `bodyShape` is a property defined directly on the `Car` class itself.

One way for us to access the value of an inherited property could be to use the public getter method to access the `makeModel` property, but in some situations it's best not to offer public getter or setter methods for a property. A better solution when you want to give methods of a subclass direct access to a property or method inherited from its superclass, without making that property or method public, is to give the property or method `protected` visibility. For that, update the `Vehicle` class as shown in [Listing 19-9](#).

```
<?php
class Vehicle
{
    ❶ protected string $makeModel;
    private int $numPassengers;
    private float $topSpeed;

    public function getMakeModel(): string
    {
        return $this->makeModel;
    }
}

--snip--
```

Listing 19-9: Using `protected` visibility for `makeModel` in the `Vehicle` class

We now declare the `Vehicle` property `makeModel` to have `protected` visibility ❶, which means it can be accessed directly by statements in the `Car` subclass. Re-execute the index script and you'll see the make, model, and shape string output with no warnings: `Ford Mustang, Sedan`.

STEREOTYPING

UML allows us to introduce visual language abbreviations to simplify common code design features through a feature called stereotyping. Stereotypes are indicated by double chevron angle brackets (called *guillemets*) in the form «*StereotypeName*». So, rather than laboriously listing public getter and setter methods for each property in our class diagrams, we can indicate that a property should have a getter and/or setter method by annotating the property with a stereotype. For example, «*get/set*» indicates that a property should have both a getter and setter public method.

[Figure 19-4](#) shows the UML diagram for our three classes, updated to indicate the visibility of each class property and method.

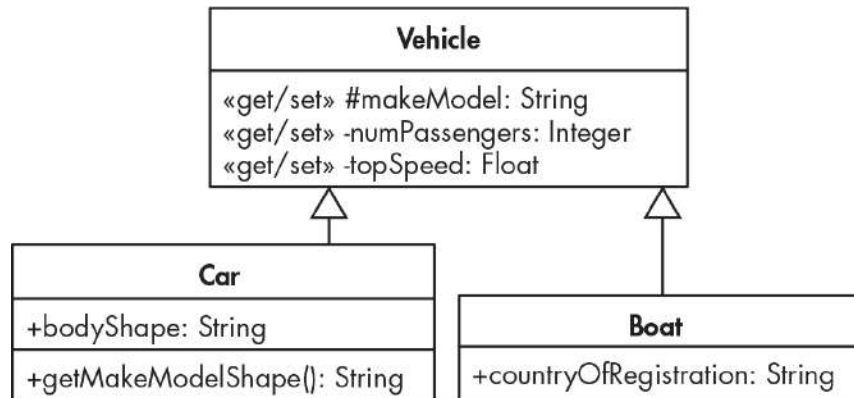


Figure 19-4: An updated UML diagram showing the visibility settings for the class hierarchy

Within the `Vehicle` class, the hash mark (#) indicates the protected visibility of the `makeModel` property, while the other two properties are shown as private with minus signs. All properties and methods for classes `Car` and `Boat` are public and so indicated with plus signs. The «*get/set*» annotations in the diagram indicate public getter and setter accessor methods for all three properties of the `Vehicle` class.

This example demonstrates the distinction between `private` and `protected`. To illustrate the difference between the `public` and `protected` visibility of properties, let's modify the `public/index.php` script to try to directly set the value of the `makeModel` property. Update the index script to match [Listing 19-10](#).

```
<?php
require_once __DIR__ . '/../src/Vehicle.php';
require_once __DIR__ . '/../src/Car.php';

$car1 = new Car();

$car1->bodyShape = 'Sedan';
$car1->makeModel = 'Ford Mustang';

$car1->setNumPassengers(5);
$car1->setTopSpeed(150);

print $car1->getMakeModelShape();
```

Listing 19-10: Illustrating the difference between *public* and *protected* visibility

We set the value of the public `bodyShape` property to 'Sedan' as before. Then we attempt to directly change the value stored in the `makeModel` property of `$car1` to the string 'Ford Mustang'. Execute this script and you'll get a fatal error, as shown here:

```
$ php public/index.php
```

```
Fatal error: Uncaught Error: Cannot access protected property Car::$makeModel
in /Users/matt/public/index.php:13
Stack trace:
#0 {main}
  thrown in /Users/matt/public/index.php on line 13
```

The `makeModel` property has *protected* visibility, so it can't be accessed from our index script. Only subclasses of `Vehicle` can access the property directly.

Abstract Classes

An *abstract class* is one that will never be used to create objects. Instead, abstract classes are commonly declared so their members (properties and

methods) can be inherited by subclasses. The `Vehicle` class, for example, is an abstract class. We'll never want to create a `Vehicle` object; we declared the class only to generalize common properties and methods from the `Car` and `Boat` classes.

Another use of abstract classes is to declare static members, properties, or methods that relate to the class as a whole rather than to individual objects. We'll explore static members in [Chapter 25](#).

It's good practice to be as specific as possible about how a class and its members will be used. For that reason, if you know a class is abstract, it's best to include the `abstract` keyword when you're declaring it. This keyword ensures that no object of the class can be created without triggering an error. To demonstrate, we'll add the `abstract` keyword to our `Vehicle` class declaration. Update `src/Vehicle.php` as shown in [Listing 19-11](#).

```
<?php

abstract class Vehicle
{
    protected string $makeModel;
    --snip--
```

Listing 19-11: Declaring `Vehicle` as an abstract class

We declare that `Vehicle` is an abstract class by adding `abstract` immediately before the `class` keyword. If you run the index script again after making this change, you'll see no difference in the behavior of the program. However, if you now attempt to create a `Vehicle` object, as shown in [Listing 19-12](#), a fatal error will occur. Update `public/index.php` to match the listing.

```
<?php
require_once __DIR__ . ' ../src/Vehicle.php';
require_once __DIR__ . ' ../src/Car.php';
```

```
❶ $vehicle1 = new Vehicle();
```

```
$car1 = new Car();  
--snip--
```

Listing 19-12: Attempting to create an object of the abstract `Vehicle` class

We attempt to create an object of the `Vehicle` class and store a reference to the created object in the `$vehicle1` variable ❶. Here's what happens if you try to run this index script:

```
$ php public/index.php
```

```
Fatal error: Uncaught Error: Cannot instantiate abstract class Vehicle in  
/Users/matt/public/index.php:5  
Stack trace:  
#0 {main}  
    thrown in /Users/matt/public/index.php on line 5
```

A fatal error occurs, with the message `Cannot instantiate abstract class`. This is exactly what we want: by declaring the `Vehicle` class to be `abstract`, we've guaranteed that no `Vehicle` objects can be created.

Overriding Inherited Methods

In some cases, you might want to change the way an inherited method behaves on a subclass, as compared to the way it's defined in the superclass. This is called *overriding* the method, and you do it by defining the method directly on the subclass and giving it the same name as the method inherited from the superclass. PHP gives precedence to properties and methods defined lower in the class hierarchy, so it will execute the method definition from the subclass rather than the one from the superclass.

Let's illustrate with a superclass that declares a `__toString()` method, then a subclass that overrides this declaration with its own `__toString()` implementation. [Figure 19-5](#) shows the UML diagram for the two classes we'll use to investigate how to override inherited methods, with notes to indicate how each `__toString()` method should behave. We'll be creating a general `Food` class and a `Dessert` subclass of `Food`.

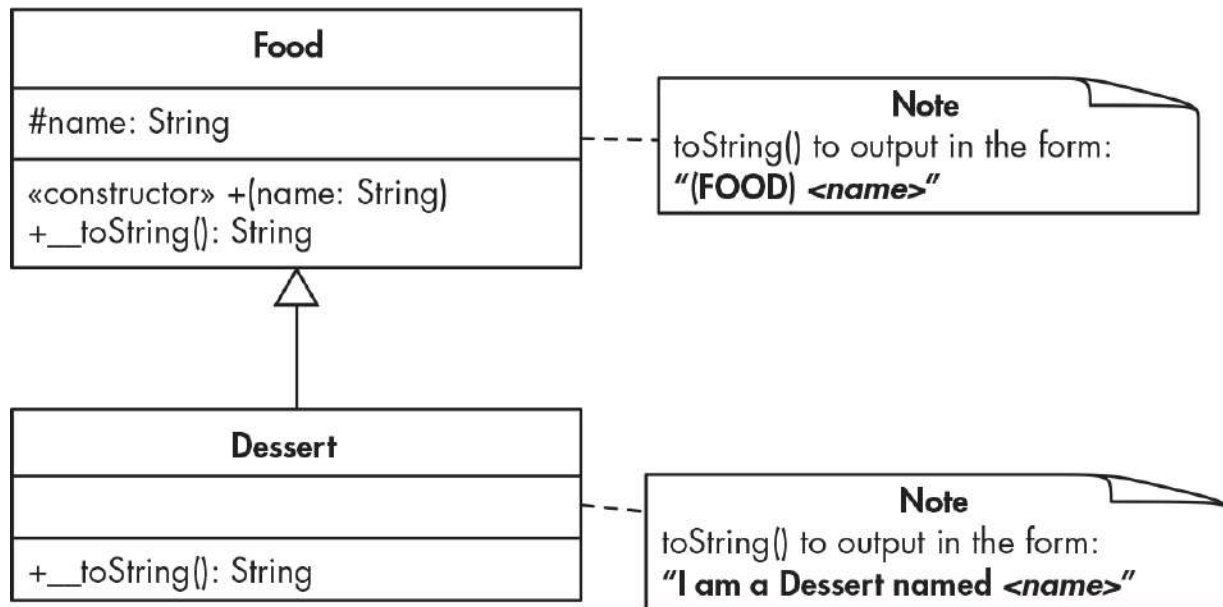


Figure 19-5: The *Dessert* subclass overrides the `__toString()` method inherited from *Food*.

The `Food` class has a `__toString()` method that generates a string in the form "(FOOD) *foodname*", such as "(FOOD) apple". Meanwhile, the `Dessert` subclass overrides this behavior with its own `__toString()` method that generates a string in the form "I am a Dessert named *foodname*", such as "I am a Dessert named strawberry cheesecake".

First, we'll declare the `Food` class. Create a new project, then create `src/Food.php` containing the code in [Listing 19-13](#).

```

<?php
class Food
{
    protected string $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function __toString(): string
    {
        return "(FOOD) $this->name";
    }
}
  
```

```
}  
}
```

Listing 19-13: The `Food` class

We declare the `Food` class with one string property, `name`, that has protected visibility so all subclasses can directly access it. The class has a constructor to initialize the `name` property when each new object is created, as well as the `__toString()` method shown in [Figure 19-5](#).

Let's now declare `Dessert` as a subclass of `Food`. Create `src/Dessert.php` and enter the contents of [Listing 19-14](#).

```
<?php  
class Dessert extends Food  
{  
    public function __toString(): string  
    {  
        $message = "I am a Dessert named $this->name";  
        return $message;  
    }  
}
```

Listing 19-14: The `Dessert` class, with its own `__toString()` method

We declare that `Dessert` is a subclass of (extends) `Food` and give it its own `__toString()` method, as illustrated in [Figure 19-5](#). Since PHP prioritizes methods declared lower in the class hierarchy, this `__toString()` method will override the `__toString()` method defined for the `Food` superclass. Notice that we've introduced a local `$message` variable in the `Dessert` class's `__toString()` method. This variable may seem unnecessary now, but we'll return to this example and add to the message later in the chapter.

Test these classes by creating the `public/index.php` script that is shown in [Listing 19-15](#). This script creates two objects, one `Food` object and one `Dessert` object, and prints out each, which will result in the objects' `__toString()` methods being invoked.

```
<?php
require_once __DIR__ . '/../src/Food.php';
require_once __DIR__ . '/../src/Dessert.php';

$f1 = new Food('apple');
❶ print $f1;
print '<br>';

$f2 = new Dessert('strawberry cheesecake');
print $f2;
```

Listing 19-15: Creating and printing `Food` and `Dessert` objects in `index.php`

We read in and execute the class declarations for `Food` and `Dessert`, starting with the superclass. Then we create and print an object of each class, which will invoke the objects' `__toString()` methods ❶. Here's the browser output of running the web server and requesting a page:

```
(FOOD) apple
I am a Dessert named strawberry cheesecake
```

Notice that the `Dessert` object outputs `I am a Dessert named strawberry cheesecake`, demonstrating that it successfully overrode the `__toString()` method inherited from `Food` with its own definition of this method. Overriding one method doesn't disrupt the subclass's inheritance of any other methods or properties from its superclass, however. In this case, the `Dessert` object still successfully inherited the `name` property and `__construct()` method from `Food`.

THE LISKOV SUBSTITUTION PRINCIPLE

In some cases of method overriding, such as the `__toString()` methods in our `Food` and `Dessert` example, the superclass and subclass methods have an identical *signature*: the method name, parameter list, and return type match exactly. However, this isn't necessary. Only the method name must match exactly between the superclass and subclass; other aspects of the method signatures can vary, as long as the subclass still satisfies the *Liskov substitution principle (LSP)*, named for Barbara Liskov, one of the first women to earn a PhD in computer science in the

United States. This principle states that you must be able to substitute an object of a superclass with an object of a subclass without creating any unexpected behavior.

To adhere to the LSP, an overriding method's signature can vary from that of the parent method in the following ways:

Make a mandatory parameter optional For example, a superclass method `validUser(int $minAge)`, which has the mandatory parameter `$minAge`, could be overridden by a subclass method that provides a default value, making the parameter optional, such as `validUser(int $minAge = 18)`.

Add new optional parameters This would be demonstrated if the `validUser(int $minAge)` superclass method could be overridden by a subclass method that adds another optional parameter (with a default value), such as `validUser(int $minAge, int $maxAge = 65)`.

Return a more specific type An example of this would be if a superclass has a `getRandomFood()` method that returns a reference to a `Food` object, this method could be overridden by a subclass method that returns a `Dessert` object instead (where `Dessert` is a subclass of `Food`). This kind of substitution with a more specific type is known as *covariance*.

Declare a parameter type to be less specific For example, if a superclass has a method that takes a `Dessert` object as a parameter, a subclass could override it with a method that instead takes a `Food` object as a parameter. This kind of substitution with a more general type is known as *contravariance*.

Augmenting Inherited Behavior

Sometimes, rather than completely replace (override) an inherited method, you might want to *augment* the inherited method with additional behavior specific to the subclass. PHP provides the keyword `parent` for this purpose: it allows you to reference a superclass's method or property from within a subclass declaration file. To illustrate, we'll modify the `__toString()` method of the `Dessert` subclass. It will now generate a string by using the `__toString()` method from the parent `Food` class, then add a special message specific to `Dessert` objects. Update `src/Dessert.php` as shown in [Listing 19-16](#).

```
<?php
class Dessert extends Food
{
    public function __toString(): string
    {
        $message = parent::__toString();
```

```
        $message .= " I am a Dessert!";  
        return $message;  
    }  
}
```

Listing 19-16: The updated `__toString()` method of the `Dessert` subclass

We invoke the `__toString()` method of the superclass (`Food`), storing the result into the `$message` variable. To access the superclass method, we use the keyword `parent`, then the double-colon *scope-resolution operator* (`::`), then the name of the method. Next, we use string concatenation to append " I am a Dessert!" to the end of the message. All this happens within the definition of the `Dessert` class's `__toString()` method, meaning we're still technically overriding the `__toString()` method from the superclass. It's just that we're using `parent` to access the method definition from the superclass as we override it.

Visit the index web page now and you should see the following:

```
(FOOD) apple  
(FOOD) strawberry cheesecake I am a Dessert!
```

The string returned by the `Dessert` object is a combination of the string "(FOOD) strawberry cheesecake" generated by the `Food` class's `__toString()` method, plus the `Dessert` object-specific "I am a Dessert!" message added to the end.

Instead of using `parent::` to access a method from a superclass, you can explicitly refer to the superclass by name, such as `Food::` in our example. This is most useful in multilevel class hierarchies (for example, `D` is a subclass of `C` is a subclass of `B` is a subclass of `A`), where you may wish to reference an inherited class higher up in the hierarchy rather than the direct parent. In this case, naming the class explicitly is the only option, since the `parent` keyword always refers to the direct parent class.

One common reason to augment a method rather than fully override it is to create a constructor tailored to a subclass. If the subclass has all the properties of the superclass plus some of its own, it's efficient to create a constructor that starts by using `parent::__construct()` to invoke the

superclass's constructor and finishes by setting the properties specific to the subclass. This way, you get to reuse any validation or other important logic in the superclass constructor method.

NOTE

Constructors are exempt from the method signature rules of the LSP. It's okay if the parameter lists of the subclass and superclass constructors aren't compatible, as when the subclass constructor has extra, mandatory parameters.

To demonstrate how to augment a constructor, we'll add a new `calories` property to `Dessert` objects. Then we'll create a constructor for `Dessert` objects that augments the `Food` constructor by setting `calories` as well as `name`. Update the `src/Dessert.php` file to match the code in [Listing 19-17](#).

```
<?php
class Dessert extends Food
{
    private int $calories;

    ❶ public function __construct(string $name, int $calories)
    {
        parent::__construct($name);
        $this->setCalories($calories);
    }

    ❷ public function setCalories(int $calories)
    {
        $this->calories = $calories;
    }

    public function __toString(): string
    {
        $message = parent::__toString();
        $message .= " I am a Dessert containing $this->calor
ies calories!";
        return $message;
    }
}
```



```
}  
}
```

Listing 19-17: Adding an augmented constructor to the `Dessert` class

We declare the new `calories` property and make it private (since we don't have any subclasses of `Dessert` for this project). Then we declare a constructor method that takes in two parameters, the name and calories for the new `Dessert` object to be created ❶. Within the method definition, we use `parent::` to invoke the constructor method for the `Food` superclass, setting the `name` property in the process. Then we finish the `Dessert` constructor by setting the `calories` property for the object. We use the setter method `setCalories()`, which we declare next ❷. Finally, we update the `__toString()` method to also output the value of the `calories` property for the object so we'll know the code is working.

We now need to add a value for the `calories` property in our index script when we create a `Dessert` object. Update *public/index.php* to match the code in [Listing 19-18](#).

```
<?php  
require_once __DIR__ . '/../src/Food.php';  
require_once __DIR__ . '/../src/Dessert.php';  
  
$f1 = new Food('apple');  
print $f1;  
print '<br>';  
  
$f2 = new Dessert('strawberry cheesecake', 99);  
print $f2;
```

Listing 19-18: Adding a `calories` argument to the index script

We create a `Dessert` object with two arguments corresponding to the `name` and `calories` properties. Here's the result of visiting the web page:

```
(FOOD) apple  
(FOOD) strawberry cheesecake I am a Dessert containing 99 ca
```

lories!

The second message demonstrates that both `Dessert` object properties were successfully set by the class's constructor method, which augments that of its superclass. Also, it shows that we successfully augmented the `__toString()` method of the `Dessert` subclass with a call to its parent superclass `__toString()` method.

Preventing Subclassing and Overriding

In certain situations, you might never want to allow a subclass to be created from a class you declare. In this case, declare the class with the `final` keyword, which prevents other classes from extending (subclassing) the class. The keyword goes at the very start of the class declaration (for example, `final class Dessert`). You can also add the `final` keyword to the declaration of an individual method to prevent the method from being overridden by a subclass.

The use of `final`, especially for whole classes, is the subject of heated debate in the OOP community. It's wise to declare classes or methods as `final` only if you have a good justification for doing so. For example, if you have an API library, you might declare a class as `final` to prevent anyone from subclassing it, since you don't want to allow or encourage programmers to expect different behaviors besides those declared in the API. The `final` declaration also helps prevent code from breaking between versions and across updates: when a class is `final`, changes to the class's internal implementation (private methods and properties) won't have any unintended consequences outside the class.

Declaring a Class final

Let's create an example of a `final` class and see how an error occurs if we try to declare a subclass of it. Start a new project and create `src/Product.php` as shown in [Listing 19-19](#).

```
<?php
❶ final class Product
{
```

```

        private int $skuId;
        private string $description;

        ❷ public function __construct(int $skuId, string $description)
        {
            $this->skuId = $skuId;
            $this->description = $description;
        }

        public function getSkuId(): int
        {
            return $this->skuId;
        }

        public function getDescription(): string
        {
            return $this->description;
        }
    }

```

Listing 19-19: The `Product` class, declared as `final`

We declare the `Product` class, designating it as `final` ❶. This simple class has properties for a stock keeping unit (SKU) number and a text description, which are set in the class’s constructor ❷. It also has a getter method for each property.

Next, we’ll attempt to extend `Product` by declaring a subclass. Create `src/MyProduct.php` containing the code in [Listing 19-20](#).

```

<?php
class MyProduct extends Product
{
}

```

Listing 19-20: The `MyProduct` subclass of `Product`

We declare that `MyProduct` is a subclass of `Product`, while leaving its body blank.

Now create a *public/index.php* script that reads in the two class declarations, as shown in [Listing 19-21](#).

```
<?php
require_once __DIR__ . '/../src/Product.php';
require_once __DIR__ . '/../src/MyProduct.php';
```

Listing 19-21: An index script reading the `Product` and `MyProduct` declarations

We don't need any more than these two `require_once` statements to demonstrate that classes declared `final` can't be extended. Execute the index script and you should see this: Fatal error: Class MyProduct may not inherit from final class Product.

Declaring a Method final

Declaring specific methods `final` prevents them from being overridden while still allowing the class to have subclasses. Among other applications, this technique ensures that a method has consistent validation logic at all levels of the class hierarchy.

To illustrate, let's modify our `Product` class, removing its overall `final` declaration and adding a new method to set the `skuId` property. The method will have validation to ensure that the SKU number is greater than 0. We'll declare this method `final`, so no subclass can replace it with a method that doesn't contain the validation logic. Update the `Product` class declaration as shown in [Listing 19-22](#).

```
<?php
class Product
{
    protected int $skuId;
    private string $description;

    public function __construct(int $skuId, string $description)
    {
        $this->skuId = $skuId;
        $this->description = $description;
```

```

    }

    public function getSkuId(): int
    {
        return $this->skuId;
    }

    public function getDescription(): string
    {
        return $this->description;
    }

    final public function setSkuId(int $skuId): void
    {
        if ($skuId > 0)
            $this->skuId = $skuId;
    }
}

```

Listing 19-22: Adding a *final* method to the *Product* class

We remove the `final` keyword from the overall class declaration, allowing this class to be extended, and we change the visibility of the `skuId` property to `protected` so it can be used in subclass methods. Then we add a new `final` setter method for the `skuId` property that confirms the desired value is greater than 0.

Now update the `MyProduct` class to match the contents of [Listing 19-23](#), where we attempt to override the `setSkuId()` method.

```

<?php
class MyProduct extends Product
{
    ❶ public function setSkuId(int $skuId): void
    {
        $this->skuId = $skuId;
    }
}

```

Listing 19-23: Overriding the `setSkuId()` method in `MyProduct`

We attempt to declare a `setSkuId()` method directly on the `MyProduct` class ❶, without the greater-than-zero validation check. This isn't permitted, since the `setSkuId()` method in the `Product` superclass has been declared `final`. If you run the index script again, you should see another fatal error noting that the `Product` class's `setSkuId()` method can't be overridden.

Summary

In this chapter, we explored several OOP features, most notably inheritance between classes. We also covered the related topics of `protected` visibility, which enables access to inherited properties and methods by subclasses, as well as method overriding and the invocation of inherited method behavior with the `parent` keyword. Finally, we looked at how to restrict the usage of classes and methods through the `abstract` and `final` keywords.

Exercises

1. Implement the `Jam` class diagrammed in [Figure 19-6](#).

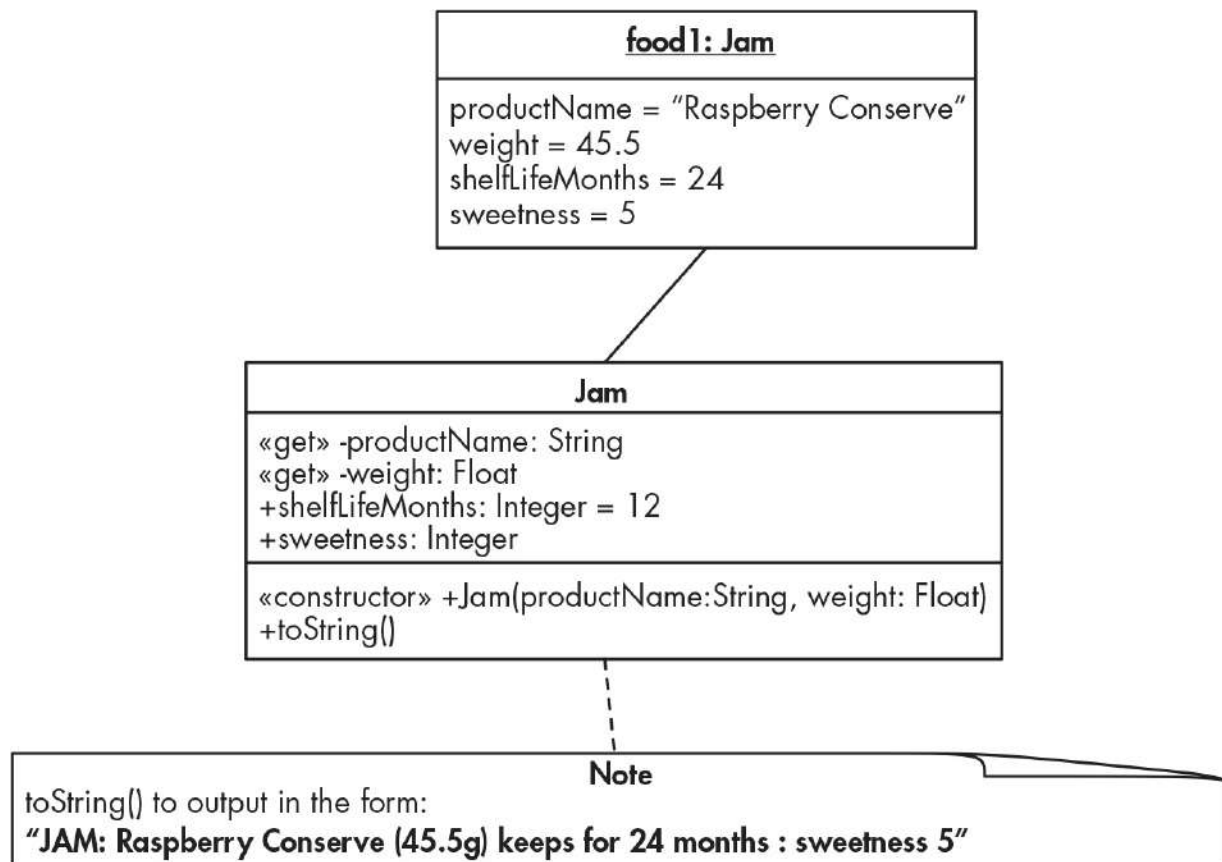


Figure 19-6: The `Jam` class and its `$food1` object

Write an index script to create the `$food1` object of the `Jam` class shown in the diagram. Use a `print` statement to invoke the object's `__toString()` method.

2. Make a copy of your project from Exercise 1. Create two new classes, `Spread` and `Honey`, and simplify the `Jam` class as shown in [Figure 19-7](#). Note the visibility of the properties for the `Spread` class: `#` for protected and `+` for public.

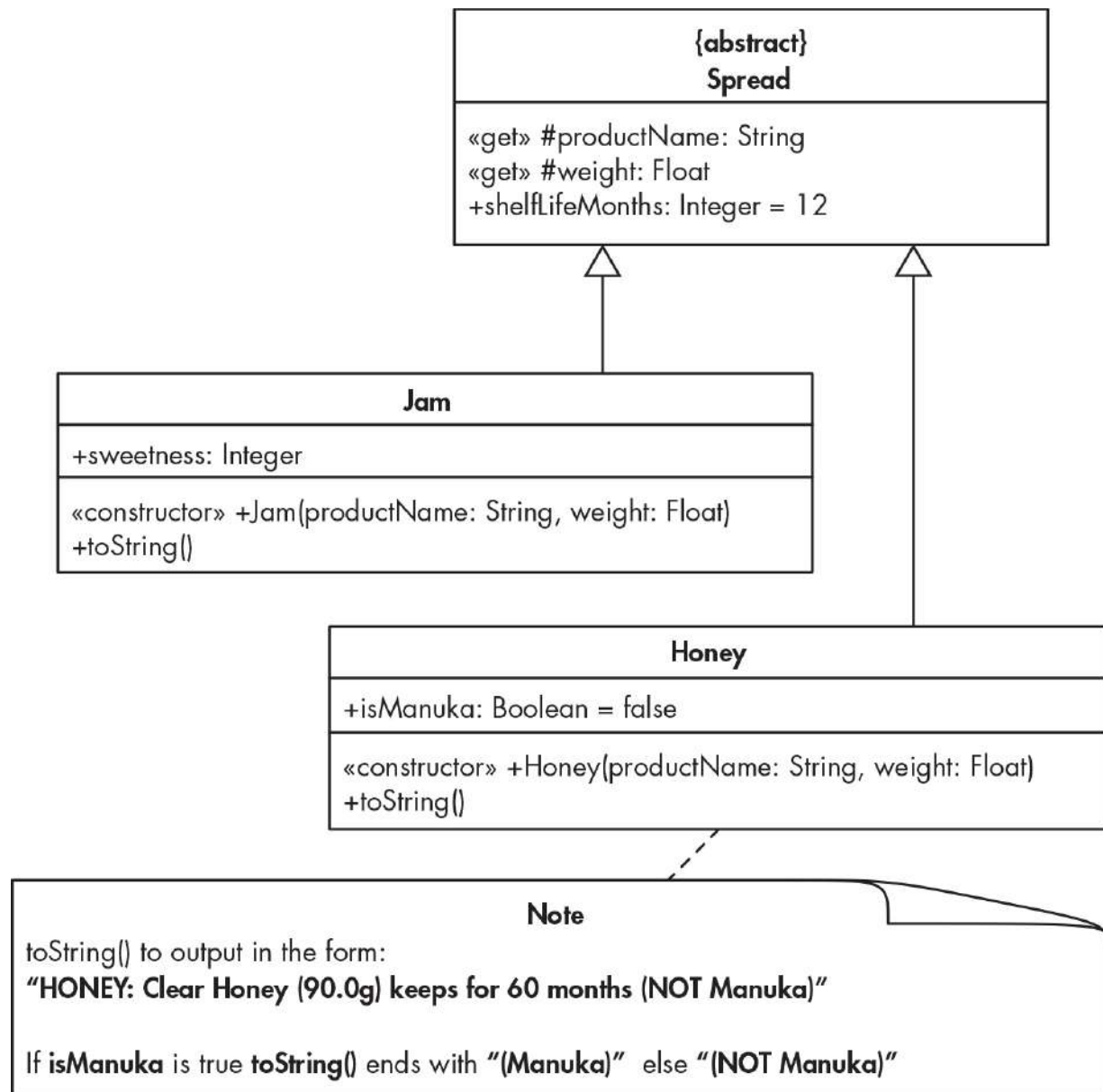


Figure 19-7: The *Jam* and *Honey* subclasses inheriting from *Spread*

Update your index script to create and print one `Jam` object and one `Honey` object.

Hint: You may wish to simplify your string-creation code by creating a private helper method `manukaString()` that returns the string (Manuka) or (NOT Manuka) depending on the value of the `isManuka` property.

3. Examine the `Car` and `Van` classes in [Figure 19-8](#) and plan out an abstract superclass to hold the common members of both classes.

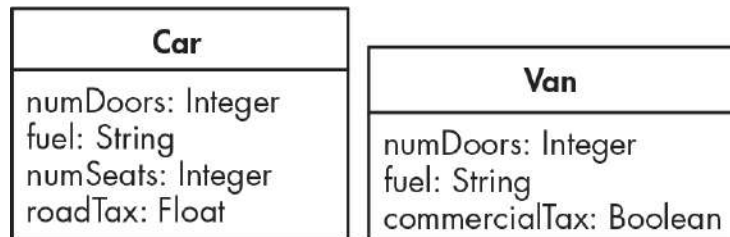


Figure 19-8: The `Car` and `Van` classes

Declare your superclass, as well as the `Car` and `Van` classes. Then write an index script that creates one `Car` and one `Van` object and uses `print` statements to invoke their `__toString()` methods.

4. Test out the use of the `final` keyword to prevent subclasses. Make a copy of your project from Exercise 2 and replace the `abstract` keyword with the `final` keyword at the beginning of your `Spread` class declaration. Run your index script, and the `Jam` and `Honey` class declarations should trigger a fatal error.

20

MANAGING CLASSES AND NAMESPACES WITH COMPOSER



As your PHP projects grow larger and more complex, you increasingly run the risk of encountering a *naming collision*, or having two classes with the same name. In this chapter, you'll learn about *namespaces*, the solution provided by object-oriented languages to avoid naming collisions. In addition, you'll learn to use the helpful Composer command line tool, which automates the process of loading class- and function-declaration files and simplifies work with namespaces. Almost every modern object-oriented PHP project uses Composer, and we'll use it throughout the remainder of the book.

You might think that a naming collision would be unlikely; after all, up until now we've been writing class declarations in PHP files with the same name as the class, and we've been placing these class declaration files in

the project's *src* directory. Since PHP doesn't allow two files with the same name in the same directory, surely we couldn't end up with two classes of the same name?

In fact, naming collisions can occur in several cases. First, you might try to declare a class with the same name as one of the built-in classes of the PHP language, such as `Error`, `Directory`, or `Generator`. Second, you might declare two classes in different directories (for example, different subdirectories of *src*). Third, you might combine your own classes with classes from third-party libraries.

Namespaces

Namespaces can be thought of as a virtual hierarchy of directories for classes, used to prevent class name collisions. Classes are organized within namespaces and sub-namespaces, much as computer files are organized within directories and subdirectories. Just as you need to state the directory location of a computer file on a hard disk, using namespaced classes requires you to specify both the name of the class and its namespace to uniquely identify a particular class.

A backslash character (`\`) separates namespaces, sub-namespaces (if any), and the class name. For example,

`\MyNamespace\MySubNamespace\MyClass` refers to a class called `MyClass` in the sub-namespace `MySubNamespace`, which is part of the larger `MyNamespace` namespace. Identifying `MyClass` with a namespace and sub-namespace prevents it from colliding with another `MyClass` class in a different namespace, such as `\YourNamespace\MyClass`. By convention, the first letter of a namespace or sub-namespace is capitalized, just like class names. Other letters in the namespace or sub-namespace can be capitalized as well.

Classes that are built into the PHP language are considered to be in the *root namespace*, which is identified with just a single backslash character. For example, you can write `\DateTime` or `\Exception` to explicitly refer to PHP's built-in `DateTime` or `Exception` classes. So far in this book, we've been omitting the backslash before built-in class names, since we haven't been using namespaces when writing our own classes. Including the

backslash makes it unambiguous that we're referring to a PHP class in the root namespace.

In the examples that follow, I'll use the namespace `Mattsmithdev`. It's the namespace I use for all the classes I write, along with a sub-namespace for each project I work on. You may want to make up your own namespace, such as `Supercoder` or `DublinDevelopers`, and use it when following these chapters and writing your own classes. We'll also encounter other namespaces in "Adding Third-Party Libraries to a Project" on [page 390](#).

Declaring a Class's Namespace

To declare the namespace of a class, use the `namespace` keyword followed by the namespace's name. This should be the first line of PHP code in the class-declaration file. To demonstrate, we'll declare a class called `Shirt` and make it part of the `Mattsmithdev` namespace. Start a new project, create a file called `src/Shirt.php`, and enter the code from [Listing 20-1](#).

```
<?php
namespace Mattsmithdev;

class Shirt
{
    private string $type = 't-shirt';

    public function getType(): string
    {
        return $this->type;
    }

    public function setType(string $type): void
    {
        $this->type = $type;
    }
}
```

Listing 20-1: The `Shirt` class in the `Mattsmithdev` namespace

Immediately after the opening PHP tag, we use `namespace Mattsmithdev` to make the class we're about to declare part of the `Mattsmithdev` namespace. We follow the namespace statement with two blank lines, which is recommended by the PHP coding standards. Then we proceed with the class declaration as usual. In this case, the `Shirt` class has a `private type` property with a default value of `'t-shirt'`, as well as public getter and setter methods for this property.

Using a Namespaced Class

Once a class is declared to be within a namespace, you need to unambiguously inform the PHP engine that it's the class you want to use. You can do this in two ways.

The first option is to always include the namespace when referencing the class; this is called using the *fully qualified name* of the class. For example, to create a new `Shirt` object, you would write `new \Mattsmithdev\Shirt()`. Let's try that now. Add a `public/index.php` file to your project and enter the code from [Listing 20-2](#).

```
<?php
require_once __DIR__ . '/../src/Shirt.php';

$shirt1 = new \Mattsmithdev\Shirt();
$shirt2 = new \Mattsmithdev\Shirt();

print "shirt 1 type = {$shirt1->getType()}";
```

Listing 20-2: Creating objects of the `\Mattsmithdev\Shirt` class in `index.php`

After reading in the class-declaration file, we create two objects of the `Shirt` class, using the class's fully qualified name. Run the project at the command line with `php public/index.php` and you should see the following:

```
shirt 1 type = t-shirt
```

The message indicates that a `Shirt` object has been successfully created via the class's fully qualified name.

The second way to unambiguously reference a class from a particular namespace is to include a `use` statement before invoking the class. For example, `use Mattsmithdev\Shirt` tells the PHP engine that any subsequent references to the `Shirt` class are specifically to the one in the `Mattsmithdev` namespace. To see how `use` statements work, update your *public/index.php* file to match [Listing 20-3](#).

```
<?php
require_once __DIR__ . '/../src/Shirt.php';

use Mattsmithdev\Shirt;

$shirt1 = new Shirt();
$shirt2 = new Shirt();

print "shirt 1 type = {$shirt1->getType()}";
```

Listing 20-3: Referencing the `Shirt` class with a `use` statement in `index.php`

We include a `use` statement after reading in the class declaration to ensure that `Shirt` later in the code will refer to `Mattsmithdev\Shirt`. Notice that we don't include a backslash before the namespace in a `use` statement. This kind of class identifier, without the initial backslash, is called a *qualified name*, as opposed to a *fully qualified name* that includes the initial backslash. We then create the two `Shirt` objects simply with `new Shirt()`, since the PHP engine knows which class we're referencing, thanks to the `use` statement. Run the index script again and you should see that the output hasn't changed. We've still successfully created some `Shirt` objects.

If you need to differentiate between two classes with the same name but different namespaces in the same section of code, you can either refer to both with their fully qualified names (for example, `\Mattsmithdev\Shirt` and `\OtherNamespace\Shirt`) or provide a `use` statement for one of the classes and qualify the other.

Referencing Namespaces in Class Declarations

Say you're writing code in the class-declaration file for a namespaced class (as opposed to in a general script like *index.php*) and you want to refer to a class from a different namespace. If you haven't written a `use` statement, you must use the fully qualified name of the other class, starting with a backslash. For example, if you were writing code for a class declared in the `Mattsmithdev` namespace and you wanted to refer to PHP's built-in `DateTime` class, you'd have to write it as `\DateTime` to indicate that it's part of the root namespace. Likewise, if you wanted to refer to a third-party class, you'd write a backslash, then the third-party namespace, then another backslash, and then the class name, such as `\MathPHP\Algebra`.

Without the initial backslash, PHP will assume you're referring to a class or sub-namespace of the current namespace. For example, in a class in the `Mattsmithdev` namespace, a reference to `DateTime()` without an initial backslash is assumed to be a reference to `Mattsmithdev\DateTime`, meaning a `DateTime` class in the `Mattsmithdev` namespace. Similarly, a reference to `MathPHP\Algebra` without an initial backslash is assumed to be a reference to `Mattsmithdev\MathPHP\Algebra`, meaning `MathPHP` is assumed to be a sub-namespace of `Mattsmithdev` and `Algebra` is assumed to be a class in that sub-namespace. Writing a fully qualified namespace beginning with the backslash ensures that the PHP engine will understand the namespace of the class you're referencing.

On the other hand, if you *are* referring to a class or sub-namespace of the current namespace, you shouldn't include a backslash before the class or sub-namespace. For example, if you're working on a class in the `Mattsmithdev` namespace, `Shirt()` is understood to refer to the `Shirt` class in the `Mattsmithdev` namespace, and `SubNamespace\Example` is understood to refer to the class `Mattsmithdev\SubNamespace\Example`.

If you're using a class from another namespace only once, it might make sense to just write the class's fully qualified name, including the initial backslash. If you'll need to refer to the class several times, however, writing a `use` statement for it at the beginning of the class declaration is more efficient. In this case, no initial backslash is needed. As you look at and write more PHP code, you'll often see many `use` statements at the

beginning of a class declaration when the code uses classes declared in other namespaces, as illustrated in [Listing 20-4](#).

```
<?php

namespace App\Controller;

use App\Entity\ChessGame;
use App\Entity\Comment;
use App\Form\ChessGameType;
use App\Repository\ChessGameRepository;
use App\Repository\CommonRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Session\SessionInterface;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * @Route("/chessgame")
 */
class ChessGameController extends AbstractController
{
    private $session;

    public function __construct(SessionInterface $session)
    {
        --snip--
    }
}
```

Listing 20-4: A class declaration with many `use` statements

This code snippet is the start of a class declaration from one of my PHP Symfony web framework chess projects. It has a whopping 11 `use` statements, drawing on classes from a variety of namespaces and sub-namespaces. The `use` statements help keep them all straight, but if juggling

all those classes still seems overwhelming, don't worry: we're about to discuss a tool for managing all the classes in a project.

Composer

Composer is a command line tool to support object-oriented PHP programming. It helps with loading class and function declaration files (your own and those from third-party libraries), and it facilitates working with classes from different namespaces. It's an essential, easy-to-use tool for professional web application projects. In this section, you'll set up Composer and learn how to use it to create command line aliases, load class-declaration files automatically, and manage a project's third-party dependencies.

NOTE

SymfonyCasts has a great free video introducing the Composer tool at <https://symfonycasts.com/screencast/composer>.

Installing and Testing Composer

For Windows, Composer offers a simple installer that can be found at <https://getcomposer.org/Composer-Setup.exe>. For macOS, you can install Composer with Homebrew, as discussed in [Appendix A](#). For Linux, you need to execute several command line statements to download and run the *composer.php* script. You can find details at <https://getcomposer.org/download/>. If you're using Replit to follow along with this book, see [Appendix C](#) for how to integrate Composer into your projects.

Once you've installed Composer, test it by opening a new command line terminal application and entering `composer`. This launches the Composer tool, bringing up a nice ASCII art logo, the version number, and a list of the command line options.

Creating the *composer.json* Configuration File

To use the Composer command line tool with a project, you need to create a *composer.json* file to house all the information Composer needs to know about the project. (See the following “JSON File Format” box for a

refresher on this type of file.) For example, the *composer.json* file features records of the namespaces and class locations for your own code, as well as of the third-party packages that the project depends on. In addition, you can declare command line *aliases* in the *composer.json* file, shortcuts that can save you from typing long commands at the command line. We'll begin our exploration of the *composer.json* file by declaring a simple alias.

THE JSON FILE FORMAT

JavaScript Object Notation (JSON) is a popular text file format for storing and exchanging data. Originally, JSON was just part of JavaScript programs, but it caught on quickly as a general-purpose format for sharing data between computer programs and is now independent of the JavaScript language.

JSON documents have two types of structures: objects and arrays. A *JSON object* is declared within curly brackets and has a comma-separated list of name/value pairs, as shown here:

```
{id: 1, name: "hammer"}
```

This object has two items, one with a name of `id` and a value of `1`, and the other with a name of `name` and a value of `"hammer"`. Objects are typically used to store multiple properties or pieces of information about the same entity, which is why the name/value pairs in an object are often called *properties*.

A *JSON array* is declared within square brackets and is a comma-separated list of values. The values in an array can be simple ones (such as strings and numbers), or they can be other arrays or objects, as in this example:

```
[{id: 1, name: "hammer"}, {id: 2, name: "bucket"}]
```

This array has two values, and each is a JSON object.

Beyond recognizing the difference between an object and an array, you don't need to know much about JSON to be able to configure the *composer.json* file for your projects. That said, you can learn more about JSON at the official site for this international file format: <https://www.json.org>.

The *composer.json* text file must be located at the top level of your PHP project directory, not inside a subfolder like *src* or *public*. Continuing with this chapter's project, create *composer.json*, save it at the top level of the project directory, and then enter the contents of [Listing 20-5](#). This code

creates an alias called `hello` that will stand in for the command `echo Hello World`.

```
{
  "scripts": {
    "hello": "echo Hello World"
  }
}
```

Listing 20-5: Declaring an alias in the `composer.json` file

The content of *composer.json* is always a JSON object and so will always begin and end with a pair of curly brackets. Inside the object, we declare a `"scripts"` property with a value that itself is an object. Within the sub-object, we declare a property named `"hello"` (the name of our alias) with a value of `"echo Hello, world!"` (the code that will be replaced by the shortcut alias).

We now have a simple but valid *composer.json* file telling Composer that there's a command line alias named `"hello"`. To see whether this has worked, enter **`composer hello`** at the terminal. You should see `Hello, world!` as a result:

```
$ composer hello
> echo Hello, world!
Hello, world!
```

In this case, we've written more characters to declare the alias than we would need to write out the `echo` statement in full at the command line. However, sometimes these script aliases can be handy. For example, here's an alias I use in some projects to output a report of how much code in the `/src` folder needs fixing to match the PHP programming standards (although the alias appears on two lines here for space reasons, it would be all one line in the file):

```
"reportfixsrc": "php php-cs-fixer.phar fix --level=psr2
--dry-run --diff ./src > ./tests/fixerReport.txt"
```

This alias lets me enter `composer reportfixsrc` at the command line rather than a long PHP command to run a PHP Archive (*.phar*) file with lots of parameters.

As you'll soon see, Composer can do a lot more than just keep track of command line aliases. For now, we've successfully created the *composer.json* file for our project, an essential first step in using this powerful tool.

Creating an Autoloader

An *autoloader* is a system that automatically retrieves class-declaration files whenever they're needed, so you don't have to load them all into an *index.php* file yourself. Autoloaders become useful as object-oriented PHP projects grow in size and complexity, involving many classes in many namespaces. If you had to write `require_once` statements for each class in your *index.php* front controller, not only would it be a lot of work, but it would also be easy to miss one or two, particularly as the project continues to evolve. This would lead to errors and would force you to keep going back to update the list of files to require. An autoloader handles the process for you, provided the classes are correctly namespaced and correctly located according to the autoloading rules.

One of the most powerful features of the Composer tool is its autoloader. It conforms to PSR-4, PHP's recommended set of rules for autoloading. According to PSR-4, you must specify the base directory containing the classes of each namespace. For example, you may want to declare that classes in the `Mattsmithdev` namespace can be found in the *src* directory. Further, PSR-4 stipulates that any sub-namespaces will be assumed to have corresponding subdirectories within the declared base directory of the namespace. For example, the class `Mattsmithdev\Trigonometry\Angles.php` should be located in *src/Trigonometry*, the class `Mattsmithdev\Utility\Security.php` should be located in *src/Utility*, and so on. As long as the subdirectory has the same name as the sub-namespace, you don't need to tell the autoloader where to find the sub-namespaced classes.

Getting the Composer autoloader to work requires three steps:

1. Declare the base directory for each namespace in the project's *composer.json* file.
2. Tell Composer to create or update its autoloader script.
3. Add a `require_once` statement for the autoloader script at the beginning of the project's *public/index.php* front controller. This single `require_once` statement replaces the separate `require_once` statements for each individual class.

We'll walk through the process of setting up the Composer autoloader to load our `Mattsmithdev\Shirt` class. First, [Listing 20-6](#) shows what to write in the *composer.json* file to declare that classes in the `Mattsmithdev` namespace can be found in the *src* directory.

```
{
    "autoload": {
        "psr-4": {
            "Mattsmithdev\\": "src"
        }
    }
}
```

*Listing 20-6: Setting up the autoloader in the *composer.json* file*

We declare an `"autoload"` property, whose value is an object. Within that object, we declare the `"psr-4"` property, whose value is another object. It contains a `"Mattsmithdev\\"` property whose value is `"src"`. This tells Composer that class files in the `Mattsmithdev` namespace are located in the *src* directory. Notice the two backslash characters (`\\`) after the namespace. This is a requirement of the PSR-4.

For the projects we'll work on in the next few chapters, the *composer.json* files will be essentially the same as [Listing 20-6](#). The only potential difference from project to project will be the actual namespace(s) and location(s) declared within the `"psr-4"` object.

NOTE

The Composer autoloader has some additional subtleties. If you want to learn more, consult the Composer documentation at

<https://getcomposer.org/doc/04-schema.md#psr-4>.

Now that we've declared the base directory for the `Mattsmithdev` namespace in the `composer.json` file, we can tell Composer to generate a class autoloader for us. Enter the following at the command line for the current project's working directory:

```
$ composer dump-autoload
```

This command creates a new folder in the project called *vendor*, if it doesn't exist already, and generates or updates several files within that folder. This *vendor* folder is where Composer keeps its working files for the project. You can look inside it, but you shouldn't change its contents. You can also delete the folder and ask Composer to rebuild it anytime, so you can safely omit this folder whenever you back up a project.

Inside *vendor*, you should see a *vendor/autoload.php* file as well as a *vendor/composer* folder containing several more scripts, including *autoload_psr4.php*, which encodes our PSR-4-compliant declaration. This file contains statements to return the location (*src/*) of classes in the `Mattsmithdev` namespace.

Now that we've generated the autoloader, we can update the *public/index.php* script to require just this one *autoload.php* file, no matter how many classes we need to refer to in our project. As long as the base directories for the namespaces have been declared in the `composer.json` file and the autoloader has been updated with the `composer dump-autoload` command, then whenever we write a `use` statement for a namespaced class, the PHP engine will load its declaration, ready for our code. [Listing 20-7](#) shows how to update *index.php*.

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Shirt;

$shirt1 = new Shirt();
$shirt2 = new Shirt();
```

```
print "shirt 1 type = {$shirt1->getType()}";
```

Listing 20-7: Reading the Composer-generated autoloader script into index.php

We change the `require_once` statement to read in and execute the Composer-generated autoloader script from the *vendor* directory. When you run the project, the output will be just as before, but now we're using the Composer autoloader to automatically read in the declaration for the `Mattsmithdev\Shirt` class from the *src* folder, instead of reading it in manually. While this may not seem to make much of a difference for our one-class project, the autoloader is a big time-saver for projects with many classes.

Adding Third-Party Libraries to a Project

Another powerful feature of Composer is its capability to add third-party libraries to a project and maintain a record of these dependencies in the *composer.json* file. Thousands of open source libraries are available, many kept up-to-date by multiple experienced software developers; in many cases, a few minutes of searching can offer up a ready-made library that does all or most of what you want. Well-maintained open source projects have been tested thoroughly and refactored to implement best practices, so careful use of third-party libraries can reduce your workload while helping to maintain the quality of a software project.

Without Composer's package-dependency features, you'd have to download the code for third-party libraries from websites or Git repositories, copy it to an appropriate location such as a *lib* folder, and update the *composer.json* file with the namespaces and locations of these library classes. Instead, you can simply tell Composer that you need a third-party library in your project, and it will do all the hard work for you. It will automatically download the code, create and copy the files to an appropriate subdirectory in *vendor*, update the autoloader, and record the dependency (and its version) in the *composer.json* file. All you need to know is the name of the package you want and the vendor supplying it.

To see how this works, we'll tell Composer to add the `math-php` package from the vendor `markrogoyski` to our chapter project. This is a

great package offering many useful mathematics operations. At the command line for the current project's working directory, enter the following:

```
$ composer require markrogoyski/math-php
```

This `require` command triggers Composer to perform a series of actions. First, if you check your project's *vendor* folder, you should see that Composer has created a new subfolder matching the package's vendor name (in this case, *vendor/markrogoyski*). Inside, you'll find a folder for the *math-php* package, containing all the necessary code.

Keep in mind that the vendor name (*markrogoyski*) and package name (*math-php*) are *not* namespaces. They're simply names that Composer uses to identify and locate the third-party scripts to be added to the project. Composer will automatically determine the namespaces for all the open source library classes, and so the contents of *vendor/composer* will be updated for all these classes that have been added to the *vendor* folder. In particular, *autoload_psr4.php* will likely be updated with the base directory for the namespaced third-party classes, since most open source libraries use the PSR-4 autoloading standard. Meanwhile, you'll need to read the package's documentation to find out the namespaces of the third-party classes so you can reference them correctly in your code.

The `require` command also prompts Composer to update the *composer.json* file with information about the *markrogoyski/math-php* package. If you check the file, you should now see something like [Listing 20-8](#).

```
{
    "autoload": {
        "psr-4": {
            "Mattsmithdev\\": "src"
        }
    },
    "require": {
        "markrogoyski/math-php": "^2.10"
    }
}
```

Listing 20-8: The composer.json file's record of the `math-php` library dependency

In addition to the "autoload" property we wrote earlier, the main object in *composer.json* now has a "require" property that Composer automatically generated. Its value is an object with entries for all the packages required for the project. In this case, there's an entry of "markrogoyski/math-php". Its value, "^2.10", indicates the acceptable versions of the package. The caret symbol (^) means we're happy to use newer versions with the same main version number (2.10.1, 2.11, 2.2, and so on) but not version 3.x or later, since that might break backward compatibility.

Now that Composer has integrated the `markrogoyski/math-php` package into our project, we can try using it. Specifically, we'll draw on the package's `Average` class to compute the average of a series of numbers. Update the contents of *public/index.php* with the code in [Listing 20-9](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use MathPHP\Statistics\Average;

$numbers = [13, 18, 13, 14, 13, 16, 14, 21, 13];
$numbersString = implode(', ', $numbers);
$mean = Average::mean($numbers);
print "average of [$numbersString] = $mean";
```

Listing 20-9: Calculating the average of an array of integers

We start with a `use` statement telling the PHP engine that `Average` refers to the namespaced class `MathPHP\Statistics\Average`. Notice that the namespace for this class is different from the vendor and package name we used earlier in the `require` statement to Composer. Then we declare a `$numbers` array and use the built-in `implode()` function to create a string version of it for user-friendly output. Next, we invoke the `mean()` method from the `Average` class, storing the result in `$mean`. We then print out the list of numbers and the calculated mean value.

Notice that we've invoked the `mean()` method without actually having to create an object of the `Average` class. This is because `mean()` is a *static method*. We'll explore this OOP concept in detail in [Chapter 25](#).

Where to Find PHP Libraries

You may be wondering how the Composer tool knew where to go on the internet to download the `markrogoyski/math-php` package files for our project. The answer is Packagist (<https://packagist.org>), a website for publishing open source PHP packages. Vendors can register with the site (I'm `mattsmithdev` on Packagist, for example) and then publish PHP packages for anyone to install via Composer.

When publishing a package, the vendor must provide information including the GitHub (or other repository) location of the package's publicly downloadable files. For example, the Packagist page for the `markrogoyski/math-php` package lists a GitHub address of <https://github.com/markrogoyski/math-php>. This is where Composer goes to get the package files. Each page on Packagist also lists the exact `require` command you need in order to make Composer add that package to your project.

Summary

In this chapter, you learned to unambiguously differentiate between classes of the same name by using namespaces. You also learned to use the powerful Composer command line tool to support object-oriented PHP programming. The time spent learning how to maintain the `composer.json` file and how to use Composer to autoload classes and incorporate third-party libraries into your projects will save you countless hours of tedious manual work.

Exercises

1. Start a new project and create a Composer script alias for a command to display the message `Hello name`, replacing `name` with your name. Then use Composer to execute your command.

Hint: Declare a script alias in *composer.json*, then run it at the command line with `composer alias`.

2. Start a new project and create a *src/Product.php* file declaring a `Product` class with the private properties `$id`, `$description`, and `$price`, and public getters and setters for each property. Declare the class to be in the `Mattsmithdev` namespace. Add a *composer.json* file to the root folder of your project, declaring that classes in the `Mattsmithdev` namespace can be found in the *src* directory. Then use Composer to generate an autoloader.

Write a *public/index.php* file that does the following:

- a. Reads in and executes the Composer autoloader in *vendor/autoload.php*
- b. Creates a new `Product` object, `$p1`, with an `id` of 7, a description of 'hammer', and a price of 9.99
- c. Uses `var_dump()` to output the details of `$p1`

When you run your code, you should see something like this:

```
object(Mattsmithdev\Product)#4 (3) {  
    ["id":"Mattsmithdev\Product":private]=>  
    int(7)  
    ["description":"Mattsmithdev\Product":private]=>  
    string(6) "hammer"  
    ["price":"Mattsmithdev\Product":private]=>  
    float(9.99)  
}
```

3. Go to the Packagist website at <https://packagist.org> and search for the `mattsmithdev/faker-small-english` package. Look at the documentation, then use Composer to require the `mattsmithdev/faker-small-english` package for a new project. Write a *public/index.php* file that loops 10 times to display 10 random names from a `FakerSmallEnglish` object.

21

EFFICIENT TEMPLATE DESIGN WITH TWIG



This chapter introduces the free, open source Twig templating library, a third-party package that applies object-oriented principles such as inheritance to page display templates. This mix of templating with OOP streamlines the process of designing web applications. We'll explore the basics of using Twig, then use it to progressively build a multipage website.

This chapter also offers a first look at how the object-oriented style of programming influences the web application architecture we arrived at in [Part III](#). For example, rather than writing functions that implement the logic of a front controller, we'll create object-oriented classes containing front-controller methods. We'll refine our approach to object-oriented application development in [Chapter 22](#) as we further develop this chapter's multipage site.

The Twig Templating Library

Web templating systems like the Twig library (which can be found at <https://twig.symfony.com>) manage the duplication that's common in web

application outputs by distinguishing between unchanging content and content that may change upon each request. This makes developing the view element of a web application's MVC architecture much more efficient. As we've discussed in previous chapters, a lot of basic HTML is often duplicated across the template scripts for each page of a website, since the multiple pages typically share elements such as a header and a navigation bar. When we created a three-page website in [Chapter 13](#), for example, most of the code in the template file for each page was redundant HTML; barely any code was unique to the page at hand.

Duplicating so much HTML across multiple templates causes two issues. First, if you need to change an aspect of the website design (for example, the look and feel of the navigation bar, or a special header or footer), you'd have to edit *every* page template, which for a large website could be tens, hundreds, or even thousands of files. Second, when editing the content or behavior of a page template, spotting the content that's specific to the current page can be difficult when it's hidden within a bunch of generic code. You want to be able to focus on just the content for the particular page you're editing.

One solution is to separate the common parts of a page into sub-templates for the header, navigation, footer, and so on. This is what we did in [Chapter 15](#), for example, when we created a `templates/_header.php` file containing header elements common to all the pages in our shopping cart website. A more elegant solution, however, is to use a dedicated PHP templating library, especially one such as Twig that supports template inheritance. As you'll see, a key advantage of template inheritance is that after we've declared that a template for a particular page inherits from a parent template, the only content in the child page is that for the child page itself; we don't have to write `require` statements for standard page headers, footers, navigation bars, and so on. This way, each child template contains fewer distractions away from the page it represents.

Another advantage of templating libraries is that they limit the behavior that can be encoded into a page template by removing any PHP code from the template itself. If you're working on a website as part of a team, these limitations mean you can safely have other members of the team (such as a marketing department) edit the page templates to refine the website's look

and feel, secure in the knowledge that they can't accidentally copy in PHP code that would break the website or create a security vulnerability. After all, template files are the view component of an MVC architecture, so they should only be “decorating” provided data with tags in a markup language like HTML.

How Twig Works

Before we create anything with Twig, let's explore the basic files, objects, and methods involved when working with Twig templates. When using Twig in our code, we work with an object of the `Twig\Environment` class. This object is responsible for generating the HTML for a web page, based on a template file and any data needed to customize the contents of the page. It does this through its `render()` method. Usually, you store a reference to the `Twig\Environment` object in a variable called `twig`, which is itself a property of a general `Application` class that encapsulates all the high-level logic for the web application.

The `render()` method requires two parameters. The first is a string containing the path to the desired template file. This string is typically stored in a variable called `$template`. The second parameter is an array of variables that need to be provided to the template to customize the page's content. This array is typically stored in a variable called `$args`, since the array is essentially providing arguments to the template. The string keys in the array should correspond to the names of any variables used in the Twig template. If no values need to be provided to the template, `$args` will be an empty array.

To illustrate, say you wanted to display a page showing the items in a shopping cart. The template for this page would be in a file that is called *shoppingCart.xhtml.twig* (by convention, Twig template files that output HTML are named in the form *<pageName>.xhtml.twig*), and it would have a `products` variable standing in for whatever products are in the user's cart (variables in Twig templates don't start with dollar signs).

You would pass the template filepath as the first argument to the `Twig\Environment` class's `render()` method. The second argument would be an `$args` array containing a `'products'` key (the same name as the variable in the template) whose value is an array of `Product` objects. The

`render()` method would then return a string containing the HTML for a web page listing all the products as a nice shopping cart, with subtotals, the grand total, links to remove items or change their quantities, and so on. This string could then be printed to the output buffer that would become the body of the HTTP response being returned to the web client.

If a web application uses Twig templating, every method that returns HTML response text will contain something similar to the code shown in [Listing 21-1](#).

```
$template = 'path/templateName.xhtml.twig';
$args = [
    'variable1NameForTwig' => $phpVariable1,
    'variable2NameForTwig' => $phpVariable2,
    ...
];
$html = $this->twig->render($template, $args);
print $html;
```

Listing 21-1: The typical code to create and print HTML from a Twig template

We store the path to a template in the `$template` variable, then build up the `$args` array with the required template variables. We pass these variables to the `render()` method. Remember, we typically store a reference to the `Twig\Environment` object that owns the `render()` method in the `twig` property of an overarching `Application` class. This is why the call to the method is written as `$this->twig->render()` rather than `$twig->render()`. We store the resulting string in `$html`, which we print. Soon you'll see this code pattern in context as we use Twig to create a basic web page.

A Simple Example

In this section, we'll use Twig to create a simple "Hello, world!" web page that displays a greeting. Thanks to Twig, we'll be able to customize the greeting by filling in a name for the person being greeted, based on the value of a PHP variable. In addition to illustrating the basics of working with Twig, this project will also offer a first glimpse at how an object-oriented web application is structured.

To create our basic greeting page, we'll start by setting up Twig. Then we'll write a Twig template featuring a `name` variable, as well as the PHP scripts needed to make the template work.

Adding Twig to the Project

The easiest way to add Twig to a project is to use Composer. Create a new empty folder for the project and then enter the following at the command line:

```
$ composer require twig/twig
```

This command triggers Composer to install the latest version of the Twig package into the project's *vendor* folder (the folder will be created, since the project doesn't already have one). The command also installs any additional dependencies that Twig requires, such as `symfony/polyfill-mbstring` and `symfony/polyfill-ctype`. If you look at the contents of the *vendor* folder after installation has finished, you should see that folders for these packages have been created. You can even look in each package's *src* folder and examine each class and configuration file in the package.

Now that the Twig package has been copied into our project folder, we can add `use` statements in our classes to create and exploit the features of the library in our project.

Writing the Twig Template

By convention, Twig template files are stored in a *templates* folder, just like the PHP template files we've written in earlier chapters. Add this folder to your project directory, then create a *hello.xhtml.twig* template in that folder and enter the contents of [Listing 21-2](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>Twig hello</title>
</head>

<body>
```



```
    ❶ Hello {{name}}, nice to meet you
</body>
</html>
```

Listing 21-2: The hello.xhtml.twig template to create an HTML greeting around the `name` variable

Like many web templating languages, Twig templates use double curly brackets to indicate a value that should be filled in, such as `{{name}}` in this template ❶. This declares that the value of a Twig variable called `name` is to be inserted here when the `render()` method generates its HTML output string. In the PHP script invoking this template, we'll need to pass Twig a value for the variable in the `$args` array, under a key of the same name. For example, if we declared the `$args` array as `['name' => 'Matt']`, the `render()` method would produce the HTML shown in [Listing 21-3](#), with **Matt** (shown here in bold) inserted in place of the `name` variable.

```
<!doctype html>
<html lang="en">
<head>
    <title>Twig hello</title>
</head>

<body>
    ❶ Hello Matt, nice to meet you
</body>
</html>
```

Listing 21-3: The HTML rendered by the Twig template when `name` contains 'Matt'

We see `Matt` has been inserted into the line saying hello in the HTML text output ❶.

Twig's double curly brackets are similar to the PHP `<?=` short echo tag, in that both set off expressions that evaluate to strings and will be inserted into the output HTML. (Just as with PHP, Twig has other tags for setting off logic, such as loops and conditionals, rather than string output statements.) However, since Twig templates can't contain PHP statements, using Twig

double curly brackets instead of PHP short echo tags protects a website from PHP security vulnerabilities in the site's templates.

Creating the Application Class

Let's now write an `Application` class for our greeting page. It will set up the `Twig\Environment` object in its constructor method and have a `run()` method to set variables and generate the page's HTML via Twig. All object-oriented MVC web applications have something like this `Application` class to perform any setup and initializations required, and then to execute the main logic for handling requests to the application.

For now, the class will always output the same HTML, but later in the chapter, we'll see some logic in the class's `run()` method to output different HTML content depending on variables received in the request to the server. To declare the class, create `src/Application.php` containing the code in [Listing 21-4](#).

```
<?php
namespace Mattsmithdev;

use \Twig\Loader\FilesystemLoader;
use \Twig\Environment;

class Application
{
    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    private Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLATES);
        ❶ $this->twig = new Environment($loader);
    }

    public function run()
    {
        $name = 'Matt';
```

```

        $template = 'hello.xhtml.twig';
        $args = [
            'name' => $name,
        ];

        ❷ $html = $this->twig->render($template, $args);
        print $html;
    }
}

```

Listing 21-4: The class `src/Application.php` to create a Twig object and a method to output HTML

We declare the `Application` class as part of the `Mattsmithdev` namespace and include two `use` statements, since we'll need to create objects of classes from the `Twig` namespace in the constructor. Then we declare a constant named `PATH_TO_TEMPLATES` holding the path to the base *templates* directory where all the templates are saved. We also declare a private `twig` property, which will be a reference to a `Twig\Environment` object.

Next, we declare the class's constructor method. Within it, we create two Twig-related objects, `FilesystemLoader` and `Environment`. The latter holds the all-important `render()` method, while the former helps the `Environment` object access the template files. The `FilesystemLoader` object is created only temporarily, since its reference, stored in the `$loader` variable, exists only within the scope of the constructor. When we create the `Environment` object (using `$loader`), we store a reference to it in the `Application` object's `twig` property ❶, which all our `Application` methods can access when `$this->twig` is written.

Notice that when we create the `FilesystemLoader` object by using the `PATH_TO_TEMPLATES` constant, we must add `self::` before the constant identifier. As we'll discuss in [Chapter 25](#), this prefix is necessary when referencing a constant declared in the same class. This is because PHP doesn't create a copy of the constant for each object of the class. Instead, there's just one constant for all objects of the class, so writing `$this->PATH_TO_TEMPLATES` isn't valid.

We next declare the `Application` class's `run()` method. In it, we define the `$name` variable containing the name of the person we want to greet. Then we create the `$template` variable, which holds the name of our template file (*hello.xhtml.twig*), and the `$args` variable, an array with a single element holding the value of the `$name` variable under the `'name'` key. As we've discussed, this key corresponds to the Twig variable enclosed in double curly brackets in our template file. Still within the `run()` method, we then invoke the `render()` method of our Twig object, storing the string it returns in the `$html` variable ❷. Finally, we print the content of `$html`, which becomes the HTML body of the response returned to the web client.

Creating the Autoloader

Let's now get Composer to create the autoloader for `Mattsmithdev` namespaced classes located in the `src` directory (such as our `Application` class). To do this, we need to add an `"autoload"` property to *composer.json* in the top-level directory for the project. This file was created automatically when we used Composer to add the Twig package to the project, and it should already contain a `"require"` property with information about Twig. Update the file as shown in [Listing 21-5](#).

```
{
    "autoload": {
        "psr-4": {
            "Mattsmithdev\\": "src"
        }
    },
    "require": {
        "twig/twig": "^3.10"
    }
}
```

Listing 21-5: Updating composer.json for class autoloading

We add an `"autoload"` property, declaring that `Mattsmithdev` namespaced classes are PSR-4 compliant and can be found in the `src` directory. Don't forget the comma after the closing curly bracket ❶; you'll get a JSON syntax error if this comma is missing.

Once you've updated *composer.json*, enter the following at the command line:

```
$ composer dump-autoload
```

This instructs Composer to generate the necessary autoload scripts in the *vendor* folder.

Adding the Index Script

The final step required to get our greeting page up and running is to create a simple index script that will read in the autoloader, create an `Application` object, and invoke its `run()` method. All our object-oriented web applications will have a simple index script like this, since all the work is being performed by the `Application` object. Create *public/index.php* as shown in [Listing 21-6](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Application;
$app = new Application();
$app->run();
```

Listing 21-6: The index.php script

We first read in and execute the generated autoloader script. Note that the autoloader created by Composer will load any class declared in the `Mattsmithdev` namespace, as well as any classes in third-party libraries that Composer has added to the project (such as Twig). Next, we add a `use` statement so we can refer to the `Application` class without specifying the namespace each time. Then we create an `Application` object and invoke its `run()` method.

If you run the PHP web server and visit the project home page, you should see something like [Figure 21-1](#). Also shown in the figure is the HTML source of the response that the web client receives. Notice that the HTML and the resulting web page both have the name *Matt* filled in at the location of the Twig `{{name}}` variable.

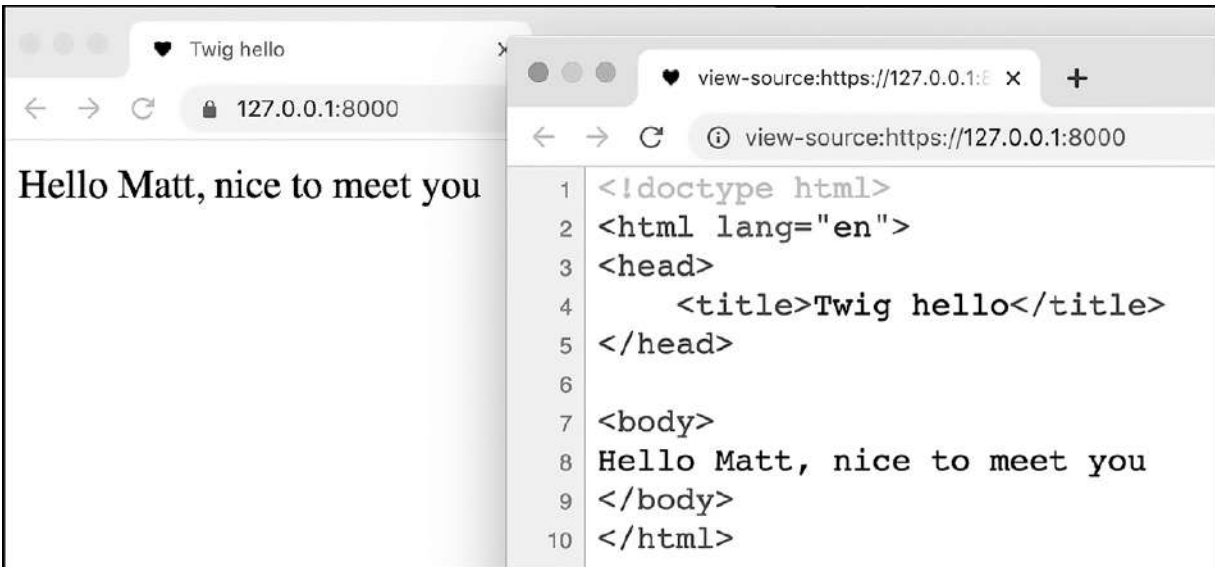


Figure 21-1: The web output, with HTML source, for our basic Twig project

This might have seemed like *a lot* of work for a simple “Hello, world!” website, but we’ve now created all the structure required for any project that uses the powerful Twig templating system. Before we apply that structure to a more substantial website, let’s explore a few more useful features of the Twig package.

Manipulating Objects and Arrays in Twig Templates

In addition to simple data types like strings, Twig templates can work with PHP objects, arrays, and so on. However, the notation is a bit different from what we’re used to in PHP, since Twig uses *dot notation* to access a property or method of an object, or an element in an array. For example, while you’d write `$product->price` in PHP to access the `price` property of a `Product` object, you’d use `{{product.price}}` to do the same in a Twig template. Conveniently, this works whether the property is public or private, provided that if it’s private, the property has a public getter method that follows the usual `getProperty()` or `isBooleanPropertyName()` naming convention. For example, `{{product.price}}` would still successfully access an object’s private `price` property as long as the object has a public `getPrice()` method. You don’t need to reference the

`getPrice()` method explicitly in the Twig template, since Twig calls the method for you.

To illustrate how Twig templates work with these more complex data types, we'll update our "Hello, world!" web page to display information obtained from a PHP object and array. First, we'll need to write a class so we can create an object and pass it to the Twig template. [Listing 21-7](#) shows a simple `Product` class that we can use as an example. Create *src/Product.php* and enter the contents of the listing.

```
<?php
namespace Mattsmithdev;

class Product
{
    private string $description;
    private float $price;

    public function getDescription(): string {
        return $this->description;
    }

    public function setDescription(string $description): void {
        $this->description = $description;
    }

    public function getPrice(): float {
        return $this->price;
    }

    public function setPrice(float $price): void {
        $this->price = $price;
    }

    public function __toString(): string {
        return "(PRODUCT) description =
            $this->description / price = $this->price";
    }
}
```

```
}  
}
```

Listing 21-7: A simple `Product` class to use in the Twig demo

The class has two private properties: `description` and `price`. The code also has public getter and setter methods for each of these properties, as well as a `__toString()` method that generates a string summary of the object.

Next, we'll modify the `run()` method of our `Application` class. The new method will create an array and a `Product` object and pass them along to the Twig template, along with the original `name` variable. Update `src/Application.php` to match the contents of [Listing 21-8](#).

```
--snip--  
public function run()  
{  
    $meals = [  
        'breakfast' => 'toast',  
        'lunch' => 'salad',  
        'dinner' => 'fish and chips',  
    ];  
  
    $product1 = new Product();  
    $product1->setDescription('bag of nails');  
    $product1->setPrice(10.99);  
  
    ❶ $template = 'demo.xhtml.twig';  
    $args = [  
        'name' => 'matt',  
        'meals' => $meals,  
        'product' => $product1  
    ];  
  
    $html = $this->twig->render($template, $args);  
    print $html;  
}  
}
```

Listing 21-8: The updated `Application` class passing an object and an array to the template

We create a `$meals` array with the keys `'breakfast'`, `'lunch'`, and `'dinner'`, along with a `Product` object with the description `'bag of nails'` and the price `10.99`. Then we declare the `$template` and `$args` variables required for the `render()` method. In `$args`, we pass values for three variables in the Twig template: `name`, `meals`, and `product`. This time we declare the value for the `'name'` key directly in the array rather than as a separate variable.

Notice that we declare the Twig template to be *demo.xhtml.twig* rather than the *hello.xhtml.twig* template we created earlier ❶. We'll create that new template now and design it to show off some of the ways Twig interacts with objects and arrays. Make a copy of *hello.xhtml.twig*, rename it *demo.xhtml.twig*, and update this new file to match [Listing 21-9](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>Twig examples</title>
</head>

<body>
❶ Hello {{name}}
<hr>
❷ for dinner you will have: {{meals.dinner}}
<hr>
the price of ❸ {{product.getDescription()}} is $ ❹ {{product.price}}

<hr>
❺ details about product: {{product}}

</body>
</html>
```

Listing 21-9: The `demo.xhtml.twig` template

In the body of the template, we first print the value of the Twig `name` variable ❶. Then we print the value inside the `meals` array variable under the `dinner` key ❷. The Twig dot notation of `{{meals.dinner}}` here corresponds to the PHP expression `$meals['dinner']`.

Next, we print the value returned from the `getDescription()` method of the `product` object variable ❸. In this case, the Twig dot notation of `{{product.getDescription()}}` corresponds to the PHP expression `$product->getDescription()`. We also print the value of the object's `price` property ❹. When Twig tries to access this `price` property, it will see that the property is `private`, so it will automatically attempt to invoke the object's `getPrice()` accessor method instead. The Twig dot notation of `{{product.price}}` therefore corresponds to the PHP expression `$product->getPrice()`.

Finally, we place the Twig `product` variable inside double curly brackets without any dot notation ❺. When Twig sees that `product` is an object, it will automatically attempt to invoke its `__toString()` method. This is similar to PHP automatically invoking an object's `__toString()` method when a reference to that object is used in a context where a string is expected. Essentially, `{{product}}` in Twig corresponds to `print $product` in PHP, which in turn corresponds to `$product->__toString()`. [Figure 21-2](#) shows how the HTML rendered from this template will appear in the browser.

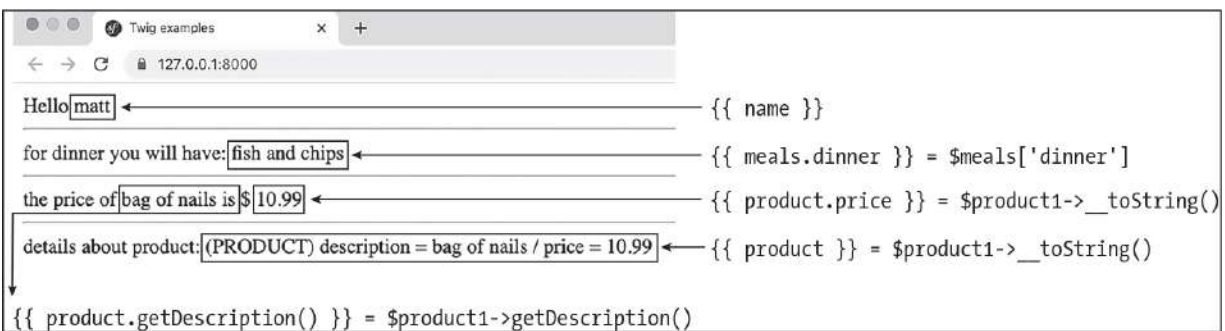


Figure 21-2: A browser rendering of the HTML from the Twig demo template

As you can see, Twig has successfully filled in all the information from the object and array passed to the template. The annotations in the figure

summarize the Twig notation used to access each piece of information, and the equivalent expressions in PHP.

Twig Control Structures

Beyond printing individual values, the Twig templating language also offers several control structures, including `if` and `for` statements, allowing you to add conditional logic and looping to your Twig templates. This greatly expands the templates' ability to adapt to whatever data they receive in the `$args` array.

Twig control statements are written within single curly brackets and percent characters, such as `{% if condition %}` or `{% for ... %}`, and each control structure must conclude with a closing tag, such as `{% endif %}` or `{% endfor %}`.

Twig can loop through all the values in an array with `{% for value in array %}`, much like a `foreach` loop in PHP. If you want the key and value for each array item, you could write `{% for key, value in array %}`. Let's try that now by creating a Twig template that will loop through the items in the PHP `$meals` array we created in the preceding section. Update the Twig template *demo.xhtml.twig* as shown in [Listing 21-10](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>Twig examples</title>
</head>

<body>
<ul>
    ❶ {% for key, value in meals %}
        <li>
            ❷ meal: {{key}} = {{value}}
        </li>
    ❸ {% else %}
        <li>
            (there are no meals to list)
        </li>
```

```
    ❷ {% endfor %}  
</ul>  
</body>  
</html>
```

Listing 21-10: Looping through and outputting meals as an HTML list

We declare a Twig `for` loop to iterate through the elements of the `meals` array ❶. The key and value of each element in the array will become part of a list item in an HTML unordered list and will be output as HTML in the form `<li>meal: lunch = salad</li>` ❷. Instead of having to write separate HTML for each item in the list, we simply write one item using the Twig variables `key` and `value`, and the `for` loop will generate all the items for us. The loop also includes a Twig `else` statement ❸, which is executed if the given array is empty. In that case, we output a message stating that no meals are in the list. The loop concludes with the closing `endfor` tag ❹. [Figure 21-3](#) shows the web page that's rendered when you serve this code with the web server.



Figure 21-3: The web page and HTML source generated by the Twig `for` loop

The Twig `for` loop has successfully generated an HTML list by using the keys and values from the `$meals` array.

Creating a Multipage Website with Twig

For the rest of this chapter, we'll harness Twig templating to create a multipage website. In addition to demonstrating the value of Twig templating, building this site will illustrate how the front-controller structure we've used in previous chapters translates to an object-oriented web application. [Figure 21-4](#) shows the site we'll progressively develop: a simplified two-page version of the website we created in [Chapter 16](#), with the login page removed.

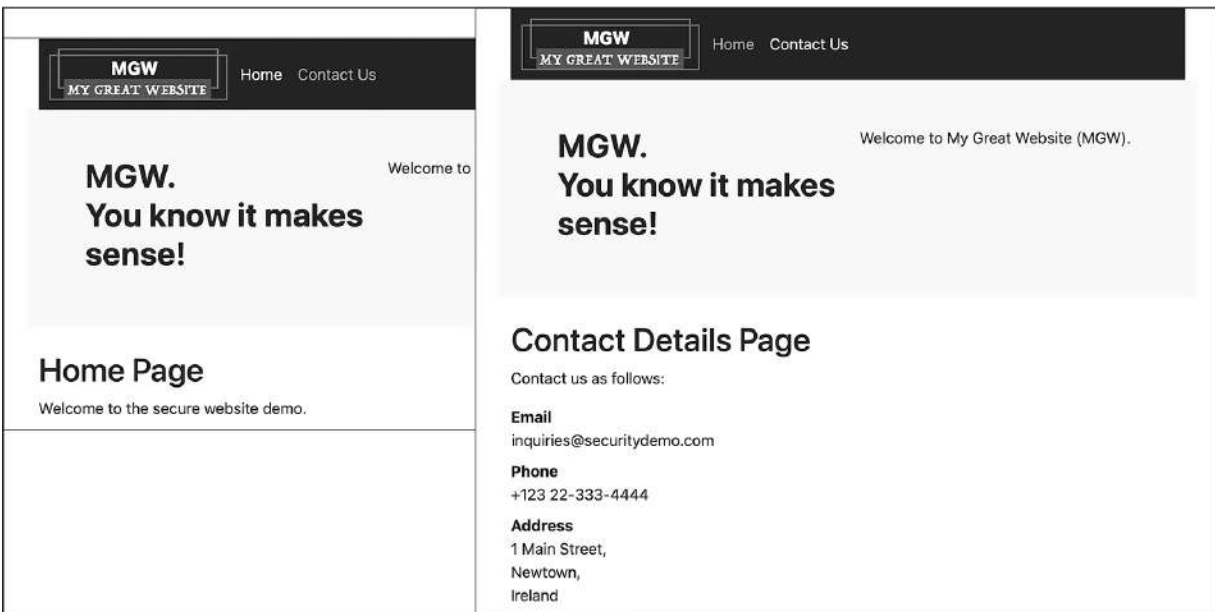


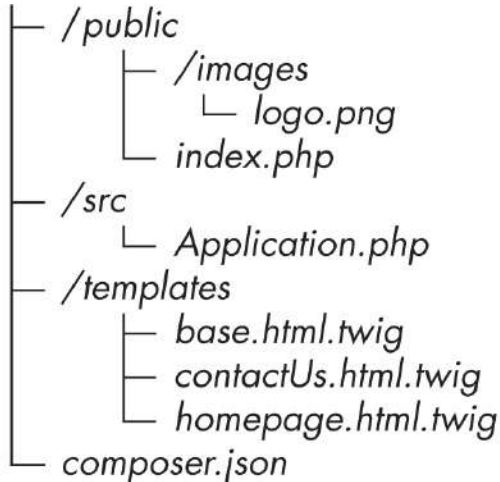
Figure 21-4: A two-page website created with Twig templating

Our site will have a home page and a Contact Us page, with the same header and navigation links shared by the two pages. All the common HTML will be declared in a *base template* from which the page-specific content templates will inherit.

This means that each *page (child) template* will contain only the special content for that page. It also means that every page in the website can be changed simply by updating the base template (for example, if we wanted to add or change the navigation links, change the logo, or make the website background turn green for St. Patrick's Day).

The File Structure and Dependencies

Let's first establish the file structure for the application. Create a new project folder. Inside it, we'll build the following directories and files:



Two files are exactly the same as the examples shown earlier in this chapter: *public/index.php* and *composer.json*. As you saw in [Listing 21-6](#), the *index.php* script simply reads in the autoloader, creates an object of the `Application` class, and calls the class's `run()` method. The *composer.json* file (Listing 21-5) provides information for the autoloader and about the project's namespace (`MattSmithDev`) as well as third-party library requirements (Twig). Copy these two files into the folder created for the project.

Also copy the *public/images/logo.png* image provided in the files accompanying this book at <https://github.com/dr-matt-smith/php-crash-course> (or use your own logo image). Finally, since we'll be using the same namespace and Twig library as before, you can also copy the *vendor* folder to get the same autoloader and library files.

The Application Class

The `Application` class for our project plays the role of a front controller: it determines which page of the website to display based on the URL-encoded navigation action. If the URL has no `action` variable, `Application` will display the home page. If `Application` finds an `action` variable with the value `contact`, it will display the Contact Us page.

The `Application` class is also responsible for creating a `Twig\Environment` object to manage the Twig templates, and for this reason the first several lines of the `Application` class declaration are the same as they were for our “Hello, world!” project earlier in the chapter. Copy *src/Application.php* from the previous project and update it to match [Listing 21-11](#).

```
<?php
namespace Mattsmithdev;

use \Twig\Loader\FilesystemLoader;
use \Twig\Environment;

class Application
{
    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    private Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLATES);
        $this->twig = new Environment($loader);
    }

    ❶ public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        switch ($action) {
            case 'contact':
                $this->contactUs();
                break;

            case 'home':
            default:
                $this->homepage();
        }
    }
}
```

```

    }

    ❷ private function homepage(): void
    {
        $template = 'homepage.xhtml.twig';
        $args = [
            'pageTitle' => 'Home Page'
        ];

        $html = $this->twig->render($template, $args);
        print $html;
    }

    ❸ private function contactUs(): void
    {
        $template = 'contactUs.xhtml.twig';
        $args = [
            'pageTitle' => 'Contact Us Page'
        ];

        $html = $this->twig->render($template, $args);
        print $html;
    }
}

```

Listing 21-11: *The `Application` class for the two-page website*

We first declare the `run()` method ❶, which takes the place of the front-controller code we've previously written in an *index.php* script. The method attempts to find a URL-encoded variable named `action`, then feeds its value to a typical front-controller `switch` statement. If the value is `'contact'`, the `contactUs()` method is invoked. Otherwise, the `homepage()` method is invoked.

Next, we declare the `homepage()` method ❷. It prints the result of running the `render()` method of the `twig` property (which contains a reference to the `Twig\Environment` object). When we call `render()`, we pass in `$template`, which has a value of `'homepage.xhtml.twig'`, and pass

in the `$args` array, which provides the Twig `pageTitle` variable with the value `'Home Page'`.

We also declare the `contactUs()` method for displaying the Contact Us page ❸. This method similarly calls `render()` and prints the result, this time passing in `$template` with a value of `'contactUs.xhtml.twig'` and `$args` providing the Twig `pageTitle` variable with the `'Contact Us Page'` value.

These two methods, `homepage()` and `contactUs()`, replace stand-alone helper functions that we earlier would have written in a *functions.php* file. In this way, our object-oriented application encapsulates all the display logic within the `Application` class.

The Twig Templates

Now all we have to do to complete our site is write the Twig template files for the two pages. We'll start with the home page. Create *templates/homepage.xhtml.twig* containing the code in [Listing 21-12](#).

```
<!doctype html>
<html lang="en">
<head>
  ❶ <title>MGW: {{pageTitle}}</title>
</head>

<body>
<header>
  ❷ 
  <ul>
    <li>
      ❸ <a href="/">
        Home
      </a>
    </li>
    <li>
      ❹ <a href="/?action=contact">
        Contact Us
      </a>
    </li>
```

```

        </ul>
    </header>

    ❸ <blockquote>
        <p>
            <b>MGW. </b>
            <br>You know it makes sense!
        </p>
        <p>
            Welcome to My Great Website (MGW).
        </p>
    </blockquote>

    ❹ <h1>{{pageTitle}}</h1>

    ❺ <p>
        Welcome to the secure website demo.
    </p>

    </body>
</html>

```

Listing 21-12: The homepage.xhtml.twig template

We declare the HTML title, outputting `MGW` followed by the contents of the Twig `pageTitle` variable ❶. Then we display the site logo image ❷. We next present a simple navigation list with links to the home page ❸ and the Contact Us page ❹. Then we use a `<blockquote>` element to present the website’s tagline and greeting ❺, followed by a level 1 heading that again uses the Twig `pageTitle` variable ❻. Finally, we declare the page-specific content; for this home page, it’s just a sentence in a paragraph ❼.

Listing 21-13 shows the parts of the page that are different for the Contact Us Twig template. Copy *homepage.xhtml.twig*, name the copy *contactUs.xhtml.twig*, and edit this file to match this listing.

```

--snip--
    </p>
    <p>

```

```

        Welcome to My Great Website (MGW).
    </p>
</blockquote>

<h1>{{pageTitle}}</h1>

❶ <p>
    Contact us as follows:
</p>

<dl>
    <dt>Email</dt>
    <dd>inquiries@securitydemo.com</dd>

    <dt>Phone</dt>
    <dd>+123 22-333-4444</dd>

    <dt>Address</dt>
    <dd>1 Main Street,<br>Newtown,<br>Ireland</dd>
</dl>
</body>
</html>

```

Listing 21-13: The contactUs.xhtml.twig template

The only content of this template that differs from the home page is the paragraph and definition list at the end of the HTML body ❶. Thanks to the use of the Twig `pageTitle` variable, the rest of the template is identical. Twig will fill in the variable with `Home Page` or `Contact Us Page` as appropriate.

At this point, the website has everything we need to display and navigate the two pages. If you run the project, you'll see something like [Figure 21-5](#).

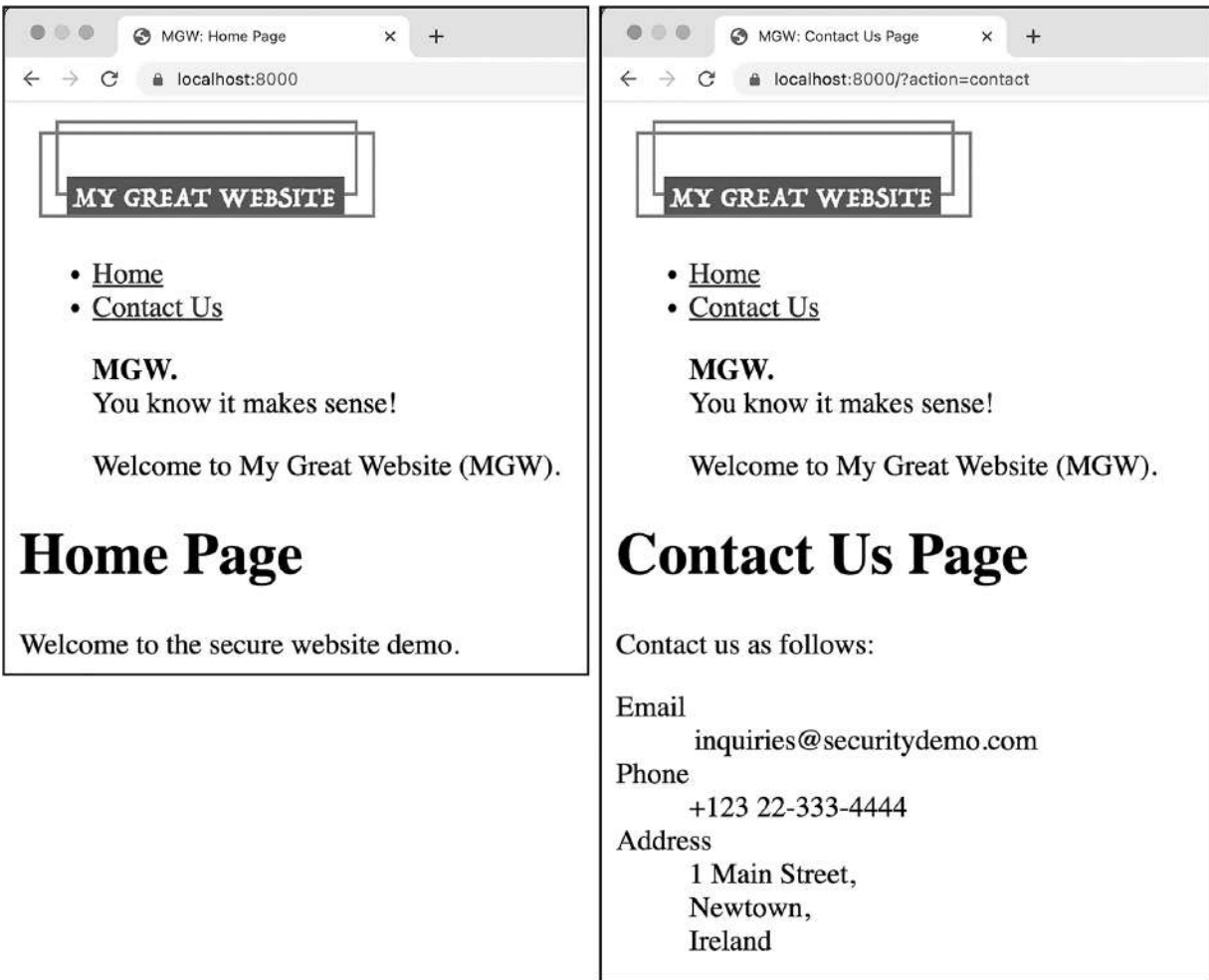


Figure 21-5: A simple two-page site built with Twig

Notice that Twig has correctly filled in the value of the `pageTitle` variable for each page.

Twig Features to Improve Efficiency

Our two-page website now works as expected, but a lot of duplicated code remains in the two Twig template files. In this section, we'll explore techniques for improving the efficiency of our templates, such as `include` statements and template inheritance. Features like these make Twig particularly useful for developing multipage web applications.

include Statements

As you've seen, pages in a website typically share much of the same HTML code. Twig `include` statements make it possible to create partial templates

containing that shared code and add the rendered output from those partial templates to the actual page templates that need it. These `include` statements take the form `{{include(templateName)}}`.

To demonstrate, we'll take all the shared content at the top of each of our two page templates and put it in a common `_header.xhtml.twig` file. (Remember, it's common to prefix partial templates like this with an underscore.) Then we'll use `include` statements to add the partial template to the top of our page template files, which we'll be able to shorten to contain just the content unique to each page.

Create `templates/_header.xhtml.twig` and copy in the code from the top of one of the page template files, as shown in [Listing 21-14](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>MGW: {{pageTitle}}</title>
</head>

--snip--
    <p>
        Welcome to My Great Website (MGW).
    </p>
</blockquote>

<h1>{{pageTitle}}</h1>
```

Listing 21-14: The partial `_header.xhtml.twig` template

Everything up to and including the level 1 heading (again, using the Twig `pageTitle` variable) has been moved into this `_header.xhtml.twig` partial template. With that, we can greatly reduce the content in the home page and Contact Us templates. [Listing 21-15](#) shows the updated home page Twig template, replacing all the duplicated content with a simple Twig `include` statement. Update `templates/homepage.xhtml.twig` to match the contents of this listing.

```
{{include('_header.xhtml.twig')}}

<p>
    Welcome to the secure website demo.
</p>

</body>
</html>
```

Listing 21-15: The simplified homepage.xhtml.twig template

We begin with a the Twig `include` statement, `{{include('_header.xhtml.twig')}}`, telling Twig to read in the partial template file `_header.xhtml.twig`. All that remains in the template is the page-specific content.

We can similarly remove the duplicated content from our Contact Us template. Update `templates/contactUs.xhtml.twig` as shown in [Listing 21-16](#).

```
{{include('_header.xhtml.twig')}}
<p>
    Contact us as follows:
</p>

<dl>
    <dt>Email</dt>
    <dd>inquiries@securitydemo.com</dd>

    <dt>Phone</dt>
    <dd>+123 22-333-4444</dd>

    <dt>Address</dt>
    <dd>1 Main Street,<br>Newtown,<br>Ireland</dd>
</dl>

</body>
</html>
```

Listing 21-16: The simplified contactUs.xhtml.twig template

Once again, we've removed the duplicated content and replaced it with a Twig `include` statement. The page-specific content follows.

Using Twig `include` statements, we've drastically simplified our individual page templates. However, notice that we still have the final `</body>` and `</html>` tags shared by both templates. In theory, we could relocate these to a partial `_footer.xhtml.twig` template. For a simple website, this could be a reasonable approach, but for more complex pages and larger websites, Twig offers an even more powerful feature for consolidating redundant content than `include` statements: template inheritance.

Template Inheritance

Template inheritance involves creating a *base template* with all the content shared by a group of web pages, then creating individual *child templates* that extend the base template by filling in or overriding just the content unique to a particular page. It's much like the OOP technique of creating subclasses that inherit from, extend, and override certain behaviors of a superclass.

The base template ensures that all website pages have all the required valid, well-formed HTML, including ending tags, freeing up the child templates to focus on their own page-specific content. As you'll see, this inheritance approach is much neater than using `include` statements to incorporate partial templates.

INCLUDE IS SOMETIMES USEFUL EVEN WHEN USING INHERITANCE

Once we are using Twig inheritance, all we need for many web applications are base and child templates, since all the common page content is declared in the base templates. So it may seem there is no need to use Twig `include` statements and partial templates. However, sometimes a web application developer may choose to declare rarely used content in partial templates, which can then be included by those few pages that need them. The chosen approach is often a personal preference between keeping everything inheritance based versus simplifying the base template to declare only the most commonly used content blocks.

To use template inheritance, we'll first convert our *templates/_header.xhtml.twig* file into a base template that the other templates can inherit from and extend. Rename *_header.xhtml.twig* to *base.xhtml.twig* and edit it to match [Listing 21-17](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>MGW: {{pageTitle}}</title>
</head>

--snip--
    <p>
        Welcome to My Great Website (MGW).
    </p>
</blockquote>

<h1>{{pageTitle}}</h1>

{% block main %}
{% endblock %}

</body>
</html>
```

Listing 21-17: The base.xhtml.twig template

The key to template inheritance is to use Twig statements in the base template to delineate blocks of code that will be filled in or overridden in each child page template. In this example, we define a block called `main`. This is where the unique content for each page will go. In our base template itself, however, the block is empty, so nothing is between the `block` and `endblock` statements. Twig blocks have names (in this case, `main`) so that child templates can specify which blocks (if any) are to be overridden with page-specific content.

Notice that in this inheritance approach, we have a full web page in the base template; that is, it isn't a partial template. In particular, the base template includes closing `</body>` and `</html>` tags. A Twig base template

is a complete HTML page in its own right, although it may have default or empty blocks meant to be overridden in the child templates.

We can now update our home page and Contact Us templates to inherit from the base template and override the `main` block with individual page content. First, update *templates/homepage.xhtml.twig* to match [Listing 21-18](#).

```
{% extends 'base.xhtml.twig' %}

{% block main %}
<p>
    Welcome to the secure website demo.
</p>
{% endblock %}
```

Listing 21-18: The homepage.xhtml.twig template, inheriting from the base template

We declare that this template extends (inherits from) *base.xhtml.twig*. Then we embed the page-specific paragraph inside the `main` block, overriding the empty contents of this block in the base template. We finish with an `endblock` statement so that Twig knows where the overriding content ends. These `endblock` statements are particularly important because in more complex pages we may be overriding two or more blocks in a child page template. Notice that we no longer have the closing HTML tags at the end of the file, since these have moved to the base template.

We next need to make the same changes to the Contact Us page. Update *templates/contactUs.xhtml.twig* to match [Listing 21-19](#).

```
{% extends 'base.xhtml.twig' %}

{% block main %}
<p>
    Contact us as follows:
</p>

<dl>
    <dt>Email</dt>
    <dd>inquiries@securitydemo.com</dd>
```

```
<dt>Phone</dt>
<dd>+123 22-333-4444</dd>

<dt>Address</dt>
<dd>1 Main Street,<br>Newtown,<br>Ireland</dd>
</dl>

{% endblock %}
```

Listing 21-19: The contactUs.xhtml.twig template, inheriting from the base template

Once again, we use `extends` so that this template will inherit from *base.xhtml.twig*, and we override the `main` block with the page-specific content. As before, we close out the `main` block with an `endblock` statement.

Blocks Instead of Variables

For this simple, static, two-page website, we shouldn't need to pass any variables to the Twig templates through the `$args` array when we call the `render()` method. At present, our controller methods are passing the page title as a Twig variable named `pageTitle`. However, we could instead make the page title a block in the base template and override the block with the appropriate text in each child template.

Let's remove the `pageTitle` variable being passed by the controller methods in our `Application` class. Update *src/Application.php* to match [Listing 21-20](#).

```
--snip--
private function homepage(): void
{
    $template = 'homepage.xhtml.twig';
    $args = [];

    $html = $this->twig->render($template, $args);
    print $html;
}
```

```

private function contactUs(): void
{
    $template = 'contactUs.xhtml.twig';
    $args = [];

    $html = $this->twig->render($template, $args);
    print $html;
}
}

```

Listing 21-20: Passing an empty `$args` array in the `Application` class

We declare `$args` as an empty array in the `homepage()` and `contactUs()` methods. While we could just pass an empty array as the second argument to `render()`, first declaring the array as a variable clarifies whether any variables are being passed to the Twig template.

We must now update the base template to declare a block for the page title, rather than printing the contents of a Twig variable. Update `templates/base.xhtml.twig` as shown in [Listing 21-21](#).

```

<!doctype html>
<html lang="en">
<head>
    <title>MGW: {% block pageTitle %}{% endblock %}</title>
</head>

--snip--
    <p>
        Welcome to My Great Website (MGW).
    </p>
</blockquote>

```

❶ `<h1>{{block('pageTitle')}}</h1>`

```

{% block main %}
{% endblock %}

```

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

Listing 21-21: Adding a page title block to the base.xhtml.twig template

We declare a new `pageTitle` Twig block whose contents will become part of the HTML `<title>` element. We also need to repeat the contents of this block later as a level 1 heading ❶. We aren't permitted to declare a second block with the same name, however. Instead, we print the contents of the block by using Twig's `block()` function, which takes an argument indicating the name of the block whose contents should be output. We need to enclose this function call in double curly brackets, just like other Twig expressions.

All that remains is to update each child page to declare a `pageTitle` block containing an appropriate page name, overriding the default empty `pageTitle` block in the base template. Update *templates/homepage.xhtml.twig* to match [Listing 21-22](#).

```
{% extends 'base.xhtml.twig' %}

{% block pageTitle %}Home Page{% endblock %}

{% block main %}
<p>
    Welcome to the secure website demo.
</p>
{% endblock %}
```

Listing 21-22: Declaring a page title block in the homepage.xhtml.twig template

We declare the `pageTitle` block with the `Home Page` content. This one declaration is enough to fill in the page title at both locations in the base template. Update *templates/contactUs.xhtml.twig* in the same way, declaring the `pageTitle` block with the `Contact Us Page` content.

When you now load the website, you should see that nothing has changed. However, our use of template inheritance has made the code much more efficient.

Improved Page Styling with CSS

Our website is working, and the template code is efficient and well organized, but the pages themselves don't look very appealing. We'll round out the website by introducing some CSS to give it a more polished design. With all the common content for the site confined to the single *base.xhtml.twig* template file, you'll see that Twig makes this process of updating the site's appearance quite straightforward.

Highlighting the Current Navigation Link

Highlighting the current page's navigation bar link is a common way to inform the user which page they're viewing. In [Chapter 16](#), we did this with PHP variables. Now you'll see how Twig template inheritance makes the process even easier. In the base template, we'll declare a uniquely named Twig block for the content of the `class` attribute for each link element in the navigation list. Then, in the child templates, we'll override the appropriate Twig block to set the current page link's `class` attribute to `active`. We'll use CSS to style the active link a different color from others.

To begin, update *base.xhtml.twig* to match [Listing 21-23](#).

```
<!doctype html>
<html lang="en">
<head>
    <title>MGW: {% block pageTitle %}{% endblock %}</title>
    <style>@import '/css/style.css'</style>
</head>

<body>
<header>
    
    <ul>
        <li>
            <a href="/" class="{% block homeLink %}{% endblock %}">
                Home
            </a>
        </li>
        <li>
```

```

        <a href="/?action=contact"
            class="{% block contactLink %}{% endblock
%}">
            Contact Us
        </a>
    </ul>
</header>

--snip--

```

Listing 21-23: The *base.html.twig* template with blocks for the navigation links

We add a style import declaration so that all pages of the site will be able to use the CSS styles declared in *public/css/style.css* (which we'll create shortly). Then we declare a `homeLink` Twig block as the content for a `class` attribute for the link to the home page. The block is empty, so if it isn't overridden, the link won't be assigned to a class. We similarly declare a `contactLink` Twig block as the content for a `class` attribute for the Contact Us link.

We now need to make the child templates override these blocks. Update the *homepage.html.twig* template file as shown in [Listing 21-24](#).

```

{% extends 'base.html.twig' %}

{% block pageTitle %}Home Page{% endblock %}

{% block homeLink %}active{% endblock %}

{% block main %}
<p>
    Welcome to the secure website demo.
</p>
{% endblock %}

```

Listing 21-24: Declaring a *homeLink* block in *homepage.html.twig*

We declare the `homeLink` block to have `active` content, thereby assigning it to a CSS class that will highlight the page link in a different

color from the default navigation links. [Listing 21-25](#) shows how to update the *contactUs.xhtml.twig* template file in the same way.

```
{% extends 'base.xhtml.twig' %}

{% block pageTitle %}Contact Us Page{% endblock %}

{% block contactLink %}active{% endblock %}

{% block main %}
<p>
    Contact us as follows:
</p>
--snip--
```

Listing 21-25: Declaring a `contactLink` block in `contactUs.xhtml.twig`

We declare the `contactLink` block with `active` content, which again will highlight the page's link when the user visits it.

To finish, we need to declare some simple CSS rules. We'll give the page header (containing the navigation list) a dark background color and define default and active colors for the links. Create a new *public/css* folder for the project, then create *style.css* within it and enter the contents of [Listing 21-26](#).

```
header {
    background-color: rebeccapurple;
}

❶ a {
    color: gray;
    text-decoration: none;
}

❷ a.active {
    color: white;
}
```

```
③ header ul {  
    display: inline-block;  
}
```

Listing 21-26: The style.css stylesheet

We set the default color of all `<a>` elements to be gray ❶, while any `<a>` elements with a `class` attribute of `active` will be white instead ❷. Finally, we declare the unordered list in the `header` element to display `inline-block` ❸ so our navigation items appear in the same row as the logo image in the header. [Figure 21-6](#) shows the updated home page for our site, as well as the page's HTML source code.



Figure 21-6: The home page and corresponding active CSS class for the page's link in the HTML source

In the HTML, notice that the `active` content from the `homeLink` block appears in the `class` element for the Home link in the navigation bar. As a result, the Home link appears white to indicate it's the page currently being viewed.

Polishing the Website with Bootstrap

Rather than hacking together our own CSS for a more professional-looking and responsive page layout, we can once again let the powerful Bootstrap CSS framework do most of the work for us. Twig makes incorporating

Bootstrap styling easy. All we need to do is make a few changes to the base template, and those changes will affect every page of the website. We don't need to change the child page templates at all.

We'll let Bootstrap style our navigation links, and we'll use predefined colors that Bootstrap provides, so we can delete the folder and file *css/style.css* altogether. Then we just need to modify the base template of the site. Edit *base.xhtml.twig* to match the contents of [Listing 21-27](#).

```
<!doctype HTML>
<html>
<head>
    <title>MGW: {% block pageTitle %}{% endblock %}</title>
    <meta name="viewport" content="width=device-width"> ❶

    <link rel="stylesheet"
          href=https://cdn.jsdelivrivr.net/npm/bootstrap@5.1.3/
dist/css/bootstrap.min.css> ❷
</head>

<body class="container"> ❸

<header class="navbar navbar-expand navbar-dark bg-dark"> ❹
    

<ul class="navbar-nav"> ❺
    <li class="nav-item">
        <a class="nav-link {% block homeLink %}{% endblock
%}" href="/">
            Home
        </a>
    </li>
    <li class="nav-item">
        <a class="nav-link {% block contactLink %}{% endbloc
k %}" href="/?action=contact">
            Contact Us
        </a>
    </li>
</ul>
```

```

</header>

<div class="row bg-light p-5 mb-4"> ❹
    <div class="col display-6">
        <span class="fw-bold">MGW. </span>
        <br>You know it makes sense!
    </div>

    <div class="col">
        <p>
            Welcome to My Great Website (MGW).
        </p>
    </div>
</div>

<h1>{{block('pageTitle')}}</h1>
--snip--

```

Listing 21-27: Updating the base.xhtml.twig template with Bootstrap

We add a `meta` element to prevent the page content from appearing too small when viewed on mobile devices ❶. Then we load in the Bootstrap stylesheet ❷ and entire HTML page body as a Bootstrap container ❸. This adds basic spacing to the left and right margins of the page after Bootstrap has determined the appropriate maximum width for page content for the web client viewport dimensions.

We assign the `header` element containing the logo and navigation bar several Bootstrap classes to render the element as a navigation bar in dark mode with the predefined `bg-dark` background color ❹. The unordered list containing the navigation links receives the `navbar-nav` style ❺ for professional-looking links. We style each link's list item as a `nav-item` and its anchor link element as a `nav-link`. Notice that the Twig `homeLink` and `contactLink` blocks still appear as part of the links' `class` attributes, alongside the `nav-link` class. This way, the currently displayed page's link element will have both the `nav-link` and `active` styles applied, and Bootstrap will highlight the link accordingly.

The background color, spacing, and multicolumn layout of the header are achieved using a combination of Bootstrap utility classes. We replace the header's old `<blockquote>` element with a `<div>` ❹ styled as a row with a light background (`bg-light`), lots of padding on all four sides (`p-5`), and a medium-spaced bottom margin (`mb-4`). The header `<div>` contains a `<div>` for the main heading and tagline styled as a column (`col display-6`), which will appear to the left of another `<div>` for the site's greeting.

With the addition of Bootstrap styling, our website now has the professional look and feel shown previously in [Figure 21-4](#). We achieved this styling simply by changing the base template file, without having to touch any of the child page templates.

Summary

In this chapter, you learned the basics of the Twig templating package, which greatly simplifies the process of creating general HTML templates that can be customized with page-specific content. Using Twig, we created a multipage website, driven by the `run()` method of an `Application` class, which performs the function of a front controller. The only part of our web application that isn't object-oriented is the code in the `public/index.php` script, which reads in and executes the Composer autoloader, creates an `Application` object, and calls its `run()` method.

Thanks to Twig, we can safely hand over responsibility for creating and modifying page templates to team members who need no knowledge of PHP programming. Through the use of Twig inheritance and overridable blocks, the templates for each page are small and focus on content specific to that individual page. We leveraged Twig's powerful inheritance feature, enabling us to add features such as professional Bootstrap styling and active link highlighting, all through declarations in the top-level base template. Overall, using a templating system such as Twig means we're strongly separating the view component of our web application from its controllers and model: Twig templates have the single responsibility of decorating provided data with HTML to create the body of responses to be returned to the requesting web client.

Exercises

1. Create a project with a single script, *public/index.php*, that returns a complete, well-formed HTML page body containing a paragraph saying *Hello name*, where *name* is a URL-encoded variable. Then progressively refactor the project in the following sequence:
 - a. Move the HTML into *templates/hello.php*, and write front-controller PHP code in *public/index.php* to display this template.
 - b. Move your front-controller logic into a `run()` method in a namespaced `Application` class. The `run()` method should extract the *name* URL-encoded variable and pass it to a `hello()` method that displays the *templates/hello.php* template. You'll also need to create a *composer.json* file for your class's namespace, generate the Composer autoloader, and update *public/index.php* to read in the autoloader, create an `Application` object, and call the `run()` method.
 - c. Convert your *templates/hello.php* file into a Twig template called *templates/hello.xhtml.twig*, and update your `Application` class to create a `twig` property in its constructor. Use this property in the `hello()` method to create and then print an `$html` variable for the body of the request to be returned to the web client.
2. Copy the project from Exercise 1 and progressively turn it into a two-page website by doing the following:
 - a. Separate the core of the HTML structure into a *base.xhtml.twig* template, and then refactor *hello.xhtml.twig* to extend this base template and override its body block with the “hello” message.
 - b. Create a second page template, *privacy.xhtml.twig*, that also extends the base template and displays the sentence *This website stores zero cookies and so does not affect your browsing privacy in any way*.
 - c. Add a footer to the *hello.xhtml.twig* template containing the text `Privacy Policy` and linking to URL `/?action=privacy`.
 - d. Add a `privacy()` method to the `Application` class. This method should display the *privacy.xhtml.twig* template.
 - e. Update the logic in the `run()` method of the `Application` class so that the value of the URL-encoded `action` variable (if found) is

stored in the `$action` variable. Then add a `switch` statement that invokes the `privacy()` method if the value of `$action` is `privacy`, or otherwise invokes the `hello()` method.

3. Create an object-oriented, inheritance-based, Twig-templated, three-page website, with a home page, a staff details page, and a privacy policy page. Include Bootstrap CSS and a three-item navigation bar, where the navigation bar item for the page being displayed is highlighted using the `active` CSS class. The staff details page should use a Twig `for` loop to display an HTML table of three staff members, from a provided array of `Staff` objects. The `Staff` class should have `name` and `jobTitle` properties.

22

STRUCTURING AN OBJECT-ORIENTED WEB APPLICATION



In the preceding chapter, we used object-oriented PHP code to create a simple two-page website controlled from an `Application` class. In this chapter, we'll revisit that website and explore how to further leverage OOP techniques to improve its structure. You'll learn how to use multiple classes to compartmentalize the application logic, and you'll see how inheritance can help share code among those classes to cut down on redundancy.

Dividing the application logic across multiple classes will help make the site more manageable. This may seem trivial for a two-page website, but imagine if the site grew to include tens, hundreds, or thousands of pages. The `Application` class would quickly become unwieldy. It would be imperative to organize the code into different types of actions and assign those actions to different classes.

For our application in [Chapter 21](#), two main types of actions need to be performed. The first is deciding what to do when a request comes into the web server. We can assign this task to a front-controller class that will examine each incoming request, including its URL pattern and any data variables received, and decide which type of page is appropriate to be returned to the web client.

The other main action is displaying the requested page. We can assign this task to a range of page-generating controller classes. One such class might be designed for displaying basic pages (such as home and Contact Us), another for displaying pages with security features like logging in and updating passwords, another for product listings, and so on. Each of these page-controller classes can operate knowing that the front controller has already made the decision to return the appropriate page.

Separating Display and Front-Controller Logic

Let's start the process of improving our application architecture by separating the front-controller decision logic (in the `Application` class) from the basic page-generation actions for the home page and Contact Us page. We'll move the latter into a new class called `DefaultController`. The name reflects that the home page is the default page displayed when the URL pattern `/` is requested, but the class could also reasonably be named `BasicPageController`, `HomeController`, or something similar.

Copy `src/Application.php`, name the copy `src/DefaultController.php`, and delete the `run()` method from this new `DefaultController` class. Also make the `homepage()` and `contactUs()` methods public so that they can still be called from the `Application` class. After these changes, the file should match [Listing 22-1](#).

```
<?php
namespace Mattsmithdev;

use \Twig\Loader\FilesystemLoader;
use \Twig\Environment;

class DefaultController
{
```

```

    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    private Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLATES);

        $this->twig = new Environment($loader);
    }

    public function homepage()
    {
        $template = 'homepage.xhtml.twig';
        $args = [];

        $html = $this->twig->render($template, $args);
        print $html;
    }

    public function contactUs()
    {
        $template = 'contactUs.xhtml.twig';
        $args = [];

        $html = $this->twig->render($template, $args);
        print $html;
    }
}

```

Listing 22-1: Declaring the *DefaultController* class

This new `DefaultController` class has a constant for the path to the template files, a `twig` property for rendering the templates, a constructor method, and `homepage()` and `contactUs()` methods for displaying the two pages of the web application.

Now that we've encapsulated the logic for displaying the web pages in a separate class, we can simplify the `Application` class to focus only on deciding which page to display. All we need to keep in `Application` is the

`run()` method, which will determine which page to display and invoke the corresponding `DefaultController` method. Update *src/Application.php* as shown in [Listing 22-2](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
    {
        $defaultController = new DefaultController();
        $action = filter_input(INPUT_GET, 'action');
        switch ($action) {
            case 'contact':
                ❶ $defaultController->contactUs();
                break;

            case 'home':
            default:
                ❷ $defaultController->homepage();
        }
    }
}
```

Listing 22-2: The simplified `Application` class

Our updated `Application` class's only content, the `run()` method, begins by creating a new `DefaultController` object. Then, in the `switch` statement, we invoke either that object's `contactUs()` method ❶ or `homepage()` method ❷ to display the appropriate page based on the action received in the HTTP request.

In this new arrangement, `Application` is functioning as a true front controller: it receives requests from the client and decides how to respond. Meanwhile, the code that generates and prints a response has been delegated to the `DefaultController` class. For our simple two-page site, this may seem like software architecture overkill, but for more sophisticated

websites, this separation of front-controller logic from page-generation logic means that when we add methods for many pages, we won't end up with a single, overcrowded `Application` class trying to do too many things.

For example, say we have some pages that can be accessed only by a logged-in user. We could encapsulate the methods for displaying these pages in a `SecureActions` controller class. Then we would check whether the user is logged in within the front-controller `Application` class and invoke methods of `SecureActions` only if the user is logged in. Otherwise, we could offer the user an error page or login page as appropriate.

Another example of the value of separating front-controller actions from page controllers is testing the incoming URL patterns for data parameters. Let's say some of the pages of our website display news items by using a `NewItem` page controller class. The methods of this class need the ID of the news item to be retrieved from a database or file store, based on a URL pattern such as `/?action=news&id=<id>`. In this case, our front controller can check for an integer ID along with the `news` action, then pass that ID to an appropriate `NewItem` object method. If no such integer ID is found in the URL, we can offer the user an error page instead.

In both of these examples, the methods in the page-controller classes can be written knowing that any required checks and decisions (determining whether the user is logged in or retrieving a news item ID) have already taken place and been satisfied. We're separating the decision of *what to do* (the front controller) from the actions that define *how we do it* (the page-controller class methods).

Using Multiple Controller Classes

Our `DefaultController` class works well for displaying simple pages with static content such as the home page, but pages with other features would benefit from being organized within their own specialized controller classes. For example, an e-commerce site would likely have several kinds of pages related to products: a list of all available products, search results showing products that match a particular user query, pages showing the details of a single product, and so on. Each of these pages would likely need a way of interacting with objects of a `Product` class, perhaps passed to the page templates in a `$products` array or a single `$product` object.

Our `DefaultController` class isn't currently equipped to handle these kinds of product-related operations. We could extend and modify the class, but a more logical approach would be to create a separate `ProductController` class to handle the specialized operations required to display pages related to products. Similarly, pages that include login forms might have their own `LoginController` class, pages for displaying and editing a shopping cart would have their own `CartController` class, and so on.

To illustrate the benefit of multiple controller classes, and to demonstrate how easy it is to add more pages and sections to an object-oriented web application, we'll add a Product List page to our website, as shown in [Figure 22-1](#), and we'll create a `ProductController` class for displaying this page.

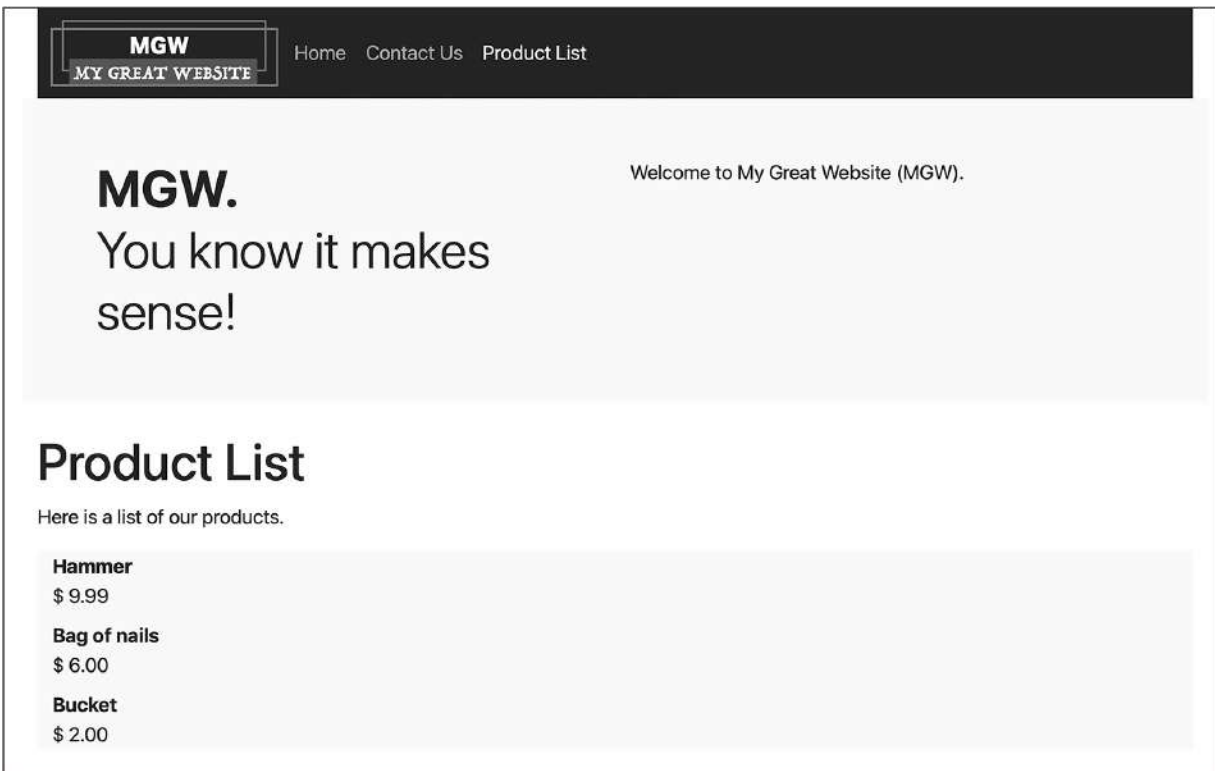


Figure 22-1: The Product List page we'll create

Our new page will display the name and price of a collection of products, where each product is an instance (object) of a `Product` class.

Using Twig template inheritance, we'll give the page the same navigation bar and header content as the other pages of the website. We'll coordinate the page's display from our new `ProductController` class, which will be designed specifically to gather `Product` objects in an array that can be passed to the Twig template.

To build the new page, we'll first create the `Product` class to represent each product's name and price. Create a new file, *src/Product.php*, containing the code in [Listing 22-3](#).

```
<?php
namespace Mattsmithdev;

class Product
{
    public string $name;
    public float $price;

    ❶ public function __construct(string $name, float $price)
    {
        $this->name = $name;
        $this->price = $price;
    }
}
```

Listing 22-3: The `Product` class

We declare two public properties for each `Product` object: `name` and `price`. Then we declare a constructor method ❶ that will take in initial values for each of these properties when creating a new `Product` object.

Now that we have a `Product` class, we can create the `ProductController` class for displaying the page. Create a new *src/ProductController.php* file as shown in [Listing 22-4](#).

```
<?php
namespace Mattsmithdev;
use \Twig\Loader\FilesystemLoader;
use \Twig\Environment;
```

```

class ProductController
{
    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    private Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLAT
ES);
        $this->twig = new Environment($loader);
    }

    public function productList()
    {
        $product1 = new Product('Hammer', 9.99);
        $product2 = new Product('Bag of nails', 6.00);
        $product3 = new Product('Bucket', 2.00);
        ❶ $products = [$product1, $product2, $product3];

        $template = 'productList.xhtml.twig';
        $args = [
            ❷ 'products' => $products
        ];

        ❸ $html = $this->twig->render($template, $args);
        print $html;
    }
}

```

Listing 22-4: The src/ProductController.php file declaring the `ProductController` class

The `ProductController` class's constructor method is similar to that of the `DefaultController` class: it performs the setup necessary for working with the Twig templates. What distinguishes this controller from the other is its `productList()` method for displaying the new Product List page.

Within that method, we create three `Product` objects and package them into a `$products` array ❶. Then we set the `$template` variable to

'productList.xhtml.twig', the new Twig template file we'll create to list all the products. We next construct the `$args` array. It maps the 'products' key (which will become a Twig variable name) to `$products`, our array of `Product` objects ❷. Then we pass the `$template` and `$args` variables to Twig to generate the HTML needed for the page ❸.

We next need to update the front-controller logic in our `Application` class to call the `ProductController` class's `productList()` method when the value of `action` in the URL is `products`. Update `src/Application.php` to match [Listing 22-5](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
    {
        $defaultController = new DefaultController();
        ❶ $productController = new ProductController();
        $action = filter_input(INPUT_GET, 'action');
        switch ($action) {
            ❷ case 'products':
                $productController->productList();
                break;

            case 'contact':
                $defaultController->contactUs();
                break;

            case 'home':
            default:
                $defaultController->homepage();
        }
    }
}
```

Listing 22-5: Updating the `Application` class to handle the `products` case

In the `run()` method, we create `$productController`, a variable referencing a new `ProductController` object ❶. Then we add a new case to the `switch` statement ❷. When the action in the URL has the value `products`, we'll send a message to our `ProductController` object to invoke its `productList()` method.

We can now write the Twig template to loop through and display the provided array of products. Create the new Twig template file `templates/productList.xhtml.twig` as shown in [Listing 22-6](#).

```
{% extends 'base.xhtml.twig' %}

{% block pageTitle %}Product List{% endblock %}

{% block productsLink %}active{% endblock %}

{% block main %}
<p>
    Here is a list of our products.
</p>

<dl class="container bg-light">
❶ {% for product in products %}
    <dt>{{product.name}}</dt>
    <dd> $ {{product.price | number_format(2)}}</dd>
{% else %}
    <dt>(sorry, there are no products to list)</dt>
{% endfor %}
</dl>

{% endblock %}
```

Listing 22-6: The `productList.xhtml.twig` template

Like our other page templates, this one inherits from `base.xhtml.twig`, giving it access to all the content shared across pages. We're therefore able to focus on just filling in the blocks from that base template that need to be customized. First, we override the `pageTitle` Twig block with the text `Product List`. Then we override the `productsLink` Twig block with the

`text active` to highlight this page's link in the navigation bar (we'll add a new navigation bar link to the base template next).

Next, we override the `main` Twig block with the page-specific body content. The centerpiece of this content is a loop through all the `Product` objects in the `products` Twig array variable to generate the items of an HTML definition list ❶. The `name` of each product is declared as the definition term (`<dt>`), and the definition data element (`<dd>`) is the `price` of the product, formatted to two decimal places by using the Twig `number_format` filter. If the `products` array is empty, a Twig `else` statement will display an appropriate message.

Our final action to get our Product List page working is to add a new item to its navigation bar in the base template. Update *templates/base.xhtml.twig* to match [Listing 22-7](#).

```
--snip--
<body class="container">

<header class="navbar navbar-expand navbar-dark bg-dark">
    

    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link {% block homeLink %}{% endblock %}" href="/">
                Home
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link {% block contactLink %}{% endblock %}" href="/?action=contact">
                Contact Us
            </a>
        </li>
        <li class="nav-item">
            <a class="
nav-link {% block productsLink %}{% endblock %}" href="/?action=products"
```



```

        >
        Product List
    </a>
</li>
</ul>
</header>
--snip--

```

Listing 22-7: Adding the product list link to the base.xhtml.twig template

We add a third item to the navigation bar for the Product List page. As with the other links, we include a `class` attribute containing a Twig block named `productsLink` so the link can be styled `active` as needed.

We’ve now added a Product List page to our website. In the new `ProductController` class, our `productList()` method creates an array of objects and uses the Twig template `templates/productList.xhtml.twig` to create the HTML for the page. Adding a new navigation link to our base Twig template was easy. Clicking that link creates a `GET` request with `action=products`. In our `Application` class front controller, an instance of the `ProductController` is created so that when this value of `action` is found in the request URL, the `productList()` method can be invoked. In all, the majority of the new code for this product list feature is well organized in its own controller class and corresponding Twig template.

Sharing Controller Features Through Inheritance

As a last step, let’s use the OOP principle of inheritance to streamline our controller classes. Right now, `DefaultController` and `ProductController` share several lines of identical code: both declare a `PATH_TO_TEMPLATES` constant, have a private `twig` property, and have identical constructor methods to create a `Twig\Environment` object. If we were to create more controller classes (for login security, shopping carts, and so on), they would also need this identical code.

To avoid all this repetition, we’ll take the common properties and behaviors all controller classes should have and make them part of a general `Controller` superclass. The individual controller classes, such as `DefaultController` and `ProductController`, will inherit from this

superclass and extend it with their own unique properties and methods. [Figure 22-2](#) shows a diagram of the class structure we'll create.

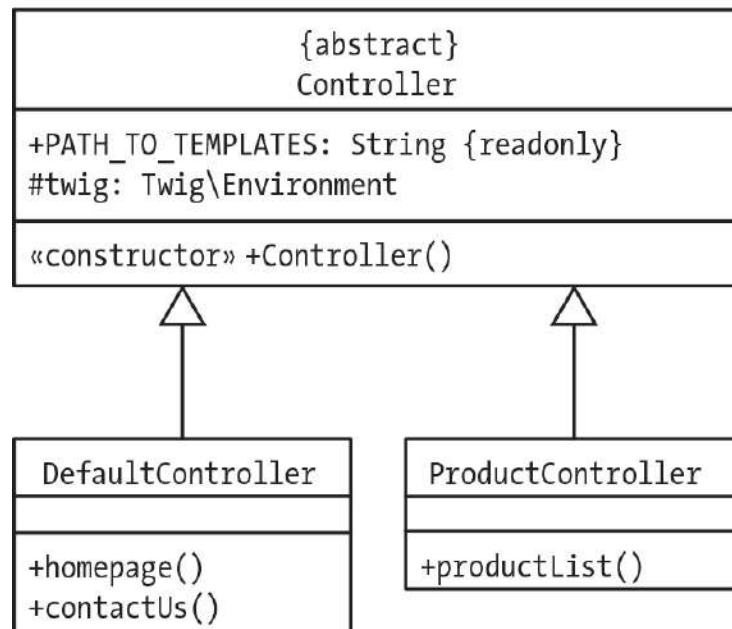


*Figure 22-2: The **Controller** superclass and its **DefaultController** and **ProductController** subclasses*

We'll declare our new **Controller** class as **abstract**, meaning we can never actually create a **Controller** object. This is fitting since the **Controller** class exists only to store the general code that all controllers should have and to be subclassed by the specific controller classes we'll want to instantiate. Remember, declaring a class **abstract** is a way to tell other programmers (and yourself in the future) that you don't want the class to be instantiated.

In [Figure 22-2](#), notice the plus signs (+) denoting public methods and constants, as well as the hash mark (#) next to the **twig** property in the **Controller** superclass, which indicates this property has **protected** visibility, not **public** or **private**. We wouldn't want this **twig** property to be **public**, since it could be incorrectly changed or used by any code in the web application with access to a **Controller** object or one of its subclasses. However, if we were to make the **twig** property **private**, the code in the methods of our subclasses wouldn't have access to it either. This would be a

problem, since using Twig to render templates is a core behavior of all our controller classes.

Giving the `twig` property protected visibility ensures that subclasses of `Controller` can access it, while preventing direct access by any code outside the `Controller` class hierarchy. This is a useful real-world example of the inheritance concepts we examined in [Chapter 19](#).

Listing 22-8 shows the code for the `Controller` superclass. Create *src/Controller.php* containing the code from this listing.

```
<?php
namespace Mattsmithdev;

use \Twig\Loader\FilesystemLoader;
use \Twig\Environment;

❶ abstract class Controller
{
    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    ❷ protected Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLATES);
        $this->twig = new Environment($loader);
    }
}
```

Listing 22-8: The `Controller` superclass

We declare the class to be `abstract` so it can't be instantiated ❶, and we designate the `twig` property as `protected` so it will be available to the subclasses ❷. Otherwise, this code is identical to the code at the start of our `DefaultController` and `ProductController` classes. Now that this code lives in the `Controller` class, the redundant parts can be removed. [Listing 22-9](#) shows the much-simplified `DefaultController` class code.

```
<?php
namespace Mattsmithdev;

class DefaultController extends Controller
{
    private function homepage()
    {
        $template = 'homepage.xhtml.twig';
        $args = [];

        $html = $this->twig->render($template, $args);
        print $html;
    }

    private function contactUs()
    {
        $template = 'contactUs.xhtml.twig';
        $args = [];

        $html = $this->twig->render($template, $args);
        print $html;
    }
}
```

Listing 22-9: *The simplified `DefaultController` class, a subclass of `Controller`*

We declare that `DefaultController` extends the `Controller` class, allowing it to inherit the constructor and `twig` property. Thanks to this inheritance, `DefaultController` now has only two methods of its own, for displaying the home page and Contact Us templates. We can similarly streamline the `ProductController` class code, as shown in [Listing 22-10](#).

```
<?php
namespace Mattsmihdev;

class ProductController extends Controller
{
    public function productList()
    {
```

```

        $product1 = new Product('Hammer', 9.99);
        $product2 = new Product('Bag of nails', 6.00);
        $product3 = new Product('Bucket', 2.00);
        $products = [$product1, $product2, $product3];

        $template = 'productList.xhtml.twig';
        $args = [
            'products' => $products
        ];

        $html = $this->twig->render($template, $args);
        print $html;
    }
}

```

Listing 22-10: The simplified `ProductController` class, a subclass of `Controller`

Again, we use `extends Controller` when declaring the class so that `ProductController` will inherit from `Controller`. The only method specific to the subclass is `productList()` for displaying the Product List page.

We’ve now successfully used inheritance to abstract the common `twig` property and its initialization into a `Controller` superclass. This has simplified the two page-controller classes, while still providing exactly the same functionality.

Summary

In this chapter, we improved the architecture of our object-oriented web application. We separated the overall front-controller logic governing the site, located in the `Application` class, from the page-controller logic for displaying the individual web pages. The latter is divided between an abstract `Controller` superclass, which contains the Twig setup code required for displaying any web page, and multiple subclasses containing just the code for logic specific to displaying particular kinds of pages.

The example site in this chapter has only three pages: a home page, a Contact Us page, and a Product List page. However, the architecture demonstrated in this chapter can easily be scaled up for complex websites

with hundreds or thousands of pages and complex features like session interactions, shopping carts, login security, and more.

Exercises

1. Make a copy of the project from this chapter and add a fourth page for a privacy policy. Follow these steps:
 - a. Create a *privacy.xhtml.twig* template.
 - b. Add a new `privacyPolicy()` method to the `DefaultController` class that displays the new template.
 - c. Add a `Privacy Policy` navigation bar link in the *base.xhtml.twig* template with the URL `?action=privacy`.
 - d. Add a new case to the `switch` statement in the `run()` method of the `Application` class that invokes the `privacyPolicy()` method of the `DefaultController` object if the value of `action` in the URL is `privacy`.
2. Make a copy of your project from Exercise 1, and add a fifth page for listing company staff. Follow these steps:
 - a. Create a `Staff` class to represent staff details, including `firstName`, `lastName`, and `email` properties.
 - b. Create a new subclass of `Controller` named `StaffController`. Give it a `list()` method that creates two or three staff objects and passes them as an array to the Twig `render()` method, along with the template name *staff.xhtml.twig*.
 - c. Add a new `Staff List` navigation bar link in the *base.xhtml.twig* template with the URL `?action=staffList`.
 - d. Create a *staff.xhtml.twig* template, based on the *productList.xhtml.twig* template, that uses Twig code to loop through and print out each `Staff` object in the received array.
 - e. In the `run()` method of the `Application` class, create a new `$staffController` object that's an instance of the `StaffController` class. Then add a new `switch` statement case that calls `$staffController->list()` if the value of `action` in the URL is `staffList`.

ERROR HANDLING WITH EXCEPTIONS



Applications don't always function the way you want. For example, a file may not upload because of a network error, or data from a user or web API may be malformed in some way. In this chapter, you'll learn to use *exceptions* to anticipate these sorts of problems and recover from them, so your application doesn't always crash when something goes wrong. You'll work with PHP's generic `Exception` class, along with other, more specialized exception classes built into the language. You'll also see how to design your own custom exception classes, as well as how to design your applications to safely handle any and all exceptions that may arise.

The Basics of Exceptions

Exceptions are classes that provide a sophisticated and customizable approach to handling and recovering from anticipated, problematic circumstances in OOP. They differ from *errors*, which arise from circumstances or events that can't be recovered from, such as the computer system running out of memory or a class declaration attempting to use a constant that can't be found. PHP has a built-in `Exception` class for handling generic problems, along with other more specialized exception classes that cater to particular types of errors. You can also develop your own custom exception classes.

Exception-based software design allows you to write code in the most natural sequence, assuming it will all work fine, and then to separately write code to capture and address any typical problems that may occur. This involves writing tests into the methods of a class that generate exception objects and disrupt the flow of program control whenever an unusual or invalid situation occurs, such as providing invalid arguments for a constructor or setter method. Thanks to these tests, code appearing later in a method can be written with the safe assumption that if execution gets that far, the exception-throwing conditions haven't occurred, and the code is working as it should.

Central to exception-based application programming are `throw` and `catch` statements. A method uses a `throw` statement to create an exception object when a problem occurs. This is called *throwing an exception*. The `throw` statement halts the execution of the method and disrupts the flow of the program. By itself, throwing an exception can lead to a fatal error, unless you *catch* the exception with a `catch` statement. The `catch` statement features code that's intended to be executed when an exception is thrown; such code may allow the application to recover from the issue, or if non-recoverable, then the problem can be logged and execution ended gracefully.

In this section, we'll explore the basics of throwing and catching exceptions. We'll also look at `finally` statements, pieces of code that are executed at the end of a process, regardless of whether an exception has been thrown.

Throwing an Exception

First, we'll consider how to throw an uncaught exception to cause a fatal error, halting an application. We'll examine the common use case of throwing an exception when an invalid argument is provided to the setter method of a class. We'll create a variation of the `Food` and `Dessert` classes from [Chapter 19](#), adding exception-based validation behavior in the `setCalories()` method of the `Dessert` class. An exception will be thrown if a negative value is provided as the number of calories for a new `Dessert` object. The project we'll create is illustrated in the UML class diagram in [Figure 23-1](#).

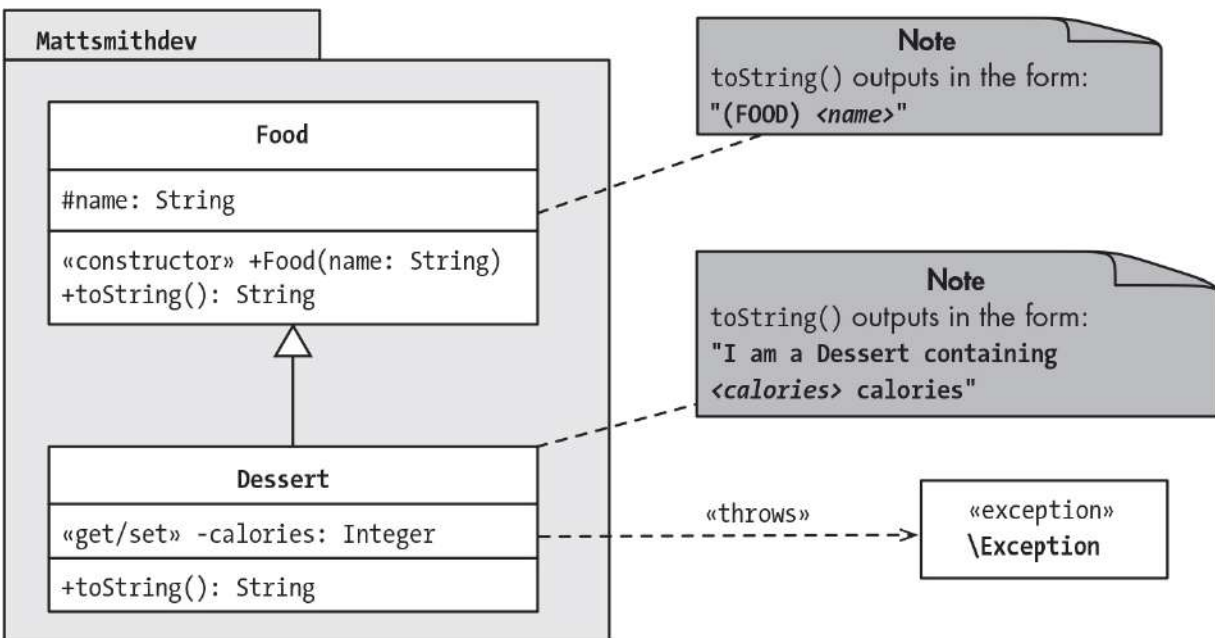


Figure 23-1: A diagram showing an exception thrown by the `Dessert` class

Recall that `Dessert` is a subclass of `Food`, with its own `__toString()` method and a `calories` property. The diagram indicates that an invalid `calories` value will throw an exception.

First, we'll declare the `Food` class. Create a new project with `src/Food.php` containing the code in [Listing 23-1](#).

```
<?php
namespace Mattsmithdev;
```

```

class Food
{
    protected string $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function __toString(): string
    {
        return "(FOOD) $this->name";
    }
}

```

Listing 23-1: The *Food* superclass

We assign the `Food` class to the `Mattsmithdev` namespace and give it a name property with protected visibility so that all subclasses can directly access it. The class has a straightforward constructor to initialize `name` when each new object is created, and a `__toString()` method to return a string in the form `"(FOOD) name"`.

Let's now declare the `Dessert` subclass of `Food`. Create *src/Dessert.php* and enter the contents of [Listing 23-2](#).

```

<?php
namespace Mattsmithdev;

class Dessert extends Food
{
    private int $calories;

    public function __construct(string $name, int $calories)
    {
        parent::__construct($name);
        ❶ $this->setCalories($calories);
    }
    public function getCalories(): int

```

```

    {
        return $this->calories;
    }

    public function setCalories(int $calories)
    {
        ❷ if ($calories < 0) {
            throw new \Exception(
                'attempting to set calories to a negative va
lue');
        }

        ❸ $this->calories = $calories;
    }

    public function __toString(): string
    {
        return "I am a Dessert containing $this->calories!";
    }
}

```

Listing 23-2: The `Dessert` class, which throws an exception if the `calories` value is invalid

The `Dessert` class has a `calories` property assigned in the constructor via the `setCalories()` method ❶. This way, we reserve any validation logic for the setter method itself, so every new `calories` value will be vetted, regardless of whether it's provided at the time of object construction or via the setter at a later point.

Within `setCalories()`, we perform the validation with an `if` statement ❷. If the provided integer argument `$calories` is less than 0, we throw a new `Exception` object, with the message 'attempting to set calories to a negative value'. If the `$calories` argument is 0 or more and no exception is thrown, the code continues by storing the provided value in the `Dessert` object's `calories` property ❸.

Notice the syntax for throwing the exception. We begin with the `throw` keyword, which tells PHP to disrupt the flow of the program if the `if` statement is `true`. Then we use the `new` keyword to create a new object of

the `Exception` class, passing the error message we want to display as an argument. We have to prefix the `Exception` class with a backslash (`\`) because it's part of PHP's root namespace, whereas `Dessert` is part of the `Mattsmithdev` namespace. Without the backslash, `Exception` would be assumed to be in the `Mattsmithdev` namespace as well.

We next need to write a `composer.json` file to autoload our classes. Create this file as shown in [Listing 23-3](#).

```
{
    "autoload": {
        "psr-4": {
            "Mattsmithdev\\": "src"
        }
    }
}
```

Listing 23-3: The `composer.json` file for autoloading

Once you have this file, generate the autoloader scripts by entering **`composer dump-autoload`** at the command line.

Now let's write an index script to attempt to create a `Food` and a `Dessert` object. Create `public/index.php` to match [Listing 23-4](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Food;
use Mattsmithdev\Dessert;

$f1 = new Food('apple');
print $f1 . PHP_EOL;

$f2 = new Dessert('strawberry cheesecake', -1);
print $f2;
```

Listing 23-4: Attempting to create an invalid `Dessert` object in `index.php`

We read and execute the autoloader and provide `use` statements for the two classes we need. Then we create and print a `Food` object and a `Dessert` object, passing an invalid argument of `-1` for the latter's `calories` property. Here's the result of running this index script at the command line:

```
$ php public/index.php
(FOOD) apple

Fatal error: Uncaught Exception: attempting to set calories
to a negative
value in /Users/matt/src/Dessert.php:23
Stack trace:
#0 /Users/matt/src/Dessert.php(12): Mattsmithdev\Dessert->se
tCalories(-1)
#1 /Users/matt/public/index.php(10): Mattsmithdev\Dessert->__
_construct
('strawberry chee...', -1)
#2 {main}
thrown in /Users/matt/src/Dessert.php on line 23
```

The first line of output shows that the `Food` object was successfully created and printed out, but then we get a fatal error due to the exception thrown by the negative calorie value. The exception is said to be *uncaught*, since we didn't write any code telling PHP what to do if an exception is thrown. As a result, the application has simply stopped running and has printed the error message we provided, followed by a *stack trace*, a report that steps through the code to show the cause of the exception.

The stack trace tells us the following:

- #0 shows that the exception was thrown when `setCalories()` was passed `-1` as an argument at line 12 of the *src/Dessert.php* file.
- #1 shows that `setCalories()` was called when the `Dessert` class's constructor method was invoked with the arguments `('strawberry chee...', -1)` (the food-name string has been shortened).
- #2 reports that the exception-throwing code is line 23 in *src/Dessert.php*.

Notice that the output ends with the stack trace, meaning the index script wasn't able to get to the point of printing out the `Dessert` object. The uncaught exception halted the flow of the program, so the final line of the index script didn't execute.

Catching an Exception

To avoid a fatal error and safely manage exceptions, we need to *catch* the exceptions by writing a `try...catch` statement in our index script. The `try` portion indicates what we want to do under normal circumstances, and the `catch` portion indicates what to do when an exception is thrown.

By catching exceptions, we prevent application users from seeing fatal errors and the resulting stack traces. Besides being embarrassing and not user-friendly, printing out a stack trace “leaks” information about the structure of the web application code (in the preceding example, for instance, we leaked the folder name *src* and the class filename *Dessert.php*). While stack traces aren't serious security vulnerability issues, any information leaked like this might be helpful to an attacker and so should be prevented where possible. Catching exceptions lets us decide what to do with the exception data, as well as what the user will see when a problem arises.

NOTE

In [Chapter 24](#), we'll explore logging, which allows useful debugging data such as stack traces to be stored for developers and site administrators to access, while not publishing such information to any public website visitor or software client.

To catch the exception raised when a negative calorie value is given, update the *public/index.php* script to match [Listing 23-5](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev \Food;
use Mattsmithdev \Dessert;
```

```

❶ try {
    $f1 = new Food('apple');
    print $f1 . PHP_EOL;

    $f2 = new Dessert('strawberry cheesecake', -1);
    print $f2;
❷} catch (\Exception $e) {
    print '(caught!) - an exception occurred!' . PHP_EOL;
    ❸ print $e->getMessage();
}

```

Listing 23-5: Adding a `try...catch` statement to `index.php`

The old code creating and printing `Food` and `Dessert` objects is now inside a `try` block ❶. If any exception occurs during this sequence, PHP checks the class of the exception against the class(es) specified in the `catch` block that follows ❷. If the class matches, the `catch` block is executed. In this case, the `catch` statement is for objects of the `\Exception` class, as specified in the parentheses after the `catch` keyword. The variable `$e`, also in the parentheses, becomes a reference to the `Exception` object that has been caught.

In the `catch` block, we print out the message `'(caught!) - an exception occurred!'` followed by a line break. Then we print the message inside the `Exception` object via its public `getMessage()` method ❸. This is the `'attempting to set calories to a negative value'` message we defined earlier.

Now that we've added code catching the exception, try running the `index` script again at the command line. You should see the following:

```

$ php public/index.php
(FOOD) apple
(caught!) - an exception occurred!
attempting to set calories to a negative value

```

Again, the `Food` object has been successfully created and printed. Next, we see the message printed from inside our `catch` statement, followed by

the message from the `Exception` object itself. In this example, having caught the exception, we're still printing out a message for the user, but we've controlled the information that's displayed. No stack trace is leaking information now that we're handling the exception with a `catch` statement.

Ending with a finally Statement

A `finally` statement is a block of code that gets executed regardless of whether an exception has been thrown. It's written after the `try` and `catch` statements and typically includes *housekeeping code*, code that gracefully ends any processes that have been started. For example, you might use a `finally` statement to ensure that any file streams or database connections are closed.

Let's add a `finally` statement to our index script to gracefully close the application every time it runs, even if an exception has been thrown. Modify *public/index.php* to match [Listing 23-6](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

try {
    $f1 = new Food('apple');
    print $f1 . PHP_EOL;

    $f2 = new Dessert('strawberry cheesecake', -1);
    print $f2;
} catch (\Exception $e) {
    print '(caught!) - an exception occurred!' . PHP_EOL;
    print $e->getMessage();
} finally {
    print PHP_EOL . '(finally) -- Application finished --';
}
```

Listing 23-6: Adding a `finally` statement to index.php

We declare a `finally` block that prints a simple message after either the `try` or `catch` block concludes. Here's the result of running this updated index script:

```
$ php public/index.php
(FOOD) apple
(caught!) - an exception occurred!
attempting to set calories to a negative value
(finally) -- Application finished --
```

The message from the `finally` block prints at the end of the output, after displaying the message from the exception.

To make sure the `finally` statement executes even when the `catch` statement doesn't, let's update our script so that the `Dessert` object is given a valid number of calories, meaning no exception will be thrown. Modify the instantiation of the `Dessert` object in *public/index.php* as shown in [Listing 23-7](#).

```
--snip--
    $f2 = new Dessert('strawberry cheesecake', 500);
--snip--
```

Listing 23-7: Creating a valid `Dessert` object in `index.php`

When you run the index script now, you should see the `Food` and `Dessert` object messages and then the `finally` message:

```
$ php public/index.php
(FOOD) apple
I am a Dessert containing 500 calories!
(finally) -- Application finished --
```

The output confirms that a `Dessert` object was successfully created without throwing an exception, and that the `finally` block was still executed regardless.

Using Multiple Exception Classes

In addition to PHP's root `Exception` class, several other classes of exception are available as part of the Standard PHP Library (SPL), such as the `InvalidArgumentException` class. These other exception classes are all

connected hierarchically to `Exception` as its subclasses, subclasses of its subclasses, and so on. You can also create your own custom exception classes that are subclasses of one of these built-in exception classes.

At first glance, it may seem unnecessary to have subclasses of `Exception`, since we could create basic `Exception` objects with custom messages for each situation throwing an exception. However, by writing code that throws objects of different `Exception` subclasses, you can include several `catch` statements, one for each `Exception` subclass, allowing you to respond differently to each type of exception.

For example, you could write multiple validation checks into a setter method and throw a certain class of exception depending on which validation check fails. Then you could write a separate `catch` statement for each of the exception classes, so each type of exception generates a customized response. You would then typically end with a `catch` statement for generic `Exception` objects, allowing you to catch any exceptions you didn't already account for. We'll look at how this works in the following sections.

Other Built-in Exception Classes

Let's use another built-in PHP exception class in conjunction with the root `Exception` class. We'll update our `Dessert` class's `setCalories()` method to throw one of two exception class objects as part of the validation of the received `$calories` argument. Our validation tests are as follows:

- If `$calories` is less than 0, throw an `\InvalidArgumentException` object because desserts can't have negative calories.
- If `$calories` is greater than 5000, throw a general `\Exception` object because that's *way too many* calories for one dessert.

Update the `setCalories()` method in `src/Dessert.php` to match [Listing 23-8](#).

```
public function setCalories(int $calories): void
{
    if ($calories < 0) {
        throw new \InvalidArgumentException(
            'attempting to set calories to a negative valu
```

```

        e');

    }

    if ($calories > 5000) {
        throw new \Exception('too many calories for one dessert!');
    }

    $this->calories = $calories;
}

```

Listing 23-8: Updating the `setCalories()` method to throw different classes of exception

First, we change the class of exception thrown when the argument received is negative to `\InvalidArgumentException`. Once again, note the forward slash before the class name, indicating that this class is declared in the root PHP namespace. Then we add a second validation test: when the number of calories is greater than 5000, an object of the general `\Exception` class will be thrown. If execution of the code gets past these two `if` statements without throwing any exceptions, we store the provided value in the object's `calories` property as before.

Next, we need to update the index script. We'll write multiple `catch` statements to handle each class of exception object appropriately. Then we'll try different values for the `Dessert` object's `calories` property to test the validation logic. Edit `public/index.php` to match [Listing 23-9](#).

```

<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Food;
use Mattsmithdev\Dessert;

$calories = -1; // Negative invalid argument
$calories = 6000; // General exception
$calories = 500; // Valid

```

```

try {
    $f2 = new Dessert('strawberry cheesecake', $calories);
    print $f2;
} ❶ catch (\InvalidArgumentException $e) {
    print '(caught!) - an Invalid Argument Exception occurred!' . PHP_EOL;
    print $e->getMessage();
} ❷ catch (\Exception $e) {
    print '(caught!) - a general Exception occurred!' . PHP_EOL;
    print $e->getMessage();

} finally {
    print PHP_EOL . '(finally) -- Application finished --';
}

```

Listing 23-9: Multiple *catch* statements in the `public/index.php` script

We start with three assignment statements for different values of the `$calories` variable. To thoroughly test the script, comment out all but one of these assignment statements, choosing a different one each time. In the `try` block, we create a new `Dessert` object, providing the `$calories` variable as an argument. Then we create two `catch` statements, one for the `InvalidArgumentException` class ❶ and the other for the general `Exception` class ❷. Each prints a different message, along with the message attached to the exception object itself, retrieved with `$e->getMessage()`.

[Table 23-1](#) shows the outputs for the three values of `$calories`, demonstrating that our exception-based logic is working as expected.

Table 23-1: Outputs for Calorie Values

Value of <code>\$calories</code>	Program output
-1	(caught!) - an Invalid Argument Exception occurred! attempting to set calories to a negative value (finally) -- Application finished --
6000	(caught!) - a general Exception occurred! too many calories for one dessert! (finally) -- Application finished --

Value of <code>\$calories</code>	Program output
500	I am a Dessert containing 500 calories! (finally) -- Application finished --

Notice that the values of `-1` and `6000` each trigger their own class of exception, while `500` allows the `Dessert` object to be successfully created and printed.

Custom Exception Classes

PHP gives you the flexibility to write your own custom exception classes, provided they're subclasses of `Exception` or one of the other built-in PHP exception classes. Also, many third-party libraries come with their own custom exception classes designed specifically for the methods in that library. Whether you're writing your own or using someone else's, custom exception classes give you even more freedom to structure your code to respond differently to a variety of anticipated problems.

Let's add a custom exception class to our `Dessert` project:

`Mattsmithdev\NegativeCaloriesException`. We'll update the project to throw an exception object of this class instead of the

`InvalidArgumentException` class. [Figure 23-2](#) shows the two classes of exception that our `Dessert` objects can throw. Notice that the

`NegativeCaloriesException` class falls within the `Mattsmithdev` namespace, while the `Exception` class is outside, since it's in the root PHP namespace.

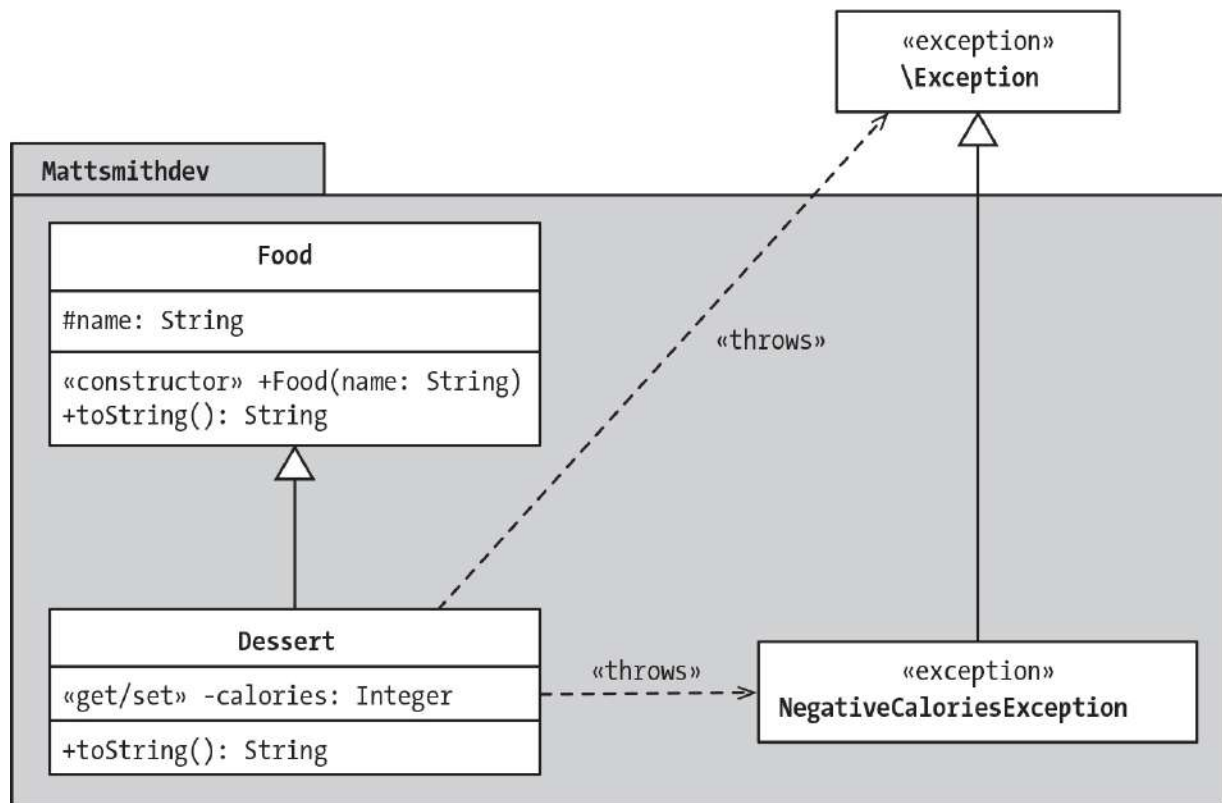


Figure 23-2: The two classes of exception that a *Dessert* object can throw

First, create a new class in *src/NegativeCaloriesException.php* containing the code in [Listing 23-10](#).

```

<?php
namespace Mattsmithdev;

class NegativeCaloriesException extends \Exception
{
}
  
```

Listing 23-10: The custom *NegativeCaloriesException* class

We declare `NegativeCaloriesException` as a subclass of the root `\Exception` class. It contains no methods. Since it doesn't have its own constructor method, it will inherit the constructor from its `Exception` superclass, allowing it to take in a message for display.

Let's now update our `Dessert` class's `setCalories()` method to throw a `NegativeCaloriesException` object when the provided calorie value is negative. As in the previous example, we'll throw a general `Exception` object when the provided value is greater than 5000. Update the `setCalories()` method in `src/Dessert.php` to match [Listing 23-11](#).

```
public function setCalories(int $calories)
{
    if ($calories < 0) {
        throw new NegativeCaloriesException(
            'attempting to set calories to a negative value');
    }

    if ($calories > 5000) {
        throw new \Exception('too many calories for one dessert!');
    }

    $this->calories = $calories;
}
```

Listing 23-11: Throwing a custom exception in the `setCalories()` method

We change the class of exception thrown when the argument received is negative to an object of the `NegativeCaloriesException` class. Since this new class is in the same namespace as our `Dessert` class, we don't write a backslash before the class identifier.

Next, we need to update the `catch` statements in our index script to handle the new custom exception class. Modify `public/index.php` as shown in [Listing 23-12](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Dessert;
use Mattsmithdev\NegativeCaloriesException;
```

```
$calories = -1; // Negative invalid argument
$calories = 6000; // General exception
$calories = 500; // Valid

try {
    $f2 = new Dessert('strawberry cheesecake', $calories);
    print $f2;
} catch (NegativeCaloriesException) {
    print '(caught!) - a Negative Calories Value Exception occurred!' . PHP_EOL;
    print $e->getMessage();
} catch (\Exception $e) {
    print '(caught!) - a general Exception occurred! ' . PHP_EOL;
    print $e->getMessage();
} finally {
    print PHP_EOL . '(finally) -- Application finished --';
}
```

Listing 23-12: Catching custom exception objects in the index.php script

We add a `use` statement so we can reference the `NegativeCaloriesException` class without the `Mattsmithdev` namespace prefix. Then we create a `catch` statement for exceptions of this class, printing an appropriate message. Here's the output you should get if you try to create a new `Dessert` object with `-1` calories, confirming that a `NegativeCaloriesException` is thrown:

```
$ php public/index.php
(caught!) - a Negative Calories Value Exception occurred!
attempting to set calories to a negative value
(finally) -- Application finished --
```

Testing for a negative value is a simple example, but it serves to illustrate how straightforward it is to create custom subclasses of `Exception`, allowing you to write different logic to address different anticipated problems at runtime.

Call-Stack Bubbling

If an exception occurs in a block of code and isn't caught by that code block, it will *bubble up* the call stack to the code that invoked the code block. If not caught there, the exception will continue to bubble up through successively higher levels of code until either it's caught and handled or the top of the call stack is reached. If the exception isn't caught in the top-level block of code, a fatal error will result, as we saw in this chapter's first example. For this reason, it's a good idea to include some code at the top level of an application to catch any exceptions that may have bubbled all the way to the top of the call stack.

As you've seen, the flow of control for a typical object-oriented PHP web application begins with the index script, which creates an object of the `Application` class and invokes its `run()` method. This in turn triggers the creation of other objects and the invocation of other methods. Some of this activity might throw exceptions. You can try to catch all those exceptions within the `Application` class, but any uncaught exceptions will ultimately bubble up to the index script and should be caught there to avoid a fatal error.

To demonstrate how this works, we'll update our `Dessert` project, taking the code that used to be in the index script and encapsulating it in an `Application` class. This class will be responsible for catching any `NegativeCaloriesException` objects thrown during the creation of `Dessert` objects, but we'll allow other miscellaneous exceptions to bubble up to the top of the call stack. Then we'll catch those in the top-level index script.

First, let's update our index script to create an `Application` object and invoke its `run()` method. We'll wrap that code in a `try...catch` statement to handle any uncaught exceptions that bubble up, and we'll include a `finally` statement to gracefully close the application. We'll also clearly indicate that the messages being printed by the `catch` and `finally` blocks are coming from this index script by prefixing them with `(index.php)`. Modify `public/index.php` to match [Listing 23-13](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';
```

```
use Mattsmithdev\Application;

try {
    $app = new Application();
    $app->run();
} catch (\Exception $e) {
    print '(index.php) Exception caught!';
} finally {
    print PHP_EOL;
    print '(index.php) finally -- Application finished --';
}
```

Listing 23-13: The simplified index.php script creating an `Application` object

Inside the `try` block, we create an `Application` object, storing a reference to the new object in the `$app` variable, and invoke its `run()` method. If any uncaught `Exception` objects bubble up from the `try` block statements, we use a `catch` block to handle them and print a message. Since every exception object is an instance of the `Exception` class (either directly or as a subclass), this `catch` statement acts as a catchall for any possible exception object that was thrown during program execution but wasn't caught elsewhere in the code. We also add a `finally` block that will print a final message regardless of whether any exceptions were thrown.

Now let's write the `Application` class. Create a new file named `src/Application.php` to match [Listing 23-14](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
    {
        $calories = -1; // Negative invalid argument
        $calories = 6000; // General exception
        $calories = 500; // Valid
    }
}
```

```

        try {
            $f2 = new Dessert('strawberry cheesecake', $calo
ries);

            print $f2;
        } catch (NegativeCaloriesException $e) {
            print
' (Application->run) - Negative Calories Value Exception caught!';
        }
    }
}

```

Listing 23-14: *The Application class*

We declare the `Application` class with a `run()` method containing many of the statements from the older index script. As before, we include three assignment statements for the `$calories` variable that you can selectively comment out to test the project. Then we create and print a new `Dessert` object in a `try` block and use a `catch` block to handle `NegativeCaloriesException` objects. [Table 23-2](#) shows the results of running the application with the different possible values of `$calories`.

Table 23-2: Catching Exceptions with Call-Stack Bubbling

Value of <code>\$calories</code>	Program output
500	I am a Dessert containing 500 calories! (index.php) finally -- Application finished --
-1	(Application->run) - Negative Calories Value Exception caught! (index.php) finally -- Application finished --
6000	(index.php) Exception caught! (index.php) finally -- Application finished --

When a valid value of 500 is used, the object properties are printed out. When the value is -1, the `NegativeCaloriesException` is caught inside the `run()` method of the `Application` class. When the too-high value of 6000 is used, the `run()` method of the `Application` class fails to catch the general `Exception` that's thrown, since the method is watching only for

`NegativeCaloriesException` objects. As a result, the exception bubbles up to the index script, where it hits the general `catch` block. In all cases, the output ends with the message from the `finally` block in the index script.

Adding a general `try...catch` statement to an index script ensures that any bubbled-up uncaught exceptions will be addressed, meaning the application will avoid runtime errors relating to exceptions. Meanwhile, the code for handling more-specific exceptions, such as our custom `NegativeCaloriesException`, is located at a lower level of the application code, which keeps the index script simple and well organized.

Summary

This chapter introduced how to work with exceptions. You learned how to create exceptions when anticipated problematic situations occur by writing `throw` statements, and how to manage exceptions with `try...catch...finally` structures. All exceptions are instances of PHP's top-level `\Exception` class, but we discussed how to refine the program logic by using provided exception subclasses such as `InvalidArgumentException` or by declaring custom exception subclasses.

We also explored a general application architecture that exploits the bubbling up of uncaught exceptions. Specific anticipated exceptions can be caught within class methods, while any remaining exceptions can be caught in the index script at the top level of the application.

Exercises

1. Create a new project and implement a simple `Product` class with private `name` and `price` properties, public accessor methods for each property, and a constructor that takes in new values for each property and sets them using the setter methods. Add validation as follows:
 - a. If a negative value for `price` is received, an `InvalidArgumentException` is thrown.
 - b. If a `price` greater than 1000000 is received, a general `Exception` is thrown.
 - c. If an empty string is provided for the `name` property, an `InvalidArgumentException` is thrown.

Create a *composer.json* file and an index script to attempt to create a `Product` object with valid and invalid names and prices. Then wrap your index code with `try...catch` statements, so you can handle the various exceptions your code throws.

2. Make a copy of your Exercise 1 solution and introduce an `Application` class similar to the one from [Listing 23-14](#). Refactor your code as follows:
 - a. Create the `Product` object in the `run()` method of your `Application` class. Catch `InvalidArgumentException` objects and print an appropriate message as part of the `run()` method as well.
 - b. In the index script, create an `Application` object and invoke its `run()` method. Catch any bubbled-up general `Exception` objects and print an appropriate message.
3. Make a copy of your solution for Exercise 2 and introduce a custom exception named `EmptyStringException` that's thrown by the `setName()` method. Add an appropriate `catch` block to catch and process this exception in the `run()` method of the `Application` class.

24

LOGGING EVENTS, MESSAGES, AND TRANSACTIONS



Almost all live, commercial web applications keep a *log*, a record of messages, errors, events, performance summaries, and other information generated by the application while it runs. In this chapter, we'll explore how to maintain logs for PHP web applications so you can analyze application performance and respond to problems when they occur. You'll learn about PHP's built-in resources for logging, as well as Monolog, a popular third-party PHP logging package, and you'll see how to log messages to various locations.

Sometimes a log records events for auditing purposes, such as to review electronic monetary transactions for irregularities. Other times, transactions are logged for backup and recovery purposes. For example, if something goes wrong while writing information to a database, the database can be returned to a correct state by reverting to a backup (called an *image*,

or *snapshot*) from a known point in time and then rerunning the sequence of transactions logged after that snapshot was created. Logging also goes hand in hand with exceptions, which we discussed in the preceding chapter. When an exception is thrown, it can be recorded in the system log for later analysis.

Built-in PHP Resources for Logging

Logging is such a core part of server programming that PHP provides many resources to facilitate it. These include a set of predefined constants corresponding to various log severity levels, as well as built-in functions for logging messages to files. We'll explore these features now.

Predefined Constants for Severity Levels

Most computer logging systems allow messages to be classified according to a particular level of urgency or importance. To that end, PHP comes with eight predefined constants establishing levels of logging severity. These severity levels, numbered 0 through 7, from most to least urgent, correspond to the eight levels laid out in RFC 5424, a widely used standard for the syslog protocol set by the IETF. You can find this protocol at <https://www.rfc-editor.org/rfc/rfc5424>.

You can use the PHP constants in conjunction with the built-in `syslog()` function, which we'll discuss next, to generate log messages of the appropriate severity level. [Table 24-1](#) shows the eight severity levels, their RFC 5424 level names, and a summary of their meanings.

Table 24-1: Levels of Severity for Log Messages from RFC 5424

Syslog severity value	RFC 5424 log level	Meaning
0	Emergency	The system is unusable or unavailable.
1	Alert	A problem has happened, and immediate action is required.
2	Critical	A problem is about to happen and must be addressed immediately.
3	Error	A failure has occurred that is non-urgent but needs

Syslog severity value	RFC 5424 log level	Meaning
		action in a given time frame.
4	Warning	An event requires action, since it is likely to lead to an error.
5	Notice	An expected but significant event has occurred that warrants logging, but no action is required.
6	Info	An expected event has occurred for reporting and measurement.
7	Debug	Used by software developers to record detailed information supporting current debugging and code analysis.

[Table 24-2](#) shows the eight named PHP constants corresponding to the RFC 5424 log levels as well as the integer values of these constants for macOS, Unix, and Windows systems.

Table 24-2: PHP Log-Level Constants

PHP constant	macOS and Unix value	Windows value
LOG_EMERG	0	1
LOG_ALERT	1	1
LOG_CRIT	2	1
LOG_ERR	3	4
LOG_WARNING	4	5
LOG_NOTICE	5	6
LOG_INFO	6	6
LOG_DEBUG	7	6

On macOS and Unix systems, each constant has an integer value corresponding to one of the eight severity levels. For example, the `LOG_EMERG` constant has a value of 0 in macOS and Unix. If you're running

PHP on a Windows server, the values of these constants are slightly different, because of different standards for system header files. For all systems, however, the severity of the log level increases as the value of the constant decreases, in line with the principles of RFC 5424. We'll refer to the macOS and Unix values throughout this chapter.

The various severity levels have their own conventional uses. When testing and debugging code, for example, it's customary to use `LOG_DEBUG` severity and perhaps to direct these log entries to their own debugging logfile. You might log messages about standard, noncritical issues, such as a user trying to upload files that are too big or of the wrong file type, with a severity of `LOG_INFO` or `LOG_NOTICE`. This way, user-interface or file-size improvements could be considered if the same issues occur many times. Much thought should go into events likely to lead to errors, and these should be logged as `LOG_ERR` severity. Likewise, it's always important when coding for exceptions to identify those that might impact the overall functioning of the web application and log them with `LOG_EMERG`, `LOG_ALERT`, or `LOG_CRIT` severity.

Classifying log messages by severity level allows you to design computer systems with logic to respond to new log messages of different importance in different ways. For example, when a new log message occurs at the top three severity levels (Emergency, Alert, or Critical), the logging system rules might perform actions such as sending text messages and automated phone calls to the site technicians listed as being on call. Meanwhile, messages of lower importance might be written to archive files or perhaps sent via a web API to a cloud logging system. We'll explore an example of creating customized responses for different severity in “Managing Logs According to Severity” on [page 466](#).

Logging Functions

PHP has two built-in functions for logging messages: `error_log()` and `syslog()`. They differ based on where the messages get logged.

The `error_log()` function appends to the PHP error logfile, whose location is defined by the `error_log` path in your *php.ini* file or your server log settings, or to another location that can be passed as a parameter when

calling the function. (See [Appendix A](#) for information on how to locate your system's *php.ini* file.)

By contrast, the `syslog()` function appends messages to your computer system's general syslog file. [Table 24-3](#) shows the default name and location of this file on macOS, Unix, and Windows.

Table 24-3: Default Names and Locations of Syslog Files

Operating system	Filename	Location
macOS	<i>system.log</i>	<i>/var/log</i>
Unix	<i>syslog</i>	<i>/etc</i>
Windows	<i>SysEvent.evt</i>	<i>C:\WINDOWS\system32\config\</i>

When setting up an application, choosing where to log messages can be difficult: Do you want to have dedicated logfiles just for this application, do you want to log your PHP web application messages to the same location as other PHP logs, or do you want logs from the application to be added to the computer system's general logging system? As you'll see in "Logging to the Cloud" on [page 472](#), using a third-party logging library provides even more options to select from: choosing the filename and location, using multiple files for different log types, or even logging to a web API.

The decision partly depends on the nature of the project. For personal project development, logging to your local machine might make sense, whereas for mission-critical reporting of a live production system, logging to files on a web server or the cloud is probably more appropriate and may be mandated by the requirements and standards of the organization you're working for.

An advantage of logging to the system's general syslog file, as with PHP's `syslog()` function, is that logs for all applications and processes will be in one place, so you can look at issues with your web application in relation to other system issues (such as memory or processing speed problems). Also, you can use a range of applications for viewing, searching, and analyzing the general logging system, whether it be for Windows, macOS, or Unix. However, general logs are large and constantly being appended to by running processes, so when developing and even when

running a production site, targeting logs for the web application to a dedicated file, as with the `error_log()` function, can make a lot of sense. With this in mind, let's take a look at how the two built-in PHP logging functions work.

You can log a message to the PHP error logfile with `error_log()` by writing a statement such as the following:

```
error_log('Some event has happened!');
```

Pass the message you want to log as an argument to the function. By default, this message will be appended to the file specified in your *php.ini* settings. You can view that file from the command line by using `cat` (macOS/Unix) or `type` (Windows), followed by the filename. For example, here's the entry added to the logfile on my macOS laptop (which logs to a file named *php_error.log*) by the previous `error_log()` call:

```
$ cat php_error.log
[28-Jan-2025 22:08:16 UTC] Some event has happened!
```

The log entry starts with a timestamp, followed by the message string passed as an argument to the function.

The `syslog()` function takes two arguments. The first is one of the integers (0 through 7) indicating the severity level, or a constant declared with that integer value. This is where the built-in PHP constants discussed earlier come in. The second argument is a string message to be logged to the system's general syslog file. Here's an example call to the function:

```
syslog(LOG_WARNING, 'warning message from Matt');
```

We use the `LOG_WARNING` constant as the first argument, which PHP defines with a value of 4, corresponding to the fifth level on the RFC 5424 severity scale. This event requires action since it's likely to lead to an error.

The syslog file often contains hundreds or even thousands of entries, logging many events and actions from many system programs and applications. Rather than display the whole logfile, filtering it to just the

entries you want is helpful. For macOS or Unix, you can use `grep` to see entries containing a certain string. Windows has an equivalent `findstr` command. Here's an example of using `grep` to view the log entry just created with the `syslog()` function:

```
$ grep "from Matt" system.log
Jan 28 22:15:15 matts-MacBook-Pro-2 php[4304]: warning message from Matt
```

Here I've used `grep` to show only log entries containing the string "from Matt". (In Windows, the command would be `findstr "from Matt" SysEvent.evtx`.) On my Apple MacBook, the log entry created by `syslog()` begins with a formatted date, followed by the computer name (`matts-MacBook-Pro-2`). Next comes the program or service appending to the log (in this case, `php`), followed by the process ID (`4304`), a number assigned by the operating system to uniquely identify each active process. Finally, the entry ends with the message string passed to the `syslog()` function. The contents of each `syslog` entry are similar for Windows, containing the event type, event ID, source, message, and so on.

NOTE

If you aren't comfortable perusing `syslog` files at the command line, many applications are available for viewing, filtering, and analyzing these files. For example, Windows has Event Viewer, and macOS has Console.

The Monolog Logging Library

Logging is so common in web applications that several third-party PHP libraries exist to help with it, including the popular Monolog library. The majority of PHP web frameworks and cloud logging systems provide integration with Monolog. It's usually the first, and sometimes only, logging system many PHP programmers learn to use. The library makes it easy to develop customized, sophisticated logging strategies, with different types of log entries handled in different ways, and messages being logged to a variety of locations, including local files, cloud-based systems, and more.

Monolog is compliant with PSR-3, a standards recommendation for PHP logging systems. This standard uses the same eight levels of log severity as the RFC 5424 syslog standard. To be PSR-3 compliant, a logging interface should have methods for each of the eight log levels. Each method should require a string argument containing the message to be logged and an optional array for more information about the context of the message.

NOTE

Monolog's source code and documentation can be found on GitHub at <https://github.com/Seldaek/monolog>, and you can learn more about the PSR-3 standard at <https://www.php-fig.org/psr/psr-3/>.

Let's create an example project that uses Monolog to log messages. Create a new project folder and then use the command **composer require monolog/monolog** to add the Monolog library. You should now have a *composer.json* file and a *vendor* folder with an autoloader and the Monolog library classes. Next, create an index script in *public/index.php* containing the code in [Listing 24-1](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$logFile = __DIR__ . '/../logs/mylogs.log';

$logger = new Logger('demo');
$logger->pushHandler(new StreamHandler($logFile));

❶ $logger->warning('I am a warning.');
```

```
$logger->error('I am a test error!');
```

Listing 24-1: Setting up and using Monolog in public/index.php

As usual, our index script begins by requiring the autoloader script. Then we provide `use` statements for the `Monolog\Logger` and `StreamHandler` classes. Next, we declare a path to *mylogs.log* in the *logs* folder for this project; this is where we'll log our messages, but you could provide a path to any file you wish. The first time Monolog tries to append a message to this file, it will create the file and directory if they don't already exist.

We next create a new `Logger` object to manage the logging, providing the channel name `demo`. We'll explore channels and why they're useful in the next section. Every `Logger` object needs one or more *log handler* classes to tell it what to do with log entries, so we also create a log handler by calling the `Logger` object's `pushHandler()` method, passing in a new object of Monolog's `StreamHandler` class. This is a class for logging messages to files (in our case, the *logs/mylogs.log* file specified in the `$logFile` variable), but Monolog has different handler classes for other actions, such as logging to the browser, a cloud API, or a database. We'll explore another log handler in "Logging to the Cloud" on [page 472](#).

Since Monolog is PSR-3 compliant, `Logger` objects have methods for logging messages with each of the eight standard severity levels. We use two of these methods. First, we use `warning()` to create a warning log entry with the text 'I am a warning.' ❶ Then we use the `error()` method to create an error log entry with the text 'I am a test error!'

After executing the index script, the contents of *logs/mylogs.log* should look something like the following:

```
[2025-01-28T23:26:51.686815 + 00:00] demo.WARNING: I am a warning. [] []  
[2025-01-28T23:26:51.688375 + 00:00] demo.ERROR: I am a test error! [] []
```

Remember, you can view the file at the command line via `cat` (macOS and Unix) or `type` (Windows).

Notice that each log entry generated by Monolog starts with a timestamp, followed by the channel name and severity level (for example, `demo .WARNING`), followed by the log message. The empty square brackets

at the end of each log entry indicate no additional information was provided. We'll add more information about the context of the log message in "Logging Exceptions" on [page 469](#).

Organizing Logs with Channels

Larger systems are organized into well-defined subsystems, and knowing which subsystems have generated which log entries greatly aids debugging, error tracking, and code evaluations. Monolog makes this possible by giving each `Logger` object a channel name. By creating multiple `Logger` objects with unique channel names, you can organize your log based on the source of the entries. For example, an online shop might have channels like `security`, `database`, and `payments` for logging different kinds of system events.

In the previous section, we created our `Logger` object to be part of the `demo` channel, and we saw how this channel name was included in each log entry. Let's now modify our project to distinguish between two channels: `demo` and `security`. Update *public/index.php* to match the contents of [Listing 24-2](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$logFile = __DIR__ . '/../logs/mylogs.log';

$demoLogger = new Logger('demo');
$demoLogger->pushHandler(new StreamHandler($logFile));

❶ $securityLogger = $demoLogger->withName('security');

$demoLogger->error('I am a test error!');
$securityLogger->warning('invalid username entered');
```

Listing 24-2: Logging to two separate channels in index.php

We create a new `Logger` object called `$demoLogger` for a channel named `demo` and set its log handler to be a `StreamHandler` to the `logs/mylogs.log` file. Then we create a second `Logger` object with a channel named `security`. Next, we use the `$demoLogger` object's `withName()` method to create a clone of the object with a channel name of `security` ❶. This saves us from having to create the second `Logger` object and its log handler (which points to the same file as `$demoLogger`) from scratch.

We now have two `Logger` objects, `$demoLogger` (channel name `demo`) and `$securityLogger` (channel name `security`). Both of these `Logger` objects use the same log handler, writing logs to `logs/mylogs.log`. Depending on which `Logger` object we use, we can ensure that log entries are marked with the appropriate channel to aid later logfile analysis. We finish the script by logging a message to each channel. The resulting contents of the `logs/mylogs.log` file should look something like this:

```
[2025-01-30T08:54:05.091158 + 00:00]
demo.ERROR: I am a test error! [] []
[2025-01-30T08:54:05.092702 + 00:00]
security.WARNING: invalid username entered [] []
```

Notice that the error log entry went to the `demo` channel, while the warning entry went to the `security` channel. We could filter the logfile to show entries from just one of the channels by using the Unix `grep` or Windows `findstr` commands. For example, we could search for `security` channel entries by entering `findstr "security." logs/mylogs.log` in a Windows command terminal.

Managing Logs According to Severity

Beyond sorting entries into channels, we can add even more sophistication to our logging strategy by treating log entries differently according to their level of severity. Monolog can do this by adding multiple log handlers, collectively referred to as a *stack*, to the same `Logger` object. When we add a log handler, we can optionally specify which severity levels it applies to. We could, for example, have one log handler for the three most severe levels that works with a web API to automatically notify IT staff via text

message to address the problem immediately. A second log handler could respond to lower severity levels and record the messages to a logfile.

Monolog handlers also have an optional feature called *bubbling* that allows log entries to be processed by one handler and also passed (*bubbled*) down the stack to be processed again by other log handlers. In addition to high-severity log entries triggering automated messages to IT staff phones, for example, those same log entries could also be stored to a logfile for archive and analysis purposes, along with the low-severity entries. [Figure 24-1](#) shows an example log handler stack that uses bubbling and manages log entries according to severity.

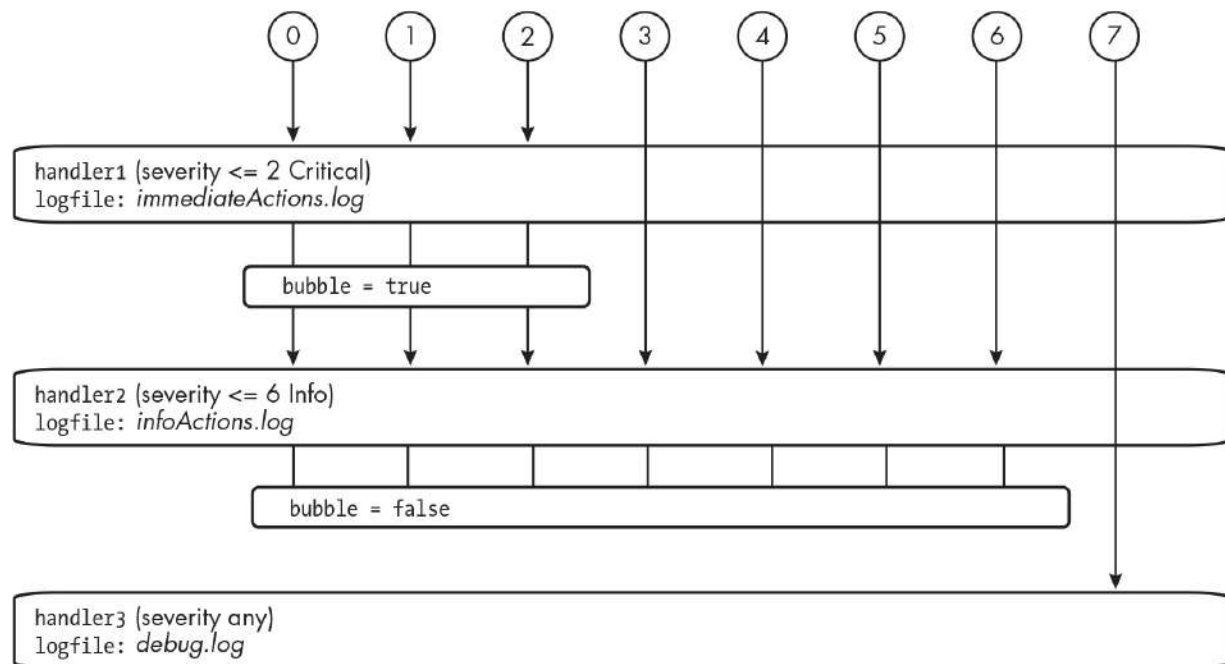


Figure 24-1: Using bubbling and severity levels to create a sophisticated logging strategy

The diagram illustrates a stack of three log handlers. At the top of the stack, `handler1` catches any log entries of Critical or more importance (severity levels 0 through 2) and records them in *immediateActions.log*. This first handler has bubbling enabled, so the high-severity log entries are also passed further down the stack for additional processing.

Next in the stack is `handler2`, which catches all log entries of Info or more importance (levels 0 through 6) and records them in *infoActions.log*.

Thanks to bubbling, the high-severity log entries will therefore be recorded in two separate files. Bubbling is turned off for `handler2`, so any log entries this handler processes won't be sent down the stack for further action. As a result, the only log entries arriving at the bottom of the log handler stack are those of severity level 7 (Debug). These are received by `handler3` and recorded in *debug.log*. Notice that `handler3` is set to receive log entries of any severity, but in practice it will receive only debug entries because all other severity levels stop at `handler2`.

Let's modify our project to implement the stack of these three log handlers. To make sure the stack works as expected, we'll generate log entries for all eight levels of severity and check the contents of the three logs. Update *public/index.php* as shown in [Listing 24-3](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Monolog\Logger;
use Monolog\Handler\StreamHandler;
❶ use Monolog\Level;

❷ $immediateActionFile = __DIR__ . '/../logs/immediateActions.
log';
$logFile = __DIR__ . '/../logs/infoActions.log';
$debugFile = __DIR__ . '/../logs/debug.log';

❸ $handler1 = new StreamHandler($immediateActionFile, Level::C
ritical);
$handler2 = new StreamHandler($logFile, Level::Info, false);
$handler3 = new StreamHandler($debugFile);

❹ $logger = new Logger('logger');
$logger->pushHandler($handler3);
$logger->pushHandler($handler2);
$logger->pushHandler($handler1);

❺ $logger->emergency('0 emergency');
```

```
$logger->alert('1 alert');
$logger->critical('2 critical');
$logger->error('3 error');
$logger->warning(' 4 warning');
$logger->notice(' 5 notice');
$logger->info('6 info');
$logger->debug('7 debug');
```

Listing 24-3: Managing log entries by severity level with a stack of three log handlers in index.php

First, we have added a `use` statement to give us access to the constants in the `Monolog\Level` class ❶. We declare three variables for the filepaths to *immediateActions.log*, *infoActions.log*, and *debug.log* ❷. Then we create three variables referencing three `StreamHandler` objects ❸. These will be the three log handlers in our stack.

For the first, `$handler1`, we pass the path to the immediate actions logfile, and we use the constant `Level::Critical` as the second argument to assign this handler to entries of Critical or greater importance. The handler has bubbling enabled by default. We provide `$handler2` the path to the info actions file and use `Level::Info` to assign it to Info-level log entries or greater (all logs except Debug entries). The third argument of `false` turns bubbling off for `handler2`.

To create `$handler3`, we simply pass the path to the debug logfile and omit the other arguments. By default, all log entries will be processed by this handler and bubbling will be enabled. However, the handler will receive only Debug-level entries, and since it will be at the bottom of the stack, no other log handler exists for log entries to bubble down to.

Next, we create a new `Logger` object ❹ and assign all three log handlers to it, one at a time. When multiple handlers are added to the same `Logger` object, the last one added is considered to be at the top of the stack and will get the chance to process all log entries first. Therefore, we add the handlers in reverse order, starting with `$handler3` and ending with `$handler1`. Finally, we log eight messages ❺, one for each level of severity, with a message confirming the level number and name.

After executing the index script, the *logs/immediateActions.log* file should look something like this:

```
[2025-02-13T10:50:52.818515 + 00:00] logger.EMERGENCY: 0 emergency [] []  
[2022-02-13T10:50:52.820236 + 00:00] logger.ALERT: 1 alert [] []  
[2022-02-13T10:50:52.820352 + 00:00] logger.CRITICAL: 2 critical [] []
```

Only Critical, Alert, and Emergency logs were processed and written to *immediateActions.log* by *\$handler1* at the top of the log handler stack. Here are the contents of *logs/infoActions.log*:

```
[2025-02-13T10:50:52.818515 + 00:00] logger.EMERGENCY: 0 emergency [] []  
[2025-02-13T10:50:52.820236 + 00:00] logger.ALERT: 1 alert [] []  
[2025-02-13T10:50:52.820352 + 00:00] logger.CRITICAL: 2 critical [] []  
[2025-02-13T10:50:52.820454 + 00:00] logger.ERROR: 3 error [] []  
[2025-02-13T10:50:52.820509 + 00:00] logger.WARNING: 4 warning [] []  
[2025-02-13T10:50:52.820563 + 00:00] logger.NOTICE: 5 notice [] []  
[2025-02-13T10:50:52.820617 + 00:00] logger.INFO: 6 info [] []
```

All logs from levels 0 to 6 were processed and written to *infoActions.log* by *\$handler2* from the middle of the log handler stack. Since we've already seen the level 0, 1, and 2 logs in *immediateActions.log* from *\$handler1*, seeing them again in *infoActions.log* confirms that the bubbling mechanism has worked, allowing these logs to also be received by *\$handler2*. Finally, here are the contents of the *logs/debug.log* file:

```
[2025-02-13T10:50:52.820672 + 00:00] logger.DEBUG: 7 debug  
[] []
```

Only the entry for severity level 7 (Debug) can be seen in *debug.log*. This demonstrates that `$handler3` at the bottom of the stack received only this single log entry.

Logging Exceptions

A common use of logs is to record when exceptions occur during program execution. In [Chapter 23](#), we explored how programs can be organized around `try...catch` statements: a `try` statement with the code that should execute under normal circumstances, and a `catch` statement for handling exceptions. When an application uses logging, the exceptions are logged as part of the `catch` statement.

Let's create a simple, single-class project to illustrate how to do this. Our project will have a `Product` class that throws an exception when we try to create a `Product` object with a negative price. We'll use Monolog to log those exceptions to a *logs/debug.log* file. We'll begin by declaring the `Product` class. Create a new project with *src/Product.php* containing the code in [Listing 24-4](#).

```
<?php  
namespace Mattsmithdev;  
  
class Product  
{  
    private string $name;  
    private float $price;  
  
    public function __construct(string $name, float $price)  
    {  
        ❶ if ($price < 0) {  
            throw new \Exception(  
                'attempting to set price to a negative valu  
e');  
        }  
    }  
}
```

```
        $this->price = $price;
        $this->name = $name;
    }
}
```

Listing 24-4: A `Product` class that throws an exception

We declare the `Product` class in the `Mattsmithdev` namespace and give it two private properties, `name` and `price`. The class's constructor method takes in `$name` and `$price` values for the new object being created. Within the constructor, we validate the `$price` argument and throw an exception if its value is negative ❶. For this simple example, we're using PHP's root `Exception` class.

We now need to create a *composer.json* file to autoload the class. [Listing 24-5](#) shows how.

```
{
    "autoload": {
        "psr-4": {
            "Mattsmithdev\\": "src"
        }
    }
}
```

*Listing 24-5: The *composer.json* file*

Next, use Composer at the command line to generate the autoloader scripts and add the Monolog library to the project:

```
$ composer dump-autoload
$ composer require monolog/monolog
```

Now we need to write an index script that attempts to create a `Product` object and logs the exception if the attempt is unsuccessful. Create *public/index.php* to match [Listing 24-6](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Product;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$debugFile = __DIR__ . '/../logs/debug.log';
$logger = new Logger('demo');
$logger->pushHandler(new StreamHandler($debugFile));

try {
    $p1 = new Product('hammer', -1);
} catch (\Exception $e) {
    ❶ $logger->error('problem creating new product', ['exception' => $e]);
}
```

Listing 24-6: Attempting to create an invalid `Product` object in `index.php`

First, we read and execute the autoloader and add `use` statements for the classes we need. Then we get set up for logging by creating a variable for the *logs/debug.log* filepath, creating a new `Logger` object for a channel named `demo` and giving it a log handler. Next, inside a `try` block, we create a new `Product` object, passing `-1` for the price. In the related `catch` block, we log an entry of Error-level severity with the `Logger` object if creating the product fails ❶.

In addition to providing a message to log (`'problem creating new product'`), we include an array as the second argument to log additional information. Specifically, we pass the entire `Exception` object `$e` and give it the key `exception`. In the Monolog documentation, this optional array is called the *context* of the log entry. It can contain multiple elements with keys of your choosing, which can be helpful when reviewing the logs and analyzing them for patterns.

After executing the index script, the *logs/debug.log* file should look something like this:

```
[2025-01-25T11:48:46.813377 + 00:00] demo.ERROR: problem creating new product  
{"exception":"[object] (Exception(code: 0): attempting to set price to a  
negative value at /Users/matt/src/Product.php:15)"} []
```

An Error-level log has been added to the logfile for the `demo` channel, with the message `problem creating new product`. The log entry also contains the details of the `Exception` object that was thrown by the `Product` constructor method, including the message associated with the exception (attempting to set price to a negative value) and the location of the exception-triggering code.

Logging to the Cloud

So far we've been logging messages to files, but most large-scale web applications log to a dedicated cloud-based logging system rather than to files on the server. One popular cloud logging system is Mezmo (previously LogDNA). Using a cloud logging API such as Mezmo provides many benefits, including historical storage of logs, powerful filtering and search features, and comprehensive analytical and reporting features. Cloud logging APIs such as Mezmo can also be linked to alert notification systems such as Atlassian's Opsgenie to send the email or text alerts for log entries requiring immediate actions.

Let's create a project that sends log entries to Mezmo. We'll log to two separate channels and try out entries of each severity level. First, visit the Mezmo website (<https://www.mezmo.com>) and create a free account. Make a note of the unique hexadecimal Mezmo ingestion key created for you in your account details; you'll need to reference it in your script.

To interact with Mezmo from your PHP code, we'll use the `monolog - logdna` package, maintained by Nicolas Vanheuverzwijn. This package adds Mezmo API communication capabilities to Monolog. Create a new project folder and add the package by entering `composer require nvanheuverzwijn/monolog-logdna` at the command line. You should now have a `composer.json` file and a `vendor` folder containing an autoloader and the Monolog and other library classes for logging to the Mezmo API. Now

create an index script in *public/index.php* containing the code in [Listing 24-7](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Monolog\Logger;
use Zwiijn\Monolog\Handler\LogdnaHandler;

❶ $INGESTION_KEY='your-MEZMO-ingestion-key-goes-here';

$generalLogger = new Logger('general');
$handler = new LogdnaHandler($INGESTION_KEY, 'host-mgw.com');
$generalLogger->pushHandler($handler);

❷ $generalLogger->emergency('0 emergency');
$generalLogger->alert('1 alert');
$generalLogger->critical('2 critical');
$generalLogger->error('3 error');
$generalLogger->warning('4 warning');
$generalLogger->notice('5 notice');
$generalLogger->info('6 info');
$generalLogger->debug('7 debug');

❸ $securityLogger = $generalLogger->withName('security');
$securityLogger->debug('7 debug - from security channel',
    ['context-1' => 'some data']);
```

Listing 24-7: Setting up and using Monolog in public/index.php

We provide `use` statements for Monolog's `Logger` class and for `LogDnaHandler`, the log handler needed to log to Mezmo. Then we declare a variable for the necessary Mezmo ingestion key; be sure to fill in your own key here ❶. Next, we create a new `Logger` object named `$generalLogger`, providing `general` as the channel name, and we give it a log handler, passing the ingestion key and naming the host source of logs as `host-`

mgw.com (short for My Great Website, as usual). Different web applications or subsites could use different hostnames in their handlers, to further differentiate the source of logs.

We log eight messages to our `$generalLogger` object ❷, one for each level of severity, with a message confirming the level number and name. Then we save a bit of work by using the `withName()` method to create a clone of the `$generalLogger` object called `$securityLogger` with a channel name of `security` ❸. Both `Logger` objects use the same log handler and so can send logs to the Mezmo API. We use this second object to log a debug entry, passing a second argument of a single-element array with the 'context-1' key and 'some data' data string. This tests how we might record extra data in a log entry.

[Figure 24-2](#) shows the logs from our executed index script, received and displayed on the Mezmo site.

The screenshot displays a list of log entries from the Mezmo cloud service. The entries are as follows:

Timestamp	Host	Channel	Level	Message	Retained
Dec 26 16:53:11	host-mgw	general	EMERGENCY 0	emergency	(not retained)
Dec 26 16:53:11	host-mgw	general	ALERT 1	alert	(not retained)
Dec 26 16:53:12	host-mgw	general	CRITICAL 2	critical	(not retained)
Dec 26 16:53:12	host-mgw	general	ERROR 3	error	(not retained)
Dec 26 16:53:12	host-mgw	general	WARNING 4	warning	(not retained)
Dec 26 16:53:12	host-mgw	general	NOTICE 5	notice	(not retained)
Dec 26 16:53:12	host-mgw	general	INFO 6	info	(not retained)
Dec 26 16:53:12	host-mgw	general	DEBUG 7	debug	(not retained)
Dec 26 16:53:12	host-mgw	security	DEBUG 7	debug - from security channel	(not retained)

Below the list, a detailed view of the selected log entry is shown:

Copy to clipboard Close

Meta Object

context-1 some data

7 debug - from security channel

LINE IDENTIFIERS

Source host-mgw App security

Figure 24-2: Log entries on the Mezmo cloud service

Mezmo shows timestamped logs from the `general` and `security` channels, with all entries coming from `host-mgw`. Each entry is marked with its severity level. The details of the final log, to the `security` channel, have been expanded in the figure, revealing the context data we passed to the `Logger` object via an array.

Summary

As you've seen in this chapter, you can create logs for a web application in several ways, from simple `error_log()` function calls to the sophisticated `Monolog` open source logging library package to APIs like Mezmo for cloud storage and analytics. The scale and importance of each project will determine the most appropriate approach to take, but for almost all projects that you need to quality-assure and maintain, you'll probably have to adopt some form of logging to record and manage errors and exceptions, and to collect historical data about use and performance of the system.

Exercises

1. Create message entries by using both the `syslog()` and `error_log()` functions. Locate the files that these functions write to on your computer system and view your messages in the logfiles.
2. Create a new project and use Composer to add the `Monolog` package. In your index script, create a new `Logger` object for a channel named `general`, and add a `StreamHandler` to append logs to the `logs/mylogs.log` file. Log several entries of different severity levels, and view the log entries in your logfile after executing your index script.
3. Create a new project with a stack of two handlers: `handler1` (appending to the `urgent.log` file) and `handler2` (appending to the `other.log` file). Add `handler2` first so that `handler1` will be on the top of the stack. Turn off bubbling for `handler1` and configure it to catch all log entries of Critical or more importance. Generate log entries for all eight levels of severity. You should see log entries of severity 0, 1, and 2 in `urgent.log`, and all others (3 through 7) in `other.log`.
4. Create an account at a cloud logging site such as Mezmo, and update the project from Exercise 3 to log entries to that site's API. View the

logs online to confirm that your program successfully sent them via the API.

25

STATIC METHODS, PROPERTIES, AND ENUMERATIONS



In this chapter, we'll explore *static members*. Unlike the instance-level properties and methods we've been using so far, which are accessible through the individual objects of a class, static properties and methods are accessed through the class as a whole. As such, you don't have to create an object of a class to use its static members.

We'll discuss how to work with static members and illustrate their usefulness in situations such as storing information about all the instances of a class or sharing resources across an entire application. We'll also touch on enumerations, which provide a way to list all possible values for a data type.

Storing Class-Wide Information

One common use of static members is to keep track of information about all instances of a class. This is handy when a message needs to be sent to all

objects of a class, or when a calculation must be based on just the class's current instances. Consider the `AudioClip` class diagrammed in [Figure 25-1](#).

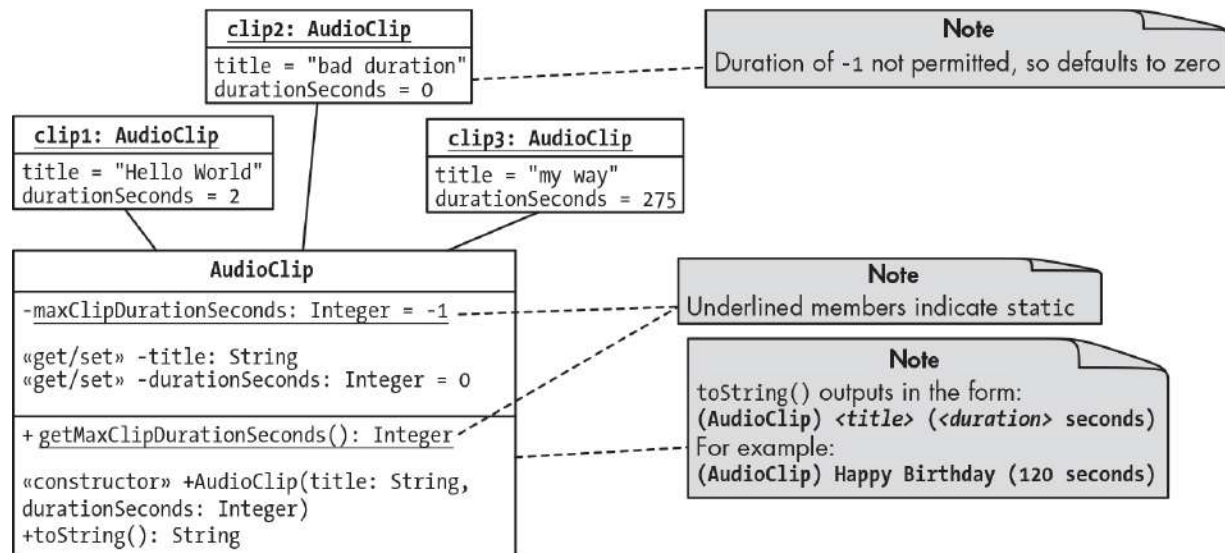


Figure 25-1: An `AudioClip` class with static members

The `AudioClip` class has instance members for storing details about a given audio file. These are the `title` and `durationSeconds` properties, with their associated getters and setters. The constructor and `__toString()` method are instance members as well, since they pertain to creating or summarizing the contents of a particular object. Meanwhile, the class has a `maxClipDurationSeconds` property that tracks the longest duration of any `AudioClip` objects that have been created. This is a good use of a static member (indicated by underlining in the diagram), since the property holds information pertaining to all the objects of the class. The property's getter method should also be static.

To see how static members are useful for storing class-wide information, we'll declare the `AudioClip` class and create three instances of it. The diagram shows how each instance will have its own values for the `title` and `durationSeconds` properties. For example, `clip1` will have a `title` of 'Hello World' and a `durationSeconds` of 2.

Create a new project folder and add the usual *composer.json* file declaring that `Mattsmithdev` namespaced classes are located in the *src* folder. Generate the autoloader file by entering `composer dump-autoload` at the command line. Then declare the `AudioClip` class by creating *src/AudioClip.php* and entering the contents of [Listing 25-1](#).

```
<?php
namespace Mattsmithdev;

class AudioClip
{
    // --- Static (per-class) members ---
    ❶ private static int $maxClipDurationSeconds = -1;

    ❷ public static function getMaxClipDurationSeconds(): int
    {
        return self::$maxClipDurationSeconds;
    }

    // --- Object (instance) members ---
    ❸ private string $title;
    private int $durationSeconds = 0;

    ❹ public function __construct(string $title, int $duration
Seconds)
    {
        $this->setTitle($title);
        $this->setDurationSeconds($durationSeconds);
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function setTitle(string $title): void
    {
        $this->title = $title;
    }
}
```

```

    }

    public function getDurationSeconds(): int
    {
        return $this->durationSeconds;
    }

    ❸ public function setDurationSeconds(int $durationSecond
s): void
    {
        // Exit with no action if negative
        if ($durationSeconds < 0) return;

        $this->durationSeconds = $durationSeconds;

        if ($durationSeconds > self::$maxClipDurationSecond
s) {
            self::$maxClipDurationSeconds = $durationSecond
s;
        }
    }

    ❹ public function __toString(): string
    {
        return "(AudioClip) $this->title ($this->durationSec
onds seconds) \n";
    }
}

```

Listing 25-1: The *AudioClip* class

We declare `maxClipDurationSeconds` by using the `static` keyword to specify that this is a static member that exists independent of any objects of the `AudioClip` class ❶. We initialize it to `-1`, ensuring that whatever the duration of the first `AudioClip` object to be created, its duration will be greater than `-1` and so will be stored in this static property. We'll see how this is done later, as part of the `setDurationSeconds()` setter method.

We set `maxClipDurationSeconds` to be private, but we declare a public getter method, `getMaxClipDurationSeconds()`, which is also static ❷. Making this method public allows code inside and outside the class to query the value of the longest `AudioClip` object that's been created since the program or request has been running. We'll explore what this implies and how this property is used shortly.

We next declare two instance-level properties for each `AudioClip` object, `title` and `durationSeconds` ❸, with a default value of 0 for the latter to ensure that it's set even if an invalid argument is provided at construction. The class's constructor method ❹ takes in initial values for these two properties and sets them in the object by invoking the appropriate setters.

The instance-level accessor methods are all straightforward, save for `setDurationSeconds()`, which has custom validation logic ❺. A clip's duration should never be negative, so we first test for this and use `return` to halt execution of the method with no further action if a negative value is provided. If we make it past that point, we know the provided argument is 0 or positive, so we store it in the object's `durationSeconds` property.

Then we check whether the object's new duration is greater than the value stored in the `maxClipDurationSeconds` static property. If it is, we update this property to equal the new duration. Since `maxClipDurationSeconds` starts off with a sentinel value of -1, and because of our validation at the beginning of the method, we know that no `AudioClip` objects will ever have a negative duration. Therefore, as soon as the first clip has been created with a valid duration, this static property will be set accordingly.

Notice that we have to use `self::` to access the `maxClipDurationSeconds` property. This *scope resolution operator* (`::`) is the syntax for accessing a static member from within the same class.

We complete the declaration of the class with the `__toString()` method ❻. It returns a string summarizing an `AudioClip` object's contents, in the form `(AudioClip) title (durationSeconds seconds)`.

Now let's put our class to work through an index script that creates `AudioClip` objects and outputs the changing value of the static `maxClip`

DurationSeconds property. Create *public/index.php* as shown in [Listing 25-2](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\AudioClip;

print '- Max AudioClip duration so far = '
    . AudioClip::getMaxClipDurationSeconds() . PHP_EOL;

$clip1 = new AudioClip('hello world', 2);
print $clip1;

print '- Max AudioClip duration so far = '
    . AudioClip::getMaxClipDurationSeconds() . PHP_EOL;

$clip2 = new AudioClip('bad duration', -10);
print $clip2;

print '- Max AudioClip duration so far = '
    . AudioClip::getMaxClipDurationSeconds() . PHP_EOL;

$clip3 = new AudioClip('My Way', 275);
print $clip3;

print '- Max AudioClip duration so far = '
    . AudioClip::getMaxClipDurationSeconds() . PHP_EOL;
```

Listing 25-2: The index.php script

First, we print a message displaying the value of `maxClipDurationSeconds`, accessed via the public static `getMaxClipDurationSeconds()` method. Since we haven't yet instantiated any `AudioClip` objects, the property should still have its initial value of `-1`. Notice that we have to prefix the static member's name with `AudioClip::` rather than `self::`, since we're writing this code from the `index` script and not from within the static members' class. We then create three `AudioClip`

objects, `$clip1`, `$clip2`, and `$clip3`, printing each one and then displaying the value of `maxClipDurationSeconds` again.

Here's the output of running the index script:

```
- Max AudioClip duration so far = -1
(AudioClip) hello world (2 seconds)
- Max AudioClip duration so far = 2
(AudioClip) bad duration (0 seconds)
- Max AudioClip duration so far = 2
(AudioClip) My Way (275 seconds)
- Max AudioClip duration so far = 275
```

The max duration starts with a value of `-1`, but each time an `AudioClip` object with a nonnegative duration is created, this value is updated if the new clip is the longest, through the logic in the `setDurationSeconds()` method. After creating the `hello world` object, the max duration goes from `-1` to `2`. The value doesn't change after the `bad duration` object is created with an invalid duration of `-10`. (Notice from the object's printout that its duration is stored as `0`, the default value, rather than `-10`, thanks to our initial validation logic in the `setDurationSeconds()` method.) Finally, after creating the `My Way` object, the max duration is updated to `275`.

Our logic for updating `maxClipDurationSeconds` has worked well for demonstration purposes, but it isn't actually a good way to keep track of the longest `AudioClip` object, since it assumes that all the objects exist for the entire run of the application. Say we decide that we no longer want one of the audio clips (the user might choose to delete it from a list). Our current logic provides no way to roll back the maximum clip duration if that clip is deleted. A better approach might be to have an array of active `AudioClip` objects. Each time a clip is removed, we could then loop through the array and recalculate the duration of the longest active clip.

Static Properties vs. Class Constants

Static properties aren't to be confused with class constants. In both cases, only one copy of the static property or class constant exists on the class itself, rather than a separate copy on each instance of the class. However,

class constants are immutable, so their value never changes. By contrast, the value of a static property can change (as you saw with `maxClipDurationSeconds`), just like the value of an ordinary property of an object. In this sense, the term *static* can be a bit misleading. All PHP object-oriented properties, whether they're per-object instance properties or per-class static properties, begin with a dollar sign, which distinguishes them from constants.

Use a class constant when you have a value that should apply to all objects of a class and should never change. We've met class constants previously; one example is the `PATH_TO_TEMPLATES` constant used to create the `Twig\Environment` object for templating. In this case, the filepath to the templates folder should never change and would apply equally to any and all `Twig\Environment` objects. Other uses of a class constant include defining special values, like setting the maximum score for a school grade point average (GPA) to `4.0` or the neutral value for the pH acidity scale to `7.0`.

In the case of our `AudioClip` class, we might have class constants defining details like the number of channels or the sampling rate of the audio files, on the assumption that these will be standard across all audio clips. To explore the difference between class constants and static properties, we'll add some class constants to our `AudioClip` project now.

[Figure 25-2](#) shows an updated diagram of the `AudioClip` class.

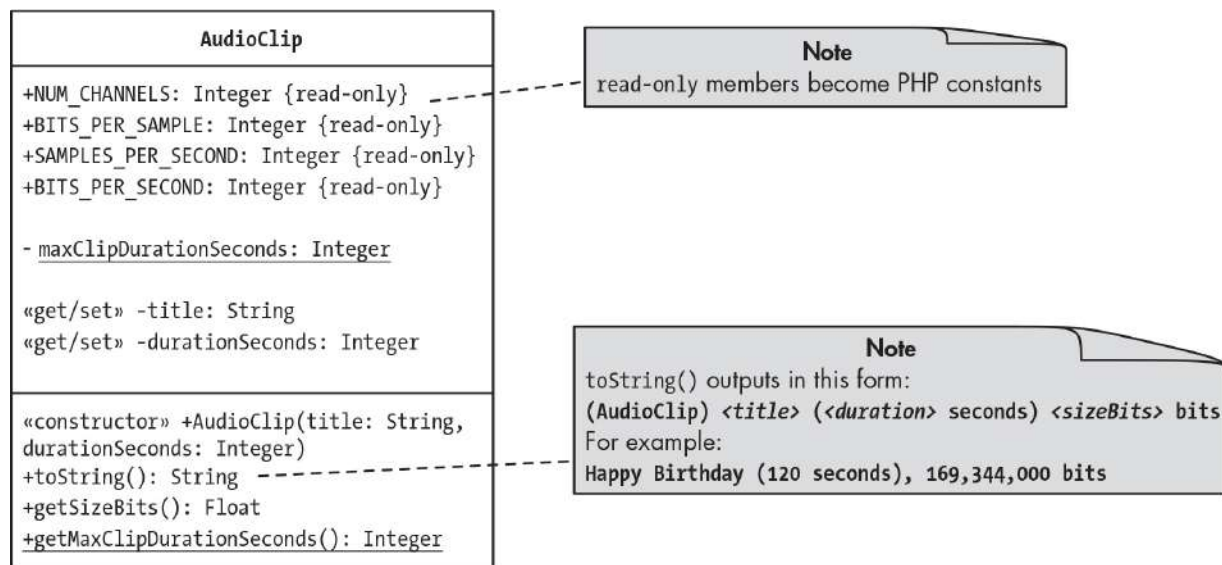


Figure 25-2: Adding class constants and an extra method to the `AudioClip` class

The diagram shows that the `AudioClip` class will now offer four public class constants, which are indicated as `{read-only}` in class diagrams. For our project, we'll make some assumptions about the audio clips that the class represents: they'll all be stereo (two channels) and store 16 bits per sample (CD-quality digital audio). The number of samples per second will always be 44,100, which is a common audio sampling rate. These values are indicated by the `AudioClip` class constants `NUM_CHANNELS`, `BITS_PER_SAMPLE`, and `SAMPLES_PER_SECOND`, and together they're used to calculate the `BITS_PER_SECOND` constant.

NOTE

For a more realistic example, these constants could be replaced by instance properties, since not all audio clips may conform to the standard laid out by the constants.

We'll also add a new `getSizeBits()` method that uses the class constants to calculate the number of bits required to store the audio clip in memory or on disk. Additionally, we'll include the value from this new method as part of the object summary returned by the `__toString()` method.

To implement all these changes, edit the *src/AudioClip.php* file as shown in [Listing 25-3](#).

```
<?php
namespace Mattsmithdev;

class AudioClip
{
    const NUM_CHANNELS = 2;

    const BITS_PER_SAMPLE = 16;

    const SAMPLES_PER_SECOND = 44100;

    const BITS_PER_SECOND = self::NUM_CHANNELS
        * self::BITS_PER_SAMPLE * self::SAMPLES_PER_SECOND;

    --snip--

    public function getSizeBits(): int
    {
        return self::BITS_PER_SECOND * $this->durationSeconds;
    }

    ❶ public function __toString(): string
    {
        $numBitsFormatted = number_format($this->getSizeBits());
        return "(AudioClip) $this->title "
            . "($this->durationSeconds seconds), $numBitsFormatted bits \n";
    }
}
```

Listing 25-3: Adding constants and the *getSizeBits()* method to the *AudioClip* class

It's common to list constants first when declaring a class, before any properties and methods, so we begin the *AudioClip* class by declaring the

constants `NUM_CHANNELS`, `BITS_PER_SAMPLE`, and `SAMPLES_PER_SECOND`, with the values described previously. Then we declare `BITS_PER_SECOND`, which is an example of a *calculated constant* since its value is determined based on the values of the other constants rather than being set directly. By precalculating this value and storing it as a constant, we avoid having to repeat the calculation every time we need to convert from seconds of audio to bits of data. Notice the use of `self::` to access the other constants in the `BITS_PER_SECOND` calculation. This syntax applies to accessing class constants from within the class, just as it does to accessing static members.

Next, we declare `getSizeBits()`, a useful extra getter method for each `AudioClip` object. It returns an integer representing the number of bits required to store the audio clip, found by multiplying the precalculated `BITS_PER_SECOND` constant by the object's `durationSeconds` property. We also update the `__toString()` method ❶, summarizing an `AudioClip` object's contents including its size in bits, thanks to our new method. Notice that we use the built-in `number_format()` function to create a temporary `$numBitsFormatted` string variable. With the function's default settings, this creates a more readable representation of the number with a comma separator every three digits.

Here's the terminal output of running the index script again (no changes to the index script are needed):

```
- Max AudioClip duration so far = -1
(AudioClip) hello world (2 seconds), 2,822,400 bits
- Max AudioClip duration so far = 2
(AudioClip) bad duration (0 seconds), 0 bits
- Max AudioClip duration so far = 2
(AudioClip) My Way (275 seconds), 388,080,000 bits
- Max AudioClip duration so far = 275
```

Each `AudioClip` object printout now ends with the integer number of bits the clip data occupies, thanks to our class constants. Even for a two-second clip, over 2 million bits are needed (which is why we've formatted the number of bits with comma separators).

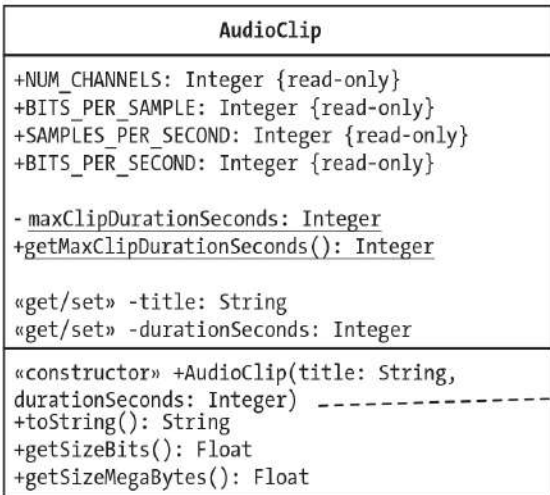
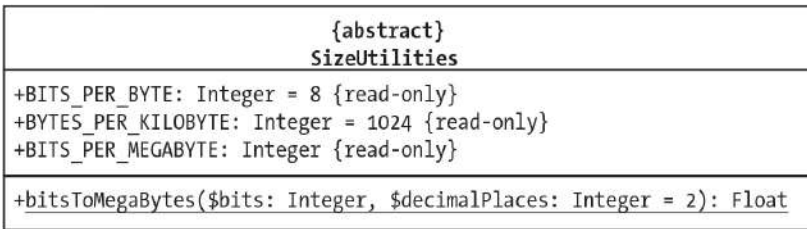
Utility Classes with Static Members

Static members may also be created as part of *utility classes*; these exist to help other classes do their work and aren't used to create objects. A utility class's static members might store helpful information or perform basic, general-purpose calculations that might have uses in other projects or other parts of the current project.

Continuing with our `AudioClip` example, let's say we want to display the size of each audio clip in megabytes rather than bits. We'll need a way to convert from bits to megabytes. We could refactor the `AudioClip` class's `getSizeBits()` method to make the necessary calculation. This calculation, along with supporting pieces of information such as the number of bits in a byte, is general enough that it might be useful to other, non-audio parts of the project or to other projects entirely. Therefore, locating the necessary code in a utility class makes sense.

We'll create a utility class called `SizeUtilities` to help the `AudioClip` class calculate memory size. Often, utility classes aren't used to create objects, and this will be the case for `SizeUtilities`. Since an object of this class will never be instantiated, we'll declare `SizeUtilities` as `abstract`. By extension, since there will never be a `SizeUtilities` object, all the class's members need to be accessible through the class as a whole. That is, `SizeUtilities` must consist of class-level constants and static members.

[Figure 25-3](#) shows a diagram of the modified `AudioClip` class and the new `SizeUtilities` utility class.



Note

toString() outputs in this form:
 (AudioClip) <title> (<duration> seconds) <sizeMb>Mb
 For example:
 (AudioClip) Happy Birthday (120 seconds) 20.19Mb

Figure 25-3: A class diagram of *AudioClip* and *SizeUtilities* demonstrating static methods and properties

The *SizeUtilities* class offers three public class constants indicated as {read-only} in the diagram: *BITS_PER_BYTE*, *BYTES_PER_KILOBYTE*, and *BITS_PER_MEGABYTE*. These will be useful for calculating file sizes. The class also offers a public static method called *bitsToMegaBytes()* that takes in a number of bits as a parameter and returns the corresponding number of megabytes to a given number of decimal places (or two by default). Meanwhile, the *AudioClip* class declares a new *getSizeMegaBytes()* method that returns a float representing the memory required for the clip in megabytes, with the help of the *SizeUtilities* class. Notice that the *AudioClip* class also has a modified *__toString()* method that includes the file size in megabytes in the object summary.

Let's declare the *SizeUtilities* class. Create *src/SizeUtilities.php* containing the code in [Listing 25-4](#).

```
<?php
namespace Mattsmithdev;
```

```

abstract class SizeUtilities
{
    const BITS_PER_BYTE = 8;
    const BYTES_PER_KILOBYTE = 1024;
    const BITS_PER_MEGABYTE =
        self::BITS_PER_BYTE * self::BYTES_PER_KILOBYTE * 1024;

    public static function bitsToMegaBytes(int $bits): float
    {
        return $bits / self::BITS_PER_MEGABYTE;
    }
}

```

Listing 25-4: The *SizeUtilities* class with constants and a static method

We declare *SizeUtilities* as abstract, since all members of this class are either constants or static. Its `BITS_PER_BYTE` and `BYTES_PER_KILOBYTE` constants have values of 8 and 1024, respectively, and the `BITS_PER_MEGABYTE` constant is calculated based on them. The class's `bitsToMegaBytes()` static method takes in a number of bits and divides it by the `BITS_PER_MEGABYTE` constant to return the equivalent number of megabytes.

Next, update *src/AudioClip.php* as shown in [Listing 25-5](#).

```

<?php
namespace Mattsmithdev;

class AudioClip
{
    --snip--

    public function getSizeBits(): int
    {
        return self::BITS_PER_SECOND * $this->durationSeconds;
    }
}

```

```

    public function getSizeMegabytes(): float
    {
        ❶ return SizeUtilities::bitsToMegabytes($this->getSize
Bits());
    }

    public function __toString(): string 2
    {
        $numMegabytesFormatted = number_format($this->getSiz
eMegabytes(), 2);
        return "(AudioClip) $this->title "
            . "($this->durationSeconds seconds), $numMegaByt
esFormatted MB \n";
    }
}

```

Listing 25-5: Updating the `AudioClip` class to use `SizeUtilities`

We add the `getSizeMegabytes()` method, which returns a float representing the size of the `AudioClip` object in megabytes. The method gets the size in bits and passes it to the public static `bitsToMegabytes()` method declared in our `SizeUtilities` class ❶. This is a good example of how to use public static methods in other classes: we don't have to create an object of the `SizeUtilities` class in order to use its public static method. Instead, we simply write `SizeUtilities::` followed by the name of the method.

In the new `__toString()` method, we create a temporary `$numMegabytesFormatted` variable holding the clip size in megabytes, formatted as a string with two decimal places. We then include the value of this variable as part of the string returned by the method. Once again, we don't need to make any changes to the index script to test our updates, since we're still using `__toString()` to output information. Here's the terminal output of the updates:

```

- Max AudioClip duration so far = -1
(AudioClip) hello world (2 seconds), 0.34 MB
- Max AudioClip duration so far = 2

```

```
(AudioClip) bad duration (0 seconds), 0 MB
- Max AudioClip duration so far = 2
(AudioClip) My Way (275 seconds), 46.26 MB
- Max AudioClip duration so far = 275
```

The size of each `AudioClip` object is now given in megabytes. Our new utility class has successfully helped the `AudioClip` class make the necessary bits-to-megabytes conversions through its static method and class constants.

Sharing Resources Across an Application

Another use of static methods is to offer a functionality (such as reading from or writing to a database, or appending messages to a logfile) to all parts of a software system, without each part of the system needing to duplicate the required setup code. The idea is to create an abstract class with static methods that do the necessary legwork, like establishing a database connection or setting up a logger and log handler. Then you can call those static methods from anywhere in your code whenever you need that functionality. This makes processes like logging or working with a database connection quite straightforward.

To illustrate how this works, let's create a project that allows logging from anywhere in the system just by writing something like this:

```
Logger::debug('my message');
```

For this to work, we'll need a `Logger` class with a public static `debug()` method that can be invoked from anywhere in the system. This method will handle the mechanics of the logging process so that the other parts of the system don't have to.

Start a new project folder and create the usual `composer.json` file to autoload the `Mattsmithdev` namespaced classes. Then enter this command at the command line to add the Monolog library to the project:

```
$ composer require monolog/monolog
```

Since we have a *composer.json* file, Composer will also generate the autoloader scripts at this step, in addition to loading the Monolog library. All the relevant files will be in your project's *vendor* directory.

Next, create a custom `Logger` class in *src/Logger.php* as shown in [Listing 25-6](#).

```
<?php
namespace Mattsmithdev;

❶ use Monolog\Logger as MonologLogger;
use Monolog\Handler\StreamHandler;

abstract class Logger
{
    const PATH_TO_LOG_FILE = __DIR__ . '/../logs/debug.log';

    ❷ public static function debug(string $message): void
    {
        $logger = new MonologLogger('channel1');
        $logger->pushHandler(new StreamHandler(self::PATH_TO_LOG_FILE));
        $logger->debug($message);
    }
}
```

Listing 25-6: The `Logger` class with a public static logging method

We declare our `Logger` class in the `Mattsmithdev` namespace to avoid a naming collision with the Monolog library's `Logger` class and designate it as `abstract` since we won't ever need to instantiate it. The `use` statements allow us to refer to the necessary Monolog classes in our code without having to write fully qualified namespaces. Notice that we alias Monolog's `Logger` class as `MonologLogger` to better differentiate it from our own class ❶.

Inside the class, we create a constant for the filepath to *logs/debug.log*. Then we declare `debug()` as a public static method that takes a `$message` string parameter ❷. The method creates a new `MonologLogger` object for

channel1 and assigns it a log handler for writing to the debug logfile, then uses the Monolog class's `debug()` method to log `$message` to the logfile.

Now create an index script in *public/index.php* containing the code in [Listing 25-7](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';
use Mattsmithdev\Application;
$app = new Application();
$app->run();
```

Listing 25-7: Creating an `Application` object in `index.php`

We read in and execute the Composer-generated autoloader, create an object of the `Application` class, and invoke its `run()` method. This is the same basic pattern for the index script of an object-oriented web application that we discussed in [Chapter 21](#).

Finally, declare the `Application` class in *src/Application.php* as shown in [Listing 25-8](#). The class includes code for logging messages via our `Logger` class's static method.

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
    {
        print 'Hello, world!';
        Logger::debug('Hello, world! printed out');
        Logger::debug('another log message');
    }
}
```

Listing 25-8: The `Application` class

In the `Application` class's `run()` method, we print out a 'Hello, world!' message. Then we invoke the static `debug()` method of our `Logger`

class to log messages to the debug logfile. Notice that we don't need to create a `Logger` object or include any setup code (such as declaring a path to the logfile or creating a log handler) in the `Application` class itself; everything is handled by the static method we defined on the `Logger` class. If the `Application` and `Logger` classes were in different namespaces, all we'd need to do is add a `use` statement or fully qualify the class when using it, like this:

```
Mattsmithdev\Logger::debug('Hello, world! printed out');
```

We can confirm that the two messages have been appended to the logfile by viewing the contents of *logs/debug.log* after running the index script. Remember to use the `cat` command (macOS and Unix) or `type` (Windows), as discussed in [Chapter 24](#). You should see something like this:

```
$ cat debug.log
[2025-01-30T10:49:54.516974 + 00:00]
channel1.DEBUG: Hello, world! printed out [] []

[2025-01-30T10:49:54.519278 + 00:00]
channel1.DEBUG: another log message [] []
```

We've successfully used our static method to append messages to the logfile. If we wanted, we could add optional arguments to this method to change the name and location of the logfile, log to different channels, or provide a context array such as an `Exception` object. This basic example has illustrated generally how a class can offer a single static method to make it easy for any part of a software system to utilize its functionality.

Saving Resources with the Singleton Pattern

The approach we used in the preceding section to share resources across an application can work in many situations, but in some cases, such as creating a connection to a database or setting up a mailing or file-writing object, the static method's task takes up enough time and memory to impair the application's performance. In cases like these, an object-oriented design

technique called the *singleton pattern* can help conserve computing resources while still making an operation available throughout an application.

Take a look back the last section of [Listing 25-6](#), where we declared our `Logger` class. According to our definition of the static `debug()` method, a new `MonologLogger` object and a new `StreamHandler` object will be created each time the method is invoked to log a debug message. In our application's `run()` method in [Listing 25-8](#), we invoked this method twice, so four objects were created to log the two messages, which is a bit wasteful. When operations are resource expensive, a more efficient approach is to perform them once and then *cache* (store) the created resources for future use. The singleton pattern is one way of doing this.

The singleton pattern declares a class with a private constructor, along with logic to ensure that, at most, we create only one object of the class. The class offers a public static method named `getInstance()` that returns a reference to the single instance of the class. If no instance exists, one is created the first time the `getInstance()` method is invoked. Otherwise, the class caches, or keeps a record of, the single instance, so it can be returned again the next time `getInstance()` is called.

Anytime you need to use the singleton class, you'd write something like this from anywhere in your code:

```
$myObject = Singleton::getInstance();
```

This stores a reference to the sole instance of the singleton class in the `$myObject` variable. You can then use the `Singleton` class's resources through the `$myObject` reference. [Listing 25-9](#) shows the typical skeleton for singleton-style classes.

```
<?php
class Singleton
{
    private static ?Singleton $instance = NULL;

    private function __construct()
    {
```

```

        // -- Do the resource-expensive work here --
    }

    public static function getInstance(): Singleton
    {
        ❶ if (self::$instance == NULL)
        {
            ❷ self::$instance = new Singleton();
        }

        return self::$instance;
    }
}

```

Listing 25-9: The *Singleton* class

We declare a private static property called `instance`, initializing it to `NULL`. Ultimately, this property will hold a reference to the only object of the singleton class. Then we declare a private constructor method where any resource-hungry work can be completed. Since the constructor has been declared as private, it can't be invoked with the `new` keyword from anywhere outside the class itself.

Next, we declare the only public member of this class, the static `getInstance()` method. This method first tests whether the `instance` is `NULL` ❶. If so, this must be the first time that this method has been invoked, so a new `Singleton` object is created (which calls the constructor, triggering the resource-heavy work), and the reference to the new object is stored in the static `instance` property ❷. Then the method returns the object reference in `instance`, making the object available for use elsewhere in the application.

Let's modify the previous section's project to save computing resources by using the singleton pattern. This way, we'll have to create the `Logger` object and log handler only once, no matter how many messages we log. We'll also make our application more flexible by making our custom `Logger` class a subclass of Monolog's `Logger` class so that we can use any of the latter's methods and optional arguments (for example, to provide

context data and to log at different severity levels through the inherited methods).

First, update the declaration of the `Logger` class in *src/Logger.php* to match [Listing 25-10](#). This redesigned class is closely modeled on the skeletal `Singleton` class demonstrated in [Listing 25-9](#).

```
<?php
namespace Mattsmithdev;

use Monolog\Logger as MonologLogger;
use Monolog\Handler\StreamHandler;

❶ class Logger extends MonologLogger
{
    const PATH_TO_LOG_FILE = __DIR__ . '/../logs/debug.log';

    private static ?Logger $instance = NULL;

    private function __construct()
    {
        parent::__construct('channell1');
        $this->pushHandler(new StreamHandler(self::PATH_TO_L
OG_FILE));
    }

    ❷ public static function getInstance(): Logger
    {
        if (self::$instance == NULL)
        {
            self::$instance = new Logger();
        }

        return self::$instance;
    }
}
```

Listing 25-10: Implementing the `Logger` class with the singleton pattern

We declare our `Logger` class as a subclass of the `Monolog Logger` class (aliased as `MonoLogger`) ❶. Since we'll be creating one instance of this class, we no longer declare it to be `abstract`. Next, we initialize the private static `instance` property to `NULL` and declare a private constructor. The constructor uses `parent::` to call the `Monolog Logger` class's constructor, creating a new object with `channel1`, then assigns it a log handler to the debug logfile. Since all this is encapsulated within the private constructor, it will happen only once.

We also declare the public static `getInstance()` method, typical for singleton classes ❷. The method follows the logic described in [Listing 25-9](#), creating and returning a `Logger` object if `instance` is `NULL`, or simply returning the instance if it's been created already.

Now let's update the `run()` method of our `Application` class. We'll obtain a reference to the single instance of our `Logger` class, then use that object reference to log some entries. Modify `src/Application.php` to match [Listing 25-11](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
    {
        print 'Hello, world!';
        Logger::getInstance()->warning('I am a warning. ');
        Logger::getInstance()->error('I am a test error!',
            ['exception' => new \Exception('example of excep
tion object')]);
    }
}
```

Listing 25-11: The `Application` class updated to use our singleton `Logger` class

We get a reference to the singleton instance of `Logger` by writing `Logger::getInstance()` and use it to log a warning-severity message via the `warning()` method inherited from the `Monolog Logger` class. Since this

is the first attempt to get the singleton instance, the `instance` property of our `Logger` class will be `NULL`, and a new `Logger` object will be created and a reference to it saved in `instance`.

Then we again get a reference to the singleton instance of `Logger` and log an error-severity message, creating and passing a second parameter of a new `Exception` object as the context. This time, the `instance` property of `Logger` isn't `NULL`, so the reference to the existing object is returned. In this way, both logs have been created by a single instance of the `Logger` class, saving time and resources while still making the logging functionality available anywhere in the application.

Use the `cat` or `type` command to view the contents of `logs/debug.log` after running the index script. You should see something like this:

```
$ cat debug.log
[2025-01-30T14:37:32.728758 + 00:00]
channell1.WARNING: I am a warning. [] []

[2025-01-30T14:37:32.730002 + 00:00]
channell1.ERROR: I am a test error! {"exception":"[object]
(Exception(code: 0): example of exception object
    at /Users/matt/src/Application.php:13)"} []
```

Both messages were successfully appended to the logfile, including the `Exception` object context data added to the error-level log entry.

You've seen how the singleton pattern can be useful. However, many programmers consider it an *antipattern*, a solution to a common problem that ends up being worse than the problem it's attempting to solve. Critics of the singleton pattern object to it being a form of global program state, which makes testing code more difficult since code using the singleton can't be tested separately from the singleton itself. Also, the singleton has *global visibility*, meaning that any code in an application might be dependent on it. More complex code analysis is therefore needed to determine which parts of an application are or aren't dependent on the globally visible singleton. On the other hand, proponents argue that the reduction in resource usage is enough to outweigh some overhead in testing or the design of other parts of the application.

Enumerations

To *enumerate* is to list out every possibility, one by one, and in computer programming, an *enumeration* is a list of all the possible values for a data type. A good example of an enumerated data type is `bool`, which lists its only two possible values as `true` and `false`. Booleans are built into the language, but since version 8.1, PHP allows you to create and use your own custom enumerations, also known as *enum classes*.

Enum classes are useful when you have data items that can take only one of a closed set of values. For example, a pizza order can be designated as being for delivery or pickup, an employee's work status can be full-time or part-time, and a playing card's suit can be hearts, diamonds, clubs, or spades. While you could use a Boolean property when you have only two potential values (for example, `$fullTime = false`), enum classes are more appropriate when you have more than two possibilities. Even for just two possible values, defining them through an enum class makes the choice more explicit.

Inside an enum class, you declare each of the possible cases for that class, where each case is an object of the class. For example, you might have a playing card `Suit` enum class with possible cases `HEARTS`, `DIAMONDS`, `CLUBS`, and `SPADES`. To reference the cases of an enum class, you use the same double-colon syntax (the scope resolution operator) as when using a static member or class constant (for example, `$card1Suit = Suit::SPADES`).

Because of this shared syntax, enum classes bear some relationship to static members and class constants. The key difference is that each case of an enum class is a proper *object* of that class, whereas class constants and static properties and methods are *members* of a class. Crucially, this means that enum classes can serve as the data types for properties of other classes, and for function parameters and return values.

As a simple example, [Listing 25-12](#) shows the declaration of an enum class for the suits of a deck of playing cards. Declare this enum class in a file called *Suit.php*, just like an ordinary class declaration file.

```
<?php
namespace Mattsmithdev;
```

```
enum Suit
{
    case CLUBS;
    case DIAMONDS;
    case HEARTS;
    case SPADES;
}
```

Listing 25-12: The `Suit` enum class

After the namespace, we use the `enum` keyword followed by the class name `Suit`. Then we declare four cases for this enum class, representing the four playing card suits. Because the enumeration cases serve a similar function as class constants (that is, defining fixed values), I tend to write them in all capital letters, although many programmers capitalize only the first letter of each case.

We can now assign an object of the `Suit` enum class to a variable. To illustrate, [Listing 25-13](#) shows a simple `Card` class that harnesses the `Suit` enum.

```
<?php
namespace Mattsmithdev;

class Card
{
    ❶ private Suit $suit;
    private int $number;

    public function getSuit(): Suit
    {
        return $this->suit;
    }

    public function setSuit(Suit $suit): void
    {
        $this->suit = $suit;
    }
}
```

```

    public function getNumber(): int
    {
        return $this->number;
    }

    public function setNumber(int $number): void
    {
        $this->number = $number;
    }

    ❷ public function __toString(): string
    {
        return "CARD: the " . $this->number . " of " . $this
->suit->name;
    }
}

```

Listing 25-13: The *Card* class with a *suit* property that uses the *Suit* enum class

We first declare *suit* as a property of the *Suit* data type ❶. Again, since enums are considered classes, they can be used as valid data types. We also give the class a *number* property, and we provide simple getters and setters for both *suit* and *number*. Then we declare a `__toString()` method to output the details of the *Card* object's properties ❷. In it, we get a string version of the enum object's name from its *name* property, accessed with the expression `$this->suit->name`.

Listing 25-14 shows a simple index script to demonstrate the use of the *Suit* and *Card* classes.

```

<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Suit;
use Mattsmithdev\Card;

$card1 = new Card();
❶ $card1->setSuit(Suit::SPADES);

```

```
$card1->setNumber(1);  
  
print $card1;
```

Listing 25-14: Using `Card` and `Suit` in an index script

After providing `use` statements for the two classes, we create a new `Card` object and set its `suit` property to be a reference to the `SPADES` case of the `Suit` enum class, using the double-colon `Suit::SPADES` syntax to reference the case ❶. We also set the `number` of this `Card` object to 1 (representing an ace). Then we print the details of the `$card1` variable, which will invoke its `__toString()` method. Here's the output of running this index script:

```
$ php public/index.php  
CARD: the 1 of SPADES
```

The output shows the `number` property of 1 and the string `SPADES`, the name of the `Suit` enum class referenced in the object's `suit` property.

Backed Enums

Besides giving enum cases names like `SPADES` and `HEARTS`, you can also associate an integer or string value with each case. When you give the cases values, the class is called a *value-backed enum*, or *backed enum* for short. The type of `int` or `string` must be declared after the enum class name, and every case must be assigned a value; otherwise, an error will occur.

Let's turn `Suit` into a backed enum by assigning a string value for each case. We could choose strings that exactly match the case names (`'HEARTS'`, `'CLUBS'`, and so on), or we could have a bit of fun and assign strings with the corresponding card suit symbols. See [Listing 25-15](#) for the updated `Suit` declaration.

```
<?php  
namespace Mattsmithdev;  
  
enum Suit: string
```



```
{
    case CLUBS = '♣';
    case DIAMONDS = '♦';
    case HEARTS = '♥';
    case SPADES = '♠';
}
```

Listing 25-15: Turning `Suit` into a backed enum class

We declare `Suit` as a string-backed enum by adding a colon and the `string` type after the class name. Then we assign the string '♣' as the value for the `CLUBS` case, as well as the corresponding symbols for the other three suits. To retrieve the value of a backed enum, use its public `value` property. For example, if the `$card1Suit` variable were a reference to a `Suit` enum object, we'd get its string value with the expression `$card1Suit->value`. These values are read-only; once they're set in the enum class declaration, they can't be changed from elsewhere in the code.

An Array of All Cases

All enum classes have a built-in static method called `cases()` that returns an array of all the case objects for the enumeration. We can use this to build an array of the values associated with the cases. For example, [Listing 25-16](#) shows an index script that does just this by looping through each `Suit` case object and appending its string value to an array for printout.

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';
use Mattsmithdev\Suit;

$cases = Suit::cases();
$caseStrings = [];
foreach ($cases as $case) {
    $caseStrings[] = $case->value;
}
print implode($caseStrings);
```

Listing 25-16: Creating an array of all `Suit` enum case values

We call the `Suit::cases()` static method to obtain a `$cases` array holding an instance of each possible `Suit` case. Then we initialize `$caseStrings` as an empty array. Next, we loop through the `Suit` case objects in `$cases` and append their string values to the `$caseStrings` array. Finally, we print out the array of strings as a single string by using the built-in `implode()` function. Here's the output of running this index script at the command line:

```
$ php public/index.php
♣♦♥♠
```

All four suit characters have been printed out on the same line. Here we've simply printed out the case values, but we could use a similar looping technique to, for example, create a drop-down menu of all the case values from an enum class so that a user can select one of the options.

Summary

In this chapter, we explored static members of classes: properties and methods that aren't related directly to individual objects but belong to the class as a whole. You saw how static members can be accessed within the class via `self::` and outside the class via `ClassName::` (assuming the static member is public). You learned about uses of static members, including to record class-wide information, provide common methods through utility classes, and share functionality across an application, through either an abstract class or the singleton design pattern. We also discussed the related concepts of class constants and enumeration classes.

Exercises

1. Create a new project to implement a `Car` class representing various cars. As shown in [Figure 25-4](#), this class should have instance properties for `make`, `model`, and `price`, and a constructor taking in values for each instance property when a new object is created. You should also have private static properties for `totalPrice` and `numInstances`.

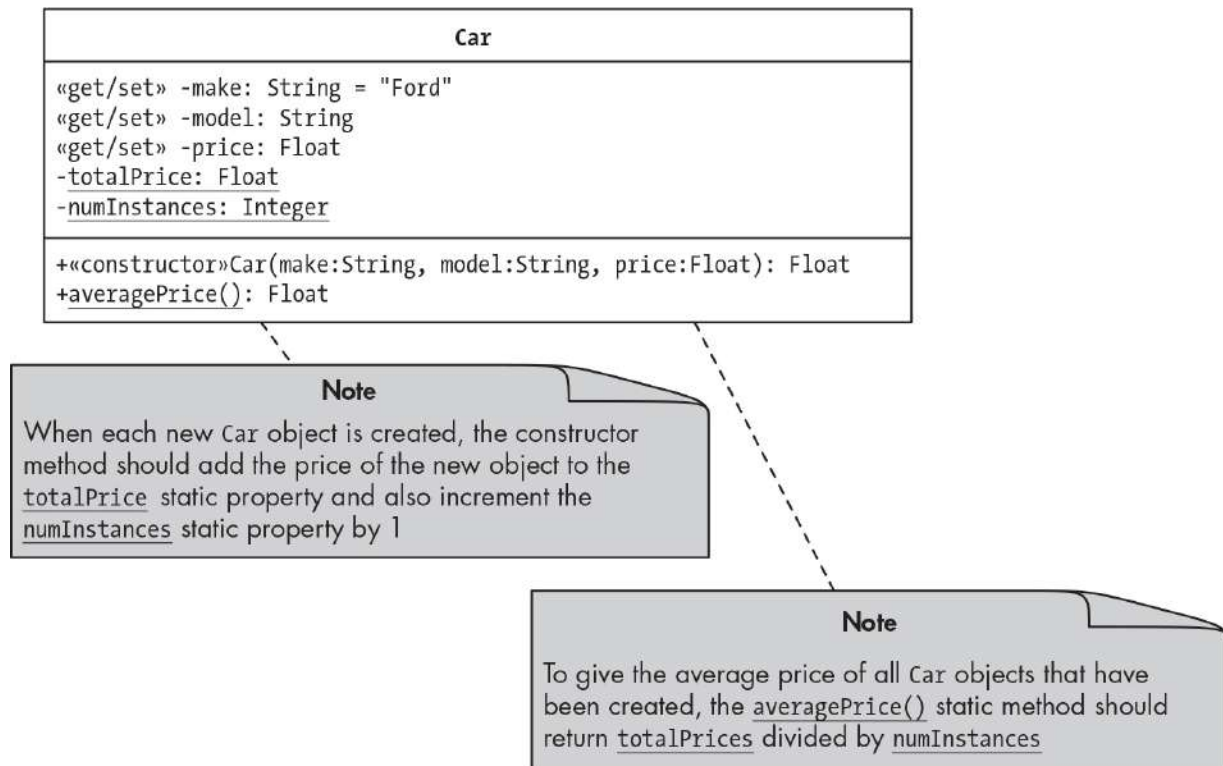


Figure 25-4: The *Car* class

Each time a new *Car* object is created, the constructor method should increment `numInstances` and add the price of the new *Car* object to `totalPrice`.

2. Add a public static `averagePrice()` method to the *Car* class from Exercise 1 that uses the static `numInstances` and `totalPrice` properties to calculate and return the average price of all *Car* objects that have been created.
3. Create a new project featuring an enum class called `DietType` with three cases: `VEGAN`, `VEGETARIAN`, and `CARNIVORE`. Also create a *Dessert* class that has a `name` property (a string) and a `diet` property (a `DietType` enum case), as well as a `__toString()` method that summarizes the *Dessert* object in the form `'(DESSERT) Dessert Name (DietType dish)'`. Write an index script that creates a *Dessert* object and prints its details. The output should be something like this:

```
(DESSERT) Eton Mess (VEGETARIAN dish)
```

26

ABSTRACT METHODS, INTERFACES, AND TRAITS



In this chapter, we'll move beyond the standard mechanism of inheritance from a superclass to a subclass and explore other strategies for sharing methods among classes. You'll be introduced to abstract methods, interfaces, and traits.

As you'll see, abstract methods and interfaces allow you to share just the signatures of methods across multiple classes, without specifying the details of how the methods should be implemented. In effect, these mechanisms act as *contracts*: to use abstract methods or interfaces, a class must agree to provide suitable implementations of those methods. Meanwhile, traits are a way to bypass inheritance and share fully implemented methods among classes in separate hierarchies. Interfaces, too, transcend class hierarchies, while abstract methods are still passed along between superclasses and subclasses through inheritance.

Together, abstract methods, interfaces, and traits can facilitate updating an application without breaking any code, since they provide assurances that certain methods will be present on a class for use by other parts of the

application. Abstract methods and interfaces, in particular, promote class interchangeability. By enforcing method signatures while remaining agnostic about the implementations, they make it easy to substitute in classes that realize those methods differently as new project requirements arise (for example, having new types of files to write to, new database management systems to communicate with, or new destinations for logging events and exceptions). Traits, on the other hand, are helpful for avoiding redundancy and promoting code reusability, since they save you from having to declare the same method on several unrelated classes. In this sense, they're somewhat similar to utility classes that are designed to make certain common operations available to all the classes in an application.

Of the topics covered in this chapter, interfaces especially are quite common in medium-to-large PHP projects. Even if you don't write many interfaces yourself, you'll probably use them, since they're a feature of many third-party libraries for core web application components, including database communication and working with HTTP requests and responses.

From Inheritance to Interfaces

In this section, we'll progressively develop a sample network of classes to illustrate the features and merits of abstract methods and interfaces. We'll start by reviewing the conventional process of a subclass inheriting methods from a superclass, then transition to using abstract methods and finally interfaces to standardize the features of unrelated classes. For simplicity, this will be a toy example. Once we've established the basics, however, we'll turn to a more realistic and practical application for interfaces.

Inheriting a Fully Implemented Method from a Superclass

As we discussed in [Chapter 19](#), inheritance makes it possible to pass down the definition of a superclass method to a related group of subclasses. If some of the subclasses need to implement the method differently, they can always override it with their own implementation, while other subclasses will simply inherit the default implementation from the superclass.

Sometimes the superclass may be abstract, meaning it will never be instantiated. In this case, one or more non-abstract subclasses must extend

the abstract superclass in order for objects to be created and the superclass's methods to be executed. As an example, [Figure 26-1](#) shows a simple class hierarchy of various animals, all of which will be able to return a string describing the type of sound they make.

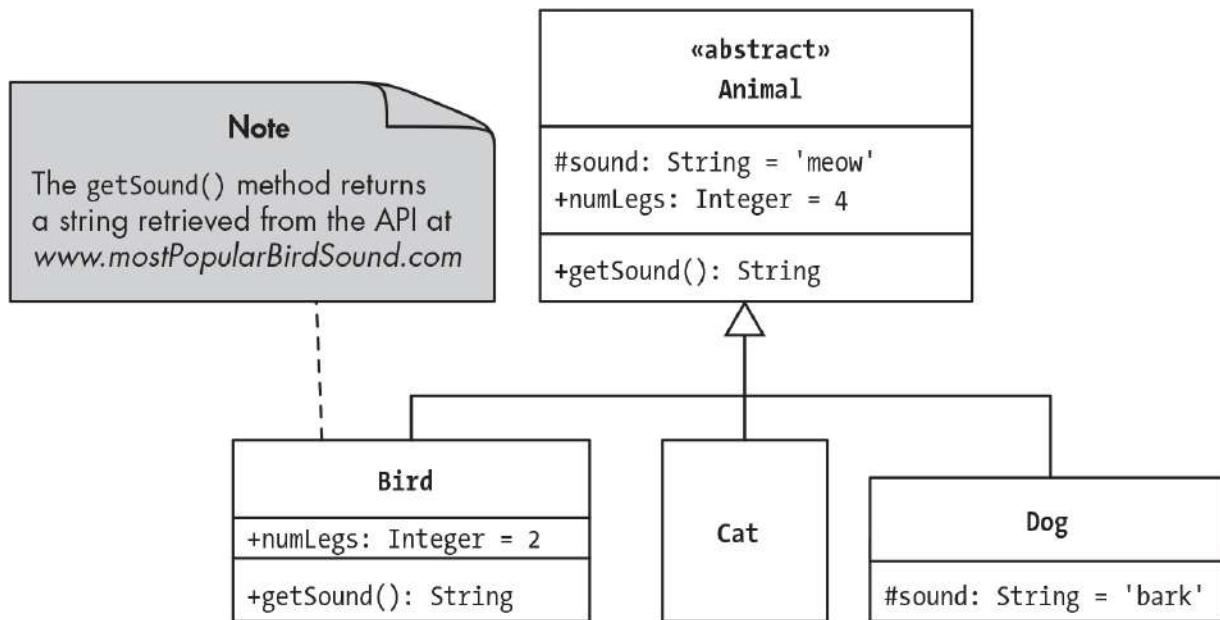


Figure 26-1: A class hierarchy with an abstract superclass passing down a `getSound()` method

At the top of this class hierarchy is the abstract `Animal` superclass. Its `sound` property is declared with a default value of `'meow'` and protected (`#`) visibility, meaning the property's value can be accessed (and overridden) by a subclass if required. Its `numLegs` property is public (`+`) and has a default value of 4. In addition, a `getSound()` method returns the string stored in `sound`.

At the next layer of the hierarchy, the `Cat` subclass extends `Animal` and so inherits the `sound` and `numLegs` properties and the `getSound()` method. This means `Cat` objects will produce a `'meow'` sound. The `Dog` subclass also extends `Animal` but declares its own `sound` property of `'bark'`, overriding the value inherited from the superclass.

Finally, let's imagine that opinions vary about the sound a bird makes; sometimes `'tweet tweet'` is more popular, and sometimes `'chirp chirp'`.

To account for this, the `Bird` subclass declares a custom implementation of `getSound()`, overriding the inherited method from the superclass. At runtime, each `Bird` object's `getSound()` method will access a (fictional) API at www.mostPopularBirdSound.com to determine the most popular bird sound string, ignoring the value of its inherited `sound` property. In addition, since birds have only two legs, the `Bird` class overrides the inherited number of legs.

Listing 26-1 shows the code for the `Animal` class.

```
<?php
namespace Mattsmithdev;

abstract class Animal
{
    protected string $sound = "meow";
    public int $numLegs = 4;

    ❶ public function getSound(): string
    {
        return $this->sound;
    }
}
```

Listing 26-1: The `Animal` class

We declare the `Animal` class to be `abstract` so that it can never be instantiated and assign it `sound` and `numLegs` properties. We also provide an implementation of the `getSound()` method ❶, which returns the value of the `sound` property. For any instance of a subclass inheriting this method, the value of the object's `sound` property will be determined at runtime when the `getSound()` method is invoked. For example, a `Cat` object will return 'meow', a `Dog` object will return 'bark', and a `Bird` object will override this method and instead return whatever string is retrieved from the www.mostPopularBirdSound.com API.

The takeaway from this example is that a superclass (whether abstract or not) provides a way to offer a default method implementation that gets

inherited by the subclasses in its class hierarchy. When required, however, this implementation can be overridden by individual subclasses.

Inheriting an Abstract Method

An *abstract method* is a method on a superclass that doesn't have an implementation. Instead, all that's declared is the method's signature: its name, its parameters, and its return type. Any subclasses that inherit from the superclass must provide their own implementation of the abstract method. The exact details of how the method is implemented are left up to each subclass, as long as the implementation matches the method signature specified on the superclass.

Abstract methods can come into play when very different classes should exhibit the same behavior. For example, cars, like animals, make sound, and they should likely have a `getSound()` method that returns a string, just like our `Animal` class does. However, cars are otherwise very different from animals, and even the way they make sounds is pretty different; cars might output a sound such as `'putt-putt-putt'`, `'purr'`, or `'vroom-vroom'`, depending on their engine size, fuel type, and so on. As such, the `getSound()` method for cars will be different from that for animals, and yet they'll still have the same signature, since in both cases the method is ultimately returning a string.

Let's address this scenario by introducing a new abstract `SoundMaker` superclass that declares an abstract `getSound()` method. Any subclasses inheriting from `SoundMaker`, such as `Animal` and `Car`, will have to provide an appropriate `getSound()` implementation. [Figure 26-2](#) shows the new, modified class hierarchy.

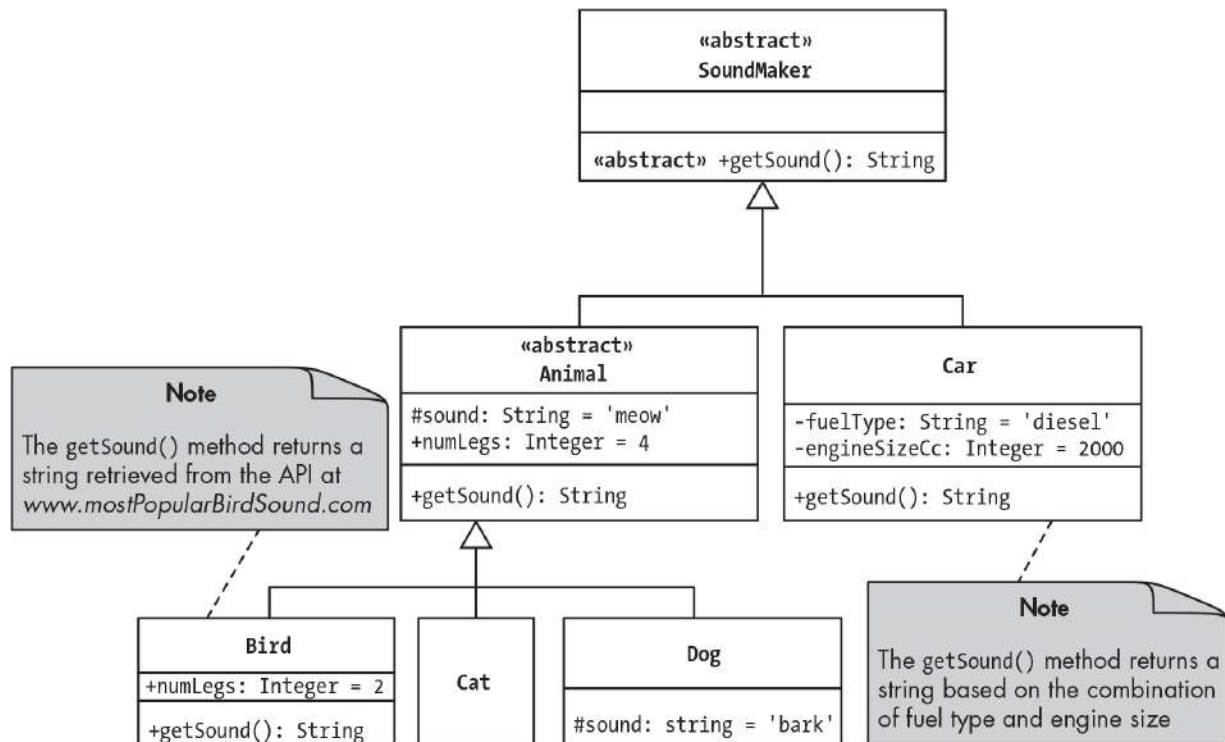


Figure 26-2: Sharing an abstract `getSound()` method through the `SoundMaker` superclass

Listing 26-2 shows the declaration of the new `SoundMaker` class. Notice that the class provides no implementation for the `getSound()` method, just its signature.

```

<?php
namespace Mattsmithdev;

abstract class SoundMaker
{
    abstract public function getSound(): string;
}
  
```

Listing 26-2: The `SoundMaker` class with an abstract method

We designate the `getSound()` method `abstract` and declare just its signature. The method declaration is considered a statement, since no implementation is provided, and so it must end with a semicolon.

The non-abstract `Car` subclass must now provide an implementation for the `getSound()` method in order to successfully inherit from `SoundMaker`. If it didn't, we'd get a fatal error like the following:

```
PHP Fatal error:  Class Mattsmithdev\Car contains 1 abstract
method and must
therefore be declared abstract or implement the remaining me
thods
```

We've already provided a `getSound()` implementation on the `Animal` class, so it can successfully inherit from `SoundMaker`. The `Cat` and `Dog` subclasses inherit the method implementation from `Animal`, so they also meet the requirements of the `SoundMaker` class. The `Bird` class can still override the `getSound()` implementation inherited from `Animal` with its own implementation.

To see the benefit of making `getSound()` an abstract method, say our application has a function that needs to know the sound an object makes. That function can require a `SoundMaker` object (or one of its subclasses) as an argument, and know that whatever subclass of `SoundMaker` is received will have a `getSound()` method that can be invoked to return a string. It doesn't matter whether it's an `Animal` object or a `Car` object; the method is guaranteed to be there. In this way, abstract methods maximize class interchangeability while still allowing for different classes to have quite distinct implementations of a method.

If a class declares one or more abstract methods, the class itself must also be abstract. This is because you can't instantiate a class with an abstract method, since no implementation of the method is provided. The opposite isn't necessarily true, however: a class may be declared abstract but not contain any abstract methods. For example, you might have an abstract class consisting of fully implemented, static members.

Requiring Method Implementations with Interfaces

An *interface* is a way to declare the signatures of one or more methods that a class should have. Classes then *implement* the interface by declaring methods with those specified signatures. Interfaces are similar to abstract

methods in that both are ways of ensuring that one or more classes should have certain methods, without specifying exactly how those methods should be implemented. Both promote class interchangeability by guaranteeing the consistency of those methods' signatures. The difference is that interfaces *aren't* classes, and therefore are independent of any class hierarchy scheme, whereas abstract methods are declared as part of a class. As such, any classes that implement the abstract methods must fall within the hierarchy of the class that declares them.

Being independent of class hierarchies, interfaces are useful when you want to share a behavior among very different classes that wouldn't belong in the same class hierarchy, or when you want to share multiple behaviors, in various combinations, among several very different classes.

Continuing with the example from the previous sections, pipe organs also make sounds, like cars and animals. Both pipe organs and cars require regular maintenance as well, whereas animals don't. Let's consider that subclasses of a `Maintainable` class must implement a `nextService()` method that returns some kind of `Date` object. The way a service date is calculated will be implemented differently for `Car` objects and `PipeOrgan` objects. `Car` service dates may be based on the type of engine and number of miles driven, while `PipeOrgan` objects may have service dates calculated based on, say, the length and material of the pipes.

We might be tempted to create a new abstract `Maintainable` class that declares an abstract `nextService()` method. The `Car` and `PipeOrgan` classes would inherit from `Maintainable` and provide their own `nextService()` implementations, while also inheriting from the `SoundMaker` class, along with `Animal`. This would be an example of *multiple inheritance*, the capability of a class to inherit from two or more superclasses at the same time. The class diagram in [Figure 26-3](#) illustrates this scheme.

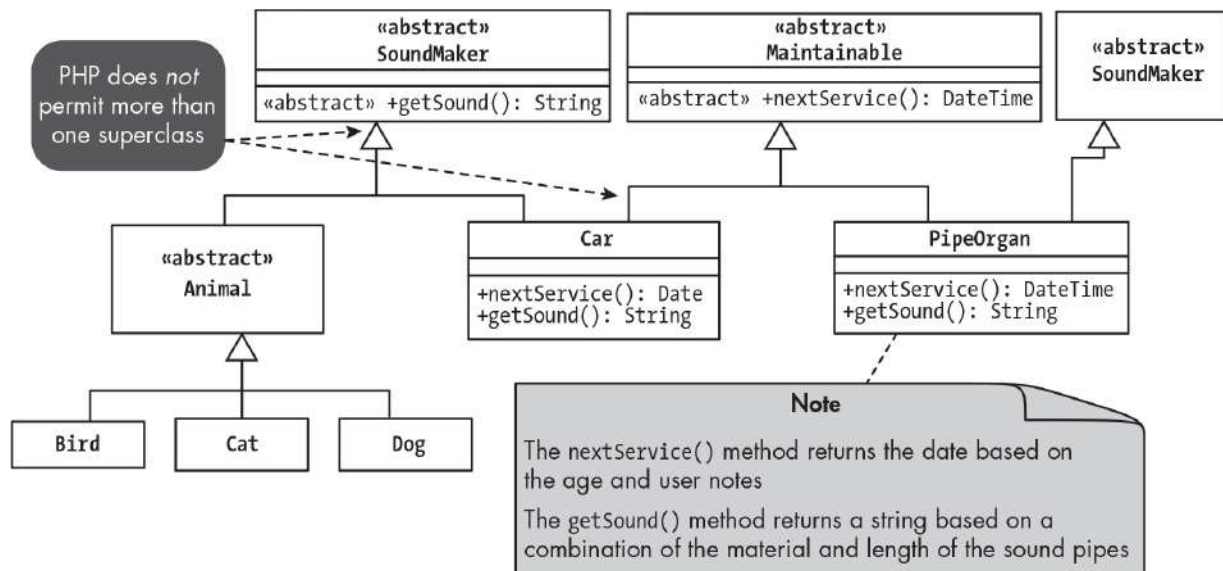


Figure 26-3: A class hierarchy with multiple inheritance

This arrangement may seem appealing: the `Car` and `PipeOrgan` classes inherit from two superclasses, thereby receiving the requirement for the `getSound()` method from the abstract `SoundMaker` class as before while also receiving the requirement for the `nextService()` method from the abstract `Maintainable` class. However, although some computer languages allow multiple inheritance, PHP does not, to avoid problems of ambiguity. If a class inherits from more than one superclass, and two or more of those superclasses declare a constant or method of the same name, how does the inheriting class know which to use?

We could try to get around the prohibition against multiple inheritance by placing the `SoundMaker` and `Maintainable` superclasses at different levels of the same class hierarchy. That is, we could make `Maintainable` a subclass of `SoundMaker`, and `Car` and `PipeOrgan` subclasses of `Maintainable`, as in [Figure 26-4](#).

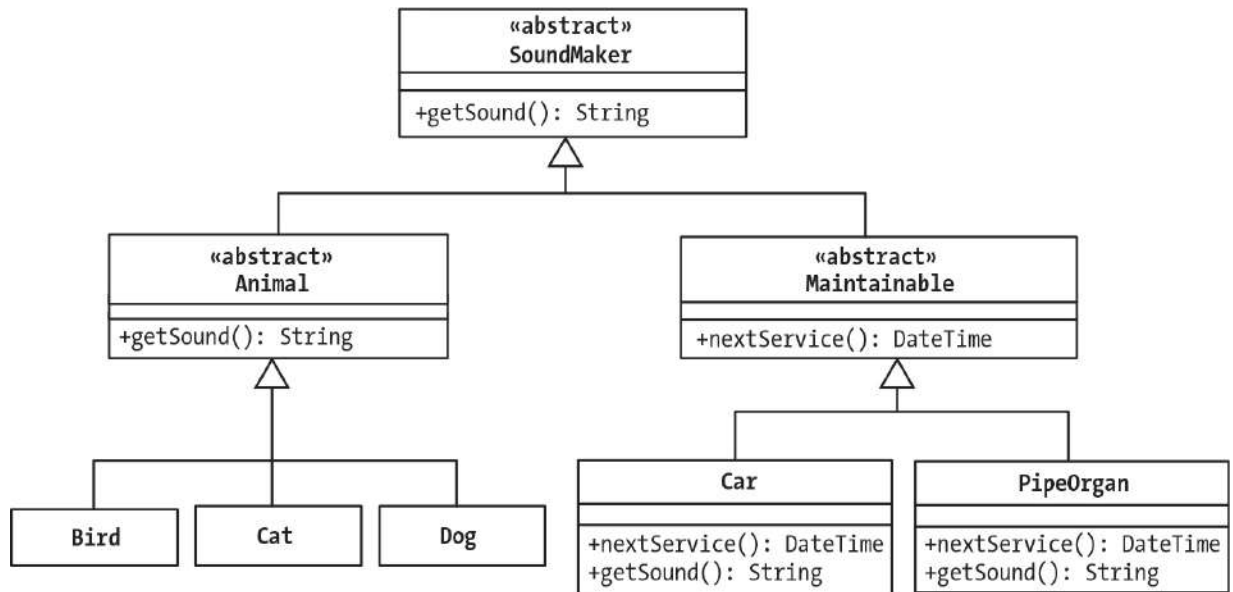


Figure 26-4: A class hierarchy where *Maintainable* is a subclass of *SoundMaker*

At first glance, this seems to work. Objects of any class in the hierarchy must have a `getSound()` method, and `Car` and `PipeOrgan` each must implement a `nextService()` method as well. However, what if we identify further behaviors that some of these classes should have but not others? Those behaviors may not make sense anywhere in the proposed hierarchy. Also, what if we want to add a `Maintainable` subclass that doesn't make sound? Chimneys, for example, require regular maintenance but are silent.

Clearly, we've created a fragile and artificial class hierarchy. Completely unrelated classes such as birds, cars, and chimneys may be forced into being subclasses of classes they have nothing to do with, all to enforce the inheritance of the `getSound()` and `nextService()` method signatures. The solution is to use an interface to define a set of required method signatures that can be implemented by classes that aren't all in the same hierarchy. This dodges the illegal solution of multiple inheritance while also skirting the requirement for a single class hierarchy.

To demonstrate, let's first define `SoundMaker` as an interface rather than a class. Then we can stipulate that the classes in our example should all implement the `SoundMaker` interface. This is illustrated in [Figure 26-5](#).

At the lower left of the diagram, the `SoundMaker` interface declares the signature of the `getSound()` method. With `SoundMaker` reframed as an interface rather than a class, we're free to break our classes into separate, more meaningful, and robust hierarchies: we have the abstract `Animal` class and its subclasses, and the abstract `Vehicle` class with subclasses `Car` and `Helicopter`. The `PipeOrgan` class, which has little to do with animals or vehicles, is off by itself. The classes that implement the `SoundMaker` interface are annotated with the interface name and *lollipop notation*.

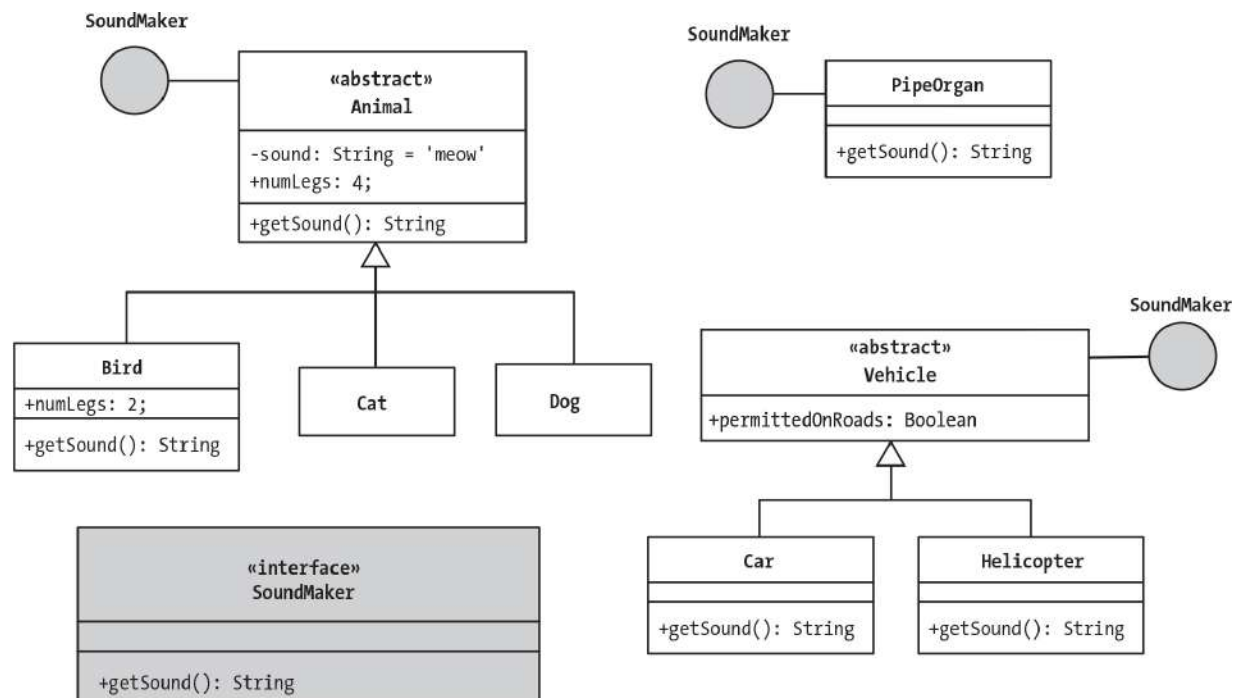


Figure 26-5: The `SoundMaker` interface, implemented by multiple classes

Notice that when an abstract class such as `Animal` or `Vehicle` implements an interface, it doesn't have to provide full implementations of all (or any) of the methods declared on the interface. The implementation can be left to the non-abstract subclasses instead. Here `Animal` provides a `getSound()` implementation (although it's overridden by the `Bird` subclass), while `Vehicle` doesn't. In the latter case, the `Car` and `Helicopter` subclasses must each provide their own custom `getSound()` implementation

in order to fulfill the promise the `Vehicle` class is making by declaring that it will implement the `SoundMaker` interface.

Declaring an Interface

The code declaring an interface goes in its own *.php* file with the same name as the interface, much like a class declaration. For example, the `SoundMaker` interface should be declared in a *SoundMaker.php* file. [Listing 26-3](#) shows its code.

```
<?php
namespace Mattsmithdev;

interface SoundMaker
{
    public function getSound(): string;
}
```

Listing 26-3: The `SoundMaker` interface

We declare `SoundMaker` by using the `interface` keyword. Its body contains just the signature for the `getSound()` method, without an actual implementation. Just as when declaring an abstract method, the `getSound()` signature must end with a semicolon to indicate the end of the statement. Notice that we give `getSound()` public visibility. Including the `public` modifier explicitly is considered best practice, though it isn't strictly necessary since all methods declared on an interface are automatically considered public so that other parts of the system can harness the behaviors of any interface-implementing classes.

NOTE

In addition to declaring method signatures, interfaces can also declare constants. A class that implements the interface will inherit the interface constant, although as of PHP 8.1, the class can override the interface constant if needed.

Implementing an Interface

Now let's look at how a class can implement an interface. As an example, [Listing 26-4](#) shows the code for the `PipeOrgan` class, which implements the `SoundMaker` interface.

```
<?php
namespace Mattsmithdev;

class PipeOrgan implements SoundMaker
{
    public function getSound(): string
    {
        return 'dum, dum, dum-dum';
    }
}
```

Listing 26-4: Implementing the `SoundMaker` interface with the `PipeOrgan` class

We declare that this class implements the `SoundMaker` interface by using the `implements` keyword. Because the `PipeOrgan` class implements `SoundMaker`, the class is obligated to provide an implementation for the `getSound()` method: in this case, it returns the string `'dum, dum, dum-dum'`. The method matches the signature declared by the `SoundMaker` interface. We would similarly declare `getSound()` methods on the `Animal` and `Vehicle` classes. The details of each implementation don't matter, as long as the method is called `getSound()` and it returns a string.

If a (non-abstract) class doesn't include a definition for a method required by an interface it's implementing, you'd get a fatal error. For example, if the code for the `PipeOrgan` class didn't declare a `getSound()` method, you'd see the following when trying to create an object of this class:

```
PHP Fatal error:  Class Mattsmithdev\PipeOrgan contains 1 ab
stract method and
must therefore be declared abstract or implement the remaini
ng methods
```

Notice that this is exactly the same fatal error as when a subclass fails to implement an abstract method declared in its superclass. The PHP engine processes the method signatures of an interface as if they were abstract methods; they must be realized in the class hierarchy implementing the interface before any object can be created.

Implementing Multiple Interfaces with One Class

A powerful feature of interfaces is that a single class may implement more than one of them. When a class implements an interface, it's promising to provide a set of public methods with the signatures declared in that interface, and there's no reason a class can't do this for multiple interfaces.

Returning to our example, the `PipeOrgan` class can implement a `Maintainable` interface, promising to declare an implementation of `nextService()`, in addition to implementing the `SoundMaker` interface by declaring a `getSound()` method. Likewise, if all vehicles require maintenance and make sounds, the `Vehicle` class can implement both the `Maintainable` and `SoundMaker` interfaces too. [Figure 26-6](#) shows how these classes can implement multiple interfaces.

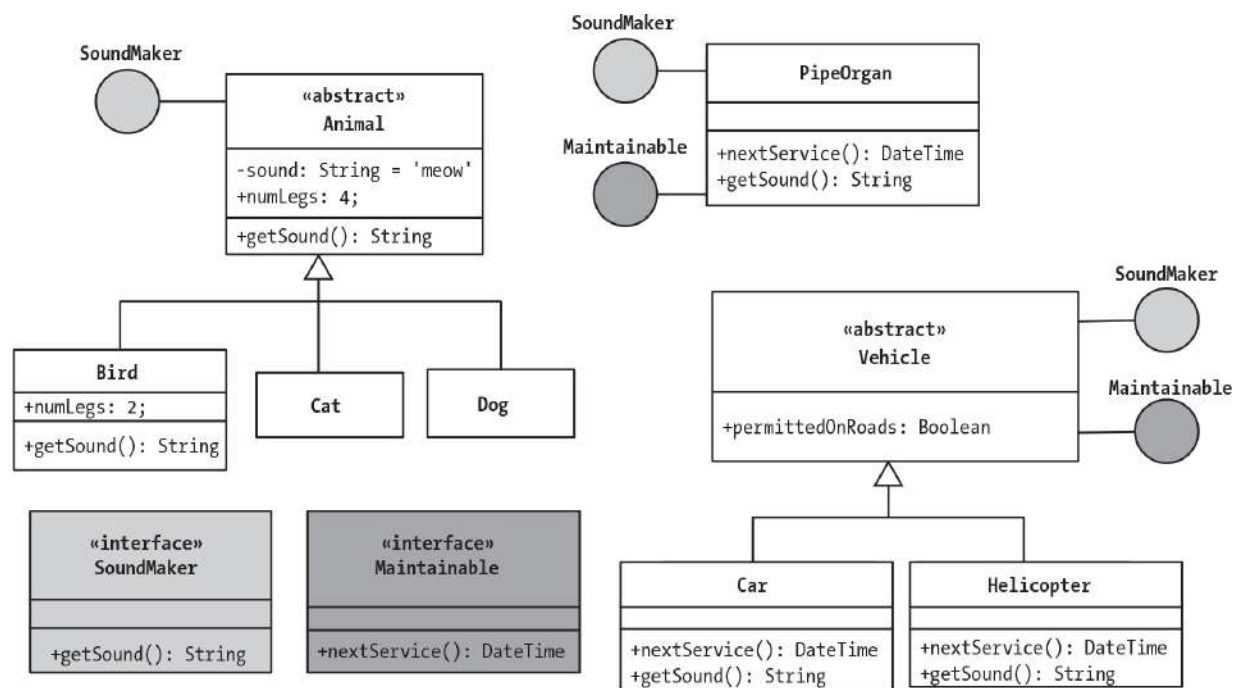


Figure 26-6: Classes implementing multiple interfaces

The figure shows the `Maintainable` interface alongside `SoundMaker`, and the `PipeOrgan` and `Vehicle` classes now have two “lollipops,” indicating that they implement both interfaces. This arrangement of classes and interfaces is much neater conceptually than the artificial class hierarchy shown in [Figure 26-4](#) or the multiple inheritance scheme shown in [Figure 26-3](#).

To see how to declare that a class implements multiple interfaces, refer to [Listing 26-5](#), which shows the updated code for the `PipeOrgan` class.

```
<?php
namespace Mattsmithdev;

class PipeOrgan implements SoundMaker, Maintainable
{
    public function getSound(): string
    {
        return 'dum, dum, dum-dum';
    }

    public function nextService(): \DateTime
    {
        return new \DateTime('2030-01-01');
    }
}
```

Listing 26-5: Implementing multiple interfaces with the `PipeOrgan` class

When a class implements multiple interfaces, you need to use the `implements` keyword only once, followed by the interface names, separated by commas, as in `implements SoundMaker, Maintainable` here. Beyond this, implementing multiple interfaces is as simple as providing definitions for all requisite methods. In this case, we’ve added the `nextService()` method, which returns a `DateTime` object as the `Maintainable` interface requires (we’ll discuss handling dates in [Chapter 31](#)).

I mentioned earlier that an argument against multiple inheritance is the ambiguity of a class attempting to inherit the same member from multiple superclasses. This isn’t a problem for a class implementing multiple

interfaces. Whether one, two, or any number of interfaces declare identical method signatures, all those interface contracts can be met by a single method of that signature implemented in a class. For example, if for some reason the `Maintainable` interface declared both a `nextService()` and a `getSound()` method, the code would still work fine provided the `PipeOrgan` class declares implementations for both methods. As long as all methods coming from interfaces are defined in the classes that implement them, no ambiguity exists and the PHP engine will work consistently, correctly, and error-free.

Comparing Interfaces and Abstract Classes

At first glance, it may seem that interfaces are the same as abstract classes, as neither can be used to instantiate objects. However, while the concepts are related, key differences exist, and each is appropriate for different situations. Above all, an abstract class is a class, while an interface is not; it's a promise, or contract, of method signatures that a class must implement. Another key difference is that a class can inherit from only one abstract class, whereas a class can implement multiple interfaces.

Interfaces can't declare or work with instance-level members, so interfaces can't have instance properties or implement methods that work with instance members. Indeed, interfaces can't implement methods at all; they only specify requirements for instance methods. An abstract class, meanwhile, can be a fully implemented class, or it can be a partially completed class including instance variables, a constructor, and a combination of implemented instance methods and unimplemented abstract methods. In the latter case, a class extending an abstract class has to complete the implementation only by fleshing out the inherited abstract methods.

Interfaces and abstract classes also differ in terms of method visibility. Methods declared on interfaces must be public, whereas abstract classes have the option to declare protected methods that are available only for internal use by objects within the class hierarchy. Also, while declaring the signature of a constructor method on an interface is technically possible, it's highly discouraged; but it's perfectly fine for an abstract class to have a constructor. Finally, it should be noted that an interface can extend another

interface, much like a subclass extending a superclass. Unlike with class inheritance, however, an interface can extend multiple interfaces.

Real-World Applications of Interfaces

Our `SoundMaker` and `Maintainable` scenario may have been a trivial example, but interfaces have significant real-world applications as well. They're particularly useful for standardizing the method signatures of classes whose behavior may change as a web application evolves. Declaring the method signatures as an interface ensures that the application will still work; even if the details of the method implementations change, the way to call the methods won't, so the rest of the application code will be unaffected.

We already used a practical, real-world interface in [Chapter 24](#) when we discussed logging. The PSR-3 standard specifies a `Logger` interface, outlining several methods that any classes implementing the interface must provide, such as `log()`, `error()`, and so on. You can work with any class that implements this interface and feel confident that these methods will be present. In [Chapter 24](#), for example, we used the Monolog library's `Logger` class, which implements the `Logger` interface, but classes from other third-party libraries implement it too. Any of these classes would work, and you could even switch between `Logger` implementations without having to change the code that uses the logging object provided.

Another functionality that interfaces can help with is the ability to temporarily cache (store) data, such as when processing form submissions or HTTP requests in a web application. Caching the data helps avoid having to pass lots of arguments between controller objects and methods; you can simply store the data to the cache in one part of the code and then retrieve it from the cache in another part.

Caching has many approaches, such as using browser sessions, a database, JSON or XML files, the PHP Extension Community Library (PECL) language extension, or perhaps an API to connect to another service. If you declare an interface for common caching operations, you can write code that will be compatible with any interface-compliant caching system. Then you can easily switch caching systems as the project requirements change. For example, you might use one caching system when

developing a project and a different caching system for the live production website.

We'll explore the approaches to caching in this section and illustrate how to standardize them through a caching interface. We'll test everything through a web application designed to cache the ID of any incoming HTTP requests and display that ID on an About page.

NOTE

PHP already has the PSR-6 and PSR-16 standards recommendations for caching interfaces, but they're too involved for our purposes. We'll create our own simpler approach to caching to explore the benefits of interfaces through a more straightforward example.

Caching Approach 1: Using an Array

First, let's implement a cache as a class called `CacheStatic` that uses a static (class-level) array for storing and retrieving values under string keys. We might use this simple approach to get the cache working quickly during the early stages of development. Besides getting and setting values, we'll want the class to provide a `has()` method that returns a Boolean indicating whether a value is currently stored for a given key.

Start a new project and give it the usual `composer.json` file declaring `src` as the location for classes in the `Mattsmithdev` namespace. Generate an autoloader with Composer, and create the usual `public/index.php` script that reads in and executes the autoloader, creates an `Application` object, and invokes its `run()` method. Once that's written, you're ready to declare the `CacheStatic` class in `src/CacheStatic.php`, as shown in [Listing 26-6](#).

```
<?php
namespace Mattsmithdev;

class CacheStatic
{
    ❶ private static array $dataItems = [];

    ❷ public static function set(string $key, string $value):
void
```

```

        {
            self::$dataItems[$key] = $value;
        }

        ❸ public static function get(string $key): ?string
        {
            if (self::has($key)) {
                return self::$dataItems[$key];
            }

            return NULL;
        }

        ❹ public static function has(string $key): bool
        {
            return array_key_exists($key, self::$dataItems);
        }
    }
}

```

Listing 26-6: The *CacheStatic* class

We initialize the private static `dataItems` property as an empty array ❶. This will be our cache. Then we declare the `set()` static method, which takes in two string arguments, a key and a value, for storage in the cache ❷. We next declare the `get()` static method, which takes in a string key and returns the value in the cache array stored for that key ❸. The method includes a test that returns `NULL` if no value exists for the given key. Finally, we declare the static `has()` method ❹, which returns `true` or `false` to indicate whether a value is cached for the given key.

Next, we'll declare the `Application` class. Its `run()` method will cache the ID from the HTTP request, then instantiate a `MainController` object (we'll declare this class shortly) to respond to the request. Create the file `src/Application.php` with the code in [Listing 26-7](#).

```

<?php
namespace Mattsmithdev;

class Application

```

```

{
    public function run()
    {
        $action = filter_input(INPUT_GET, 'action');
        ❶ $id = filter_input(INPUT_GET, 'id');
        if (empty($id)) {
            $id = "(no id provided)";
        }

        // Cache ID from URL
        ❷ CacheStatic::set('id', $id);

        $mainController = new MainController();
        ❸ switch ($action) {
            case 'about':
                $mainController->aboutUs();
                break;

            default:
                $mainController->homepage();
        }
    }
}

```

Listing 26-7: The `Application` class

After retrieving the URL-encoded `action` variable as usual, we attempt to retrieve another URL-encoded variable called `id` and store its value in the `$id` variable ❶. If this query-string variable turns out to be empty, we set `$id` to `'(no id provided)'` instead. Then we use the `CacheStatic` class's `set()` static method to store the string in the `$id` variable in the cache with a key of `'id'` ❷. We can now retrieve the stored string with the `CacheStatic` public static method `get('id')` if needed. The `run()` method concludes with a typical `switch` statement that invokes either the `homepage()` or `aboutUs()` method of the `MainController` object, depending on the value of the `action` variable ❸.

Now we'll declare the `MainController` class. Create `src/MainController.php` as shown in [Listing 26-8](#).

```
<?php
namespace Mattsmithdev;

class MainController
{
    public function homepage()
    {
        require_once __DIR__ . '/../templates/homepage.php';
    }

    public function aboutUs()
    {
        ❶ $id = CacheStatic::get('id');
        require_once __DIR__ . '/../templates/aboutUs.php';
    }
}
```

Listing 26-8: *The `MainController` class*

The `homepage()` method simply outputs the Home page template. In the `aboutUs()` method, we use the `CacheStatic` class's `get()` method to retrieve the ID from the cache array, storing the result in the `$id` variable ❶. Then we read in and execute the About page template, which will have access to `$id`.

Listing 26-9 shows the content of the Home page template. Enter this code into *templates/homepage.php*.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>home page</title>
</head>
<body>
<?php
❶ require_once '_nav.php'
?>

<h1>home page</h1>
```



```
<p>
    welcome to home page
</p>
</body>
</html>
```

Listing 26-9: The homepage.php template

This basic HTML Home page template reuses some code by outputting the navigation bar from the partial template file *templates/_nav.php* ❶.

[Listing 26-10](#) shows the content of that partial template.

```
<ul>
    <li>
        <a href="/">
            Home
        </a>
    </li>
    <li>
        <a href="/?action=about">
            About Us
        </a>
    </li>
    <li>
        ❶ <a href="/?action=about&id=<?= rand(1,99) ?>">
            about (with ID in URL)
        </a>
    </li>
</ul>
<hr>
```

Listing 26-10: The _nav.php partial template

The navigation bar starts with two simple links, with the /URL for the home page and */?action=about* for the About page. We also provide an extra, more complex link to the About page ❶, using PHP's `rand()` function to pick an integer from 1 to 99 and pass it as the value of the `id` query-string variable. This value will be cached and then displayed in the content of the About page to confirm that the cache is working.

Listing 26-11 shows the About page template in *templates/aboutUs.php*.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>about page</title>
</head>
<body>
<?php
require_once '_nav.php'
?>

<h1>about page</h1>
<p>
    welcome to about page

    <br>
    ❶ your ID = <?= $id ?>
</p>
</body>
</html>
```

Listing 26-11: The aboutUs.php template

As for the home page, we draw on the partial *_nav.php* template to simplify the file at hand. Then we incorporate the value of the `$id` variable into the body of the page ❶. [Figure 26-7](#) shows the resulting web page.

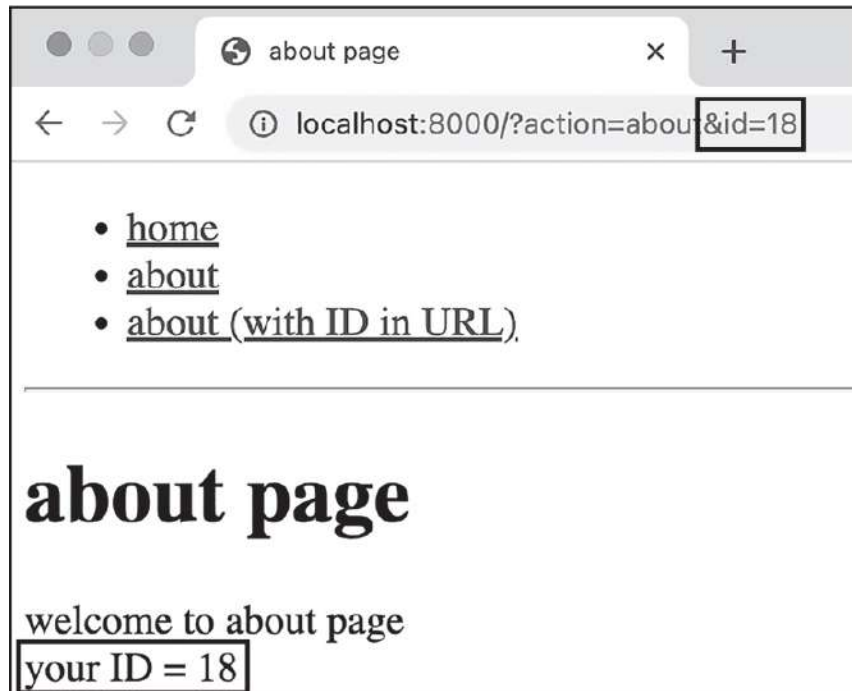


Figure 26-7: The About page, including the cached ID value

Notice that the value of the URL-encoded `id` variable has been printed to the page. This indicates that the ID was successfully cached by the `run()` method in the `Application` class, then retrieved by the `aboutUs()` method of the `MainController` class, and finally printed by the `aboutUs.php` template.

Caching Approach 2: Using a JSON File

Suppose we later decide to add a second caching approach that caches data in a JSON file. This JSON approach would, for example, make it much easier to log different states of the cache at different times to a logging API that accepts JSON data. Let's declare a new caching class named `CacheJson` to implement this other approach. Create `src/CacheJson.php` containing the code in [Listing 26-12](#).

```
<?php
namespace Mattsmithdev;

class CacheJson
{
```

```
private const CACHE_PATH = __DIR__ . '/../var/cache.json';
```

```
❶ public function set(string $key, string $value): void
{
    $dataItems = $this->readJson();
    $dataItems[$key] = $value;
    $this->writeJson($dataItems);
}
```

```
❷ public function get(string $key): ?string
{
    $dataItems = $this->readJson();
    if ($this->has($key)) {
        return $dataItems[$key];
    }

    return NULL;
}
```

```
❸ public function has(string $key): bool
{
    $dataItems = $this->readJson();
    return array_key_exists($key, $dataItems);
}
```

```
private function readJson(): array
{
    $jsonString = file_get_contents(self::CACHE_PATH);
    if (!$jsonString) {
        return [];
    }

    $dataItems = json_decode($jsonString, true);
    return $dataItems;
}
```

```
private function writeJson(array $dataItems): bool
{

```

```

        $jsonString = json_encode($dataItems);
        return file_put_contents(self::CACHE_PATH, $jsonString);
    }
}

```

Listing 26-12: The `CacheJson` class

Within `CacheJson`, we declare public `set()` ❶, `get()` ❷, and `has()` ❸ methods. Outwardly, they're similar to those in our `CacheStatic` class, except they're instance methods, belonging to each object of the class, rather than static methods belonging to the class as a whole. Internally, however, the method definitions are different from those of `CacheStatic`: they read and write information to a JSON file by using the private `readJson()` and `writeJson()` methods, which in turn use the built-in `file_get_contents()` and `file_put_contents()` functions introduced in [Chapter 9](#).

These details are hidden from the rest of the application, though, so the impact of these changes on our code is minimal. For example, [Listing 26-13](#) shows the only changes we need to make to the `Application` class.

```

<?php
namespace Mattsmithdev;

class Application
{
    public function run()
    {
        $action = filter_input(INPUT_GET, 'action');
        $id = filter_input(INPUT_GET, 'id');
        if (empty($id)) {
            $id = "(no id provided)";
        }

        $cache = new CacheJson();
        $cache->set('id', $id);

        --snip--
    }
}

```

```
}  
}
```

Listing 26-13: Updating the *Application* class to use a *CacheJson* object

We replace `CacheStatic::set('id', $id)` with two statements that create a `CacheJson` object and invoke its `set()` method. The `MainController` class requires a similar small tweak, shown in [Listing 26-14](#).

```
<?php  
namespace Mattsmithdev;  
  
class MainController  
{  
    public function homepage()  
    {  
        require_once __DIR__ . '/../templates/homepage.php';  
    }  
  
    public function aboutUs()  
    {  
        ❶ $cache = new CacheJson();  
        $id = $cache->get('id');  
        require_once __DIR__ . '/../templates/aboutUs.php';  
    }  
}
```

Listing 26-14: Updating the *MainController* class to use a *CacheJson* object

Instead of the `$id = CacheStatic::get('id')` statement, we create a `CacheJson` object and invoke its `get()` method to retrieve the value cached under the 'id' key ❶. If you now test the application again, it should work just as it did before. The only difference is that the ID is being cached to a JSON file instead of an array.

Caching Approach 3: Creating a Cacheable Interface

We've already used two methods of caching for our application, and in the future we might want to use still others. This situation lends itself to

abstracting the caching classes' common operations as an interface, then writing classes that implement the interface. This way, as long as our code can create an object of any class that implements the interface, we know we'll be able to use that class's `get()`, `set()`, and `has()` methods without having to worry about which class the caching object is an instance of or how the class is doing the work.

To make this change, we'll first declare a generic `Cacheable` interface. In addition to the `get()`, `set()`, and `has()` methods, we'll also stipulate a fourth method, `reset()`, that completely empties the cache of any stored values. Create `src/Cacheable.php` and enter the contents of [Listing 26-15](#).

```
<?php
namespace Mattsmithdev;

interface Cacheable
{
    public function reset(): void;
    public function set(string $key, string $value): void;
    public function get(string $key): ?string;
    public function has(string $key): bool;
}
```

Listing 26-15: The `Cacheable` interface

We declare the `Cacheable` interface with the signatures for the four methods that any class implementing the interface must have. These methods are all public instance methods, with appropriate typed arguments and return types. For example, `set()` takes in strings for the desired key and value being cached and returns `void`, while `get()` takes in a string key and returns a string or `NULL`.

When we switched from using `CacheStatic` to `CacheJson`, we had to make some updates to the `Application` and `MainController` classes. We'll now refactor those classes so that we can switch implementations of the `Cacheable` interface without having to change anything. We'll start with the `Application` class. [Listing 26-16](#) shows the updates to `src/Application.php`.

```
<?php
namespace Mattsmithdev;

class Application
{
    ❶ private Cacheable $cache;

    ❷ public function __construct(Cacheable $cache)
    {
        $this->cache = $cache;
        $this->cache->reset();
    }

    ❸ public function getCache(): Cacheable
    {
        return $this->cache;
    }

    public function run()
    {
        $action = filter_input(INPUT_GET, 'action');
        $id = filter_input(INPUT_GET, 'id');
        if (empty($id)) {
            $id = "(no id provided)";
        }

        ❹ $this->cache->set('id', $id);

        ❺ $mainController = new MainController($this);
        switch ($action) {
            case 'about':
                $mainController->aboutUs();
                break;

            default:
                $mainController->homepage();
        }
    }
}
```



```
}  
}
```

Listing 26-16: Refactoring the `Application` class to use the `Cacheable` interface

We first add a private `cache` property to the class, whose value is a reference to a `Cacheable` object ❶. This is a powerful feature of interfaces: we can provide an interface name as a data type for a variable, method parameter, or method return value, and any object from any class that implements the interface will work fine.

We next obtain a `Cacheable` object reference for this property as an argument passed to the constructor method ❷. Whatever object is passed as an argument when an `Application` object is created must therefore be of a class that implements the `Cacheable` interface. The constructor invokes the `reset()` method of the provided `Cacheable` object, so we know we'll have an empty cache when we start processing the current HTTP request. Because the `cache` property is private, we declare a public getter method so that it can be accessed outside the `Application` class ❸.

Notice that all these new statements so far have been written in such a way that the `Application` class doesn't need to know which implementation of the `Cacheable` interface is referenced by the argument provided to the class's constructor. You'll see later how the `Cacheable` object is created in the index script, so this is the only place where the code needs to change if we choose to use a different `Cacheable` implementation.

Inside `run()`, we use the `Cacheable` object's expected `set()` method to store the value of the `$id` variable in the cache ❹. Then, when we create a `MainController` object, we provide `$this` as an argument ❺, meaning that the `MainController` object will have a reference back to this `Application` object. By extension, the `MainController` object will also have access to the `Cacheable` object through the `Application` object's `cache` property.

Now let's update the `MainController` class. [Listing 26-17](#) shows the revised `src/MainController.php` file.

```
<?php  
namespace Mattsmithdev;
```

```

class MainController
{
    ❶ private Application $application;

    ❷ public function __construct(Application $application)
    {
        $this->application = $application;
    }

    public function homepage()
    {
        require_once __DIR__ . '/../templates/homepage.php';
    }

    public function aboutUs()
    {
        ❸ $cache = $this->application->getCache();
        $id = $cache->get('id');
        require_once __DIR__ . '/../templates/aboutUs.php';
    }
}

```

Listing 26-17: Refactoring the `MainController` class to use the `Cacheable` interface

We declare a private `application` property ❶ whose value is the reference to the `Application` object passed as an argument to the constructor method ❷. Then, in the `aboutUs()` method, we use the public `getCache()` method of the `application` object to obtain a reference to the `Cacheable` object ❸. This way, we can call the `get()` method as before to retrieve the stored ID from the cache for use within the page template.

Next, we need to update the *public/index.php* script to create a caching object and pass it to the `Application` object when the latter is created. As mentioned earlier, this is the only part of the code that needs to know which implementation of the `Cacheable` interface we want to use. Update the index script as shown in [Listing 26-18](#).

```

<?php
require_once __DIR__ . '/../vendor/autoload.php';

```

```
use Mattsmithdev\Application;
use Mattsmithdev\CacheJson;
use Mattsmithdev\CacheStatic;

$cache1 = new CacheJson();
$cache2 = new CacheStatic();

app = new Application($cache2);
$app->run();
```

Listing 26-18: Choosing a `Cacheable` implementation in `index.php`

We create two objects: `$cache1` is a `CacheJson` object, and `$cache2` is a `CacheStatic` object. Then we pass one of these variables when we construct the `Application` object. Try the code with both variables, and it should work the same way each time.

The final step is to revise our cache classes to implement the `Cacheable` interface. [Listing 26-19](#) shows the updated `CacheStatic` class. To meet the contractual obligations of the `Cacheable` interface, we need to make `set()`, `get()`, and `has()` instance (rather than static) methods, and we also must add a public `reset()` instance method. Update *src/CacheStatic.php* as shown in the listing.

```
<?php
namespace Mattsmithdev;

class CacheStatic implements Cacheable
{
    private static array $dataItems = [];

    ❶ public function reset(): void
    {
        self::$dataItems = [];
    }

    public function set(string $key, string $value): void
    {
```

```

        self::$dataItems[$key] = $value;
    }

    public function get(string $key): ?string
    {
        if (self::has($key)) {
            return self::$dataItems[$key];
        }

        return NULL;
    }

    public function has(string $key): bool
    {
        return array_key_exists($key, self::$dataItems);
    }
}

```

Listing 26-19: Revising `CacheStatic` to implement the `Cacheable` interface

We declare that the class implements the `Cacheable` interface, then provide an implementation for the requisite `reset()` method that sets `$dataItems` to an empty array **❶**. The `set()`, `get()`, and `has()` implementations are the same as before, except we've changed them all from static to instance methods. The `$dataItems` array itself remains a static member.

Listing 26-20 shows the modified `CacheJson` class in *src/CacheJson.php*.

```

<?php
namespace Mattsmithdev;

class CacheJson implements Cacheable
{
    private const CACHE_PATH = __DIR__ . '/../var/cache.json';

    ❶ public function reset(): void

```

```

    {
        $directory = dirname(self::CACHE_PATH);
        $this->mkdirIfNotExists($directory);
        $this->makeEmptyFile(self::CACHE_PATH);
    }

    private function mkdirIfNotExists(string $directory):
bool
    {
        return is_dir($directory) || mkdir($directory);
    }

    private function makeEmptyFile(string $path): bool
    {
        return file_put_contents($path, '');
    }

    public function set(string $key, string $value): void
--snip--

```

Listing 26-20: Our refactored `CacheJson` class, implementing the `Cacheable` interface

Again, we have to provide an implementation for the `reset()` method ❶. It uses the private `mkdirIfNotExists()` and `makeEmptyFile()` methods (declared next in the listing) to ensure that an empty file and directory exist after `reset()` is invoked. The remainder of the code, including the `set()`, `get()`, and `has()` methods, is the same as it was in [Listing 26-12](#).

As this example has illustrated, declaring a useful feature like caching as an interface means you can create different implementations of that feature while writing most of your code (in this case, the `Application` and `MainController` classes) in a general way. This enables you to switch implementations of the interface, or create new ones later, without having to update all your code because of hardcoded references to the old way of doing things. All that has to change is the code that actually instantiates the class implementing the interface. We’ve conveniently located that code in the index script, where it can easily be updated without breaking the application.

Traits

A *trait* is a way to provide default versions of methods that are shared among multiple, unrelated classes. This feature offers not just the method signatures, as with an interface or abstract method, but actual method implementations. When a class uses a trait, it's called *insertion*, since the trait is essentially inserting a method into the class without the class having to define the method itself. That said, a trait can be inserted onto a class and then be overridden by the class's own method implementations if necessary. This is useful when most, but not all, classes inserting a trait can use the same method implementation.

NOTE

In some other programming languages, traits are known as mixins, after the extra ingredients such as nuts or candy that can be mixed into ice cream.

Traits are a way of permitting code reuse across class hierarchies without resorting to multiple inheritance, a sort of copy-and-paste feature for methods that allows the methods to still be overridden if needed. This is helpful, for example, if you have several classes implementing the same interface, all with identical versions of some of the methods the interface calls for. In this case, a lot of code would be duplicated across the classes, a violation of the DRY principle. Declaring those methods as a trait would allow you to write the code once and then add it to all the relevant classes by telling them to use the trait.

More broadly, traits may come into play when classes in multiple class hierarchies need to perform common actions. For example, several classes may need the behaviors of `mkdirIfNotExists()` and `makeEmptyFile()`, methods we declared earlier as part of the `CacheJson` class. One solution could be to make these methods public members of some kind of utility class (say, `FileUtilities`), so each class needing that functionality could create a `FileUtilities` object and invoke the methods; or we could declare the methods as public static members of the utility class to avoid having to create an object at all.

However, the application might change over time, and some of the classes may need specialized variations of the main method implementations. As such, instead of relegating the methods to a utility class, we can declare them as a trait. The methods will then be available for any class to use, but each class can replace them with custom implementations if required, without affecting any other part of the codebase.

Ultimately, traits and utility classes are similar concepts, in that both can provide the same fully implemented methods to classes from different class hierarchies. Traits are a little more sophisticated and flexible than utility classes, however, since they can be overridden if needed. A class's reliance on a trait may be more obvious than its reliance on a utility class, since the trait must be referenced with a `use` statement, whereas utility class method calls might be hidden within the implementation of a method; in this way, traits make code dependencies more transparent. On the other hand, traits can be harder to test directly since their methods are often private or protected, whereas utility class methods are typically public.

Declaring Traits

You declare a trait much like a class, but with the `trait` rather than the `class` keyword. To see how it works, let's move the declarations of the `mkdirIfNotExists()` and `makeEmptyFiles()` methods from the class to a `FileSystemTrait` trait. Continuing the project from the previous section, create a new `src/FileSystemTrait.php` file and copy over the two method definitions, as shown in [Listing 26-21](#).

```
<?php
namespace Mattsmithdev;

trait FileSystemTrait
{
    private function mkdirIfNotExists(string $directory):
bool
    {
        return is_dir($directory) || mkdir($directory);
    }
}
```

```
private function makeEmptyFile(string $path): bool
{
    return file_put_contents($path, '');
}
}
```

Listing 26-21: The *FileSystemTrait* trait

We use the `trait` keyword to declare `FileSystemTrait` as a trait. It contains method declarations for `mkdirIfNotExists()` and `makeEmptyFile()`. The implementation of these two methods is exactly as it was when they were in the `CacheJson` class.

While we're at it, let's extract the two JSON file methods from the `CacheJson` class, `readJson()` and `writeJson()`, and declare them as a second trait, `JsonFileTrait`, since these methods also define functionality that several classes might need. Copy the method definitions into a new `src/JsonFileTrait.php` file and update them as shown in [Listing 26-22](#).

```
<?php
namespace Mattsmithdev;

trait JsonFileTrait
{
    private function readJson(string $path): array
    {
        $jsonString = file_get_contents($path);
        if (!$jsonString) {
            return [];
        }

        $dataItems = json_decode($jsonString, true);
        return $dataItems;
    }

    private function writeJson(string $path, array $dataItem
s): bool
    {

```



```
        $jsonString = json_encode($dataItems);
        return file_put_contents($path, $jsonString);
    }
}
```

Listing 26-22: The *JsonFileTrait* trait

We declare the `JsonFileTrait` trait with two methods, `readJson()` and `writeJson()`. Once again the method implementations are virtually identical to the original methods on the `CacheJson` class, but this time we use a string `$path` parameter to indicate the JSON file that needs to be read or written to instead of a hardcoded class constant. This makes the methods more generally applicable.

Inserting Traits

Now let's look at how to insert traits onto a class by refactoring `CacheJson` to use our two traits. [Listing 26-23](#) shows the modified `src/CacheJson.php` file.

```
<?php
namespace Mattsmithdev;

class CacheJson implements Cacheable
{
    ❶ use FileSystemTrait, JsonFileTrait;

    private const CACHE_PATH = __DIR__ . '/../var/cache.json';

    ❷ public function reset(): void
    {
        $directory = dirname(self::CACHE_PATH);
        $this->mkdirIfNotExists($directory);
        $this->makeEmptyFile(self::CACHE_PATH);
    }

    public function set(string $key, string $value): void
    {
```

```

        $dataItems = $this->readJson(self::CACHE_PATH);
        $dataItems[$key] = $value;
        $this->writeJson(self::CACHE_PATH, $dataItems);
    }

    public function get(string $key): ?string
    {
        $dataItems = $this->readJson(self::CACHE_PATH);
        if($this->has($key)){
            return $dataItems[$key];
        }

        return NULL;
    }

    public function has(string $key): bool
    {
        $dataItems = $this->readJson(self::CACHE_PATH);
        return array_key_exists($key, $dataItems);
    }
}

```

Listing 26-23: Updating the `CacheJson` class to use traits

We start with a `use` statement containing a comma-separated list of the traits to insert onto the class ❶. In the `reset()` method ❷, notice how we invoke the `mkdirIfNotExists()` and `makeEmptyFile()` methods, which now come from the trait, just as we did before. We don't need to mention the trait when using these methods; we simply call the methods by name, as usual. Similarly, we're able to use the `readJson()` and `writeJson()` methods as before, but now we pass the `CACHE_PATH` constant as an argument.

We now have a much simpler `CacheJson` class. The commonly used methods for filesystem and JSON file operations have been refactored as traits, which keeps `CacheJson` itself focused on tasks related to caching. Meanwhile, the methods on the traits are also available for any other class to use.

Resolving Trait Conflicts

If a class uses two or more traits, the same member could be declared in multiple traits. This potential problem is similar to issues that occur in languages that allow multiple inheritance. In such cases, you'll get a fatal error if you try to call that method, since the PHP engine won't know which implementation to invoke.

To resolve the ambiguity and avoid the error, use the `insteadof` keyword to specify which version of the method you want to use. Here's an example:

```
use TraitA, TraitB {  
    TraitA::printHello insteadof TraitB;  
}
```

This code snippet specifies that if `printHello()` is declared in both `TraitA` and `TraitB`, it's the `TraitA` implementation that should be inserted into the class.

What to Use When?

The various strategies we've discussed in this chapter have considerable overlap. Deciding which to use in a given situation may be a matter of personal preference or the preference of a larger team. That said, [Figure 26-8](#) offers some guidance by summarizing the similarities and differences between the approaches we've discussed.

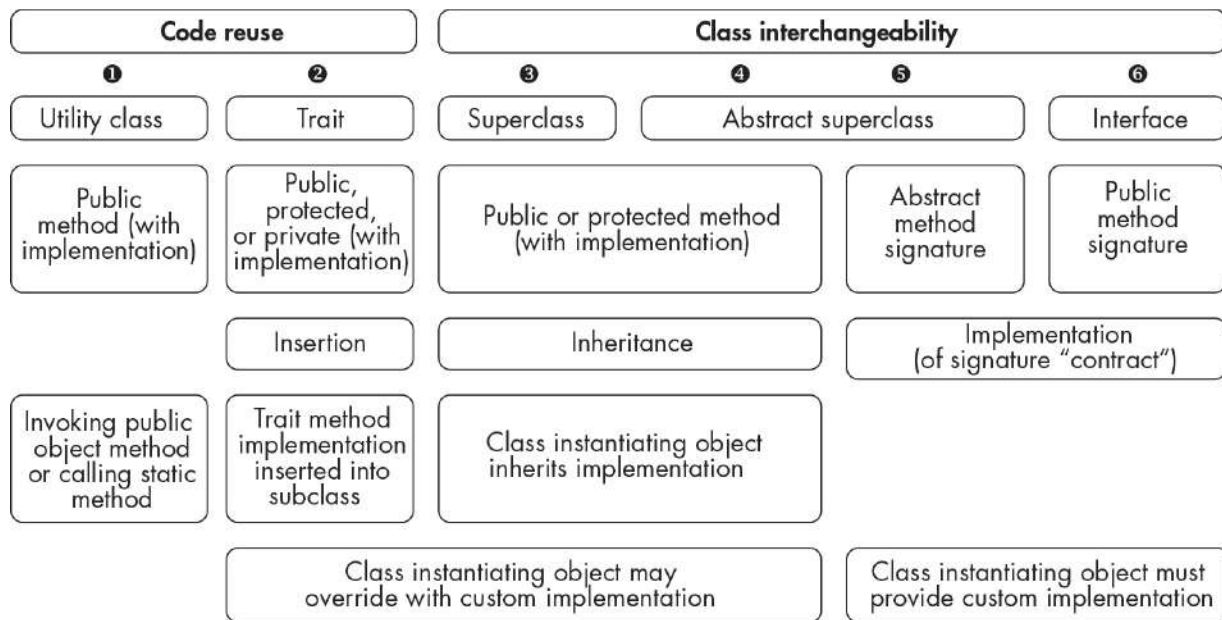


Figure 26-8: Comparing strategies for code reuse and class interchangeability

For basic class hierarchies, much can be achieved with simple inheritance, allowing subclasses to inherit fully implemented methods from concrete ③ or abstract ④ superclasses. If many subclasses will require custom implementations of the inherited methods, you might declare them on the superclass as abstract methods instead ⑤. This way, only the methods' signatures will be specified, with the implementation details left up to the subclasses. Interfaces are another way to declare just the signatures of methods, but in this case the methods can be shared across class hierarchies ⑥.

Taken together, inheritance, abstract methods, and interfaces promote class interchangeability while also loosening the dependencies among the components in a software system. This greatly facilitates cooperative software development. By standardizing method signatures while allowing for flexibility in method implementation, interfaces in particular can be a contract between software components as well as a contract between cooperating developers, each with responsibilities for coding different parts of the system. The whole team can be confident that the system will behave as expected as long as the interface requirements are adhered to. This approach to software design is sometimes referred to as *loose coupling*: the number and form of breakable dependencies among software components

are reduced, so changes in any one component are much less likely to affect performance or require refactoring of other components.

Meanwhile, if your goal is to reduce code duplication, you can use traits to offer sets of default method implementations that can be explicitly inserted into classes from different hierarchies ❷. For small systems, utility classes, perhaps with public static methods, can be another way to offer the same functionality to different parts of the system ❶. Traits provide more flexibility (for example, a class that inserts a trait may still override a method from that trait with its own custom implementation), but the public methods on utility classes are more readily exposed for thorough testing.

Summary

In this chapter, we looked at several strategies for sharing methods among classes, both within and outside the confines of class hierarchies. You saw how abstract methods and interfaces enforce method signatures without providing implementations. The rest of the application can safely call the relevant methods, regardless of the implementation, since the signatures are guaranteed to always be the same. You saw this at work when we created a `Cacheable` interface that allowed us to switch approaches to caching (using a static array versus an external JSON file) with virtually no impact on the rest of the application code.

You also saw how to use traits to insert fully implemented methods onto unrelated classes, while still having the flexibility to override those methods if necessary. We harnessed traits to make general-purpose filesystem and JSON-handling methods available for any class in our caching project to use. This promoted code reusability and allowed us to declare simpler, more narrowly focused classes.

Exercises

1. Declare a `Book` class that has the following members:
 - A private string `title` property with get and set methods
 - A private float `price` property with get and set methods
 - A public `getPriceIncludingSalesTax()` method that returns a float, calculated as `price` plus 5 percent sales tax

Write a main script that creates a `Book` object and prints its price with and without sales tax, like so:

```
Book "Life of Pi"
    price (excl. tax) = $20.00
    price (incl. tax) = $21.00
```

2. Refactor your project for Exercise 1 to declare an interface named `SalesTaxable` requiring classes to implement a `getPriceIncludingSalesTax()` method that returns a float. The `Book` class should implement the `SalesTaxable` interface.

Next, declare a `Donut` class that also implements the `SalesTaxable` interface and has these members:

- A private string `topping` property with get and set methods

- A private float `price` property with get and set methods

- A public `getPriceIncludingSalesTax()` method implementing the `SalesTaxable` interface and returning `price` plus 7 percent sales tax

Finally, write a main script that creates the following two objects and prints their price with and without sales tax:

```
Book "Life of Pi"
    price (excl. tax) = $20.00
    price (incl. tax) = $21.00
```

```
Donut "strawberry icing"
    price (excl. tax) = $10.00
    price (incl. tax) = $10.70
```

3. Write a `TaxFunctions` utility class that declares a public static `addTaxToPrice()` method that takes in a float price and float tax rate and returns the value of the price with the tax added. Refactor the implementations of the `getPriceIncludingSalesTax()` methods in the `Book` and `Donut` classes to use this utility class method, to avoid code duplication.

4. Change your `TaxFunctions` utility class to a trait, declaring a (nonstatic) `addTaxToPrice()` method. Refactor the `Book` and `Donut` classes to insert the trait and use its `addTaxToPrice()` method in their implementations of `getPriceIncludingSalesTax()`.
5. Refactor your project as a class hierarchy, with an abstract `SellableItem` superclass that declares a fully implemented `getPriceIncludingSalesTax()` method with protected visibility. Make `Book` and `Donut` subclasses of `SellableItem`, and delete the interface and trait files; they aren't needed in this design. Sometimes, for a simple situation, the simplest solution is the most appropriate.

PART VI

DATABASE-DRIVEN APPLICATION DEVELOPMENT

INTRODUCTION TO DATABASES



Databases are one of the ways computer systems can persistently store data so that it can be remembered when code runs at a later time. We'll explore how to use PHP to work with databases over the next several chapters, starting in this chapter with database fundamentals. We'll go over basic database terminology, look at how databases are structured, and consider the motivation for and benefits of connecting a web application to a database.

We'll also discuss how a database fits with our coverage so far about web application architecture. You'll learn how to map the contents of a database onto an object-oriented structure of classes and objects, and you'll see how a database forms the backbone of the model component of the MVC architecture described in earlier chapters.

NOTE

This book doesn't aim to provide a comprehensive guide to relational database design, a complex field in its own right. Our focus will be on interacting with databases by using PHP. Some books for learning more about SQL and databases include Practical SQL, 2nd edition (2022), by Anthony DeBarros; The Manga Guide to Databases (2009) by Mana Takahashi; and MySQL Crash Course (2023) by Rick Silva, all from No Starch Press.

Relational Database Basics

The majority of modern database systems are *relational*, meaning they're composed of a set of interrelated tables. Each table represents a type of entity. For example, a `customer` table might store information about the customers at an e-commerce site.

A table consists of columns and rows. Each column represents an attribute of the entity (for example, the `customer` table might have columns for a customer's name, address, phone number, and so on). Each row represents a single instance of the entity (for example, an individual customer). Each row is also called a *record*.

The relationships between database tables are established through keys; each *key* is a unique identifier associated with one record in a table. Referencing one table's keys from within another table creates a link between the two tables, while also avoiding duplication of data. Continuing our e-commerce example, each customer in our `customer` table could be given a *primary key* in the form of a unique customer ID number. Meanwhile, we might also have an `invoice` table for recording transactions, with a unique ID number for each invoice. Every invoice should be related to a single customer (the person who initiated the transaction), while each customer may be associated with multiple invoices, since a person can initiate multiple transactions.

We would establish this relationship by storing the customer ID associated with each invoice as a column in the `invoice` table, unambiguously associating each invoice with one—and only one—customer. In the context of the `invoice` table, the customer ID is known as a *foreign key*, since it connects to a field in a different table. Thanks to the foreign key, the `invoice` table doesn't need to duplicate the name, address,

and other information about the customer; we can simply look up those details in the `customer` table based on the customer ID assigned to a given invoice. This is the power of the relational database.

Assigning a unique key to each row also helps maintain the correctness, or *integrity*, of the database. When database changes are being attempted, these keys act as links between data items in different tables. The system can ensure there's an associated data item corresponding to a key referenced by another item in a different table. Rules can be established in the database to prevent new data from being created if it attempts to link to a nonexistent data item. For example, this might save a customer from being charged for a nonexistent invoice, or save an invoice from being assigned to a nonexistent customer. Other rules can be related to deletions of data, creating a warning or exception if we try to delete an item that other items are linked to.

Overall, the structure of a database's tables, the relationships between the tables, and the rules governing data integrity are referred to as the *relational schema* for that database. Complex web applications often require several relational schemas that operate side by side. For example, one schema might be for the financial records of an organization, another for human resource details, and another for stock items and customer orders.

Database Management Systems

The software that creates, modifies, retrieves from, and stores a database is called a *database management system (DBMS)*. For a relational database, we sometimes refer more specifically to a *relational database management system (RDBMS)*. For the purposes of this book, we'll focus on two (R)DBMSs: MySQL and SQLite. These are two of the most popular free and open source systems in use today.

Some DBMSs run as server applications requiring usernames and passwords. They may run on the same computer system as the web application that uses them, or they may run on a completely independent internet server. MySQL is an example of a server-based DBMS. Other DBMSs, such as SQLite, are file-based, meaning the data is stored in files on the same computer as the web application. Server-based DBMSs like MySQL can work with multiple database schemas, whereas SQLite and

most other file-based DBMSs store a single database schema in each file. One SQLite file might hold the financial records database, for example, another file could hold the human resource details database, and so on.

For a computer language like PHP to communicate with a particular DBMS, you need a *database driver*. This piece of software allows a program to communicate with a DBMS through its own standard protocol. For example, PHP has a driver for MySQL, a driver for SQLite, and other drivers for other DBMSs. The MySQL and SQLite PHP drivers may already be enabled on your system. If not, you'll get driver errors when you try to run the code in the following chapters, and you may need to tweak the settings in your *php.ini* configuration file. See [Appendix B](#) for instructions on setting up one or both of these database systems locally, or [Appendix C](#) if you're working in Replit.

When your PHP program needs to work with a DBMS, it uses a database *connection* at runtime. This connection is an active communication link between a computer program and a DBMS. To create a connection with a server-based database system, you must provide the host and port details, and usually the appropriate username and password authentication details as well. In some cases, a connection can be made directly to work with a particular database schema (for example, the human resource details schema); in other cases, a general connection is made to the DBMS, and either a new schema is created or the existing schema to be used is selected after the connection has been created. Once a connection has been established to work with a specific schema, the desired actions can be executed, which might include creating tables and relationships, inserting or deleting data, or retrieving data from the tables of the schema.

One key advantage of databases over other persistent storage methods such as files is that many DBMSs are designed to safely be used by multiple people simultaneously. Incorporating a database into a web application thus allows many people to interact with the application at the same time, while still ensuring the security and integrity of the system data. This is one area where server-based DBMSs like MySQL shine over file-based DBMSs like SQLite. While SQLite allows multiple simultaneous users to work on its file-based database, it locks the entire database file when a user is making changes. This is fine for local machine testing and development, but it would result in unacceptable delays for a fully

deployed, real-world web application with heavy traffic. Systems such as MySQL can handle large numbers of concurrent connections, locking only single tables or even single database rows to minimize any interference with other users.

Server-based DBMSs also have the potential to be run as multiple instances, allowing multiple versions of both the web application and its database to help a system handle massive numbers of simultaneous users. Instances can be added and removed as needed to cope with a varying load of traffic over time. This technique is known as *load balancing*, and many cloud services implement it automatically.

Structured Query Language

The majority of modern RDBMSs are manipulated using Structured Query Language (SQL). SQL is designed to accomplish three key aspects of working with relational databases:

- Defining the structure of the related tables
- Manipulating the stored data (creating, updating, or deleting data)
- Querying the data (searching the database, given criteria to match against)

Listing 27-1 illustrates SQL statements for each of these kinds of actions.

```
CREATE TABLE IF NOT EXISTS product (  
    id integer PRIMARY KEY AUTO_INCREMENT,  
    description text,  
    price float  
)  
  
DELETE FROM product WHERE price < 0  
  
SELECT * FROM product WHERE price > 99
```

Listing 27-1: Examples of SQL definition, manipulation, and querying statements

The first SQL statement creates a `product` table and defines its structure. Each entry in this table will store the `id`, `description`, and `price`

of a product. Each of these columns is given a data type (such as `float` for the `price` column), and the `id` column is designated as the table's primary key, meaning each table entry should have a unique `id` value.

The second SQL statement demonstrates how to manipulate the data stored in the database; the statement deletes all entries from the `product` table that have a negative price. Finally, the third SQL statement is an example of a database query; it uses `SELECT` to request all the entries in the `products` table that have a price greater than 99.

Although SQL isn't case-sensitive, it's common practice to write SQL keywords like `SELECT`, `FROM`, and `WHERE` in all caps, and to use lowercase letters for table and column names as well as for strings forming a condition in a statement. Following this convention helps make SQL statements more readable.

Databases and Web Application Architecture

Databases naturally fit into the architecture of object-oriented web applications. Classes of objects can be written to closely map to the data items stored in database tables, and databases and their classes are the usual choice for the model (M) component in the MVC web application pattern.

Object-Oriented Programming

The class structure of OOP easily maps to relational database tables. A common and straightforward way to structure a web application working with a database is to design a class that corresponds to each table in the database. These *entity classes* have properties mirroring the table's columns, and an instance of the class would correspond to a record (row) of the table.

If we need to write data to a database table, we would first create an object of the appropriate class containing the new data, then use our database connection to send the object's data to the database table. We can even send data back to that object; for example, if the database needs to choose a new unique key for the new record, this value can be sent back to the web application and stored in the corresponding object for future reference. Conversely, if we need to read a whole record out of a database table, we would read the retrieved data into a new object of the appropriate

class, at which point the rest of the application can utilize the database data by accessing that object.

Consider a web application (and its database) implementing products of various categories. We may have categories with names such as `food`, `hardware`, and `furniture`, and each product must be related to one of these categories. [Figure 27-1](#) shows the database’s relational schema. This kind of diagram is called an *entity-relationship (ER) model*.

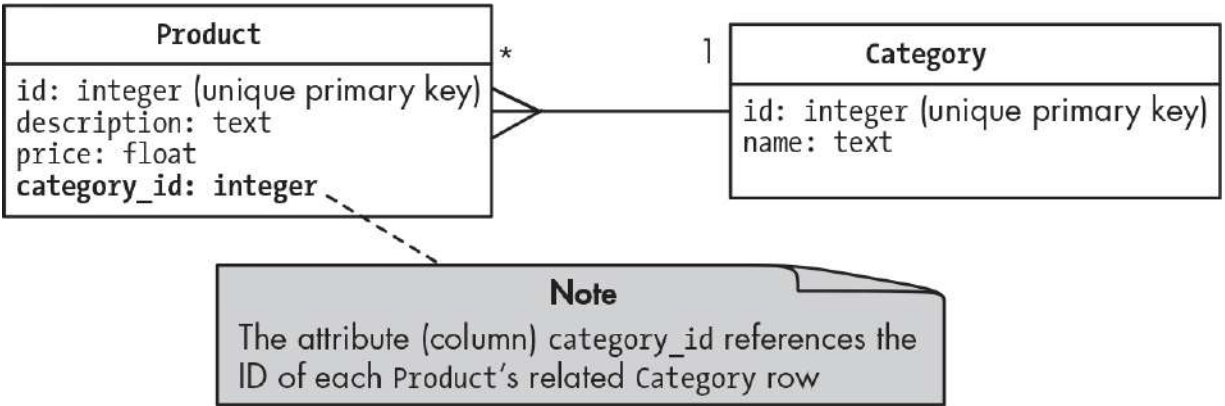


Figure 27-1: An ER diagram showing related *Product* and *Category* entities

Our database will have a `Product` table containing an entry for each product as well as a `Category` table with the possible categories the products can fall into. The line connecting the tables illustrates how entries from these tables can relate. The 1 at the `Category` end of the connecting line expresses the relationship “each product is associated with precisely one category.” The crow’s foot link and asterisk (*) at the `Product` end expresses the relationship “each category is associated with zero, one, or more products.”

Each record in the `Category` table will have a unique integer `id` property (a primary key) and a text description. [Table 27-1](#) shows sample entries in the table.

Table 27-1: Example Rows for the `Category` Table

id (primary key)	name
1	"food"

id (primary key)	name
2	"hardware"
3	"furniture"

Every record in the `Product` table will similarly have a unique integer `id` property as a primary key, along with a text `description` and a float `price`. Each product will also be related to exactly one category via the `category_id` column, which will store a reference to the key for one of the records in the `Category` table. Again, this is known as a *foreign key*. [Table 27-2](#) shows example rows for the `Product` table.

Table 27-2: Example Rows for the `Product` Table

id (unique key)	description	price	category_id
1	"peanut bar"	1.00	1 (food)
2	"hammer"	9.99	2 (hardware)
3	"ladder"	59.99	2 (hardware)

We can easily map our database tables to object-oriented classes. [Figure 27-2](#) shows the corresponding class diagram for `Product` and `Category`. Notice that this UML diagram is essentially the same as the ER model from [Figure 27-1](#).

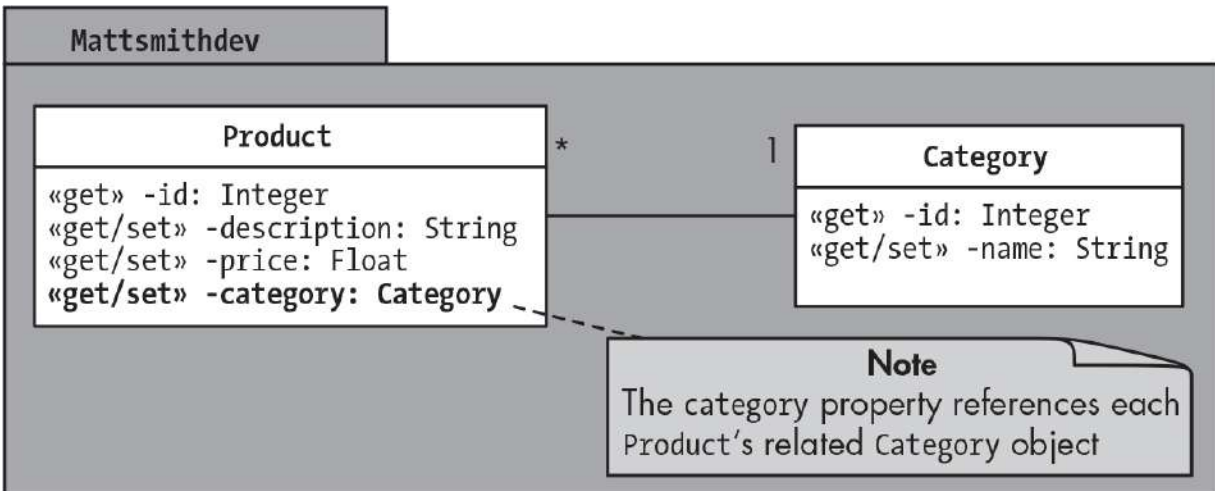


Figure 27-2: A class diagram of the related *Product* and *Category* classes

Each class has properties for all the columns of its corresponding database table; for example, the `Product` class has `id`, `description`, `price`, and `category` properties. Each `Product` object will be related to exactly one `Category` object via its `category` property, which will store a reference to a `Category` object. Notice that this is the main difference between our class structure and our database structure. In the `Product` database table, the `category_id` column simply stores the integer ID of the related category, whereas with our classes we can store a reference to a complete `Category` object.

[Figure 27-3](#) shows the objects that will be created when we read the sample database rows from [Tables 27-1](#) and [27-2](#) into our web application.

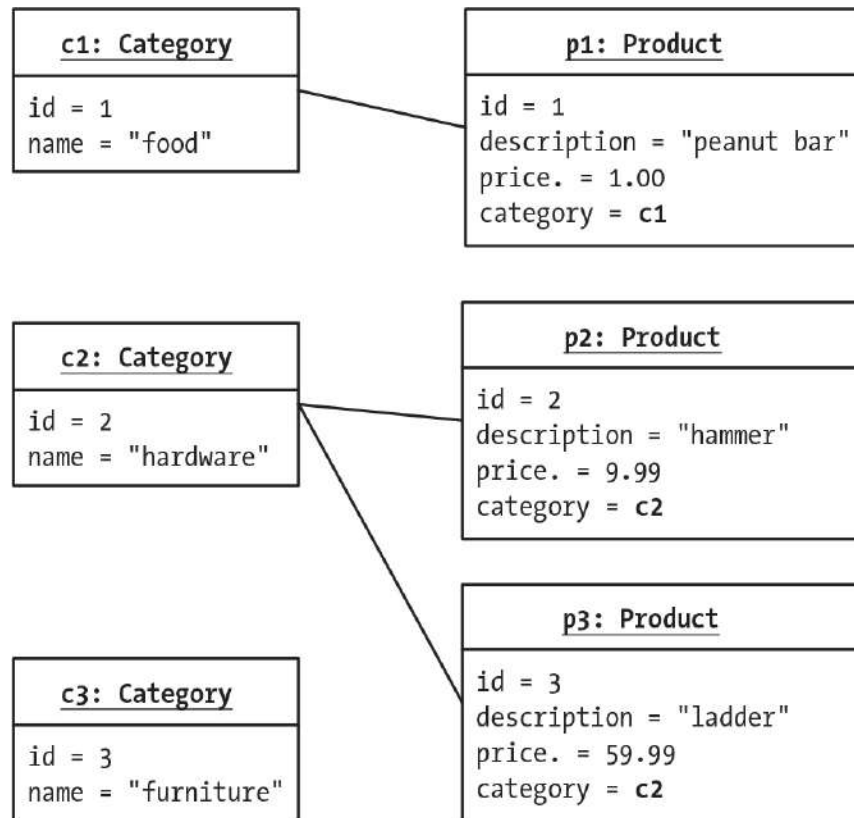


Figure 27-3: Connecting *Product* objects to *Category* objects

We end up with three `Product` objects, linked to their corresponding `Category` objects. Notice that each `Product` object is associated with only one `Category` object. In contrast, a `Category` object can be related to zero, one, or many `Product` objects, since at a given point we may have no products for some categories, perhaps just one product, or more.

The Model-View-Controller Pattern

In previous chapters, we've discussed the MVC software architecture, which assigns the various tasks required for operating a web application to different parts of the system. We've focused primarily on how a templating library like Twig provides the view component of MVC, preparing content to be displayed to the user, and on how a front controller and other specialized controller classes provide the controller component, making decisions about how to respond to each user request.

I've said little so far about the model part of this architecture, the actual data that underlies the web application. This is where a database comes in.

It stores the data in an organized format and serves it up or modifies it when prompted by the controller classes. The classes corresponding to the database tables are part of the model component of the application as well. [Figure 27-4](#) illustrates the database's place in the MVC web application architecture.

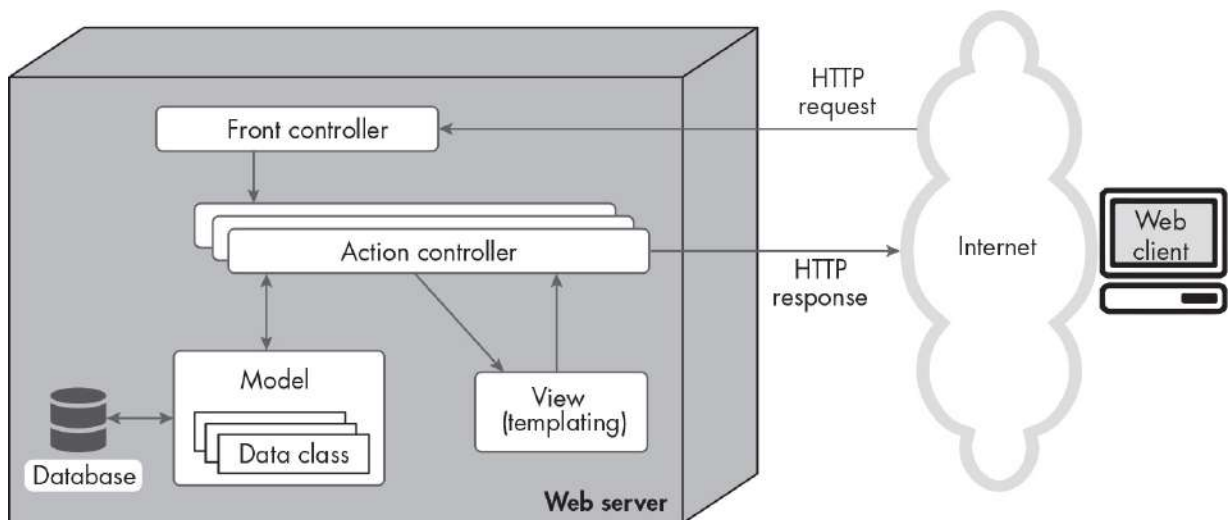


Figure 27-4: The MVC architecture, highlighting the model component

Notice that the action controller classes communicate with (read from and modify) the *model classes*. As you'll learn in the following chapters, these are the object-oriented classes that communicate with the database. All database interactions are kept completely separate from the controller and view components of the application. This compartmentalization means that we could change the underlying database (from a file-based DBMS to a server-based DBMS, for example) without having to make any changes to the front controller, the action controller classes, or the templates.

Summary

In this chapter, we reviewed the concept of databases, especially SQL-based relational databases, and considered some of the advantages of adding databases to web applications. We also explored how databases fit into web application architecture, including how databases are the heart of the MVC pattern's model component. We observed the close mapping among the

tables, columns, and rows of a relational database and the classes, properties, and instances used in OOP. With this introduction, you're now ready for the remaining chapters in the book, where you'll learn to use PHP to connect to, create, modify, and retrieve data from MySQL and SQLite relational databases.

Exercises

1. Read up on some of the history of databases in the phoenixNAP article "What Is a Database?" by Milica Dancuk (<https://phoenixnap.com/kb/what-is-a-database>).
2. DB Fiddle (<https://www.db-fiddle.com>) is a great online resource for practicing SQL statements and designing a database. You can create and populate tables, query them, and view the results. Try using DB Fiddle to implement the `Product` and `Category` database tables discussed in this chapter. Insert the three rows of sample data for each database table (see [Tables 27-1](#) and [27-2](#)), then run queries to select the data from each table.

DATABASE PROGRAMMING WITH THE PDO LIBRARY



Incorporating a database into a web application requires writing code to perform operations such as opening a connection to the database system; creating a database and its table structure; manipulating database data through insertions, deletions, and updates; and querying and retrieving data matching your desired criteria. In this chapter, you'll learn about the PHP Data Objects (PDO) library, which makes it easy to carry out these sorts of database operations. We'll use the library to progressively develop a simple, multipage web application that pulls information from a database.

The PDO Library

The PDO library for database operations has been a built-in feature of the PHP language since 2005 and is compatible (through various drivers) with

many DBMSs, including MySQL and SQLite, as we'll see in this chapter. This makes it incredibly easy to develop flexible web applications that can switch DBMSs with minimal changes required to the code. Before PDO, switching to a different DBMS meant using a different library.

In addition to offering a standard (and therefore reusable) way to run SQL commands on different relational database systems, PDO also makes it much easier to write more secure database communication code through the use of *prepared statements*. These are templates for database queries, including placeholders for certain fields that can be set to actual values when the query is to be executed. The basic pattern is to build the SQL statement as a string (including any placeholders), pass that string to a PDO connection object to “prepare” the statement, pass along any values for filling in the placeholders, and then execute the prepared statement on the database.

Handling SQL code through prepared statements avoids problems of SQL injection attacks, and so we'll be using only prepared statements in this book. In a *SQL injection*, text that's received from the user (for example, through a web form or a login field) is concatenated into an SQL query string and executed on the database. Malicious users can take advantage of this common web application vulnerability to modify the original SQL query or add an additional SQL query that will then also be executed on the database. The web comic XKCD took a humorous look at SQL injection, shown in [Figure 28-1](#).



Figure 28-1: Randall Munroe's "Bobby Tables" cartoon (<https://xkcd.com/327/>) is a lighthearted example of the damage an SQL injection attack can do.

Yet another bonus of using PDO is that it offers an *object fetch mode*, whereby data queried from the database is automatically packaged into objects of the appropriate classes in your PHP code (*model classes*). All you have to do is tell PDO which classes correspond to which database tables. Without this feature, you'd have to write code to handle the details of building the objects based on the results of the query, which often requires fussing with multidimensional arrays and column headers.

We'll explore the basics of using the PDO library throughout this chapter as we develop an object-oriented, database-driven web application.

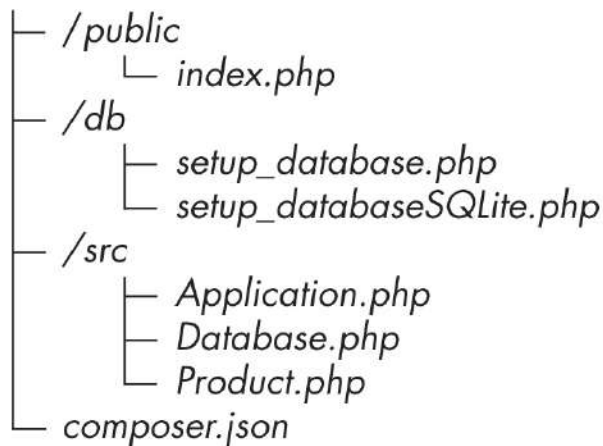
NOTE

This chapter only scratches the surface of the PDO library's capabilities. For more information about what it can do, I recommend the modestly titled "(The Only Proper) PDO Tutorial," available online at <https://phpdelusions.net/pdo>.

A Simple Database-Driven Web Application

To get started using PDO, we'll first create a bare-bones application with a single page that retrieves and displays information about a selection of products stored in a database. This will illustrate how to connect to a database, create a table, populate it with data, and retrieve that data for use in the application, all in an object-oriented way. In "A Multipage Database-Driven Web Application" on [page 553](#), we'll expand the application to include multiple pages, well-organized controller logic, and Twig templating.

For now, the project will have the following file structure:



To begin, create a new project folder and add the usual *composer.json* file declaring that `Mattsmithdev` namespaced classes are located in the *src* folder. Then add a *public* folder containing the usual *index.php* script that reads in the autoloader, creates an `Application` object, and invokes its `run()` method. With that, we're ready to set up the databases. The *db* folder will contain the scripts to create both MySQL and SQLite versions of a database to support our application.

Setting Up the Database Schema

Our web application will be able to use MySQL or SQLite as a DBMS, and in this section we'll write PHP scripts to set up a new database schema using both systems. For small, local projects, an SQLite database file in the *var* folder of the project is often sufficient. For large-scale, production-ready web applications, MySQL is more common, with the database running on a different server (or multiple servers).

For this simple example, we'll save the MySQL and SQLite database setup scripts in the project's *db* folder. In a more realistic scenario, however, the database structure would be fixed and the database already set up, so such scripts wouldn't usually be kept as part of the application's folder structure.

For our database, we'll create a simple schema consisting of a single table called `product` and insert two example records into that table. [Figure 28-2](#) shows an ER model of this table.

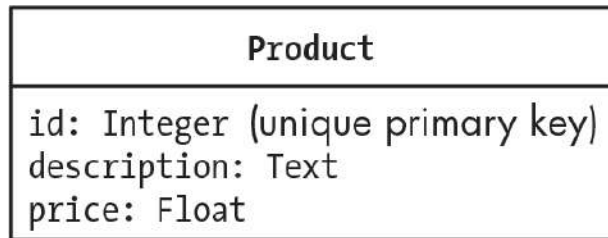


Figure 28-2: An ER model for the *product* table

As the diagram shows, the `product` table will have fields for the product's `id` (a unique numerical identifier), its `description`, and its `price`.

MySQL

Listing 28-1 uses PDO to create a MySQL database schema, define the structure of a `product` table, and insert two example records into that table. Name this script *create_database.php* and save it in the *db* folder.

```
<?php
❶ define('DB_NAME', 'demo1');

❷ $connection = new \PDO(
    'mysql:host=localhost:3306',
    'root',
    'passpass'
);

❸ $sql = 'CREATE DATABASE ' . DB_NAME ;
$stmt0 = $connection->prepare($sql);
$stmt0->execute();

$connection = new \PDO(
    ❹ 'mysql:dbname=' . DB_NAME . ';host=localhost:3306',
    'root',
    'passpass'
);

$sql = 'CREATE TABLE IF NOT EXISTS product ('
    ❺ . 'id integer PRIMARY KEY AUTO_INCREMENT,'
    . 'description text,'
```

```

        . 'price float'
        . '));
$stmt1 = $connection->prepare($sql);
$stmt1->execute();

$sql = "INSERT INTO product (description, price) VALUES ('hammer', 9.99)";
$stmt2 = $connection->prepare($sql);
$stmt2->execute();

$sql = "INSERT INTO product (description, price) VALUES ('ladder', 59.99)";
$stmt3 = $connection->prepare($sql);
$stmt3->execute();

```

Listing 28-1: A script to create our MySQL database

First, we define a `DB_NAME` constant to hold the database schema name, 'demo1' ❶. Putting the name in a constant makes this script easy to edit and reuse on other database schemas—just update the name in the constant.

Next, we create a new `PDO` object to establish a connection with a database, storing the result in the `$connection` variable ❷. The first argument is the *data source name (DSN)*, a standardized string providing information about the database connection. The DSN string begins with 'mysql:', telling PDO that we want to connect to a MySQL server, followed by one or more *key=value* pairs, separated by semicolons. For now, we need just one *key=value* pair to specify that the host the MySQL database is running on is the `localhost` server at port 3306. We don't include the schema name here, since we haven't created the schema yet. The second and third arguments passed to the `PDO` constructor provide the username '`root`' and the password '`passpass`'. Replace these with the database username and password for your MySQL setup (see [Appendix B](#)).

We next build and execute an SQL statement to create the database schema named in the `DB_NAME` constant ❸. We create the SQL statement as a string in the `$sql` variable, consisting of the SQL keywords 'CREATE DATABASE' plus the schema name. We pass that string to the `$connection` object's `prepare()` method to securely prepare the statement, placing the

result, an object of the PDO library's `PDOStatement` class, in the `$stmt0` variable. Then we execute the prepared statement. We'll use this basic pattern of preparing and executing SQL statements over and over when working with databases. In most cases, we'd then perform an action after the statement has been executed, such as returning a list of retrieved objects or confirming the database has been changed as expected and then informing the user if it has.

Now that we've created the schema, we need to create the `product` table within it. First, we overwrite `$connection` with a new connection to the database schema itself, rather than to the MySQL server in general. Notice that this time we specify the schema name as part of the DSN string when constructing the PDO object ❹. Specifically, we assign the `DB_NAME` constant as the value of the `dbname` key, followed by a semicolon (`;`) to separate this key/value pair from the others in the DSN string. Then we build and execute another SQL statement creating the `product` table with `id`, `description`, and `price` fields. MySQL databases support auto-incrementing, which automatically generates unique numeric keys in sequence. Our SQL statement uses this feature as part of the primary-key declaration for the `id` field, so we don't have to worry about manually setting the product IDs ❺.

We finish the script by creating and executing two `INSERT SQL` statements to add two rows to the `product` table: a 'hammer' costing 9.99 and a 'ladder' costing 59.99. We don't include values for each product's `id` field since MySQL will automatically generate them.

Once you've written this script, run it by entering `php db/create_database.php` at the command line. This will create and populate the MySQL schema.

SQLite

Now let's adapt the script from [Listing 28-1](#) to create the same schema as an SQLite database file. As you'll see, the process is similar, since the PDO library can work with SQLite just as easily as with MySQL. Save the contents of [Listing 28-2](#) as `create_databaseSQLite.php` in your project's `db` subdirectory. The SQLite database file itself will be located in the `var` subdirectory, which is created as part of the script.

```

<?php
define('FILENAME', 'demo1.db');
❶ define('FOLDER_PATH', __DIR__ . '/../var/');

if (!file_exists(FOLDER_PATH)) {
    mkdir(FOLDER_PATH);
}

$connection = new \PDO(
    ❷ 'sqlite:' . FOLDER_PATH . FILENAME
);

$sql = 'CREATE TABLE IF NOT EXISTS product ('
    ❸ . 'id integer PRIMARY KEY AUTOINCREMENT,'
    . 'description text,'
    . 'price float'
    . ')';
$stmt1 = $connection->prepare($sql);
$stmt1->execute();

$sql = "INSERT INTO product (description, price) VALUES ('hammer', 9.99)";
$stmt2 = $connection->prepare($sql);-
$stmt2->execute();

$sql = "INSERT INTO product (description, price) VALUES ('ladder', 59.99)";
$stmt3 = $connection->prepare($sql);
$stmt3->execute();

```

Listing 28-2: A script to create our SQLite database

First, we define constants for the database filename ('demo1.db') and folder path. Remember that a double dot (..) in a path refers to a parent directory, so ../var indicates that var should be at the same hierarchy level as the running script's directory ❶. We create this directory if it doesn't already exist.

Then we create a new `PDO` object, once again passing a DSN string as an argument to provide information about the database we want to connect to ❷. This time, the DSN string begins with `'sqlite:'`, telling PDO that we want to connect to an SQLite server, followed by the full filepath, including the directory path and filename, to the desired database. Unlike with MySQL, we don't need to write and execute an SQL statement creating the database schema; if necessary, the SQLite database file is created when the connection is established. Also, since SQLite is just working with a file, there's no need for any username or password.

Once PDO establishes a database connection, it mostly doesn't matter what DBMS it's working with, so the rest of the script is virtually identical to [Listing 28-1](#): we create and execute SQL statements to create the `product` table and add two entries to it. The only difference is that SQLite uses the keyword `AUTOINCREMENT` with no underscore (unlike MySQL's `AUTO_INCREMENT`) ❸.

As with the MySQL version, you need to run this script to create and populate the SQLite database schema. Enter `php db/create_databaseSQLite.php` at the command line.

Writing the PHP Classes

Now we'll organize the logic of our simple web application by writing some PHP classes. For now, we need three. As usual, we'll create an `Application` class to serve as a front controller for the application. We'll also write a `Product` class with properties corresponding to the fields in our database's `product` table, to make it easy to move data back and forth between the database and our PHP code. Finally, we'll design a `Database` class to encapsulate the logic of creating a database connection. Not only will this help keep our code tidy and object-oriented, but it will also enable us to easily refactor the application to switch between MySQL and SQLite with little to no effect on the rest of the code.

We'll start with the `Application` class. Declare this class in `src/Application.php`, as shown in [Listing 28-3](#).

```
<?php
namespace Mattsmithdev;
```

```

use Mattsmithdev\Product;

class Application
{
    ❶ private ?\PDO $connection;

    public function __construct()
    {
        $db = new Database();
        ❷ $this->connection = $db->getConnection();
    }

    public function run()
    {
        if (NULL != $this->connection){
            ❸ $products = $this->getProducts();
            print '<pre>';
            var_dump($products);
            print '</pre>';
        } else {
            print '<p>Application::run() - sorry '
                . '- there was a problem with the database c
onnection';
        }
    }

    public function getProducts(): array
    {
        $sql = 'SELECT * FROM product';
        $stmt = $this->connection->prepare($sql);
        $stmt->execute();
        ❹ $stmt->setFetchMode(\PDO::FETCH_CLASS, Product::clas
s);
        $products = $stmt->fetchAll();

        return $products;
    }
}

```

Listing 28-3: The `Application` class

We start by giving the class a private `connection` property ❶. This property has the nullable data type of `?\PDO`, so it will be either a reference to a `PDO` database connection object or `NULL` (if the connection fails). In the class's constructor method, we create a new `Database` object and invoke its `getConnection()` method (we'll define that class and method shortly).

Next, we store the resulting database connection reference into the `Application` class's `connection` property ❷. This may seem more roundabout than directly connecting to a database as we did earlier in the setup code, but relegating the details of establishing the connection to the `Database` class allows this `Application` class to work regardless of the DBMS we're using.

We next declare the application's `run()` method. In it, we test the `connection` property to make sure it isn't `NULL` and invoke the `getProducts()` method if it isn't ❸, which returns an array of `Product` objects retrieved from the database. For simplicity, we print the array preceded by an HTML `<pre>` tag. (We'll refine this project to output valid HTML when we expand the application later in the chapter.) If `connection` is `NULL`, we print an error message instead.

We close out the class by declaring the `getProducts()` method. It uses the `connection` property to prepare and execute an SQL statement that selects all the rows from the database's `product` table. The raw results of this query are in the `PDOStatement` object referenced by the `$stmt` variable, but we want to represent the results as `Product` objects.

This is where the PDO library's object fetch mode comes in handy. We set it up by invoking `$stmt->setFetchMode()` ❹, passing the `\PDO::FETCH_CLASS` constant as a first argument to indicate that we want the results to be objects of a class. The second argument, `Product::class`, tells PDO which (namespaced) class to use. The `::class` magic constant returns the fully qualified class name string (in this case, `'Mattsmithdev\\Product'`). Then we invoke `$stmt->fetchAll()` to retrieve the results. Since we selected multiple rows from the database, this creates an array of `Product` objects rather than just a single object. We return this array via the `$products` variable.

Now we'll create the `Product` model class. The properties of this class must correspond to the columns of the `product` database table (that is, have the same names and data types) for PDO to be able to successfully return query results as `Product` objects. Save the contents of [Listing 28-4](#) in *src/Product.php*.

```
<?php
namespace Mattsmithdev;

class Product
{
    private int $id;
    private string $description;
    private float $price;
}
```

Listing 28-4: The `Product` class

All this code does is declare three private properties for the class (`id`, `description`, and `price`) with names and data types matching the fields in the `product` table. That's all the PDO library needs in order to retrieve rows from the table as objects of this class. Since for now our application is simply using `var_dump()` to display an array of `Product` objects, we never need to access the class's private properties. When we expand the application, we'll add accessor methods to the `Product` class so that we can write a more elegant template page that loops through and outputs each object's properties in customizable and valid HTML.

Finally, let's declare the `Database` class to manage the process of establishing and storing a live MySQL database connection. Create the file *src/Database.php* containing the contents of [Listing 28-5](#).

```
<?php
namespace Mattsmithdev;

class Database
{
    ❶ const MYSQL_HOST = 'localhost';
```

```

const MYSQL_PORT = '3306';
const MYSQL_USER = 'root';
const MYSQL_PASSWORD = 'passpass';
const MYSQL_DATABASE = 'demo1';

const DATA_SOURCE_NAME = 'mysql:dbname=' . self::MYSQL_
DATABASE
    ❷ . ';host=' . self::MYSQL_HOST . ':' . self::MYSQL_PO
RT;

    ❸ private ?\PDO $connection;

public function getConnection(): ?\PDO
{
    return $this->connection;
}

public function __construct()
{
    ❹ try {
        $connection = new \PDO(
            self::DATA_SOURCE_NAME,
            self::MYSQL_USER,
            self::MYSQL_PASSWORD
        );
        $this->connection = $connection;
    } catch (\Exception $e) {
        ❺ print "Database::__construct() - Exception '
            . '- error trying to create database connect
ion";
    }
}
}

```

Listing 28-5: *The Database class*

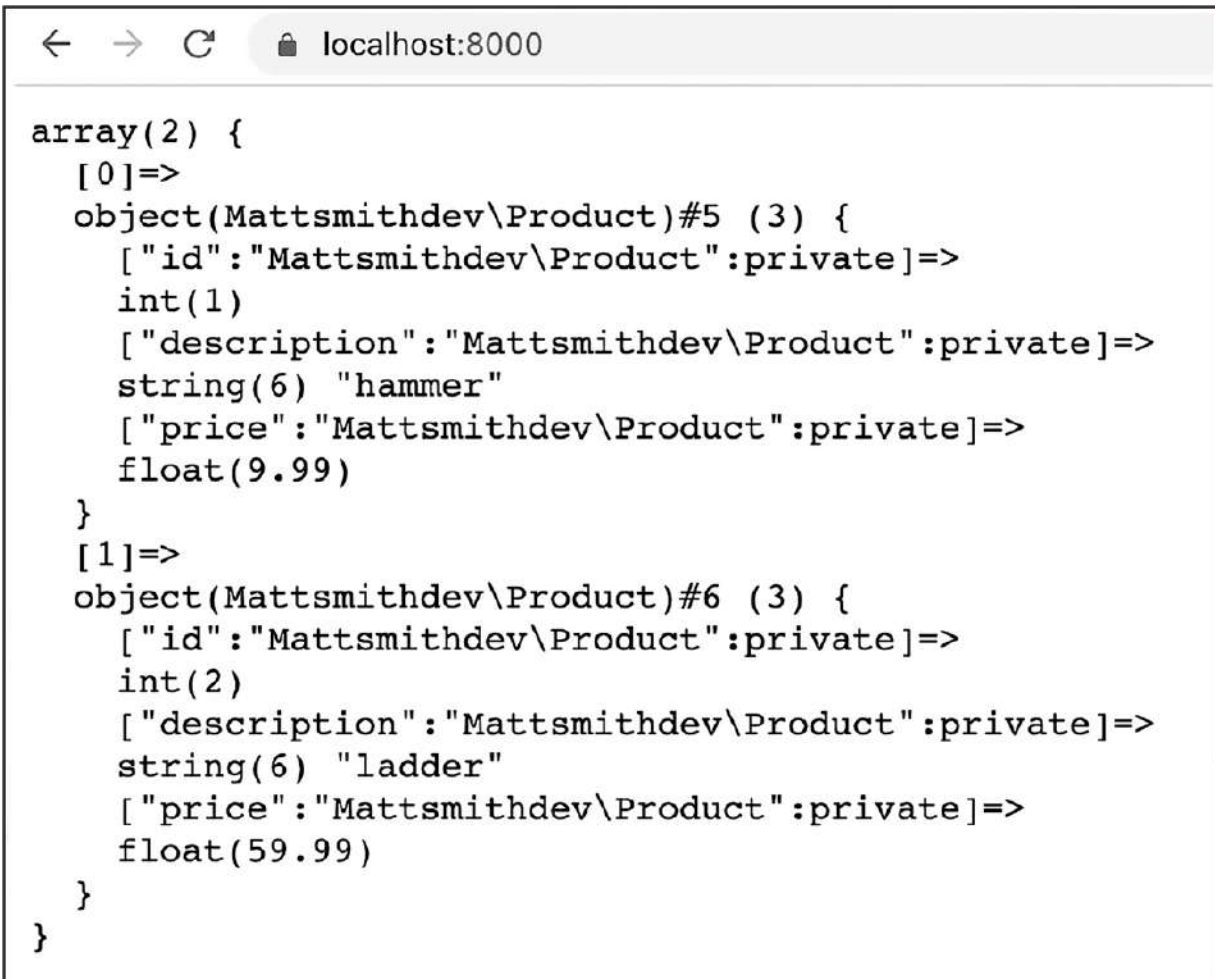
We declare class constants for the five individual pieces of data needed to create a live connection to a MySQL database ❶: the host (`localhost`), the port (`3306`), the MySQL username and password (fill these in as

appropriate), and the name of the database schema we want to work with (`demo1`). Then we combine some of these constants into another constant representing the DSN string that we'll need to pass as the first argument when creating the `PDO` object ❷.

We next declare a private `connection` property for the class ❸, along with a public `getConnection()` method to return its value. This property has the nullable data type of `?\PDO`, so it will be either `NULL` or a reference to a `PDO` object.

In the `Database` class's constructor method, we attempt to connect to the MySQL database by creating a new `PDO` object, using the class constants to provide the necessary DSN string, username, and password. A reference to the database connection is stored in the class's `connection` property. These actions are wrapped inside a `try` statement ❹, so any exception thrown in the process will be caught ❺ and an error message will be printed out. Therefore, whenever a new `Database` object is created (from within the `Application` class), the constructor method will attempt to connect to the database. A subsequent call to `getConnection()` will return either a `PDO` connection object or `NULL` if a problem occurred when creating the connection.

With that, we're ready to run the application and see the results. When you visit the localhost server running the web application, you should see something like [Figure 28-3](#).

A screenshot of a web browser window. The address bar shows 'localhost:8000'. The main content area displays a PHP var_dump() output for an array named \$products. The array has two elements. The first element is an object of class Product with id 1, description 'hammer', and price 9.99. The second element is an object of class Product with id 2, description 'ladder', and price 59.99. The browser's user interface includes back, forward, and refresh buttons in the top left, and a lock icon and address bar in the top right.

```
array(2) {  
  [0]=>  
  object(Mattsmithdev\Product)#5 (3) {  
    ["id":"Mattsmithdev\Product":private]=>  
    int(1)  
    ["description":"Mattsmithdev\Product":private]=>  
    string(6) "hammer"  
    ["price":"Mattsmithdev\Product":private]=>  
    float(9.99)  
  }  
  [1]=>  
  object(Mattsmithdev\Product)#6 (3) {  
    ["id":"Mattsmithdev\Product":private]=>  
    int(2)  
    ["description":"Mattsmithdev\Product":private]=>  
    string(6) "ladder"  
    ["price":"Mattsmithdev\Product":private]=>  
    float(59.99)  
  }  
}
```

Figure 28-3: The web page showing the contents of the `$products` array

At this stage, the web page doesn't look like much; all it shows is a `var_dump()` of the `$products` array. Since we haven't yet included any decision logic in the `run()` method of the `Application` class, this page will always be displayed, regardless of any URL-encoded request variables. However, the printed contents of the array indicate that we've successfully retrieved entries from the `products` MySQL database table and mapped them to objects of our custom `Product` class, an important first step in database-driven application development.

Switching from MySQL to SQLite

Earlier, we set up the same database schema in both MySQL and SQLite; what would it take to refactor our application to use the SQLite schema

rather than MySQL? We've designed the application so that all the DBMS-specific logic is encapsulated in the `Database` class, and we reference this class only once, in the constructor method for the `Application` class (see [Listing 28-3](#)). There we use the statement `$db = new Database()` to get a reference to a new `Database` object before invoking its `getConnection()` method to obtain a PDO database connection.

Let's replace this statement with one that creates an instance of a `DatabaseSQLite` class that will connect to SQLite instead of MySQL.

[Listing 28-6](#) shows the necessary change to *src/Application.php*.

```
--snip--
    public function __construct()
    {
        $db = new DatabaseSQLite();
        $this->connection = $db->getConnection();
    }
--snip--
```

Listing 28-6: Updating the `Application` class to create a `DatabaseSQLite` object instead of a `Database` object

Now we need to declare the `DatabaseSQLite` class to encapsulate the work of creating and storing a live SQLite database connection. For the rest of our application code to work as before, it needs to have a `getConnection()` method that returns a reference to a PDO connection object, just like the `Database` class. Create *src/DatabaseSQLite.php* containing the code in [Listing 28-7](#).

```
<?php
namespace Mattsmithdev;

class DatabaseSQLite
{
    const DB_DIRECTORY = __DIR__ . '/../var';
    const DB_FILE_PATH = self::DB_DIRECTORY . '/demo1.db';

    const DATA_SOURCE_NAME = 'sqlite:' . self::DB_FILE_PATH;
}
```

```

        private ?\PDO $connection = NULL;

        public function getConnection(): ?\PDO
        {
            return $this->connection;
        }

        public function __construct()
        {
            try {
                $this->connection = new \PDO(self::DATA_SOURCE_NAME);
            } catch (\Exception $e) {
                print 'DatabaseSQLite::__construct() - Exception
- '
                    . 'error trying to create database connection'
                    . PHP_EOL;
            }
        }
    }
}

```

Listing 28-7: The DatabaseSQLite class

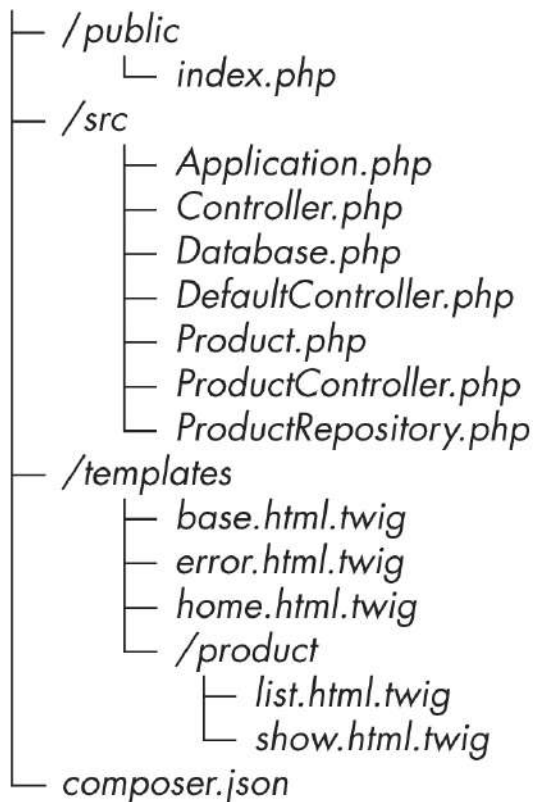
This new class follows the same contours as the old `Database` class; only the SQLite-specific details differ. We first declare constants for the data needed to create a live SQLite database connection: the location of the directory containing the database file (`DB_DIRECTORY`); the full filepath, including the directory location and filename (`DB_FILE_PATH`); and the DSN string, including the full filepath (`DATA_SOURCE_NAME`). Then we declare a private `connection` property with a nullable `\PDO` data type of `?\PDO` and its public `getConnection()` getter method, as before. Finally, we declare a constructor method that uses `try` and `catch` statements to attempt to create a new `PDO` database connection object and report any errors—again, just like the `Database` class.

Try running the web server again and you should see the application function exactly as before, displaying the contents of the `$products` array

that features data retrieved from the database. It's just that now we're using SQLite rather than MySQL. This switch required virtually no updates to the code aside from declaring the new `DatabaseSQLite` class and changing one line to create a `DatabaseSQLite` object instead of `Database`.

A Multipage Database-Driven Web Application

Now let's expand our database-driven web application to encompass multiple pages, including a home page, a product list page, a page for displaying details about a single product, and a page for showing error messages. We'll also use Twig templating to streamline the process of designing these pages. This expanded project will have the following file structure:



[Figure 28-4](#) shows the four pages of this website.

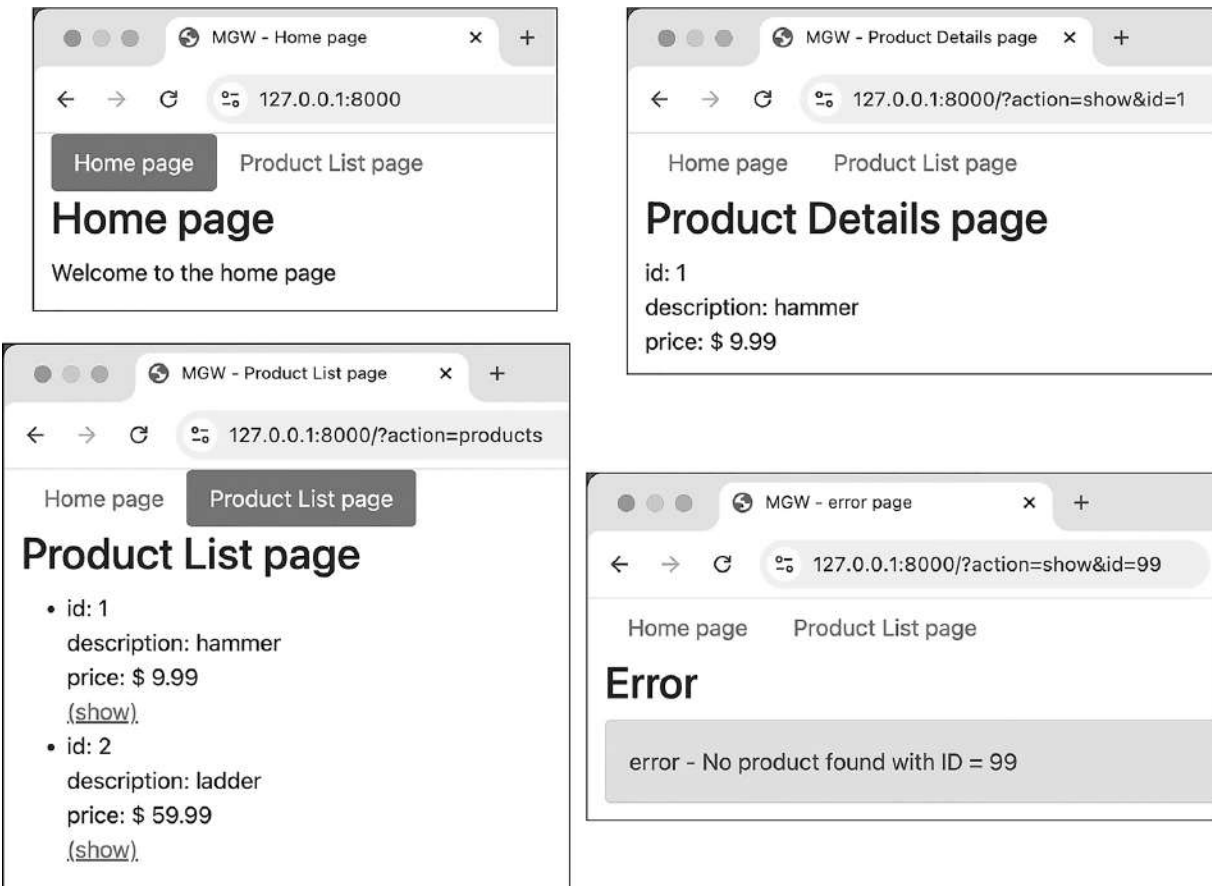


Figure 28-4: The expanded web application

The pages feature HTML formatted with Bootstrap and offer a navigation bar with links to the home page and the list of products. When the *(show)* link is clicked next to a listed product, the details of that product are displayed on a new page. Notice that the ID of the clicked product appears as part of the query string in the page's URL (for example, *id=1* on the page displaying information about the hammer). If an error occurs, such as a missing ID or an ID that doesn't match a row in the database, an error page will be shown with an appropriate error message; the example in [Figure 28-4](#) shows an ID of 99 in the browser address bar, and an error message stating that no product could be found with this ID.

Since the application will involve several actions, we'll create two controller classes: `ProductController` for listing all products and displaying a details page for an individual item, and `DefaultController` with methods to display the home page and the error message page. We'll

revert to using MySQL and the original `Database` class to manage the database connection, but keep in mind that you can always sub in SQLite by switching to the `DatabaseSQLite` class instead. Since the project will use Twig templating, be sure to run `composer require twig/twig` at the command line to add the Twig library to the project.

Managing the Product Information

Let's begin expanding the application by focusing on the classes that manage product information through database interaction. To start, we'll add getter and setter methods to the `Product` class. This is necessary since the application will now need to individually access a `Product` object's properties to display them in a more elegant way than a simple `var_dump()`. Update `src/Product.php` to match the contents of [Listing 28-8](#).

```
<?php
namespace Mattsmithdev;

class Product
{
    private int $id;
    private string $description;
    private float $price;

    public function getId(): int
    {
        return $this->id;
    }

    public function setId(int $id): void
    {
        $this->id = $id;
    }

    public function getDescription(): string
    {
        return $this->description;
    }
}
```



```

        public function setDescription(string $description): void
        {
            $this->description = $description;
        }

        public function getPrice(): float
        {
            return $this->price;
        }

        public function setPrice(float $price): void
        {
            $this->price = $price;
        }
    }

```

Listing 28-8: The `Product` class, now with getters and setters for each property

Here we add simple getter and setter methods for each of the three properties of the `Product` class: `id`, `description`, and `price`. That's a total of six new methods for this class.

We'll now start organizing our application code a little better by putting any logic pertaining to retrieving products from the database in a `ProductRepository` class. It's common in database-driven applications to use these sorts of classes, called *repository classes*, to separate logic accessing the database from the other logic in the application. The repository class methods take in and return objects, and handle any access to the database (with help establishing the database connection from the `Database` class itself). The rest of the application works with the resulting objects and has nothing at all to do with the database.

Our `ProductRepository` class will include a method to retrieve all the products, as we originally had in the `Application` class, as well as a new method to retrieve a single product with a given ID. (Often repository classes have methods for other database operations as well, such as adding, updating, or deleting entries. We'll discuss these operations in [Chapter 29](#).) Since this new class needs to interact with the database, it will now be

responsible for creating the necessary Database object. These changes will free up the main Application class to focus on controller logic.

Create the new ProductRepository class in *src/ProductRepository.php* as shown in [Listing 28-9](#). The black code is brand new; the grayed-out code has been taken from the Application class (see [Listing 28-3](#)).

```
<?php
namespace Mattsmithdev;

class ProductRepository
{
    private ?\PDO $connection = NULL;

    public function __construct()
    {
        $db = new Database();
        $this->connection = $db->getConnection();
    }

    public function findAll(): array
    {
        ❶ if (NULL == $this->connection) return [];

        $sql = 'SELECT * FROM product';
        $stmt = $this->connection->prepare($sql);
        $stmt->execute();
        $stmt->setFetchMode(\PDO::FETCH_CLASS, Product::class);

        $products = $stmt->fetchAll();

        return $products;
    }

    public function find(int $id): ?Product
    {
        if (NULL == $this->connection) return NULL;

        $sql = 'SELECT * FROM product WHERE id = :id';
```

```

        $stmt = $this->connection->prepare($sql);

        ❷ $stmt->bindParam(':id', $id);
        $stmt->execute();

        $stmt->setFetchMode(\PDO::FETCH_CLASS, Product::class);

        ❸ $product = $stmt->fetch();

        ❹ if ($product == false) {
            return NULL;
        }

        return $product;
    }
}

```

Listing 28-9: The new *ProductRepository* class

We declare a private `connection` property, followed by a constructor method that creates a new `Database` object and invokes its `getConnection()` method to retrieve a database connection. This is just like the original `Application` class. Next, we declare a `findAll()` method containing the logic from the `Application` class's `getProducts()` method. (Since this is now a method in the `ProductRepository` class, we don't need to use the word *product* in the method name.) The method starts with an extra line of code that tests whether the database connection is `NULL` and returns an empty array if it is ❶. This is necessary because the `Application` class no longer has access to the database connection. If the connection isn't `NULL`, the method retrieves all the products from the database and returns them as an array of `Product` objects, just like before.

We next declare the new `find()` method. It takes in an integer `$id` argument, which is used to retrieve a single `Product` object from the database table. Again, the first statement of this method is a `NULL` test on the database connection. If the connection is `NULL`, this method will immediately return `NULL` and finish executing. If the connection isn't `NULL`, we prepare the `'SELECT * FROM product WHERE id = :id'` string as an

SQL query. The `:id` at the end is a named PDO placeholder, consisting of a colon followed by an identifier (`id`) for a part of the SQL statement that needs to be filled in by the value of a variable.

We use the `bindParam()` method of the PDO statement object to connect the placeholder with the value of the `$id` argument ❷. This mechanism is what makes PDO's prepared statements safe from SQL injection attacks. The placeholder syntax forces the value of `$id` to be treated as a possible value to look for in the `id` column. The value of the variable can't possibly change the query itself into something more malicious.

The method uses PDO's object fetch mode to retrieve the result of the query as a `Product` object (or `NULL` if no product in the database has a matching ID). Notice that we obtain the `Product` object by calling `$stmt->fetch()` ❸ rather than `$stmt->fetchAll()` as we did in the `findAll()` method, since this time we're expecting only a single result. Since the `fetch()` method returns `false` (not `NULL`) on failure, we test for this value and return `NULL` if no object was successfully retrieved ❹.

Implementing the Controller Logic

Next, we'll focus on the classes that implement the application's controller logic. First, we'll remove the database-related code from the `Application` class (since that code now lives in `ProductRepository`) and update the class to take in requests and delegate them to the appropriate controller. Modify `src/Application.php` to match the contents of [Listing 28-10](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    private DefaultController $defaultController;
    private ProductController $productController;

    public function __construct()
    {
        $this->defaultController = new DefaultController();
```

```

        $this->productController = new ProductController();
    }

    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');

        switch ($action)
        {
            case 'products': ❶
                $this->productController->list();
                break;

            case 'show': ❷
                $id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT);
                if (empty($id)) { ❸
                    $this->defaultController->
error('error - To show a product, an integer ID must be provided');
                } else { ❹
                    $this->productController->show($id);
                }
                break;

            default: ❺
                $this->defaultController->homepage();
        }
    }
}

```

Listing 28-10: The updated `Application` class with simple front-controller logic

We declare two private `defaultController` and `productController` properties and then use the constructor to fill them with `DefaultController` and `ProductController` objects. Then we declare the `run()` method, which retrieves the value of `$action` from the incoming URL and passes it to a `switch` statement to decide what to do. If the value is

'products' ❶, we invoke the `list()` method of the `ProductController` object to display the page listing all the products.

If the value of `$action` is 'show' ❷, we want to display a page with details about just one product. For that, we attempt to extract an integer `$id` variable from the URL, using `FILTER_SANITIZE_NUMBER_INT` to remove any non-integer characters from the variable. If `$id` ends up being empty ❸, we display an error page by passing a string error message to the `error()` method of the `DefaultController` object. If the value of `$id` isn't empty ❹, we pass it to the `show()` method of the `ProductController` object to display the product page. Finally, we declare the `switch` statement's default action ❺, which is to display the home page by invoking the `homepage()` method of the `DefaultController` object.

Now we'll create an abstract `Controller` class that will become the superclass for both our controller classes, `DefaultController` and `ProductController`. Create *src/Controller.php* containing the contents of [Listing 28-11](#). Note that this class is just the same as Listing 22-8 on [page 436](#).

```
<?php
namespace Mattsmithdev;

use Twig\Loader\FilesystemLoader;
use Twig\Environment;

abstract class Controller
{
    const PATH_TO_TEMPLATES = __DIR__ . '/../templates';

    protected Environment $twig;

    public function __construct()
    {
        $loader = new FilesystemLoader(self::PATH_TO_TEMPLATES);
        $this->twig = new Environment($loader);
    }
}
```

```
}  
}
```

Listing 28-11: The `abstract Controller` superclass, providing a `twig` property

We declare this class to be `abstract` so that it can't be instantiated. We declare a class constant for the path to the Twig templates directory. Then we declare a `twig` property with `protected` visibility so that it will be available to methods of subclasses to this class. Within the class's constructor, we create two Twig objects: `FilesystemLoader` object and `Environment`. The latter holds the all-important `render()` method and is stored in the `twig` property, while the former helps the `Environment` object access the template files.

With this `Controller` superclass declared, we can now declare the subclasses that will inherit from it. We'll start with `DefaultController`, which will handle displaying the home page and error page. Declare the class in `src/DefaultController.php` as shown in [Listing 28-12](#).

```
<?php  
namespace Mattsmithdev;  
  
class DefaultController extends Controller  
{  
    ❶ public function homepage(): void  
    {  
        $template = 'home.xhtml.twig';  
        $args = [];  
        print $this->twig->render($template, $args);  
    }  
  
    ❷ public function error(string $message): void  
    {  
        $template = 'error.xhtml.twig';  
        $args = [  
            'message' => $message  
        ];  
        print $this->twig->render($template, $args);  
    }  
}
```

```
}  
}
```

Listing 28-12: The `DefaultController` class for simple page actions

We declare that this class extends `Controller` so that it will inherit the superclass's `twig` property. The class's `homepage()` method ❶ invokes the `render()` method of the inherited `twig` property to render the *home.xhtml.twig* template, then prints out the text received. Similarly, the `error()` method ❷ renders and prints the *error.xhtml.twig* template. For this template, we pass along the value of the `$message` argument so that the error page will include a custom error message.

Now we'll create the other controller subclass, `ProductController`, for displaying product-related pages. Create *src/ProductController.php* to match the code in [Listing 28-13](#).

```
<?php  
namespace Mattsmithdev;  
  
class ProductController extends Controller  
{  
    private ProductRepository $productRepository;  
  
    public function __construct()  
    {  
        parent::__construct();  
        $this->productRepository = new ProductRepository();  
    }  
  
    ❶ public function list(): void  
    {  
        $products = $this->productRepository->findAll();  
  
        $template = 'product/list.xhtml.twig';  
        $args = [  
            'products' => $products  
        ];  
        print $this->twig->render($template, $args);  
    }  
}
```



```

    }

    ❷ public function show(int $id): void
    {
        $product = $this->productRepository->find($id);

        if (empty($product)) {
            $defaultController = new DefaultController();
            $defaultController->error(
                'error - No product found with ID = ' . $id
            );
        } else {
            $template = 'product/show.xhtml.twig';
            $args = [
                'product' => $product
            ];
            print $this->twig->render($template, $args);
        }
    }
}

```

Listing 28-13: The *ProductController* class

Like `DefaultController`, this class is declared as extending `Controller`. Its constructor method first invokes the parent (`Controller`) constructor, which sets up the inherited `twig` property. Then the constructor creates a new `ProductRepository` object and stores it in the `productRepository` property. The class will be able to use this object to get product information from the database.

The class's `list()` method ❶ obtains an array of `Product` objects by invoking the `ProductRepository` object's `findAll()` method. Then it renders and prints the *list.xhtml.twig* template, passing along the array of products, to display the full product list page. The `show()` method ❷ is similar to `list()`, but it uses the provided integer `$id` argument to retrieve a single `Product` object from the database (via the `ProductRepository` object's `find()` method). Then it displays a page with just this product's information by rendering and printing the *show.xhtml.twig* template. The `if`

statement in this method tests whether `NULL` was received from the repository instead of an object, indicating no product with the provided ID exists in the database. If so, an error page will be displayed.

Designing the Templates

All that remains is to design the Twig templates for the application's various pages. The templates will all extend a common base template that defines the HTML skeleton for each page and includes the Bootstrap CSS stylesheets. We'll write that base template first. Create *templates/base.xhtml.twig* containing the Twig code in [Listing 28-14](#).

```
<html lang="en">
<head>
    <title>MGW - {% block title %}{% endblock %}</title> ❶
    <meta name="viewport" content="width=device-width">
    <link rel="stylesheet"
        href="https://cdn.jsdelivrivr.net/npm/bootstrap@5.1.
3/dist/css/bootstrap.min.css">
</head>

<body class="container">
<ul class="nav nav-pills"> ❷
    <li class="nav-item">
        <a class="nav-link {% block homeLink %}{% endblock
%}" href="/">Home page</a>
    </li>
    <li class="nav-item">
        <a class="nav-link {% block productLink %}{% endbloc
k %}"
            href="/?action=products">Product List page</a>
    </li>
</ul>

{% block body %} ❸
{% endblock %}
</body></html>
```

Listing 28-14: *The top-level base.xhtml.twig template*

The HTML `<head>` element includes a Twig `title` block where the page title can be inserted ❶. Each individual page template will declare its own title to be appended to the `MGW` text immediately before this block. The `<head>` element also contains a link to download the Bootstrap 5 minimized stylesheets from <https://www.jsdelivr.com>.

Inside the HTML `<body>`, we declare an unordered list styled with the `nav nav-pills` CSS class to represent the navigation bar at the top of each page ❷. We want the bar to include links to the home page and the product list. We declare each item as a list element styled with the `nav-item` CSS class and an anchor link element styled with the `nav-link` CSS class. The class declaration for each anchor link element features an empty Twig block (called `homeLink` and `productLink`, respectively), which we can override in the page templates to add the `active` CSS class. This way, the current page will be highlighted in the navigation bar. The base template ends with a `body` Twig block, where we'll fill in the page-specific content ❸.

Now that we have the base Twig template, we can begin declaring the individual child templates, starting with the home page. Create the new Twig template `templates/home.xhtml.twig` containing the code in [Listing 28-15](#).

```
{% extends 'base.xhtml.twig' %}

{% block title %}Home page{% endblock %}

{% block homeLink %}active{% endblock %}

{% block body %}
    <h1>Home page</h1>
    <p>
        Welcome to the home page
    </p>
{% endblock %}
```

Listing 28-15: The `home.xhtml.twig` template for the home page

We first declare that this template extends the base template. As such, the file is very short, since we have to fill in only the page-specific content.

We override the `title` block with the text content `home page` so that the page's title will be `MGW - home page`. Then we override the `homeLink` block with the text content `active`, so the home page link will appear as a colored button when this template page is displayed. Finally, we override the `body` block with a basic heading and paragraph.

Now we'll create the Twig template for the error page. Enter the contents of [Listing 28-16](#) into `templates/error.xhtml.twig`.

```
{% extends 'base.xhtml.twig' %}

{% block title %}error page{% endblock %}

{% block body %}
    <h1>Error</h1>
    <p class="alert alert-danger">
        {{message}}
    </p>
{% endblock %}
```

Listing 28-16: The error.xhtml.twig template for the error page

First, we override the `title` block with the text content `error page`, making this page's title `MGW - error page`. Then we override the `body` block with a heading and paragraph. The paragraph is styled with the `alert alert-danger` CSS class to make it a nicely spaced, pink warning message to the viewer. The text content of this paragraph is the Twig `message` variable, which will be passed in via the `$args` array from the `DefaultController` class's `error()` method. [Figure 28-5](#) shows this error page displayed in a web browser.

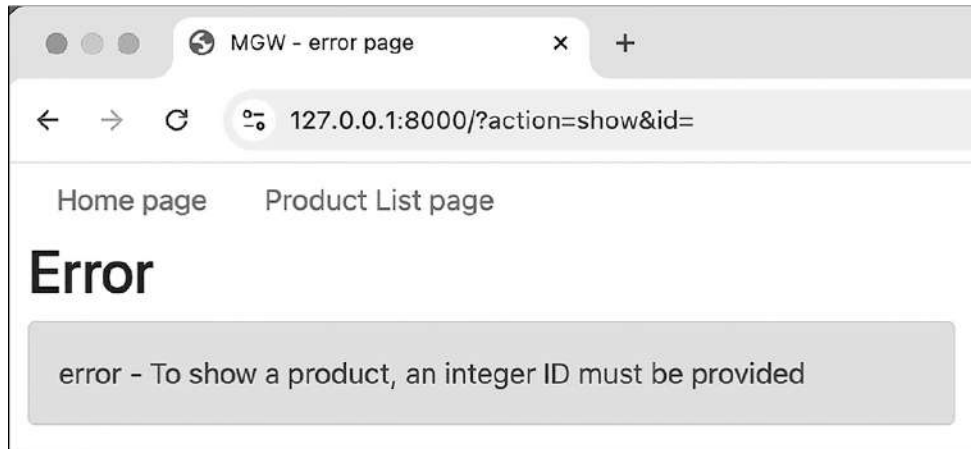


Figure 28-5: The error page when an integer product ID hasn't been provided

Next, we'll create a Twig template for the Product List page in *templates/product/list.xhtml.twig*.

To prepare for templates for other model classes, we create a subdirectory of *templates* named *product*, and we will create our templates for the `Product` class here. Also, since templates for the `Product` class are in this folder, we don't need to prefix the template names with the class name. For example, we can name this list template *list.xhtml.twig* rather than *productList.xhtml.twig* and so on.

The code for *templates/product/list.xhtml.twig* is shown in [Listing 28-17](#).

```
{% extends 'base.xhtml.twig' %}

{% block title %}Product List page{% endblock %}

{% block productLink %}active{% endblock %}

{% block body %}
    <h1>Product List page</h1>

    <ul>
        ❶ {% for product in products %}
            <li>
                id: {{product.id}}
                <br>
            </li>
        {% endfor %}
    </ul>
</block %}
```

```

        description: {{ product.description }}
        <br>
        ❷ price: $ {{product.price | number_format(2)}}
        <br>
        ❸ <a href="/?action=show&id={{product.id}}">(show)
    </a>
</li>

    {% endfor %}
</ul>
{% endblock %}

```

Listing 28-17: The list.xhtml.twig template for listing all products

We override the `title` block with the text content `product list page` and override the `productLink` block with the text content `active`, much as we did for the home page. Then, inside the `body` block, we use a Twig for loop ❶ to iterate over the `Product` objects in the `products` variable, formatting each one as a list item in an unordered list. (These `Product` objects were passed to the template in an array as part of the `list()` method of the `ProductController` class.)

We extract each property of each object individually for display on its own line, using Twig's double curly bracket notation. For example, `{{product.id}}` accesses the `id` property of the current `Product` object. Notice that we format the product's price to include two decimal places ❷. Each product's list item ends with an anchor link element to show the details page for just that one product ❸. We insert the product's ID into the link's URL. This ID, in turn, will be passed to the `find()` method of the `ProductRepository` class to retrieve just that product's information from the database.

Finally, we'll create the individual Product Details page template in `templates/product/show.xhtml.twig`. [Listing 28-18](#) shows how.

```

{% extends 'base.xhtml.twig' %}

{% block title %}Product Details page{% endblock %}

```

```
{% block body %}
    <h1>Product Details page</h1>

    id: {{product.id}}
    <br>
    description: {{ product.description}}
    <br>
    price: $ {{product.price | number_format(2)}}
{% endblock %}
```

Listing 28-18: The show.xhtml.twig template to display a single product's details

This template is simpler than the main Product List page template, since only a single `Product` object's properties are being displayed inside the `body` block. The object is the Twig `product` variable that was passed to this template from the `ProductController` class's `show()` method. Just as in the Product List page template, we output each of the object's properties individually, accessing them via double curly bracket notation.

With that, the application is complete. Try launching it and visiting its four pages, clicking the (*show*) link to view the Product Details page about each product. You should see that the application is able to retrieve all the products from the database, or just one of the products based on the appropriate integer ID. You should also be able to toggle between MySQL and SQLite simply by substituting the `Database` class with the `SQLiteDatabase` class.

Summary

In this chapter, we explored the basics of PHP's built-in PDO library for interacting with databases. We used the library to create a database schema and to insert data into a table. We then retrieved data from a table, mapping the results of a query to objects of a PHP model class using PDO's object fetch mode. We also used prepared SQL statements, which add a layer of protection against SQL injection attacks.

We integrated our database schema into a well-organized, multipage web application. The code to manage the database connection was abstracted into a suitable class, either `Database` or `SQLiteDatabase`, allowing us to switch seamlessly between MySQL and SQLite as the

application's DBMS. The logic to retrieve data from the database and into `Product` objects was encapsulated in the `ProductRepository` class, front-controller logic was placed in an `Application` class, and logic for displaying simple pages was located in the `DefaultController` class. Actions relating to requests for one or many products went into the `ProductController` class, which used `ProductRepository` methods to query the database.

This architecture could easily be scaled up, with additional repository, model, and controller classes for other database tables (users, customers, orders, suppliers, and so on). The entire application was styled using Twig templating, with a base (parent) template to efficiently share common page elements across all the individual child templates.

Exercises

1. Create a new model class called `Book` with the following properties:

- `id` (integer), an auto-incrementing primary key
- `title` (string)
- `author` (string)
- `price` (float)

Create a setup script (based on [Listings 28-1](#) and [28-2](#)) to create a database with a `book` table, mapped to the `Book` class's properties and types (use `int`, `text`, and `float` for the SQL data types). Also insert at least two book records into the database table with title, author, and price properties of your choice. Then write an object-oriented project (or adapt the example from this chapter) to retrieve and list all the books from the database. The project should include the following:

- A `Database` class to create a connection to your database

- A `public/index.php` script that creates an `Application` object and invokes its `run()` method

- An `Application` class with a `run()` method to get an array of `Book` objects and `var_dump()` them

- Your `Book` model class

2. Extend your project from Exercise 1 as follows:

- a. Add getter and setter accessor methods for each property in your `Book` class.
- b. Change your `Application` logic to test for an `action` variable in the URL. The default action should be to display a home page using an appropriate template. If the action is `books`, the application should display a page with information about all the books, using an appropriate template.

PROGRAMMING CRUD OPERATIONS



In the preceding chapter, we began developing a database-driven web application, with a focus on learning how to read data from the database. However, reading is just one of the four primary database operations known collectively as *CRUD*, short for *create*, *read*, *update*, *delete*. In this chapter, we'll look at the other components of *CRUD* as we expand our web application. We'll write code that allows users to change the database data by deleting, adding, or updating entries through interactive links and web forms.

Just about any database-driven mobile or web application revolves around the four *CRUD* operations. Take an email app as an example: when you write a new email and send it, this *creates* an item in the database representing the receiver of the email, as well as an item in your own system's Sent mailbox database. It's common to *read* or *delete* email in

your inbox, and you may also draft an email and then *update* it later before sending (or deleting) it.

As we start adding the remaining CRUD features to our web application, you'll notice a pattern. Each change will begin with a new case in the front-controller `switch` statement inside the `Application` class, invoking a new method in the `ProductController` class. This method, in turn, will call a new repository class method, where the actual database interaction will take place. Finally, we'll update the appropriate page templates to add the necessary user interface for the new feature.

Deleting Data

Sometimes we need to delete data from a database table. For example, a car manufacturer may stop making a particular model of a car. The model's details might be copied into an archive database table, and then the model is deleted from the main table of car models. To delete data from a table, we use the `DELETE` SQL keyword. If no criteria are given, all records are deleted from the named table.

When deleting a specific row or rows matching certain criteria, we need to provide an SQL `WHERE` clause. For example, to delete the row with an ID of 4 from a `model` table, the SQL statement would be as follows:

```
DELETE FROM model WHERE id = 4
```

We'll look at examples of deleting an entire table and selectively deleting entries from a table in this section.

Deleting Everything from a Table

Let's first add a feature to our web application from the previous chapter that deletes all the products from the `products` table in the database. [Figure 29-1](#) shows the Delete All Products link we'll create, along with a pop-up confirmation dialog. It's always a good idea to offer users a chance to reconsider and cancel destructive operations such as permanently deleting data (assuming they have the option to delete data at all).

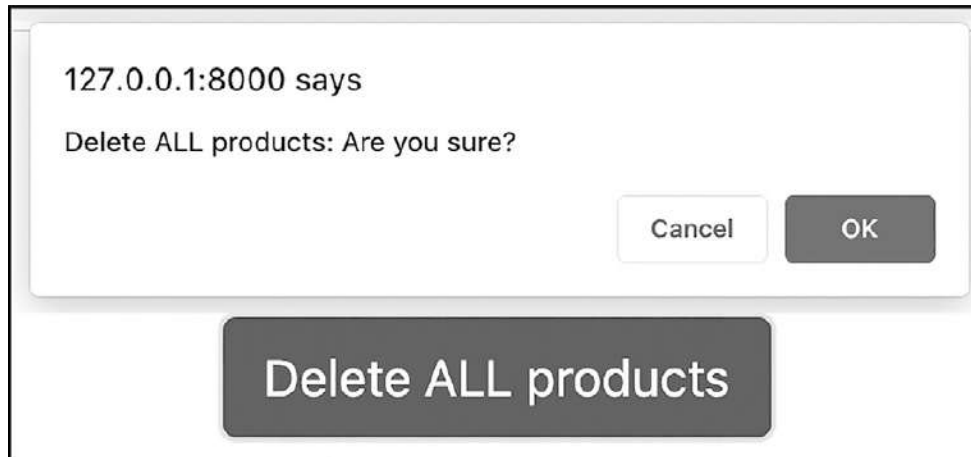


Figure 29-1: Deleting all products

First, we'll add a new route to detect a `POST` submission with the `action =deleteAll` variable. Update the `run()` method in the `Application` class to match [Listing 29-1](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    --snip--
    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        $isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
        'POST');

        switch ($action)
        {
            --snip--
            case 'deleteAll':
                if ($isPostSubmission) {
                    $this->productController->deleteAll();
                } else {
                    $this->defaultController->
```

```

        error('error - not a POST request');
    }
    break;

    default:
        $this->defaultController->homepage();
    }
}
}

```

Listing 29-1: The updated *Application* class to act on the *deleteAll* action

We add a new `$isPostSubmission` variable that will be `true` if the received request uses the `POST` method. While it's technically possible to write a web application that changes the server state (such as the database contents) in response to `GET` messages, this would violate the definition of the HTTP `GET` method. For this reason, we'll use the `POST` method and an HTML `<form>` element for any database-changing requests (deletions, creations, or updates) in this chapter.

We next add a new case to the front-controller `switch` statement for when the value of `action` in the URL is `'deleteAll'`. When this action is received through a `POST` request, we invoke the `deleteAll()` method of the `ProductController` object. If `$isPostSubmission` is `false`, we instead use the `defaultController` to return an error message to the user.

We'll define the `deleteAll()` method next. Update `src/ProductController.php` to match the contents of [Listing 29-2](#).

```

<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;

    --snip--

    public function deleteAll(): void
    {

```

```
$this->productRepository->deleteAll();

$this->list();
}
}
```

Listing 29-2: Adding the `deleteAll()` method to `ProductController`

We declare the `ProductController` class's `deleteAll()` method to in turn invoke the `deleteAll()` method of the `ProductRepository` class (which manages communication with the database). Then we invoke the `list()` method to make the application display the Product List page, using the `header()` function and location URL `/?action=products`. The user should therefore see an empty list of products after they've all been deleted.

Now we'll add the `deleteAll()` method to the `ProductRepository` class. Update `src/ProductRepository.php` as shown in [Listing 29-3](#).

```
<?php
namespace Mattsmithdev;

class ProductRepository
{
    private ?\PDO $connection = NULL;
    --snip--

    public function deleteAll(): int
    {
        if (NULL == $this->connection) return 0;

        $sql = 'DELETE FROM product';
        $stmt = $this->connection->prepare($sql);
        $stmt->execute();
        ❶ $numRowsAffected = $stmt->rowCount();

        return $numRowsAffected;
    }
}
```

Listing 29-3: Adding the `deleteAll()` method to `ProductRepository`

We declare the `deleteAll()` method to return an integer value indicating the number of rows deleted from the database. If the connection is `NULL`, we return 0. Otherwise, we declare, prepare, and execute the `'DELETE FROM product'` SQL query string, which deletes every entry from the `product` table. Then we invoke the `rowCount()` method of the PDO statement object ❶, which returns the number of rows affected by the most recently executed query. We return this integer value at the end of the method.

Finally, we need to update the template for the Product List page to offer a link for deleting all the products. Update `templates/product/list.xhtml.twig` to match the contents of [Listing 29-4](#).

```
{% extends 'base.xhtml.twig' %}
{% block title %}product list page{% endblock %}
{% block productLink %}active{% endblock %}

{% block body %}
    <h1>Product list page</h1>
    <ul>
        {% for product in products %}
            --snip--
        {% endfor %}
    </ul>
    <p>
        <form method="POST" action="/?action=deleteAll">
            <button class="btn btn-danger m-1"
                onclick="return confirm('Delete ALL products: Are you sure?');">
                Delete ALL products</button>
        </form>
    </p>
{% endblock %}
```

Listing 29-4: The `list.xhtml.twig` template for listing all products

Here we add a paragraph to the end of the template declaring a `POST` method form containing a Bootstrap-styled button with the text `Delete ALL`

products. The action for our controller to receive (`deleteAll`) is sent through the form's `action` attribute. This button includes a pop-up confirmation message (launched by the JavaScript `confirm()` function) declared in its `onclick` attribute, so the user will be able to confirm or cancel the request.

Deleting Individual Items by ID

Just as we can show a particular product based on its ID, we can also use an ID to specify which individual product to delete from the database. Let's add that feature now. [Figure 29-2](#) shows a screenshot of the page we'll create: each product in the Product List page will get its own Show and Delete button-styled links, each triggering a database action based on the product's ID.

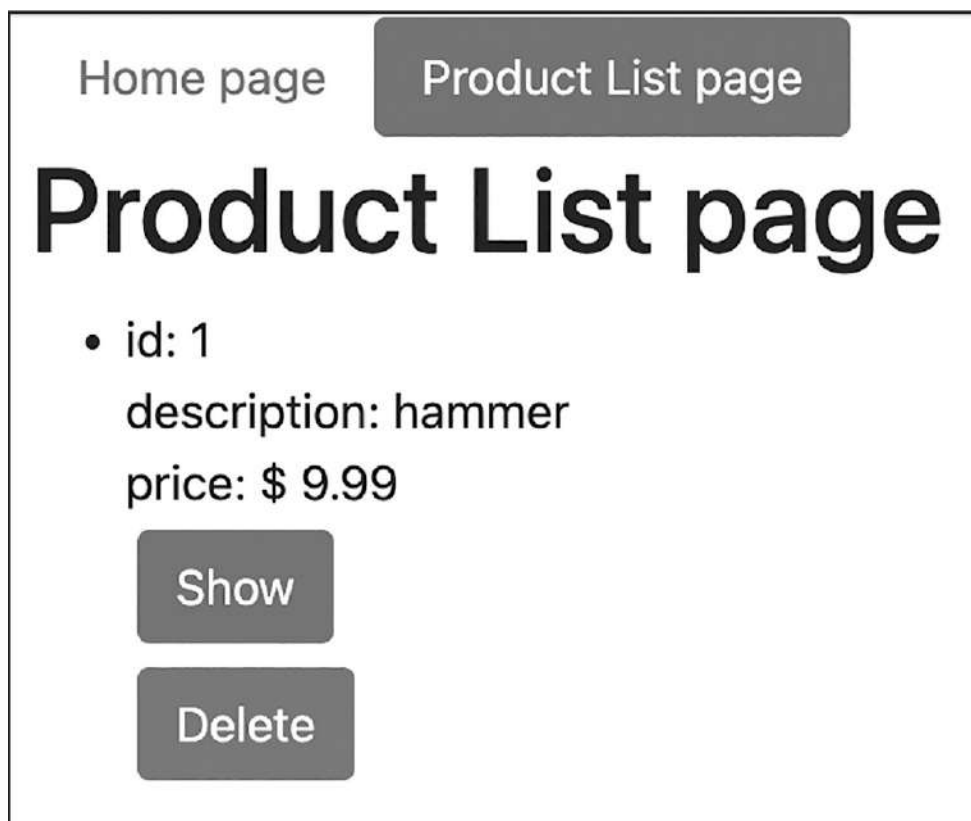


Figure 29-2: The Show and Delete buttons for an individual product

We first need to add a new route URL pattern of `action=delete`, where the ID of the product to be deleted is passed through a POST form submission as a variable `id`. Update the `Application` class code to match [Listing 29-5](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    --snip--
    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        $isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
        'POST');

        switch ($action)
        {
            --snip--

            case 'delete':
                ❶ $id = filter_input(INPUT_POST, 'id',
                                FILTER_SANITIZE_NUMBER_IN
T);

                if ($isPostSubmission && !empty($id)) {
                    $this->productController->delete($id);
                } else {
                    $this->defaultController->error('error -
to delete a
                                product an integer id must be pro
vided by a
                                POST request');
                }
                break;

            default:
                $this->defaultController->homepage();
        }
    }
}
```

```
}  
}
```

Listing 29-5: The `Application` class, updated to act on the `delete` action

Here we add a new case in the front-controller `switch` statement for when the value of `action` in the URL is `'delete'`. In this case, we attempt to extract an integer variable `id` from the `POST` variables received in the request ❶. If `$isPostSubmission` is `true` and the ID isn't empty, we pass the ID to the `delete()` method of the `ProductController` object. Otherwise, we pass an appropriate error message to the `error()` method of the `DefaultController` object for display.

To define the `delete()` method, update `src/ProductController.php` according to [Listing 29-6](#).

```
<?php  
namespace Mattsmithdev;  
  
class ProductController extends Controller  
{  
    private ProductRepository $productRepository;  
  
    --snip--  
  
    public function delete(int $id): void  
    {  
        $this->productRepository->delete($id);  
  
        $this->list();  
    }  
}
```

Listing 29-6: Adding the `delete()` method to `ProductController`

The `delete()` method takes in an integer product ID and passes it to the `delete()` method of the `ProductRepository` object. Then it makes the application display the Product List page via the `list()` method. The user

should therefore see the list of products, less the one deleted, after clicking the link to delete an item.

Now we'll add the `delete()` method to the `ProductRepository` class in `src/ProductRepository.php`. [Listing 29-7](#) shows how.

```
<?php
namespace Mattsmithdev;

class ProductRepository
{
    private ?\PDO $connection = NULL;

    --snip--

    public function delete(int $id): bool
    {
        if (NULL == $this->connection) return false;

        $sql = 'DELETE FROM product WHERE id = :id';
        $stmt = $this->connection->prepare($sql);
        $stmt->bindParam(':id', $id);
        $success = $stmt->execute();

        return $success;
    }
}
```

Listing 29-7: Adding the `delete()` method to `ProductRepository`

The `delete()` method takes in an integer argument (the ID) and returns a Boolean indicating the success of the deletion. If the connection is `NULL`, we return `false`. Otherwise, we declare the SQL query string `'DELETE FROM product WHERE id = :id'` to delete just the product with the specified ID. We then prepare the statement and bind the `$id` argument to the `:id` placeholder. Executing the statement then produces a Boolean success value, which we store and return.

Finally, we need to update the Product List template to offer the button-styled Show and Delete links for each product. Modify

templates/product/list.xhtml.twig to match the contents of [Listing 29-8](#).

```
{% extends 'base.xhtml.twig' %}
{% block title %}product list page{% endblock %}
{% block productLink %}active{% endblock %}

{% block body %}
    <h1>Product list page</h1>

    <ul>
        {% for product in products %}
            <li class="mt-5">

                id: {{product.id}}
                <br>
                description: {{product.description}}
                <br>
                price: $ {{product.price | number_format(2)}}
                <br>
                <a href="/?action=show&id={{product.id}}"
❶ class="btn btn-secondary m-1">Show</a>
                <br>
                <form method="POST" action="/?action=delete">
                    ❷ <input type="hidden" name="id" value="{{product.id}}">
❸ <button class="btn btn-danger m-1"
                        onclick="return confirm(
                            'Delete product with ID =
{{product.id}}:
Are you sure?');">
                        Delete</button>
                </form>
            </li>
        {% else %}
            <li>
                (there are no products to display)
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

```
</ul>

<p>
    <form method="POST" action="/?action=deleteAll">
--snip--
```

Listing 29-8: The list.xhtml.twig template offering deletion by ID

We style the existing `Show` link as a secondary button ❶. Then we declare a `POST` submission form with `action=delete` and a button `Delete`, passing `id` as a hidden variable filling in the product ID with the Twig `{{product.id}}` placeholder ❷. As with the `Delete ALL products` form, this form button includes a pop-up confirmation message declared in an `onclick` attribute, so the user will be able to confirm or cancel the request. We also add a Twig `else` block ❸ so that the message (there are no products to display) is shown if no products are found in the database.

Creating New Database Entries

Let's turn to the *C* in CRUD: creating new database entries by using SQL `INSERT` statements. For example, here's an SQL statement that inserts a new row into a table called `cat`:

```
INSERT INTO cat (name, gender, age) VALUES ('fluffy', 'female', 4)
```

Three values are provided for the `name`, `gender`, and `age` columns. The `INSERT` SQL statement requires us to first list the sequence of column names, and then follow this with the values to be inserted into those columns.

We touched on how to create new database entries when we first set up our application's database with its two initial products in “Setting Up the Database Schema” on [page 543](#). Now we'll make the process interactive by adding a form to our application that allows users to define new products and submit them to the database. [Figure 29-3](#) shows the form we'll create.

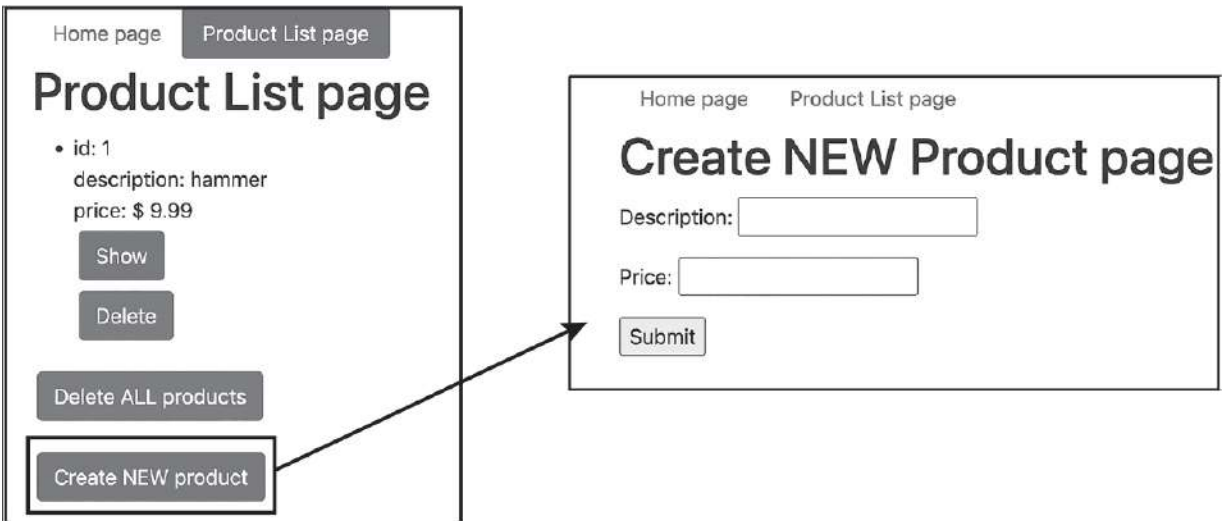


Figure 29-3: The link to create a new product and its associated form

We'll add a button link at the bottom of the Product List page to create a new product. This link will launch a Create NEW Product page with form fields for submitting the new product's description and price.

Adding Products Through a Web Form

To offer the Create NEW Product page form feature, we first need to add two new route actions to the application, one to display the form (`action=create`) and one to process the form submission (`action=processCreate`). [Listing 29-9](#) shows how to add cases for these actions to the front controller in the `Application` class.

```
<?php
namespace Mattsmithdev;

class Application
{
    --snip--
    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        $isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
        'POST');
```

```

switch ($action)
{
--snip--

    case 'create': ❶
        $this->productController->create();
        break;

    case 'processCreate': ❷
        $description = filter_input(INPUT_POST, 'description');
        $price = filter_input(INPUT_POST, 'price', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_FRACTION);

        if ($isPostSubmission && !empty($description) && !empty($price)) {❸
            $this->productController->processCreate($description, $price);
        } else {
            $this->defaultController->error('error - new product needs a description and price (via a POST request)');
        }
        break;

    default:
        $this->defaultController->homepage();
}
}
}

```

Listing 29-9: Adding the 'create' and 'processCreate' routes to the front controller

First, we add a new case in the front-controller switch statement for when the value of action in the URL is 'create' ❶. This invokes the create() method of the ProductController object.

Next, we declare the case for the 'processCreate' action ❷. For that, we retrieve the description and price values from the POST submission variables. Notice the use of two filters for the float price variable; the `FILTER_FLAG_ALLOW_FRACTION` argument is required to permit the decimal-point character.

If `$isPostSubmission` is true and both `$description` and `$price` are not empty ❸, the description and price are passed to the `processCreate()` method of the `ProductController` object. Otherwise, an appropriate error message will be displayed using the `error()` method of the `DefaultController` object.

This example is assuming that the `product` database table uses auto-incrementing to choose a new, unique integer ID when a new row is added, as demonstrated in [Chapter 28](#). Without this feature, we'd also have to supply an ID for the new product, perhaps using logic that first finds the highest current ID in the database and then adds 1 to it.

We'll now add the `create()` and `processCreate()` methods to the `ProductController` class. Update `src/ProductController.php` to match [Listing 29-10](#).

```
<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;

    --snip--

    ❶ public function create(): void
    {
        $template = 'product/create.xhtml.twig';
        $args = [];
        print $this->twig->render($template, $args);
    }

    ❷ public function processCreate(string $description, float
```



```

$price): void
{
    $this->productRepository->insert($description, $price);

    $this->list();
}
}

```

Listing 29-10: Adding the `create()` and `processCreate()` methods to `ProductController`

The `create()` method ❶ simply renders the `templates/product/create.xhtml.twig` template to display the new product form (we'll create this template shortly). The `processCreate()` method ❷ takes in a string for the new description and a float for the new price and passes them along to the `insert()` method of the `ProductRepository` object for insertion into the database. Then `processCreate()` invokes the Product List page via the `list()` method so that the user will see the updated list of products, including the newly created one.

If we were being completely correct, our `processCreate()` method would not call the `list()` method, but instead would force a redirect, sending a new request to the server to list all products. By not redirecting, we'll get a problem: if the user refreshes their browser page after submitting a form, the form will be submitted a second time. However, adding redirects now would make our work in the next section more complex, so we'll just call the `list()` method for now and formulate a better redirect solution at the end of this chapter.

To add the `insert()` method to the `ProductRepository` class, update `src/ProductRepository.php` as shown in [Listing 29-11](#).

```

<?php
namespace Mattsmithdev;

class ProductRepository
{
    private ?\PDO $connection = NULL;

```

```

--snip--

public function insert(string $description, float $price): int
{
    if (NULL == $this->connection) return -1;

    $sql = 'INSERT INTO product (description, price)'
        . ' VALUES (:description, :price)';
    $stmt = $this->connection->prepare($sql);

    $stmt->bindParam(':description', $description);
    $stmt->bindParam(':price', $price);

    $success = $stmt->execute();

    ❶ if ($success) {
        return $this->connection->lastInsertId();
    } else {
        return -1;
    }
}

```

Listing 29-11: Adding the `insert()` method to `ProductRepository`

The new `insert()` method takes in a string argument (`$description`) and a float argument (`$price`) and returns an integer—either the ID of the newly created database record or `-1` if no record is created. If the database connection is `NULL`, we return `-1` right away. Otherwise, we declare and prepare the SQL query string `'INSERT INTO product (description, price) VALUES (:description, :price)'` to add a new entry to the `product` table.

We then bind the `$description` argument to the `:description` placeholder and the `$price` argument to the `:price` placeholder before executing the statement. Finally, we test the Boolean `$success` value from the execution ❶. If `true`, we use the `lastInsertId()` method of the PDO connection object to return the ID of the most recently inserted database

entry, which should correspond to the new product. If `false`, we return `-1` instead.

Now let's revise the Product List page template to include the link for adding a new product. Update *templates/product/list.xhtml.twig* to match the contents of [Listing 29-12](#).

```
{% extends 'base.xhtml.twig' %}
{% block title %}product list page{% endblock %}
{% block productLink %}active{% endblock %}

{% block body %}
    --snip--
    Delete ALL products</button>
</form>
</p>

<p>
    <a href="/?action=create" class="btn btn-secondary m
-1">
        Create NEW product
    </a>
</p>
{% endblock %}
```

Listing 29-12: Adding the new product link to the list.xhtml.twig template

Here we add a paragraph to the end of the template containing a Bootstrap button–styled link with the text `Create NEW product`. The link triggers the `create` URL action.

Now let's add the Twig template to display the Create NEW Product page form. Create the *templates/product/create.xhtml.twig* template file containing the code shown in [Listing 29-13](#).

```
{% extends 'base.xhtml.twig' %}

{% block title %}create product page{% endblock %}

{% block body %}
```

```

    <h1>Create NEW Product page</h1>

    ❶ <form method="POST" action="/?action=processCreate">
        <p>
            Description:
            <input name="description">
        </p>
        <p>
            Price:
            <input name="price" type="number" min="0" step="0.0
1">
        </p>
        <input type="submit">
    </form>
{% endblock %}

```

Listing 29-13: The create.xhtml.twig template for the new product form

This template presents an HTML form ❶ whose submit action is `action=processCreate`, so the submitted values will be passed along to the `processCreate()` method of the `ProductController` class described earlier. The form contains two paragraphs, for the description and price, and then a Submit button.

Highlighting the Newly Created Product

When something is changed, it's helpful to highlight the change to the user. Let's update our application to highlight the new product in the product list after it's been added to the database. [Figure 29-4](#) shows the effect we want to achieve; it shows we've added a very expensive bag of nails costing \$999!

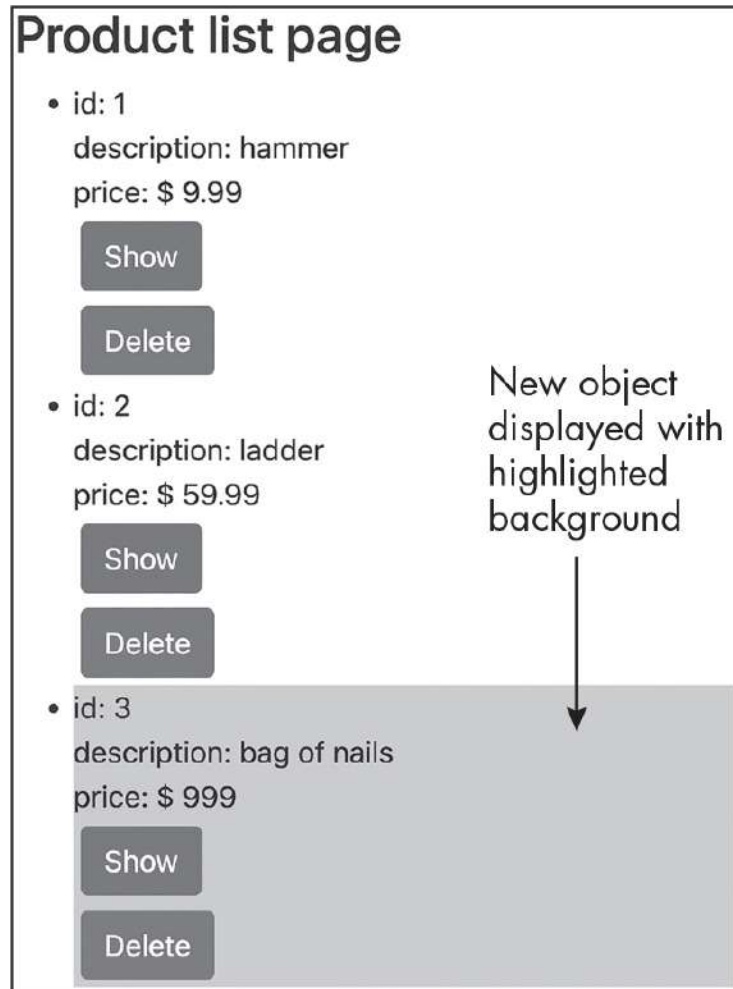


Figure 29-4: Displaying the newly created product with a highlighted background

To implement this feature, we can take advantage of the return value from the `ProductRepository` class's `insert()` method, which we declared in the preceding section. This value indicates the ID of the newly created product, so we can add logic to the application to highlight the product whose ID matches this value. First, we need to update the `ProductController` class, shown in [Listing 29-14](#).

```
<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;
```

```

public function list(?int $newProductId = NULL): void
{
    $products = $this->productRepository->findAll();

    $template = 'product/list.xhtml.twig';
    $args = [
        'products' => $products,
        ❶ 'id' => $newProductId
    ]
    print $this->twig->render($template, $args);
}

--snip--

public function processCreate(string $description, float
$price): void
{
    $newProductId =
        $this->productRepository->insert($description,
$price);

    $this->list($newProductId);
}
}

```

Listing 29-14: Updating the `list()` and `update()` methods in the `ProductController` class

We update the `list()` method (which displays the complete product list) to take in an optional `$newProductId` parameter with a default value of `NULL`. We pass this parameter to the Twig Product List template, along with the array of products ❶. Next, we update the `processCreate()` method to receive the new product ID returned from `insert()` and pass it along to the `list()` method.

Now we can update the Product List template to highlight the product matching the `id` variable passed to the template. Since the product IDs start at 1 and auto-increment, a value of `-1` will never match an object retrieved from the database, so the `list()` method's default `$newProductId`

parameter value of `-1` will result in no products being highlighted. Modify *templates/product/list.xhtml.twig* as shown in [Listing 29-15](#).

```
{% extends 'base.xhtml.twig' %}
{% block title %}product list page{% endblock %}
{% block productLink %}active{% endblock %}

{% block body %}
    <h1>Product list page</h1>

    <ul>
        {% for product in products %}

            ❶ {% if id == product.id %}
                {% set highlight = 'active' %}
            {% else %}
                {% set highlight = '' %}
            {% endif %}

            ❷ <li class="{{highlight}}">

                id: {{product.id}}
                <br>

--snip--

        {% endblock %}
```

*Listing 29-15: Updating the *list.xhtml.twig* template to highlight the newly added product within the list*

We've added a Twig `if` statement inside the loop through the products that sets the Twig `highlight` variable to `'active'` if the ID of the current product matches the received Twig variable `id` ❶. Otherwise, the `highlight` variable is set to an empty string. We include the value of `highlight` in the CSS style classes for each list item ❷, so each product will either be highlighted or not, as appropriate.

Finally, we need to add a `<style>` element for the `active` CSS class in the base template. Update */templates/product/base.xhtml.twig* according to

[Listing 29-16.](#)

```
<!doctype html>
<html lang="en">
<head>
  <title>MG- - {% block title %}{% endblock %}</title>
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.
3/dist/css/bootstrap.min.css">
  <style>
    li.active {background-color: pink;}
  </style>

</head>

<body class="container">
--snip--
```

Listing 29-16: Declaring a `<style>` element in the `base.xhtml.twig` Twig template

In the `<head>` element, we add a CSS rule that `active` list items should have a pink background.

Updating a Database Entry

The last CRUD operation to explore is the *U* for *update*. This operation is necessary since the data in a database constantly needs to be changed to reflect changes in the real world, such as a person's new address, the increase in the price of a product, a user changing their subscription status, and so on. To modify an existing record in a table, we can use the SQL `UPDATE` keyword. For example, here's an SQL statement that changes the age of a cat to 5, for the row whose ID is 1:

```
UPDATE cat SET age = 5 WHERE id = 1
```

Let's add a way to update an existing product to our web application. Much like creating a new product, we'll do this through a web form. [Figure](#)

[29-5](#) shows how this new feature will work.

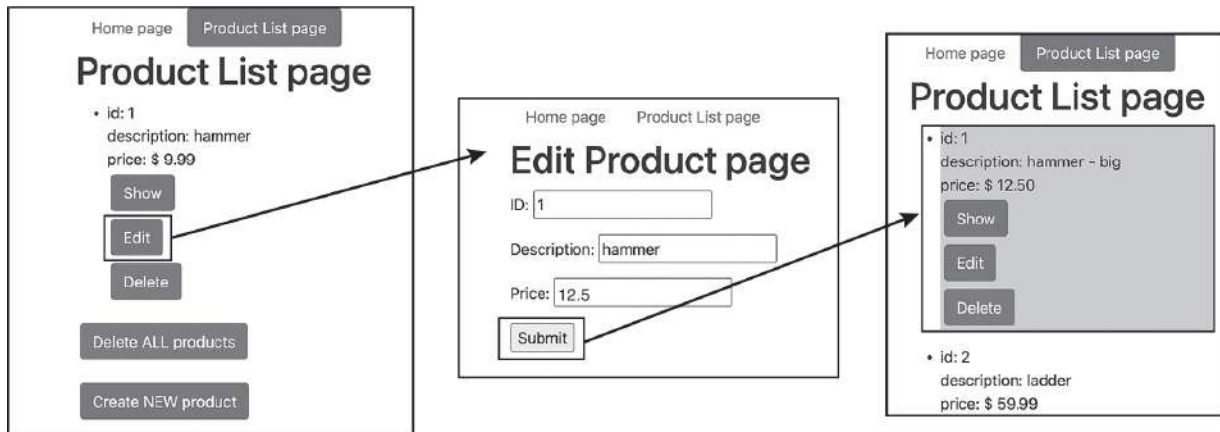


Figure 29-5: Updating an existing product

We'll add an Edit button to each product in the Product List page, which will take the user to an Edit Product page with form fields to modify the product's description and price (the ID will be read-only). These fields will start out with the current values filled in. Once the changes are submitted, the newly updated product will be highlighted on the Product List page.

To implement this feature, we must first add two new route actions, one to display the editing form (`action=edit`) and one to process the form submission (`action=processEdit`). [Listing 29-17](#) adds these two new cases to the front controller in the `Application` class.

```
<?php
namespace Mattsmithdev;

class Application
{
    --snip--
    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        $isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
        'POST');
```

```

switch ($action)
{
    --snip--

    case 'edit': ❶
        $id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT);
        if (empty($id)) {
            $this->defaultController->error(
                'error - To edit a product, an integer ID must be provided');
        } else {
            $this->productController->edit($id);
        }
        break;

    case 'processEdit': ❷
        $id = filter_input(INPUT_POST, 'id', FILTER_SANITIZE_NUMBER_INT);
        $description = filter_input(INPUT_POST, 'description');
        $price = filter_input(INPUT_POST, 'price', FILTER_SANITIZE_NUMBER_FLOAT,
                                FILTER_FLAG_ALLOW_FRACTION);

        if ($isPostSubmission && !empty($id) && !empty($description)
            && !empty($price)) {
            $this->productController->processEdit($id, $description, $price);
        } else {
            $this->defaultController->error(
                'error - Missing data (or not POST method) when trying to update product');
        }
        break;

```

```

        default:
            $this->defaultController->homepage();
        }
    }
}

```

Listing 29-17: Adding the 'edit' and 'processEdit' routes to the front controller

We add a new case in the front-controller `switch` statement for when the value of `action` in the URL is 'edit' ❶. As with the 'show' and 'delete' cases, we attempt to extract an integer `id` variable from the URL-encoded variables received in the request. If the value of `id` is empty, we display an appropriate error message by passing a string message to the `error()` method of the `DefaultController` object. If the value isn't empty, we pass it to the `edit()` method of the `ProductController` object.

Next, we add the 'processEdit' case ❷, which starts by retrieving `id`, `description`, and `price` from the `POST` submitted variables. If `$isPostSubmission` is true and all three variables (`id`, `description`, and `price`) aren't empty, we pass the values to the `processEdit()` method of the `ProductController` object. Otherwise, we again display an appropriate error message by using the `error()` method of the `DefaultController` object.

Now we'll add the new methods to the `ProductController` class. Update `src/ProductController.php` to match the contents of [Listing 29-18](#).

```

<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;

    --snip--

    public function edit(int $id): void
    {
        ❶ $product = $this->productRepository->find($id);
    }
}

```

```

        $template = 'product/edit.xhtml.twig';
        $args = [
            'product' => $product
        ];
        print $this->twig->render($template, $args);
    }

    public function processEdit(int $id, string $description,
                                float $price): void
    {
        ❷ $this->productRepository->update($id, $description,
        $price);

        $this->list($id);
    }
}

```

Listing 29-18: Adding the `edit()` and `processEdit()` methods to `ProductController`

The `edit()` method uses the provided integer `$id` argument to retrieve a single `Product` object from the database ❶. Then it passes this object to the `/templates/product/edit.xhtml.twig` template, which displays the form for editing the product. The `processEdit()` method takes in an `$id` integer, `$description` string, and `$price` float and passes them to the `update()` method of the `ProductRepository` object ❷. Then it makes the application display to the Product List page via the `list()` method. As with the `processCreate()` method, we pass the ID of the updated product to `list()` so that product will be highlighted.

Listing 29-19 shows how to add the new `update()` method to the `ProductRepository` class.

```

<?php
namespace Mattsmithdev;

class ProductRepository
{

```

```

private ?\PDO $connection = NULL;

--snip--

public function update(int $id, string $description, float $price): bool
{
    if (NULL == $this->connection) return false;
    $sql = 'UPDATE product SET description = :description, price = :price WHERE id=:id';

    $stmt = $this->connection->prepare($sql);

    $stmt->bindParam(':id', $id);
    $stmt->bindParam(':description', $description);
    $stmt->bindParam(':price', $price);

    $success = $stmt->execute();

    return $success;
}
}

```

Listing 29-19: Adding the `update()` method to the `ProductRepository` class

The `update()` method takes in a product's ID, description, and price, and returns a Boolean value indicating the success or failure of the update. If the database connection is `NULL`, then `false` is returned. Otherwise, we declare the SQL query string `'UPDATE product SET description = :description, price = :price WHERE id=:id'`, using the `WHERE` clause with the object's ID to specify the particular database row to be updated. After preparing the statement, we bind the `$id`, `$description`, and `$price` variables to their corresponding placeholders.

Then we execute the statement and return the resulting Boolean success value. Note that we're returning a Boolean here, rather than the product ID as we did previously for the `insert()` method. The difference here is that the calling method already knows the product ID in question, so it's

sufficient to simply return the true/false success of executing the database update statement.

Now we need to offer an Edit button for each product on the Product List page. Update the *templates/product/list.xhtml.twig* file as shown in [Listing 29-20](#).

```
--snip--
{% block body %}
    <h1>Product list page</h1>
    --snip--
    <li class="{{highlight}}">

        id: {{product.id}}
        <br>
        description: {{product.description}}
        <br>
        price: $ {{product.price | number_format(2)}}
        <br>
        <a href="/?action=show&id={{product.id}}"
            class="btn btn-secondary m-1">Show</a>
        <br>
        <a href="/?action=edit&id={{product.id}}"
            class="btn btn-secondary m-1">Edit</a>
        --snip--
    </li>
{% endblock %}
```

Listing 29-20: Adding an Edit button to the list.xhtml.twig template

Inside the Twig `for` loop for the current product, we add a Bootstrap button-styled link with the text `Edit`. This link for the `edit` action includes the `id` of the current product.

Finally, let's add the Twig template to display the form to edit the details of a product. Create *templates/product/edit.xhtml.twig* containing the code shown in [Listing 29-21](#).

```
{% extends 'base.xhtml.twig' %}

{% block title %}edit product page{% endblock %}
```

```

{% block body %}
    <h1>Edit Product page</h1>

    ❶ <form method="POST" action="/?action=processEdit">
        <p>
            ID:
            ❷ <input name="id" value="{{product.id}}" readonly
        >
        </p>
        <p>
            Description:
            <input name="description" value="{{product.description}}">
        </p>
        <p>
            Price:
            <input name="price" value="{{product.price}}" type
            ="number"
                min="0" step="0.01">
        </p>
        <input type="submit">
    </form>
{% endblock %}

```

Listing 29-21: The edit.xhtml.twig template for the form to edit a product

The main work of this template is to present an HTML form ❶ whose submit action is `action=processEdit`, so the submitted values will go to the `processEdit()` method of the `ProductController` class described earlier. This form contains three paragraphs, for the ID, description, and price, and then a Submit button. The ID, description, and price form inputs are populated with the values of the `Product` object for those properties. The ID input has the `readonly` attribute; since we don't want the user to be able to edit this value, it's displayed but isn't editable ❷.

Avoiding Double Form Submission with Redirects

In our current implementation, we call the `list()` method after processing a form submission to add or edit a product. By passing a product ID to this method, we can make our template highlight the product that's been created or updated. If the user were to refresh their browser page after submitting a form, however, the browser would attempt to submit the form data a second time by repeating the HTTP `POST` request. We can avoid this problem by making the server *redirect* to request the Product List page after a form submission (a `GET` request for the URL `?action=products`). If the page is refreshed, this `GET` request to list all products will be repeated rather than the `POST` request. This technique is sometimes called the *post-redirect-get (PRG) pattern*.

Let's update our application to use this redirect approach. Rather than passing the product ID as an argument to the `list()` method, we'll need to store the ID in the `$_SESSION` array. As we discussed in [Chapter 14](#), this is a special array for storing data about the user's current browser session. First, we'll update the `list()` method and add a new session helper method in `src/ProductController.php`, as shown in [Listing 29-22](#).

```
<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;

    --snip--

    public function list(): void
    {
        $products = $this->productRepository->findAll();
        ❶ $id = $this->getIdFromSession();

        $template = 'product/list.xhtml.twig';
        $args = [
            'products' => $products,
```



```

        'id' => $id
    ];
    print $this->twig->render($template, $args);
}

private function getIdFromSession(): ?int
{
    $id = NULL;
    ❷ if (isset($_SESSION['id'])) {
        $id = $_SESSION['id'];

        // Remove it now that it's been retrieved
        unset($_SESSION['id']);
    }

    return $id;
}
}

```

Listing 29-22: Updating the `list()` method and adding `getIdFromSession()` to the `ProductController` class

The `list()` method no longer has any parameters as input. Instead we attempt to retrieve an ID from the session by using the `getIdFromSession()` method ❶. This method initializes the `$id` variable to `NULL`, then tests whether the `$_SESSION` array contains a variable with a key of `'id'` ❷. If such a key exists, its value is retrieved and stored in `$id`, then that element of the array is unset so that it will no longer be stored in the session once retrieved. The method returns the value of `$id`, which will be either `NULL` or the value retrieved from the session.

Now we can update the `ProductController` class methods to use redirects after storing the ID in the session. Update these methods in `src/ProductController.php` as shown in [Listing 29-23](#).

```

<?php
namespace Mattsmithdev;

class ProductController extends Controller

```

```

{
    private ProductRepository $productRepository;

    --snip--

    public function delete(int $id): void
    {
        $this->productRepository->delete($id);

        ❶ $location = '/?action=products';
        header("Location: $location");
    }

    public function deleteAll(): void
    {
        $this->productRepository->deleteAll();

        $location = '/?action=products';
        header("Location: $location");
    }
    --snip--

    public function processCreate(string $description, float
$price): void
    {
        $newObjectId =
            $this->productRepository->insert($description,
$price);

        ❷ $_SESSION['id'] = $newObjectId;

        $location = '/?action=products';
        header("Location: $location");
    }

    --snip--

    public function processEdit(
        int $id, string $description, float $price): void

```

```

        {
            $this->productRepository->update($id, $description,
            $price);

            // Store ID of product to highlight in the SESSION
            ❸ $_SESSION['id'] = $id;

            $location = '/?action=products';
            header("Location: $location");
        }
    }
}

```

Listing 29-23: Updating the `POST` action methods to use redirects in `ProductController`

Both the `delete()` and `deleteAll()` methods have been updated to use the built-in `header()` function to redirect the server to process a `GET` request for the URL `/?action=products` ❶. This value of `action` will result in the products being listed by our `list()` method.

We've also updated `processCreate()` to store the ID of the newly created product (`$newObjectId`) in the session with key `'id'` ❷. Then it redirects the server to process a `GET` request for the URL `/?action=products`. Likewise, we've updated the `processEdit()` method to store the ID of the edited product's ID (`$id`) in the session with the key `'id'` ❸ and to redirect to `/?action=products` in the same way. We've now improved our web application to properly redirect after processing a `POST` form submission, so a refresh of the browser will not result in a repeat submission of the form data.

Since we're storing the ID in the session, we have to ensure that the session is started by our front-controller index script each time a request is received. Update `/public/index.php` as shown in [Listing 29-24](#).

```

<?php
require_once __DIR__ . '/../vendor/autoload.php';

session_start();

use Mattsmithdev\Application;

```

```
$app = new Application();  
$app->run();  
}
```

Listing 29-24: Updating index.php to ensure that sessions are active for our web application

We now call the `session_start()` function before any actions are taken. This ensures that our web application can store and retrieve values from the user's HTTP session.

Summary

In this chapter, we explored the full range of standard database operations: creating, reading, updating, and deleting entries, collectively known as *CRUD*. As in the preceding chapter, we used prepared statements for all our database queries, making it easy to bind parameters to the four actions such as deleting a row by its ID or inserting new values into multiple fields of a database entry.

We continued to see how much of the architecture of a database-driven web application is identical to that of a non-database application. The core of our code is focused around a front controller interrogating the requests received from the web client and invoking appropriate controller methods. By encapsulating our database actions in a repository class, we were able to keep the logic in our controller classes focused on responding to requests by arranging data and rendering the appropriate Twig template.

We also saw how forms for creating and updating database rows are presented and processed just like any other web form, only now the data is passed to the repository method to work with the database. We then saw how to improve the system by using redirects after processing form submissions to avoid a repeat of the form actions if the browser page happens to be refreshed.

Exercises

1. Open a web application you regularly use, such as a social media app or e-commerce site. Explore the actions available to you as a user and reflect on which CRUD operations are being executed for each action.

How is data coming from and being saved to the database sitting behind the web application you're using?

2. Create a CRUD web application for `Book` objects with these properties:

- `id` (integer), an auto-incrementing primary key

- `title` (string)

- `author` (string)

- `price` (float)

You can either create a new project from scratch or reuse classes from this chapter and the [Chapter 28](#) exercises. I suggest you follow this sequence when incrementally adding CRUD features to your application:

- a. List all objects.
- b. List one object, given an ID.
- c. Delete all objects.
- d. Delete one object, given an ID.
- e. Create a new object.
- f. Edit an object, given an ID.

ORM LIBRARIES AND DATABASE SECURITY



In this chapter, we'll explore techniques that make working with databases easier and more secure. First, much of the CRUD code in repository classes can become tedious and repetitive to write, varying only in terms of the names of the model classes and their properties. *Object-relational mapping (ORM) libraries* relieve this problem, automating lower-level work like preparing and executing SQL queries based on the way an application's model classes are named and structured. You'll see how to use such a library to simplify or replace our repository classes with just a few lines of code. We'll start by adding a simple ORM library to our example web application, then later integrate the professional-grade Doctrine ORM library with the project.

On the security front, adopting an ORM library will push us to remove any hardcoded database credentials from our code, instead placing those credentials in a separate data file. We'll also explore best practices for handling login information in a web application, including using *password hashing* to avoid storing plaintext passwords in a database. As you'll see, PHP provides built-in functions that make this process straightforward.

Simplifying Database Code with an ORM Library

One approach to making web applications communicate with databases is to design and write the necessary low-level code from scratch for each project. This includes the code to connect to database servers, create schemas and tables, and perform the four CRUD operations so that the database tables can store the data to support the application. Implementing this code requires careful analysis of the project requirements, especially the requirements for which data needs to be persisted to a database. The result is code that's tailored to the application at hand and can be written for computational efficiency to maximize speed.

We've followed this approach of designing and writing custom, application-specific database code in the last few chapters. It's been helpful for learning about how to work with a database, but it also comes with disadvantages. First, it takes time to design, write, and test code for every new application. Second, if the application requirements change, both the web application's database communication code and the database structure itself need to be changed accordingly. Finally, any new developers joining a software team for an ongoing project will have to learn all the details of the system's design to communicate with the database.

An alternative approach is to use an ORM library to abstract away the lower-level work of communication with the database. ORM libraries use the structure and associations of an application's model classes (often with a little additional metadata) to automatically create and update the structure of the corresponding database tables. If changes in the application requirements lead to changes in the model classes (perhaps new model classes are added, or existing classes are given new properties or associations), then the ORM library can automatically update the database

table structures accordingly and manage updated database queries based on the new model class declarations.

ORM libraries can be less computationally efficient than custom-written low-level database communication code. If speed isn't the most important feature for a web application, however, they have several strengths. For one, the database structure and queries are updated as soon as the model classes are updated, which streamlines the coding process. Also, if the project uses a well-known, industrial-standard ORM library, new developers joining a project will likely already be familiar with the abstracted ways to use the ORM library to handle database operations.

Before we get into the details of how to use an ORM library, let's consider an example that illustrates this approach's benefits, compared to using custom code. [Listing 30-1](#) shows an excerpt of the `ProductRepository` class developed in the previous two chapters.

```
<?php
namespace Mattsmithdev;

class ProductRepository
{
    private ?\PDO $connection = NULL;

    public function __construct()
    {
        $db = new Database();
        $this->connection = $db->getConnection();
    }

    public function findAll(): array
    {
        if (NULL == $this->connection) return [];

        $sql = 'SELECT * FROM product';
        $stmt = $this->connection->prepare($sql);
        $stmt->execute();
        $stmt->setFetchMode(\PDO::FETCH_CLASS, 'Mattsmithdev
\\Product');
        $products = $stmt->fetchAll();
    }
}
```



```

        return $products;
    }

    public function find(int $id): ?Product
    {
        if (NULL == $this->connection) return NULL;

        $sql = 'SELECT * FROM product WHERE id = :id';
        $stmt = $this->connection->prepare($sql);

        $stmt->bindParam(':id', $id);
        $stmt->execute();

        $stmt->setFetchMode(\PDO::FETCH_CLASS, 'Mattsmithdev
\\Product');
        $product = $stmt->fetch();

        return $product;
    }

    --snip--
}

```

Listing 30-1: *Some of the contents of the `ProductRepository` class*

We developed this class manually, meaning we had to implement lower-level methods such as a constructor to retrieve a database connection and CRUD methods such as `findAll()` and `find()` to prepare and execute SQL queries. Compare this code with [Listing 30-2](#), which declares an equivalent `ProductRepository` class with the help of an ORM library.

```

<?php
namespace Mattsmithdev;

use Mattsmithdev\PdoCrudRepo\DatabaseTableRepository;

class ProductRepository extends DatabaseTableRepository

```

```
{  
}
```

Listing 30-2: A `ProductRepository` class inheriting from an ORM library

For straightforward database interactions, an ORM library can do almost all the work for us. Instead of implementing custom methods in the repository class, we simply inherit those methods from an ORM library superclass (in this case, `DatabaseTableRepository`). The superclass is designed to use *reflection*, a technique of inspecting the classes and objects it interacts with, such as the `Product` model class. Then the superclass uses what it finds to generate SQL queries suitable for objects of those classes.

In the coming sections, we'll explore how this works in more detail by using a simple ORM library, one that I maintain as an open source project on GitHub. Later we'll also try out an industrial-strength ORM library called Doctrine, one of the most popular ORM libraries available for modern PHP. For now, though, take a moment to appreciate how much shorter the ORM-assisted `ProductRepository` class declaration is than the manually coded version.

Adding an ORM Library to a Project

Let's extend our database-driven web application from the previous chapters to work with a simple ORM library called `pdo-crud-for-free-repositories` that I maintain (<https://github.com/dr-matt-smith/pdo-crud-for-free-repositories>). It has limited features but is straightforward to use, making it a good tool for introducing the basics of ORM libraries. To get started, enter the following at the command line to add the library to the project:

```
$ composer require mattsmithdev/pdo-crud-for-free-repositories
```

This will add a *mattsmithdev* folder inside *vendor* containing the library code.

At the time of this writing, the released version of the `pdo-crud-for-free-repositories` is compatible only with MySQL, so we'll focus on

the MySQL version of our web application rather than the SQLite version.

Moving Database Credentials to a .env File

The ORM library we're using requires all our database credentials to be declared in a file named *.env*, commonly known as a *dotenv file*, rather than hardcoded in the `Database` class where we currently have them. Dotenv files are human-readable text files defining name/value pairs necessary for a program to run; other common file types for such variables include XML and YAML. This requirement is not a bad thing, since it also makes the application more secure.

Typically, we'd exclude dotenv files when using version-control systems such as Git so that when code is archived or pushed to open source projects, sensitive database credentials won't be included. This reduces the chance of a security breach from code being published or distributed to unauthorized people. Another advantage of this approach is that different environments can be set up in multiple dotenv files, such as for local development, remote development, testing, and live production systems.

To satisfy this ORM library requirement, create a file called *.env* and save it in the main project directory. Enter the contents of [Listing 30-3](#) into the file, changing values such as the password and port to match the MySQL server properties running on your computer.

```
MYSQL_USER=root
MYSQL_PASSWORD=password
MYSQL_HOST=127.0.0.1
MYSQL_PORT=3306
MYSQL_DATABASE=demo1
```

Listing 30-3: The database credentials in the .env file

These MySQL attributes were all previously defined as constants in our `Database` class. We can now delete the `Database` class from our project, since the ORM library comes with its own class for managing the connection with the database, based on the information in the dotenv file.

Relegating Product Operations to the ORM Library

Now that the ORM library has access to the database, we can shift responsibility for all CRUD operations relating to the `product` table from our `ProductRepository` class to the ORM library. We'll still use the `ProductRepository` class, but as hinted earlier, instead of manually filling it with methods that prepare and execute SQL statements, we'll simply declare it to be a subclass of one of the ORM library classes. Replace the contents of `src/ProductRepository.php` with the code shown in [Listing 30-4](#).

```
<?php
namespace Mattsmithdev;

use Mattsmithdev\PdoCrudRepo\DatabaseTableRepository;

class ProductRepository extends DatabaseTableRepository
{
}
```

Listing 30-4: The much-simplified `ProductRepository` class

The `use` statement specifies that we want to refer to the `DatabaseTableRepository` class in the `Mattsmithdev\PdoCrudRepo` namespace. Then we declare `ProductRepository` as a subclass of `DatabaseTableRepository`, with no code whatsoever in the class body. And that's it! We now have a working `ProductRepository` class with just those few lines of code. It will inherit all the methods declared in the `DatabaseTableRepository` superclass that happen to follow the same naming convention we used previously: `find()`, `findAll()`, `delete()`, `deleteAll()`, and so on.

But how does the `DatabaseTableRepository` class know that we want it to work with a `product` table with `id`, `description`, and `price` fields? This is where reflection comes into play. The `DatabaseTableRepository` class uses this technique to infer the details about how to construct appropriate SQL statements based on the classes and objects it comes into contact with. In this case, the reflection code assumes that the `ProductRepository` repository class manages database methods for a

model class in the same namespace called `Product`, and that a corresponding `product` table in the database has fields matching the property names of the `Product` class. As long as all the names align, the ORM library will be able to do its job.

For the reflection process to work, the `DatabaseTableRepository` methods need to receive objects of the appropriate model class, rather than free-floating variables as we'd previously designed the CRUD methods in [Chapter 29](#). To finalize the shift to the ORM library, we therefore need to refactor our `ProductController` class to pass in `Product` objects when calling the `insert()` and `update()` methods to process new and updated products. Change the `src/ProductController.php` file as shown in [Listing 30-5](#).

```
<?php
namespace Mattsmithdev;

class ProductController extends Controller
{
    private ProductRepository $productRepository;

    --snip--

    public function processCreate(string $description, float
$price): void
    {
        $product = new Product();
        $product->setDescription($description);
        $product->setPrice($price);

        ❶ $newObjectId = $this->productRepository->insert($pro
duct);

        $_SESSION['id'] = $newObjectId;

        $location = '/?action=products';
        header("Location: $location");
    }
}
```

```

--snip--

public function processEdit(int $id, string $description,
                           float $price): void
{
    ❷ $product = $this->productRepository->find($id);
    $product->setDescription($description);
    $product->setPrice($price);

    $this->productRepository->update($product);

    $_SESSION['id'] = $id;

    $location = '/?action=products';
    header("Location: $location");
}
}

```

Listing 30-5: The updated src/ProductController.php class

In the revised `processCreate()` method, we first create a new `Product` object and set its `description` and `price` properties to the values passed into the method. We then pass this `Product` object to the `ProductRepository` object's `insert()` method to add the new product to the database ❶. The ORM library assumes that every database table has an auto-incrementing primary key named `id`, so no value for the product ID is needed when creating a new row in the database. We make a similar change to the `processEdit()` method, using the `id` to get a reference to the `Product` object to be updated ❷, setting the other properties received from the form, and passing the object reference to the repository class's `update()` method.

Run the web server and you should now see the web application is working just as before, but with significantly less code! In this way, working with an ORM library greatly simplified the task of executing standard database CRUD operations.

NOTE

Before moving on, make a copy of your project at this point. In “The Doctrine ORM Library” on [page 615](#), we’ll modify that copy to use Doctrine.

Adding a New Database Table

Now that we’ve incorporated the ORM library into the project, let’s expand our web application by adding another table to the database. With the library handling all the CRUD operations, the process will be much more efficient than our effort in [Chapter 29](#) to get CRUD working for the `product` table. When we look at security in “Security Best Practices” on [page 608](#), we’ll discuss best practices for handling passwords, so we’ll go ahead and add a `user` table storing username and password information.

Along with the new database table, we’ll need a `User` model class, a `UserRepository` repository class (so named to match the ORM library’s requirements), a `UserController` controller class, and a page for displaying all the users. [Figure 30-1](#) shows that page. Of course, this page exists just to illustrate that the database methods are working; displaying a list of usernames and passwords is *not* an example of secure web development.

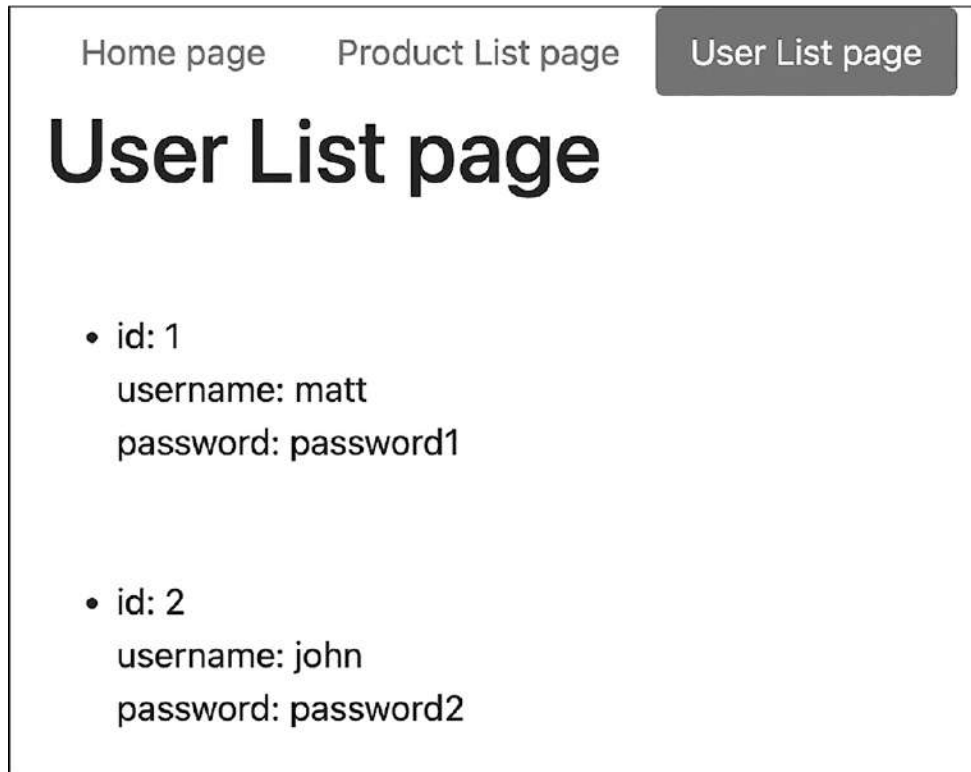


Figure 30-1: The User List page

We'll start by declaring the `User` model class. Add a `src/User.php` file containing the code in [Listing 30-6](#) to the project.

```
<?php
namespace Mattsmithdev;

class User
{
    private int $id;
    private string $username;
    private string $password;

    public function getId(): int
    {
        return $this->id;
    }

    public function setId(int $id): void
```



```

    {
        $this->id = $id;
    }

    public function getUsername(): string
    {
        return $this->username;
    }

    public function setUsername(string $username): void
    {
        $this->username = $username;
    }

    public function getPassword(): string
    {
        return $this->password;
    }

    public function setPassword(string $password): void
    {
        $this->password = $password;
    }
}

```

Listing 30-6: The *User* class

The class has an integer `id` property (a requirement for the ORM library) and string properties for `username` and `password`. We declare standard getter and setter methods for each of these properties.

Now we'll create the `UserRepository` class in `src/UserRepository.php`. As with `ProductRepository`, we'll have this class extend the ORM library's `DatabaseTableRepository` class. [Listing 30-7](#) shows the code.

```

<?php
namespace Mattsmithdev;

use Mattsmithdev\PdoCrudRepo\DatabaseTableRepository;

```

```
class UserRepository extends DatabaseTableRepository
{
}
```

Listing 30-7: The simple *UserRepository* class

We don't need to declare any methods for this repository class, since it will inherit all the necessary CRUD methods from the `DatabaseTableRepository` class. Thanks to the naming of the `UserRepository` and `User` classes, these CRUD methods will know to work with the `user` table in the database.

Next, let's create a `UserController` class, with a method to retrieve all users from the database and display them with a Twig template. Create `src/UserController.php` containing the code in [Listing 30-8](#).

```
<?php
namespace Mattsmithdev;

class UserController extends Controller
{
    private UserRepository $userRepository;

    public function __construct()
    {
        parent::__construct();
        $this->userRepository = new UserRepository();
    }

    public function list(): void
    {
        ❶ $users = $this->userRepository->findAll();

        $template = 'user/list.xhtml.twig';
        $args = [
            'users' => $users,
        ];
        print $this->twig->render($template, $args);
    }
}
```

```
}  
}
```

Listing 30-8: The `UserController` class declaring a `list()` method

We declare `UserController` as a subclass of `Controller` so that it will inherit a `twig` property for rendering templates. We declare a private `userRepository` property and initialize it in the constructor (where we also must first invoke the parent `Controller` class's constructor to set up the `twig` property). We then declare a `list()` method, which uses the `UserController` object's `findAll()` method (inherited from the ORM library) to retrieve all users from the database ❶. The results are returned as an array of objects, which we store as `$users`. We pass this array as the Twig variable `users` for rendering by the `templates/user/list.xhtml.twig` template.

Now we'll add a navigation bar link for the User List page to the base Twig template that all other templates inherit from. Update `templates/base.xhtml.twig` to match the contents of [Listing 30-9](#).

```
<!doctype html>  
<html lang="en">  
--snip--  
<body class="container">  
  
<ul class="nav nav-pills">  
  <li class="nav-item">  
    <a class="nav-link {% block homeLink %}{% endblock  
%}"  
      href="/">Home page</a>  
  </li>  
  <li class="nav-item">  
    <a class="nav-link {% block productLink %}{% endblock  
k %}"  
      href="/?action=products">Product List page</a>  
  </li>  
  <li class="nav-item">  
    <a class="nav-link {% block userLink %}{% endblock  
%}"
```

```

        href="/?action=users">User List page</a>
    </li>
</ul>

{% block body %}
{% endblock %}
</body></html>

```

Listing 30-9: Adding a user list link to /templates/base.xhtml.twig

Here we add a navigation list item with the text `User List page`. The anchor element has an action of `users`, and its CSS class attribute declares an empty Twig block called `userLink`. As with the other navigation bar items, this block can be overridden with the text `active` to highlight the link.

With the base template updated, we can now create the *templates/user/list.xhtml.twig* child template for the User List page. [Listing 30-10](#) shows how.

```

{% extends 'base.xhtml.twig' %}

{% block title %}User List page{% endblock %}

❶ {% block userLink %}active{% endblock %}

{% block body %}
    <h1>User List page</h1>

    <ul>
        ❷ {% for user in users %}
            <li class="mt-5">
                id: {{user.id}}
                <br>
                username: {{ user.username}}
                <br>
                password: {{ user.password}}
            </li>
        {% endfor %}
    </ul>

```

```
</ul>
{% endblock %}
```

Listing 30-10: The list.html.twig template

In this template, we override the `userLink` block to contain the text `active` ❶, highlighting the User List page link in the navigation bar. In the `body` block, we use a Twig `for` loop ❷ to iterate through the `users` array, creating a list item for each user displaying the associated ID, username, and password.

Now we need to add a case for the `action=users` route to our front-controller `Application` class. Update `src/Application.php` to match the contents of [Listing 30-11](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    private DefaultController $defaultController;
    private ProductController $productController;
    private UserController $userController;

    public function __construct()
    {
        $this->defaultController = new DefaultController();
        $this->productController = new ProductController();
        $this->userController = new UserController();
    }

    public function run(): void
    {
        $action = filter_input(INPUT_GET, 'action');
        $isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
        'POST');

        switch ($action)
        {
            case 'products':
```

```

        $this->productController->list();
        break;

        case 'users':
            $this->userController->list();
            break;

        --snip--
    }

```

Listing 30-11: Adding a route to the user list in the *Application* class

We declare a `userController` property and initialize it as a new `UserController` object in the constructor. Then, in the `switch` statement, we declare a case for when the action is `'users'`, invoking the `list()` method of the `UserController` object.

All we need to do now is add the `user` table to the database schema and insert user rows into the table. Create a new helper script, *db/setup_users.php*, as shown in [Listing 30-12](#).

```

<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\User;
use Mattsmithdev\UserRepository;

$userRepository = new UserRepository();

❶ $userRepository->resetTable();

$user1 = new User();
$user1->setUsername('matt');
$user1->setPassword('password1');
$userRepository->insert($user1);

$user2 = new User();
$user2->setUsername('john');
$user2->setPassword('password2');
$userRepository->insert($user2);

```

```
$users = $userRepository->findAll();
print '<pre>';
var_dump($users);
print '</pre>';
```

Listing 30-12: The setup script for the `user` table in `db/setup_users.php`

When we first set up the `product` table in [Chapter 28](#), we had to manually type out and execute each SQL statement to add a new row to the database. Now we can instead build each row as an instance of the `User` class and add it to the table by calling the `insert()` method of the `UserRepository` class (inherited from the ORM library). In this script, we do that for two users, assigning them usernames and passwords.

First, though, we invoke the `UserRepository` class's `resetTable()` method ❶, which drops any existing table mapped to the `User` class and creates a new table based on the names and data types of the `User` class. This is another “free” method automatically available to our repository class through inheritance from the ORM library's `DatabaseTableRepository` class. To confirm the database table has been created and two `User` records have been inserted, the script ends by retrieving all users from the database with the `findAll()` method and printing them with `var_dump()`.

Enter `php db/setup_users.php` at the terminal to run this setup script. You should see the following output:

```
$ php db/setup_users.php
<pre>array(2) {
  [0]=> object(Mattsmithdev\User)#8 (3) {
    ["id":"Mattsmithdev\User":private]=> int(1)
    ["username":"Mattsmithdev\User":private]=> string(4) "ma
tt"
    ["password":"Mattsmithdev\User":private]=> string(9) "pa
ssword1"
  }
  [1]=> object(Mattsmithdev\User)#9 (3) {
    ["id":"Mattsmithdev\User":private]=> int(2)
```

```
        ["username":"Mattsmithdev\User":private]=> string(4) "john"
        ["password":"Mattsmithdev\User":private]=> string(9) "password2"
    }
}
```

The terminal output shows an array containing two `User` objects, proof that the `user` database table has been added to the database schema, complete with two users. At this point, you can also launch the web application again and visit the User List page. It should look like [Figure 30-1](#).

By working with my `pdo-crud-for-free-repositories` library, we've seen how using an ORM library can remove the need to code low-level database queries. This reduces the amount of code required for each individual web application, simplifying the development process. We'll continue to use this library as we turn our attention to application security, but later we'll return to the topic of ORM libraries to see the added benefits of working with a more sophisticated library like Doctrine.

Security Best Practices

Security is an essential part of software development, both in your local development environment and when deploying web applications to the real world as public websites. Perhaps the most common manifestation of security a user meets these days is a username/password login form. We'll explore best practices for securing login information in this section.

Storing Hashed Passwords

You should never store plaintext passwords in your application's database. Otherwise, if someone gets access to the database, all those accounts would be compromised. One option for securely storing data is to *encrypt* it, encoding data in such a way that it can be decoded back to its original form at a later time. When sending confidential messages, for example, it's common to encrypt them first and to provide the intended recipient with the method for decrypting the message once received. For passwords, however, encryption isn't the best solution; if the database were accessed, brute-force

techniques could allow attackers to eventually decrypt the data (although depending on the speed of their computers, it might take a long time).

A better technique for password storage is *hashing*. This is an irreversible way of creating a new piece of data from the original data; there's no way to reconstruct the plaintext password from the hashed version. The same password passed through the same hash algorithm will always yield the same hash, however. When a user is logging into an application, you can therefore test whether their password is valid by hashing what they've entered and comparing it with the hash stored in the database. With this mechanism, there's no need to ever store the original plaintext password.

Let's make our web application more secure by storing hashes rather than plaintext passwords in the `user` database table. Conveniently, modern PHP offers a built-in `password_hash()` function for calculating the hash of a string. We'll change the `setPassword()` method of the `User` entity class to take advantage of this function. Update `src/User.php` to match the contents of [Listing 30-13](#).

```
<?php
namespace Mattsmithdev;

class User
{
    private int $id;
    private string $username;
    private string $password;

    --snip--

    public function setPassword(string $password): void
    {
        $hashedPassword = password_hash($password, PASSWORD_
DEFAULT);
        $this->password = $hashedPassword;
    }
}
```

Listing 30-13: Storing a hashed password in the `User` class

The revised `setPassword()` method takes in the plaintext password for a new user and passes it to the `password_hash()` function for hashing. The `PASSWORD_DEFAULT` constant means the function will use the strongest hashing algorithm available in the installed version of PHP, though there are other constants for explicitly choosing a particular hashing algorithm. We store the hash in the `$hashedPassword` variable and assign this as the value of the `User` object's `password` property.

With this change, any new `User` objects created and passed to the database will contain hashed rather than plaintext passwords. To prove it, rerun our `user` table setup script by entering `php db/setup_users.php` at the command line. This will delete and re-create the table with the modified `User` class. Here's the resulting `var_dump()` output in the terminal:

```
$ php db/setup_users.php
<pre>array(2) {
  [0]=> object(Mattsmithdev\User)#8 (3) {
    ["id":"Mattsmithdev\User":private]=> int(1)
    ["username":"Mattsmithdev\User":private]=> string(4) "ma
tt"
    ["password":"Mattsmithdev\User":private]=> string(60)
"$2y$10$k25neEiR.2k8j4gM7Gn6aeiHK8T7ZNgS18QUVsTdm592fGfN
23SZG"
  }
  [1]=> object(Mattsmithdev\User)#9 (3) {
    ["id":"Mattsmithdev\User":private]=> int(2)
    ["username":"Mattsmithdev\User":private]=> string(4) "jo
hn"
    ["password":"Mattsmithdev\User":private]=> string(60)
"$2y$10$telY8TmtAD7a/niym3/W5OvlKIFbu.CYOfrX0u3yRKdPEyD1
V6KRi"
  }
}
```

The black text lines show the hashes in each `password` field. Each hash is a long character string that has no discernible relationship with the original password.

Verifying Hashed Passwords at Login

Another useful built-in PHP function is `password_verify()`, which takes in a plaintext password, hashes it, and compares it with an existing hash to determine whether the password is correct. With this function, we can implement a login page for our application, where the user inputs a username and password for verification against their record in the `user` database table. [Figure 30-2](#) shows the login page we'll create.

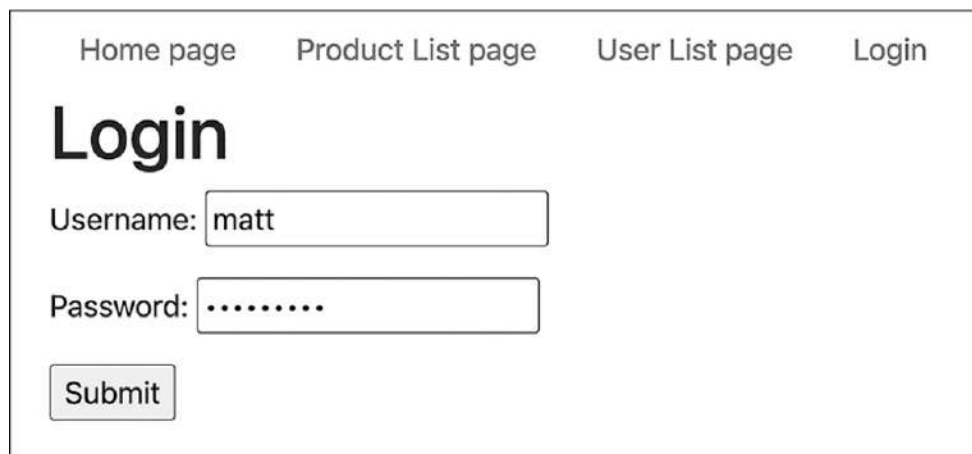


Figure 30-2: The new login page

Our application will need two new routes, one to request display of the login page (`action=login`) and one to request processing of the submitted data from the login form (`action=processLogin`). First, we'll add cases for these routes to our front-controller `Application` class. Update the `switch` statement in `src/Application.php` to match [Listing 30-14](#).

```
<?php
namespace Mattsmithdev;

class Application
{
    --snip--

    public function run(): void
    {
```

```

$action = filter_input(INPUT_GET, 'action');
$isPostSubmission = ($_SERVER['REQUEST_METHOD'] ===
'POST');

switch ($action)
{
    case 'login':
        $this->userController->loginForm(); ❶
        break;

    case 'processLogin':
        $username = filter_input(INPUT_POST, 'username');
        $password = filter_input(INPUT_POST, 'password');
        if (empty($username) || empty($password))
        { ❷
            $this->defaultController->error(
                'error - you must enter both a username and a password to login');
        } else {
            $this->userController->processLogin($username, $password); ❸
        }
        break;
}
--snip--
}

```

Listing 30-14: Adding login routes to the *Application* class

For the 'login' case, we invoke the `loginForm()` method of the `UserController` object ❶. For the 'processLogin' case, we first attempt to extract the 'username' and 'password' values from the variables received in the `POST` request. If either is empty ❷, an appropriate error message is displayed by passing a string message to the `error()` method of the `DefaultController` object. Otherwise, the username and password are passed to the `processLogin()` method of the `UserController` object ❸.

Now we need to add the new methods to the `UserController` class. Update `src/UserController.php` as shown in [Listing 30-15](#).

```
<?php
namespace Mattsmithdev;

class UserController extends Controller
{
    --snip--

    public function loginForm(): void
    {
        $template = 'user/login.xhtml.twig';
        $args = [];
        print $this->twig->render($template, $args);
    }

    public function processLogin(string $username, string $password): void
    {
        ❶ $loginSuccess = $this->isValidUsernamePassword($username, $password);
        if ($loginSuccess) {
            print 'success - username and password found in database';
        } else {
            print 'sorry - there was an error with your username/password';
        }
    }

    private function isValidUsernamePassword($username, $password): bool
    {
        ❷ $user = $this->userRepository->findOneByUsername($username);

        // False if no user for username
        if ($user == NULL) {
```

```

        return false;
    }

    // See if entered password matches stored (hashed) one
    ❸ return password_verify($password, $user->getPassword());
}

```

Listing 30-15: Adding login methods to the `UserController` class

For the `loginForm()` method, we simply render the appropriate Twig template, which doesn't require any arguments. For the `processLogin()` method, we take in the received `$username` and `$password` variables and pass them to the `isValidUsernamePassword()` helper method ❶, which returns a Boolean. If `true`, we print a success message, or an error message if `false`. In a full web application, at this stage, we would store the login success in the session as we did in [Chapter 16](#).

The `isValidUsernamePassword()` helper is responsible for determining whether the database holds a record matching the received username and password. First, we call the `UserRepository` class method `findOneByUsername()`, which attempts to retrieve a record (in the form of a `User` object) from the `user` table matching the provided username ❷. If a single user can't be retrieved, `findOneByUsername()` returns `NULL`, in which case the validation method returns `false`. Otherwise, we call PHP's built-in `password_verify()` function, passing it the submitted password (`$password`) and the correct password hash (accessed with the `User` object's `getPassword()` method) ❸. The `password_verify()` function hashes the provided plaintext password and returns a Boolean indicating whether it matches the provided hash.

TIMING ATTACKS

You may be wondering why PHP's `password_verify()` function exists. Couldn't we just use `password_hash()` on the submitted password and compare the resulting

string with the correct password hash stored in the database? We could, but that would open up the application to a security vulnerability known as a *timing attack*.

Simple string-comparison functions examine two strings character by character and return `false` as soon as they encounter a pair of characters that don't match. The longer the function takes before returning `false`, the more characters at the start of the strings must have matched. In timing attacks, hackers use this fact to their advantage by deliberately entering incorrect passwords with known hashes and measuring how long the application takes to report these passwords as incorrect. Based on the timings, they can infer information about the correct hash.

The `password_verify()` function is specially designed to guard against timing attacks. After hashing the input password, it uses a *constant-time function* to compare it with the stored hash. The comparison takes the same amount of time no matter how many characters do or don't match, so no information about the stored hash is exposed.

Now let's write the `findOneByUsername()` method for the `UserRepository` class. Update `src/UserRepository.php` to match the code in [Listing 30-16](#).

```
<?php
namespace Mattsmithdev;

use Mattsmithdev\PdoCrudRepo\DatabaseTableRepository;

class UserRepository extends DatabaseTableRepository
{
    public function findOneByUsername(string $username): ?User
    {
        $users = $this->searchByColumn('username', $username);

        if (count($users) != 1) {
            return NULL;
        }

        return $users[0];
    }
}
```

Listing 30-16: Adding the `findOneByUsername()` method to the `UserRepository` class

The new `findOneByUsername()` method has a nullable `?User` return type. It uses the `searchByColumn()` method inherited from the ORM library, which takes in a column name (`'username'`) and a value (in the `$username` variable) and returns an array of records where the value in that column of the database table is a match. If the resulting array doesn't have a length of exactly 1 (either because it's empty or because multiple records were retrieved), `findOneByUsername()` returns `NULL`. However, if a single user matches the submitted username string, the corresponding `User` object is returned.

Note that the logic in this method could have been made part of the `isValidUsernamePassword()` method in `UserController`, but what's needed is a query for a user with a given username, which is a model database query. It therefore makes sense to create this as a custom method in our `UserRepository` class, where all the code for querying the `user` database table lives. It's also worth highlighting that even though we're relying on an ORM library for generic methods such as `find()` and `findAll()`, it's often still necessary to extend a repository class with custom database methods that support the more specialized controller logic specific to the application at hand. In this case, we need to search by the `username` column rather than `id`, so the inherited `find()` method wouldn't do. The ORM library is still helping us through the `searchByColumn()` method, but we still need the custom logic of verifying that exactly one `User` object has been retrieved.

Next, we'll add a login page link to the navigation bar in the base template. Update `templates/base.xhtml.twig` as shown in [Listing 30-17](#).

```
<!doctype html>
<html lang="en">
--snip--
<body class="container">

<ul class="nav nav-pills">
    --snip--
    <li class="nav-item">
```



```
        <a class="nav-link" href="/?action=login">Login</a>
    </li>
</ul>

{% block body %}
{% endblock %}
</body></html>
```

Listing 30-17: Adding a login link to the base.xhtml.twig template

Here we add a navigation list item with the text `Login` and a URL route of `action=login`. With that added, we can create the child template for the login page itself in `/templates/user/login.xhtml.twig`. [Listing 30-18](#) shows the code.

```
{% extends 'base.xhtml.twig' %}

{% block title %}login page{% endblock %}

{% block body %}
    <h1>Login</h1>

    <form method="POST" action="/?action=processLogin">
        <p>
            Username:
            <input name="username">
        </p>
        <p>
            Password:
            ❶ <input name="password" type="password">
        </p>
        <input type="submit">
    </form>
{% endblock %}
```

Listing 30-18: The login.xhtml.twig template

The body of the page features a `<form>` element with a `POST` action of `processLogin`. The form features fields for a username and password, along

with a Submit button. Notice that the password input is of type "password"
❶. With this setting, the browser will display placeholder characters such as dots or asterisks, hiding the actual characters the user enters.

Try testing out the new login form with the username `matt` and password `password1`, or with any incorrect username/password combination. Thanks to PHP's secure, handy `password_verify()` function, you should find that the form works, even though the database is storing password hashes rather than plaintext passwords.

Securing Database Credentials

Another important security measure for web applications is to avoid exposing your database credentials. Whether you declare these credentials as class constants or in a completely separate file such as `.env`, as we did earlier in the chapter, it's important not to have them in any public-facing files.

To begin, you should have only a single file for your credentials. If you're using class constants rather than a `.env` file, I recommend having a completely separate class that just declares the constants. Then you can reference this class from your `Database` class (or whatever other class is responsible for establishing the database connection).

Next, mark the file containing your credentials to be ignored by any backup or archiving system. For example, if you're using the Git distributed version control system, you'd list this file in your project's `.gitignore` file.

The Doctrine ORM Library

The open source Doctrine project is a well-maintained, fully featured PHP ORM library. It's widely used; for example, the Symfony framework uses Doctrine for all database communications. My small ORM library is fine for small projects and for learning the basics, but for larger projects with many interrelated model classes, Doctrine is a more robust, sophisticated solution. Some of its features include easily facilitating object-to-object references that become foreign keys in the database schema and providing low-level control of the database table and column names beyond the default naming conventions.

After [Listing 30-5](#) (before adding the `User` model class), you were asked to make a copy of your project. (Don't worry if you didn't make a copy of your project at that point; you can copy my `listing30-05` from the book codes at <https://github.com/dr-matt-smith/php-crash-course>.) The coming sections will show you how to adapt that copy of the project to use Doctrine rather than my `pdo-crud-for-free-repositories` ORM library.

Removing the Previous ORM Library

First, let's remove the previous ORM library features from the project. Enter the following at the command line to remove the `pdo-crud-for-free-repositories` library from the project's `/vendor` folder and `composer.json` project dependencies file:

```
$ composer remove mattsmithdev/pdo-crud-for-free-repositories
```

We also need to remove the references to the old library's `DatabaseTableRepository` class from the `ProductRepository` class declaration. [Listing 30-19](#) shows how to update the `src/UserRepository.php` file.

```
<?php
namespace Mattsmithdev;

class ProductRepository
{
}
```

Listing 30-19: The `ProductRepository` class, without inheriting from the ORM library

For now, we're left with an empty class declaration, but later we'll return to the class and integrate it with Doctrine.

Adding Doctrine

Now we'll use Composer to add the Doctrine ORM library to the project, along with two other required libraries. Enter the following at the command line:

```
$ composer require doctrine/orm
$ composer require symfony/cache
$ composer require symfony/dotenv
```

Doctrine requires a cache to aid its performance, and `symfony/cache` is the recommended choice. Additionally, `symfony/dotenv` will make it easy to access values from the project's `.env` file.

Next, we need to connect Doctrine with the database. Create a script in the project's top-level directory named *bootstrap.php*, containing the code in [Listing 30-20](#). This script is based on Doctrine's documentation pages at <https://www.doctrine-project.org>.

```
<?php
require_once "vendor/autoload.php";

use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\ORMSetup;
use Symfony\Component\Dotenv\Dotenv;

❶ $dotenv = new Dotenv();
$dotenv->load(__DIR__ . '/.env');

// Get Doctrine to create DB connection
$connectionParams = [
    'dbname' => $_ENV['MYSQL_DATABASE'],
    'user' => $_ENV['MYSQL_USER'],
    'password' => $_ENV['MYSQL_PASSWORD'],
    'host' => $_ENV['MYSQL_HOST'],
    'driver' => 'pdo_mysql',
];

$config = ORMSetup::createAttributeMetadataConfiguration(
    paths: [__DIR__ . '/src'],
    isDevMode: true,
);

❷ $connection = DriverManager::getConnection($connectionParam
```

```
s, $config);  
❸ $entityManager = new EntityManager($connection, $config);
```

Listing 30-20: The bootstrap.php script to set up Doctrine

We read in the Composer autoloader, create a `Dotenv` object to load the database credentials from the project's `.env` file ❶, and package those credentials into a `$connectionParams` array. We then use this array and some Doctrine static methods to establish a database connection ❷ and create an `EntityManager` object ❸. The `EntityManager` class is key to the way Doctrine works; the class maintains the link between the model class objects in the PHP code and their corresponding database table rows defined with unique primary keys.

Any other script that reads in *bootstrap.php* will now have access to a database connection through the `$connection` variable and to Doctrine's entity manager through the `$entityManager` variable.

Verifying That Doctrine Is Working

Before we go any further, let's make sure Doctrine is successfully linked with the project's database. [Listing 30-21](#) shows a simple script that tests Doctrine by retrieving `Product` objects from the database as an associative array. Save this script as *public/doctrinel.php*.

```
<?php  
require_once __DIR__ . '/../vendor/autoload.php';  
require_once __DIR__ . '/../bootstrap.php';  
  
$sql = 'SELECT * FROM product';  
$stmt = $connection->executeQuery($sql);  
$result = $stmt->fetchAllAssociative();  
  
// Print results  
foreach ($result as $row) {  
    print "ID: {$row['id']}, Description: {$row['description']}\n";  
}
```

Listing 30-21: The doctrine1.php script to retrieve existing rows from the database

After reading in the autoloader and the Doctrine bootstrap script, we create an SQL query to select all rows from the `product` database table, then execute the query by using the Doctrine database connection (in the `$connection` variable). The results are returned as a nested array; each inner array maps the column names to the values in a particular row of the `product` database table. We loop through this array and print each row. If you run this *public/doctrine1.php* script, you should see the following output:

```
ID: 1, Description: bag of nails
ID: 2, Description: bucket
```

We've successfully retrieved the two products from the database, indicating that Doctrine is up and running.

Creating Database Tables

One of Doctrine's strengths is its ability to update the structure of a database based on the classes it encounters in the application's PHP code, creating new tables and columns as needed. To see how this works, let's switch our project over to a new, empty database. Then we can use Doctrine to create the `product` table from scratch.

To begin, open the project's `.env` file and change the value associated with the `MYSQL_DATABASE` key to `demo2`. Next, we need to write a script to create this new `demo2` database schema. Create *db/create_database.php* and enter the contents of [Listing 30-22](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Symfony\Component\Dotenv\Dotenv;
use Doctrine\DBAL\DriverManager;

$dotenv = new Dotenv(); ❶
$dotenv->load(__DIR__ . '/../.env');
```

```

$connectionParams = [
    'user' => $_ENV['MYSQL_USER'],
    'password' => $_ENV['MYSQL_PASSWORD'],
    'host' => "{$_ENV['MYSQL_HOST']}:{$_ENV['MYSQL_PORT']}",
    'driver' => 'pdo_mysql',
];

try {
    // Get connection
    $connection = DriverManager::getConnection($connectionPa
rams); ❷

    $databaseNames = $connection->createSchemaManager()->lis
tDatabases();
    $databaseExists = array_search($_ENV['MYSQL_DATABASE'],
$databaseNames); ❸
    // Drop database if exists already
    if ($databaseExists) {
        $connection->createSchemaManager()->dropDatabase($_E
NV['MYSQL_DATABASE']);
    }

    // Create database
    $connection->createSchemaManager()->createDatabase($_ENV
['MYSQL_DATABASE']); ❹

    print "succeeded in (re)creating database: {$_ENV['MYSQL
_DATABASE']}\n";
} catch (Exception $e) { ❺
    print "there was a problem creating the database: $e";
}

```

Listing 30-22: The db/create_database.php script to create the database named in the .env file

We use a `Dotenv` object ❶ to read the database credentials from the `.env` file and create an array of connection parameters. Then, inside a `try...catch` block, we create a connection to the MySQL database server by using Doctrine's `DriverManager::getConnection()` method ❷. We

then get an array of all the database names and search that array for the database name from our *.env* file, storing the result (`true` or `false`) in the `$databaseExists` variable ❸.

If the database exists, we drop it by using the `dropDatabase()` method. Then we create the database anew by using the `createDatabase()` method ❹ and print a success message. If any `Exception` is caught ❺, we print an error message instead. Run this script and you should now have a new, empty database schema called `demo2`.

The basic usage of Doctrine is to run a command line script that reads metadata about model classes in the PHP code (called *entity classes* in Doctrine's parlance) and executes SQL statements to create corresponding structures in the database schema. The command line script is usually placed in a file called `/bin/doctrine` (without the *.php* file extension). Create this file as shown in [Listing 30-23](#).

```
<?php
require_once __DIR__ . '/../bootstrap.php';

use Doctrine\ORM\Tools\Console\ConsoleRunner;
use Doctrine\ORM\Tools\Console\EntityManagerProvider\SingleManagerProvider;

ConsoleRunner::run(new SingleManagerProvider($entityManager), []);
```

Listing 30-23: The `/bin/doctrine` command line script

This script invokes the `run()` method of Doctrine's `ConsoleRunner` class. The method takes in the arguments from the command line and uses them to run whatever Doctrine command has been entered after `bin/doctrine` in the terminal. Let's run this script to try updating the new database schema. Enter `php bin/doctrine orm:schema-tool:create` at the command line. You should see the following output:

```
$ php bin/doctrine orm:schema-tool:create
[OK] No Metadata Classes to process.
```

The script hasn't done anything because we haven't yet added any of the necessary metadata for Doctrine to know which model classes and properties should be mapped to which database tables and columns. We'll now add metadata to the `Product` model class so that Doctrine will have a table to create in the database. As you'll see, each metadata tag is preceded by a hash mark (`#`) and enclosed in square brackets. Modify the `src/Product.php` file as shown in [Listing 30-24](#).

```
<?php
namespace Mattsmithdev;

use Doctrine\ORM\Mapping as ORM;

❶ #[ORM\Entity]
#[ORM\Table(name: 'product')]
class Product
{
    ❷ #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private ?int $id;

    #[ORM\Column(type: 'string')]
    private string $description;

    #[ORM\Column()]
    private float $price;

    --snip--
}
```

Listing 30-24: Adding Doctrine metadata to the `Product` class

To keep the metadata easier to read, we start with a `use` statement aliasing the `Doctrine\ORM\Mapping` class as `ORM`. Then we add metadata to the class itself and to each of its properties. We declare the class as an `Entity` ❶, indicating that it should correspond to a database table, and specify that this table should be named `product`. Without the latter,

Doctrine would default to `Product` (starting with a capital letter) as the table name, to match the class name.

For the class's `id` property, the `Id` tag indicates that this property should be used as the primary key ❷, `Column` indicates the property should correspond to a column in the database table, and `GeneratedValue` means the property should be auto-incremented in the database system. For the remaining properties, all we need is the `Column` tag. Notice that we can either specify the database column's data type as part of the `Column` tag or let Doctrine guess the appropriate data type.

With this metadata added, we can run our Doctrine command line script again. First, let's add the `--dump-sql` option, which will show the SQL that Doctrine *would* execute, without actually executing it yet:

```
$ php bin/doctrine orm:schema-tool:create --dump-sql
CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL, description VARCHAR(255) NOT NULL, price DOUBLE PRECISION NOT NULL, PRIMARY KEY(id))
DEFAULT CHARACTER
SET utf8 COLLATE `utf8_unicode_ci` ENGINE = InnoDB;
```

This shows that Doctrine will issue SQL code to create a `product` table with an auto-incrementing integer primary key `id`, a text `description`, and a floating-point `price`. Exactly what we want! Now run the command line script again without the `--dump-sql` option to execute that SQL:

```
$ php bin/doctrine orm:schema-tool:create
! [CAUTION] This operation should not be executed in a production environment!
Creating database schema...
[OK] Database schema created successfully!
```

Doctrine has now created the `product` table in the `demo2` database schema.

Adding Records to a Table

Now that we've used Doctrine to map our `Product` class to the `product` database table, we can create new `Product` objects and store their data in the database. [Listing 30-25](#) shows the `public/doctrine2.php` script to do this. Add this file to the project.

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';
require_once __DIR__ . '/../bootstrap.php';

use Mattsmithdev\Product;

❶ $product1 = new Product();
$product1->setDescription("small hammer");
$product1->setPrice(4.50);

$entityManager->persist($product1);
$entityManager->flush();

// Retrieve products from Database
❷ $productRepository = $entityManager->getRepository(Product::
class);
$products = $productRepository->findAll();
foreach ($products as $product) {
    print "Product OBJECT = ID: {$product->getId()}, "
        . "Description: {$product->getDescription()}\n";
}
```

Listing 30-25: The `public/doctrine2.php` script to insert and retrieve a database row

We create a `Product` object ❶ and set its description and price. Then we use the Doctrine `EntityManager` object to add this product's data to a queue (the `persist()` method) and insert the object into the database (the `flush()` method).

To confirm this has worked, we use `EntityManager` to create and get a reference to a Doctrine repository object for the `Product` class ❷. This is a custom repository object linking the `Product` class with the records in the

product database table. We use this repository object to retrieve all the records (in this case, just the one) from the table with the object's `findAll()` method. Then we loop through the resulting `$products` array and print each object. Here's the output of running this script:

```
Product OBJECT = ID: 1, Description: small hammer
```

This output confirms that Doctrine has successfully added the "small hammer" object to the product database table.

Integrating Doctrine into the Application Code

All the code is in place now to integrate the Doctrine ORM library into our main web application so that we can easily map objects and database table rows. First, to minimize changes required throughout the application code, we'll add a helper class called `OrmHelper` that manages access to the Doctrine `EntityManager` instance. [Listing 30-26](#) shows how to declare this class in *src/OrmHelper.php*.

```
<?php
namespace Mattsmithdev;

use Doctrine\ORM\EntityManager;

class OrmHelper
{
    private static EntityManager $entityManager;

    public static function getEntityManager(): EntityManager
    {
        return self::$entityManager;
    }

    public static function setEntityManager(
        EntityManager $entityManager): void
    {
        self::$entityManager = $entityManager;
    }
}
```

```
}  
}
```

Listing 30-26: The `OrmHelper` class storing and providing access to the `$entityManager` property

This class declares a private static `entityManager` property, with public static getters and setters. We use static members to allow retrieval of a reference to the Doctrine `EntityManager` object from anywhere in our application code (after the variable has been set), without having to create an object or pass an object reference down through several constructor methods when creating the application, controller, or repository classes.

Notice that the setter method takes in a reference to an `EntityManager` object and passes it along to the class's `entityManager` property. We've already created that reference in the *bootstrap.php* script, so we just need to read in the bootstrap script before invoking the setter method. We'll do that now by updating the *public/index.php* script as shown in [Listing 30-27](#).

```
<?php  
require_once __DIR__ . '/../vendor/autoload.php';  
❶ require_once __DIR__ . '/../bootstrap.php';  
  
session_start();  
  
use Mattsmithdev\Application;  
❷ use Mattsmithdev\OrmHelper;  
  
OrmHelper::setEntityManager($entityManager);  
  
$app = new Application();  
$app->run();
```

*Listing 30-27: Updating the *index.php* script to bootstrap Doctrine and store the `EntityManager` object reference*

We add a `require_once` statement to read and run our Doctrine bootstrap script ❶. We add a `use` statement so that we can refer to the `OrmHelper` class in our code ❷. Then we store a reference to the script's

EntityManager object by calling the `setEntityManager()` static method of our `OrmHelper` class. This means the `EntityManager` object is now available anywhere in our web application logic via the public static method `OrmHelper::getEntityManager()`.

Finally, we need to fill out our `ProductRepository` class, which we left as an empty class declaration when we switched over to Doctrine. Our `ProductController` class expects `ProductRepository` to have CRUD methods like `find()`, `findAll()`, `insert()`, `delete()`, and so on. [Listing 30-28](#) shows how to update *src/ProductRepository.php* accordingly.

```
<?php
namespace Mattsmithdev;

use Doctrine\ORM\EntityManager;
use Doctrine\ORM\EntityRepository;
use Mattsmithdev\Product;

class ProductRepository extends EntityRepository
{
    private EntityManager $entityManager;

    public function __construct()
    {
        ❶ $this->entityManager = OrmHelper::getEntityManager
();
        $entityClass = Product::class;
        $entityMetadata = $this->entityManager->
            getClassMetadata($entityClass);
        ❷ parent::__construct($this->entityManager, $entityMet
adata);
    }

    public function insert(Product $product): int
    {
        $this->entityManager->persist($product);
        $this->entityManager->flush();

        return $product->getId();
    }
}
```

```

    }

    public function update(Product $product): void
    {
        $this->entityManager->persist($product);
        $this->entityManager->flush();
    }

    public function delete(int $id): void
    {
        $product = $this->find($id);
        $this->entityManager->remove($product);
        $this->entityManager->flush();
    }

    public function deleteAll(): void
    {
        $products = $this->findAll();
        foreach ($products as $product) {
            $this->entityManager->remove($product);
        }
        $this->entityManager->flush();
    }
}

```

Listing 30-28: Updating *ProductRepository* with Doctrine-based CRUD methods

We declare `ProductRepository` as a subclass of `Doctrine\ORM\EntityRepository`. This means it will inherit methods such as `find()` and `findAll()` from its parent. The class declares one instance variable, an `EntityManager` object, which is assigned its value in the constructor via our `OrmHelper` class ❶. The remaining lines in the constructor retrieve the required metadata about the `Product` class and pass it along to the parent class's constructor to tailor the repository class to the `product` table ❷.

We continue the class by declaring the remaining CRUD methods our application expects. For `insert()` and `update()`, we use the `persist()` and `flush()` methods of the `EntityManager` object methods to add or modify a

database record. The `delete()` method uses the `remove()` and `flush()` methods of the `EntityManager` object to remove a record. Finally, the `deleteAll()` method retrieves all objects with the inherited `findAll()` method, then loops through them to remove each one from the database.

Creating Foreign-Key Relationships

It may seem like we've done a lot of work to incorporate Doctrine while gaining little or no functionality beyond the previous ORM library. However, we can begin to see some of the real power of the Doctrine ORM library when we start creating foreign-key relationships between database tables and their corresponding model classes. In our code, we establish this relationship by adding a property to a model class whose value is a reference to an object of another model class. With the right metadata, Doctrine can see this relationship and generate all the SQL needed to realize it in the database.

To illustrate, let's add a `Category` model class to our project along with the equivalent `category` database table. Then we'll modify the `Product` model class so that each product is associated with a category. In the process, we'll see how Doctrine manages the foreign-key relationship behind this association. [Listing 30-29](#) shows the `src/Category.php` script declaring the new `Category` class.

```
<?php
namespace Mattsmithdev;

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
❶ #[ORM\Table(name: 'category')]
class Category
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    ❷ #[ORM\GeneratedValue]
    private ?int $id;

    #[ORM\Column(type: 'string')]
```



```

    private string $name;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function setId(?int $id): void
    {
        $this->id = $id;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }
}

```

Listing 30-29: The `Category` model class, including Doctrine ORM metadata

The initial metadata before the class name indicates that this simple model class (or Doctrine entity) should correspond to a database table called `category` ❶. The class has two properties: a unique integer `id` and a string `name`. As with the `Product` class, we include a tag specifying that `id` is to be autogenerated by the database ❷. For each property, we declare basic getter and setter methods.

Now let's add a `category` property to the `Product` class so that each `Product` object will be associated with one `Category` object. [Listing 30-30](#) shows how to modify *src/Product.php*.

```

<?php
namespace Mattsmithdev;

```

```

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'product')]
class Product
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private ?int $id;

    #[ORM\Column(type: 'string')]
    private string $description;

    #[ORM\Column()]
    private float $price;

    ❶ #[ORM\ManyToOne(targetEntity: Category::class)]
    private Category|NULL $category = NULL;

    public function getCategory(): ?Category
    {
        return $this->category;
    }

    public function setCategory(?Category $category): void
    {
        $this->category = $category;
    }
    --snip--
}

```

Listing 30-30: Adding a *category* property to the *Product* class

We declare the `category` property as either `NULL` or a reference to a `Category` object, and give it public getter and setter methods. The metadata attribute preceding the property ❶ tells Doctrine that this field in the database should hold a foreign-key reference to a row in the `category` table. Here `ManyToOne` indicates that the foreign key establishes a *many-to-one*

relationship, where many products can be of the same category, and `targetEntity` sets the model class (and database table) on the other end of the relationship.

Since we've changed the structure of the `Product` model class, as well as adding the new `Category` class, we need Doctrine to update the structure of the database accordingly. First, let's use our *bin/doctrine* command-line script to drop the old `product` table from the database schema:

```
$ php bin/doctrine orm:schema-tool:drop --force
[OK] Database schema dropped successfully!
```

This drops *all* tables from the schema (in our case, that's just the `product` table). Now we'll use the command-line script again to create the database schema anew, complete with the `product` and `category` tables and the foreign-key relationship between them. As before, we'll first use the `--dump-sql` option to view the SQL statements Doctrine wants to run:

```
$ php bin/doctrine orm:schema-tool:create --dump-sql
CREATE TABLE category (id INT AUTO_INCREMENT NOT NULL, name
VARCHAR(255)
NOT NULL, PRIMARY KEY(id))
--snip--
CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL, category_id INT DEFAULT
NULL, description VARCHAR(255) NOT NULL, price DOUBLE PRECISION NOT NULL,
INDEX IDX_1x (category_id), PRIMARY KEY(id))
--snip--
ALTER TABLE product ADD CONSTRAINT FK_1x FOREIGN KEY (category_id) REFERENCES
category (id);
```

This shows that Doctrine will issue SQL code to create the `category` and `product` tables, where `product` has a `category_id` field with a foreign-key reference to a `category` database row. Real-world databases

abound with foreign-key references like this, and here we see how Doctrine excels at managing the SQL for these relationships so we don't have to.

Run the command-line script once more without the `--dump-sql` option to execute the SQL statements and create these related database tables. To make sure the related tables have been successfully created in the database, we'll write a one-off script creating related `Product` and `Category` objects, saving them to the database, and retrieving them. [Listing 30-31](#) shows *public/doctrine3.php* implementing these actions. Add this file to your project.

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';
require_once __DIR__ . '/../bootstrap.php';

use Mattsmithdev\ProductRepository;
use Mattsmithdev\OrmHelper;

OrmHelper::setEntityManager($entityManager);

// -- Create 2 categories ---
$category1 = new \Mattsmithdev\Category();
$category1->setName('HARDWARE');
$entityManager->persist($category1);

$category2 = new \Mattsmithdev\Category();
$category2->setName('APPLIANCES');
$entityManager->persist($category2);

// Push category objects into DB
❶ $entityManager->flush();

// -- Create 2 products ---
$productRepository = new ProductRepository();
$productRepository->deleteAll();

$product1 = new \Mattsmithdev\Product();
$product1->setDescription("small hammer");
$product1->setPrice(4.50);
```

```

$product1->setCategory($category1);
$productRepository->insert($product1);

$product2 = new \Mattsmithdev\Product();
$product2->setDescription("fridge");
$product2->setPrice(200);
$product2->setCategory($category2);
$productRepository->insert($product2);

// Retrieve products from Database
❷ $products = $productRepository->findAll();
if (empty($products)) {
    print 'no products found in DB';
} else {
    foreach ($products as $product) {
        print "Product OBJECT = ID: {$product->getId()}, "
            . "Description: {$product->getDescription()} //"
            . "Category = {$product->getCategory()->getName()}\n";
    }
}

```

Listing 30-31: The public/doctrine3.php script to insert related records into the database

We create two `Category` objects for `HARDWARE` and `APPLIANCES`, then store them in the database by using the `Doctrine EntityManager` object from the bootstrap script. Notice that we call the `persist()` method on each `Category` object individually, then call the `flush()` method once ❶; `flush()` will batch-process any operations that have been queued up for it with methods like `persist()`. We next use our `ProductRepository` class to create and insert two `Product` objects into the database, one for each category. Then we retrieve an array of all the products from the database with the `ProductRepository` class's `findAll()` method ❷. If the array isn't empty, we loop through it and print each product. Here's the output of running this script:

```
Product OBJECT = ID: 1, Description: small hammer // Category = HARDWARE
Product OBJECT = ID: 2, Description: fridge // Category = APPLIANCES
```

Each product is shown with its associated category. With just a little bit of metadata in the `Product` model classes (the Doctrine `ManyToOne` attribute added before the `category` property), we've created a whole database of foreign-key declaration and storage mapping.

Overall, although switching from my simple ORM library to Doctrine added complexity to the code, such as the need for a bootstrap script and metadata tags in the model classes, Doctrine comes with added benefits like increased flexibility and support for foreign-key relationships. Using a popular ORM library like Doctrine for a project also means that developers you collaborate with will be more likely to already be familiar with its operations, which can save time in code development and maintenance. Another advantage of ORM libraries like Doctrine is that they allow you to seamlessly switch from one DBMS to another (such as from MySQL to PostgreSQL), without any of your core web application code having to change. The only downside may be the effort of learning the library in the first place, and perhaps some performance reduction due to the extra layer of abstraction. Still, the advantages will in many cases outweigh any minor performance reduction.

Summary

In this chapter, we used my `pdo-crud-for-free-repositories` library to explore the basics of ORM libraries, seeing how they can simplify the process of working with a database by eliminating the need for writing a lot of repetitive code for CRUD applications. When we transitioned to the Doctrine ORM library, we saw that this more robust and feature-complete library has added benefits like greater flexibility and support for foreign-key associations between model classes and their corresponding database tables.

This chapter also outlined important practices in web application security. In previous chapters, we were already using prepared SQL

statements, which help protect against SQL injection attacks. Now we've added the ability to store and verify against hashed passwords, so we never need to store plaintext passwords in a database. We've also emphasized the importance of keeping database credentials in a separate file so that they won't be published or archived and accidentally exposed.

Exercises

1. I've created a publicly shared sample project to help explore the `pdo-crud-for-free-repositories` library. The project uses PHP templates (not Twig) to illustrate how to use the ORM library for a `Movie` model class and its associated `MovieRepository` class. To check out the project, do the following:
 - a. Enter the following at the command line to create a new project named *demo1* based on my published project template:

```
$ composer create-project mattsmithdev/pdo-repo-project demo1
```

- b. In the *demo1* directory that was created, edit the MySQL credentials in the `.env` file to match your computer's setup.
 - c. Run the database setup script in `db/migrateAndLoadFixtures.php`.
 - d. Run a web server and visit the home page and movie list page.
 - e. Examine the `Movie` model class, and the `listMovies()` method in the `Application` class.
2. Use the `pdo-crud-for-free-repositories` library to create a MySQL CRUD web application for `Book` objects with these properties:
 - `id` (integer), an auto-incrementing primary key
 - `title` (string)
 - `author` (string)
 - `price` (float)

You can either create a new project from scratch, extend the demo project from the previous exercise, or adapt your work from Exercise 2 of the previous chapter. You may find it helpful to adapt the database schema creation and initial data script from [Listing 30-12](#), or if you're

using the demo project, you can adapt the database setup script in *db/migrateAndLoadFixtures.php*.

3. Web application security is an enormous topic (the subject of entire books), and covering it exhaustively here would be impossible. Learn more about PHP security best practices by exploring the following resources:

The Paragon Initiative Enterprises PHP security guide, <https://paragonie.com/blog/2017/12/2018-guide-building-secure-php-software>

The Open Web Application Security Project, <https://owasp.org>
PHP The Right Way's security chapters, by Josh Lockhart (codeguy), <https://phptherightway.com/#security>

WORKING WITH DATES AND TIMES



Web applications use dates and times in many ways: to maintain calendars, log when invoices are created, record when a message has been sent, and so on. In this chapter, we'll explore PHP's built-in resources for storing and manipulating dates and times, including ways to handle complexities such as time zones and daylight saving time. We'll also look at how to send date-time information back and forth between PHP and a database system like MySQL, which has its own, separate way of storing that information.

The ISO 8601 Standard

Countries have varying conventions for representing dates and times in day-to-day life. For example, people in the United States write 11/2 for November 2, but people in Ireland and the United Kingdom write 2/11 for that date. Likewise, some countries use 24-hour time, while others use 12-hour time with AM and PM designations. Computer programs have no room

for such ambiguity when storing and manipulating dates and times, so it's important to agree on a standard. These days, pretty much everyone in computing uses the date-time formats declared in *ISO 8601*, a standard originally published in 1988 and updated several times since.

Two key principles of ISO 8601 are that dates and times are represented with numbers and that they're ordered from most to least significant. Dates therefore start with the year, then the month, then the day. For example, November 22, 1968, would typically be written as 1968-11-22, with four digits for the year, two for the month, and two for the day, adding leading zeros if required (such as 03 for March rather than just 3). Following the same principle of most to least significance, times are written as hours, then minutes, then seconds. For example, 5 minutes and 30 seconds past 9 AM would be written as 09:05:30. You can add a decimal point and more digits for fractions of a second if needed, so 09:05:30.01 is one-hundredth of a second after 09:05:30. The standard uses 24-hour time, so 11 PM is given as 23:00:00.

NOTE

The creator of PHP, Rasmus Lerdorf, was born on November 22, 1968. The web doesn't say what time he was born, so I've made up the time of 9:05 AM and 30 seconds for the examples in this chapter.

These examples illustrate the *extended version* of ISO 8601's date and time formatting, which adds dashes between the fields in the date, and colons between the fields in the time, to aid with human readability. Internally, the computer might not use these separators, but it's recommended to always use them when presenting a date or time to a human, since something like 1968-11-22 is a lot easier for a human to interpret than 19681122.

When combining a date and a time, ISO 8601 calls for a capital T between the date and time components. Therefore 5 minutes and 30 seconds past 9 AM on November 22, 1968, would be written as 1968-11-22T09:05:30. That said, the original ISO 8601 standard allowed the use of a space between the date and time rather than a T, so many computer implementations will also accept 1968-22-22 09:05:30 as a valid format.

NOTE

A concept found in most programming languages is date-time. So while we humans think about dates and times separately, or might talk about a date and time, in computer programming we typically work with objects that store data about a date and time all together. If we're interested in only the date, we ignore the time components in our code, often defaulting to a zeroed time component for the beginning of a new day (00:00:00). If we're interested in only the time, we ignore the date components in our code and often default to the current date.

Time-zone characters or time offsets can be added to the end of a date-time string; we'll look at how this works in “Formatting the Date-Time Information” on [page 634](#). ISO 8601 also defines other date and time components (such as week numbers), but the components I've described cover 99 percent of the date and time formatting needed to create and work with temporal data in PHP programs.

Creating Dates and Times

The fundamental PHP class for working with individual dates and times is `DateTimeImmutable`. If you create a `DateTimeImmutable` object without providing arguments to the constructor, the new object defaults to the current time and day at the moment of its creation, based on the local time-zone settings for your computer system. [Listing 31-1](#) illustrates how to instantiate this class.

```
<?php
$today = new DateTimeImmutable();
var_dump($today);
```

Listing 31-1: Creating a `DateTimeImmutable` object for the current date and time

We create a new `DateTimeImmutable` object without any arguments and output its value by using the `var_dump()` function. If you execute this script at the command line, you should see something similar to the following:

```
object(DateTimeImmutable)#1 (3) {  
  ["date"]=>  
    string(26) "1968-11-22 09:05:30.000000"  
  ["timezone_type"]=>  
    int(3)  
  ["timezone"]=>  
    string(3) "Europe/Dublin"  
}
```

The `var_dump()` function returns the object's internal `date` value as a date-time string. Notice that this string output uses a space rather than a `T` between the date and the time to aid in human readability, and that the time component uses six decimal places to capture the time down to the nearest microsecond (millionth of a second). The object's remaining properties pertain to the time zone (I'm in Dublin, Ireland, for example), as we'll discuss in detail in "Time Zones" on [page 641](#).

NOTE

To get the current date and time, you can also use `new DateTimeImmutable('now')`. The `'now'` argument is useful if you need to provide a second argument specifying a different time zone from the default.

To create a `DateTimeImmutable` object holding another (noncurrent) date and time, pass a string containing the desired date and time to the constructor, like this:

```
$rasmusBirthdate = new DateTimeImmutable('1968-11-22T09:05:30');
```

This creates a new `DateTimeImmutable` object holding the date November 22, 1968, and the time 9:05 and 30 seconds.

Formatting the Date-Time Information

The `DateTimeImmutable` class's `format()` method makes it possible to adjust the output style for dates and times in different ways. For example,

[Listing 31-2](#) shows how to format a date-time string to the ISO 8601 standard.

```
<?php
$now = new DateTimeImmutable();

$atomDateString = $now->format(DateTimeInterface::ATOM);
print 'now (ISO-8601): ' . $atomDateString . PHP_EOL;
```

Listing 31-2: Formatting a `DateTimeImmutable` object constant

We instantiate `$now` as a default `DateTimeImmutable` object, then call its `format()` method, passing in the `ATOM` constant from `DateTimeInterface` (the `DateTimeImmutable` class is an implementation of this interface). This constant indicates that the date and time should be formatted according to the ISO 8601 standard. The `format()` method returns a string, which we store in `$atomDateString` and print. The output should look something like this:

```
now (ISO-8601): 1968-11-22T09:05:30 + 00:00
```

Notice that the output follows ISO 8601 format, including the `T` between the date and the time. The `+00:00` at the end has to do with time zones, which we'll discuss later.

The `format()` method can also take in a string specifying custom formatting. This allows you to create more readable date-time output. For example, you can tell `format()` to spell out the name of the month, include the day of the week, convert from 24-hour to 12-hour time, add a suffix to the date number (as in *1st*, *2nd*, or *23rd*), and so on. [Listing 31-3](#) illustrates how it's done.

```
<?php
$now = new DateTimeImmutable();

$formattedDateString = $now->
    format('l \t\h\e jS \o\f F Y \a\t i \m\i\n\s \p\a\s\t g
a');
```

```
print 'now (nice format): ' . $formattedDateString . PHP_EOL;
```

Listing 31-3: Customizing the output string format for a date-time

This time, we provide a string to `format()` that defines a custom format for the date-time information. The string uses letter codes, such as `l`, `j`, and `s`, to stand in for parts of the date and time. Here's what the letters in [Listing 31-3](#) mean:

- l** The full name of the day of the week (Monday, Tuesday, and so on)
- j** The day of the month as an integer with no leading space (1, 4, 20, 31, and so on)
- s** A two-letter suffix for the numeric date of the month (*st* for 1st, *nd* for 2nd, and so on)
- F** The full name of the month (January, May, and so on)
- Y** The four-digit year (2000, 2019, 2025, and so on)
- i** Minutes with leading zeros (00 through 59)
- g** The hour in 12-hour format without leading zeros (1 through 12)
- a** The appropriate AM or PM abbreviation

The case of each character is important, since several characters represent different values depending on whether they're upper- or lowercase. For example, a lowercase `d` represents the date of the month (with a leading zero if required to ensure that it's two digits), while an uppercase `D` is the three-letter abbreviation of the day of the week (Mon, Tue, and so on).

NOTE

The PHP documentation provides a full list of all these special formatting characters at <https://www.php.net/manual/en/datetime.format.php>.

Any text that you want to be included verbatim in the formatted string needs to be escaped (preceded by a backslash), character by character. Thus, the string argument to `format()` in [Listing 31-3](#) includes character sequences like `\t\h\e` (the word *the*) and `\o\f` (the word *of*). [Listing 31-3](#) should output something like this:

```
now (nice format): Friday the 22nd of November 1968 at 05 mi  
ns past 9am
```

You can also use letter codes with the `createFromFormat()` static method to create new `DateTimeImmutable` objects based on formatted strings. Consider this example:

```
$date = DateTimeImmutable::createFromFormat('j-M-Y', '15-Feb  
-2009');
```

The `createFromFormat()` method takes two arguments. The first is a formatter string built using the letter codes we've discussed. The second is a string that follows the indicated format and sets the value for the new `DateTimeImmutable` object.

Using `DateTimeImmutable` vs. `DateTime`

Our focus has been on the `DateTimeImmutable` class, but PHP also offers the similar `DateTime` class. The only difference (but an important one) is that once created, a `DateTimeImmutable` object won't change the values it contains, whereas a `DateTime` object's value can be updated. One consequence is that `DateTimeImmutable` methods that might change the date or time return a new `DateTimeImmutable` object rather than modifying the original object's values.

Use `DateTimeImmutable` whenever possible rather than `DateTime` to avoid the unexpected behavior of an object changing and returning a reference to itself. To see why, look at [Listing 31-4](#).

```
<?php  
$today = new DateTime();  
print 'today (before modify) = ' . $today->format('Y-m-d') .  
PHP_EOL;  
  
$tomorrow = $today->modify('+1 day');  
print 'today = ' . $today->format('Y-m-d') . PHP_EOL;  
print 'tomorrow = ' . $tomorrow->format('Y-m-d') . PHP_EOL;
```

Listing 31-4: Creating a `DateTime` object and modifying it

We create a new `DateTime` (rather than `DateTimeImmutable`) object called `$today` and print its value (in year-month-day format). Then we invoke the object's `modify()` method to move the date one day forward, storing the result in the `$tomorrow` variable. Rather than creating a new `DateTime` object for the next day, however, calling `modify()` changes the original `DateTime` object and returns a reference to that same object. To confirm this, we print both `$today` and `$tomorrow`. The output will look something like this:

```
today (before modify) = 1968-11-22
today = 1968-11-23
tomorrow = 1968-11-23
```

Initially, the value of `$today` indicates November 22. After calling `modify()`, both `$today` and `$tomorrow` indicate November 23. Both variables reference the same `DateTime` object, whose value was changed by the `modify()` method.

While having changeable date-time information may sometimes be desirable, it's important to make sure that's what you want before using `DateTime` over `DateTimeImmutable`. The example also illustrates how inappropriate variable names can make code hard to understand and debug. If we deliberately create a mutable `DateTime` object and make a change to its date, that object shouldn't be named for a particular date (like `$today` or `$tomorrow`), since at some point the name of the variable won't correctly refer to its value.

To really see the difference between the two PHP date-time classes, try changing the class from `DateTime` to `DateTimeImmutable` in [Listing 31-4](#) and then run the script again. Here's the output:

```
today (before modify) = 1968-11-22
today = 1968-11-22
tomorrow = 1968-11-23
```

This time, the value of `$today` remains the same, even as `$tomorrow` moves one day ahead. This is because the `modify()` method of a `DateTimeImmutable` object creates and returns a completely new `DateTimeImmutable` object with the modified value while leaving the original object unchanged. Therefore, `$today` and `$tomorrow` refer to two different `DateTimeImmutable` objects, and both variable names are reasonable, since they always correctly refer to the values of the object they reference.

Manipulating Dates and Times

PHP's `DateTimeImmutable` class features several methods that create a new `DateTimeImmutable` object based on the value of an existing one. These methods make it possible to programmatically manipulate date and time information.

You already saw one example, the `modify()` method, in [Listing 31-4](#). This method takes a string argument indicating how to set a new date or time relative to the current `DateTimeImmutable` object's value. For instance, the strings `'yesterday'` and `'tomorrow'` yield midnight (time `00:00:00`) on the previous or next day; the `'noon'` and `'midnight'` strings yield noon (`12:00:00`) or midnight on the current day; and `'first day of this month'` or `'last day of this month'` change the date as appropriate while leaving the time the same. These modifier strings can be combined; for example, `'tomorrow noon'` and `'first day of this month midnight'` are both valid.

Other modifier strings use `+` or `-` followed by a quantity and unit of time to give more granular control over the new `DateTimeImmutable` object's value, as in `'+1 day'`, `'-2 hours'`, or `'+30 seconds'`. These can also be combined into longer strings, such as `'+1 day +30 seconds'`.

[Listing 31-5](#) shows some of these modifiers in action.

```
<?php
function showModify(string $modifier): void
{
    print PHP_EOL. $modifier . PHP_EOL;
    $date1 = new DateTimeImmutable();
```

```

        $date2 = $date1->modify($modifier);
        print 'date1 = ' . $date1->format(DateTimeInterface::ATOM) . PHP_EOL;
        print 'date2 = ' . $date2->format(DateTimeInterface::ATOM) . PHP_EOL;
    }

    showModify('first day of this month');
    showModify('+1 day');
    showModify('+30 seconds');
    showModify('-10 seconds');
    showModify('+1 month +3 days +1 seconds');

```

Listing 31-5: Passing relative date-time strings to the `modify()` method

To help show the date-times before and after modification, we first declare a `showModify()` function that takes in a modifier string as an argument. The function prints the modifier string itself, creates a new `DateTimeImmutable` object with the current time, and passes the string to the `modify()` method to create another, modified `DateTimeImmutable` object. Then it prints both objects in ISO 8601 format. Next, we make a series of `showModify()` calls to demonstrate various modifier strings. The output should be something like this:

```

first day of this month
date1 = 1968-11-22T09:05:30 + 00:00
❶ date2 = 1968-11-01T09:05:30 + 00:00

+1 day
date1 = 1968-11-22T09:05:30 + 00:00
❷ date2 = 1968-11-23T09:05:30 + 00:00

+30 seconds
date1 = 1968-11-22T09:05:30 + 00:00
date2 = 1968-11-22T09:06:00 + 00:00

-10 seconds
date1 = 1968-11-22T09:05:30 + 00:00

```

```
date2 = 1968-11-22T09:05:20 + 00:00
```

```
+1 month +3 days +1 seconds
```

```
date1 = 1968-11-22T09:05:30 + 00:00
```

```
date2 = 1968-12-25T09:05:31 + 00:00
```

The value of `$date1` is the same for each function call, November 22, 1968, at 9:05 and 30 seconds. With the `'first day of this month'` modifier, the date component of `$date2` ends up as November 1, but the time component is left unchanged ❶. With `'+1 day'`, the date component moves forward to November 23, but again the time component is unchanged ❷. The `'+30 seconds'` and `'-10 seconds'` strings move the time component forward and backward without changing the date component, while `'+1 month +3 days +1 seconds'` changes both the date and time.

If you provide a date to the `modify()` method, that date will replace the original one in the new `DateTimeImmutable` object created, leaving the time unchanged. For example, the `'2000-12-31'` string applied to an object holding `1968-11-22T09:05:30` would result in a new object of `2000-12-31T09:05:30`, the same time but for December 31, 2000. Likewise, providing a time results in a new object with the original date but the new time.

That said, modifying just the date or time component of a `DateTimeImmutable` object can be accomplished more easily using the `setDate()` and `setTime()` methods. For example, if `$date1` is a `DateTimeImmutable` object, both the following statements result in the same new object being created:

```
$date2 = $date1->modify('2000-12-31');  
$date2 = $date1->setDate(2000, 12, 31);
```

Notice that the `setDate()` method takes three separate integers as arguments, rather than a single string. These integers represent the desired year, month, and day. Similarly, the `setTime()` method takes four integers for the new hour, minute, second, and microsecond. The latter two default to 0.

Using Date-Time Intervals

The `DateInterval` class represents a span of time rather than a specific date-time. This class provides another useful way of manipulating date-time information and of thinking about the relationship between different date-times. For example, the `add()` and `sub()` methods of the `DateTimeImmutable` class take in a `DateInterval` object and return a new `DateTimeImmutable` object offset forward or backward in time from the original object by the specified interval. [Listing 31-6](#) illustrates how this works.

```
<?php
$interval1 = DateInterval::createFromDateString('30 second
s');
$interval2 = DateInterval::createFromDateString('1 day');

$date1 = new DateTimeImmutable();
$date2 = $date1->add($interval1);
$date3 = $date1->sub($interval2);

print '$date1 = ' . $date1->format(DateTimeInterface::ATOM)
. PHP_EOL;
print '$date2 = ' . $date2->format(DateTimeInterface::ATOM)
. PHP_EOL;
print '$date3 = ' . $date3->format(DateTimeInterface::ATOM)
. PHP_EOL;
```

Listing 31-6: *Creating new `DateTimeImmutable` objects offset by time intervals*

First, we use the `createFromDateString()` static method to create two `Date Interval` objects. With this method, we can express the desired interval by using strings like `'30 seconds'` or `'1 day'`. Next, we create a `DateTimeImmutable` object for the current date and time, then call its `add()` and `sub()` methods, passing in the `DateInterval` objects. This creates two more `DateTimeImmutable` objects, offset according to the given time intervals. The three date-time strings in the output should look something like this:

```
$date1 = 1968-11-22T09:05:30 + 00:00
$date2 = 1968-11-22T09:06:00 + 00:00
$date3 = 1968-11-21T09:05:30 + 00:00
```

Notice that `$date2` has the same date, but its time is 09:06:00, 30 seconds later than `$date1`. Meanwhile, `$date3` has the same time, but its date is one day earlier: November 21 instead of November 22. Note that we can create negative as well as positive `DateInterval` objects. For example, we could create a `DateInterval` by using the string `'-1 day'`.

A more common way of creating a `DateInterval` object is to use the `diff()` method of the `DateTimeImmutable` class. Given one `DateTimeImmutable` object, you call its `diff()` method, passing in a second `DateTimeImmutable` object, and the method returns a `DateInterval` object representing the difference between those two date-times. This is useful when a user has provided a start and end date, and some logic or calculation needs to be performed based on the size of the date interval between them. For example, a hotel-booking web application might calculate the cost of a stay based on the number of days between the desired start and end date.

[Listing 31-7](#) shows how this mechanism works.

```
<?php
$date1 = new DateTimeImmutable('1968-11-22');
$date2 = new DateTimeImmutable('1968-11-16');

$interval = $date1->diff($date2);
print '$interval = ' . $interval->
    format('%m months, %d days, %i minutes, %s seconds');
```

Listing 31-7: Obtaining the interval between two `DateTimeImmutable` objects

We create two `DateTimeImmutable` objects, `$date1` and `$date2`, specifying just the date for each; the objects are six days apart. Then we call the `diff()` method on `$date1`, passing `$date2` as an argument. This produces a `DateInterval` object holding the difference between the two dates, which we format and print. Here's the result:

```
$interval = 0 months, 6 days, 0 minutes, 0 seconds
```

As expected, the `DateInterval` object indicates that the two dates are four days apart. Note that the `format()` method of the `DateInterval` class works differently from formatting actual dates. It takes in a string using the percent character (%) to indicate where values from the `DateInterval` object should be inserted. For example, %d is replaced by the number of days (6) in the output string.

Looping at Regular Intervals

For displaying a series of date-time values, updating calendars, or generating historical reports, it's often useful to loop at a regular interval between two dates. PHP's `DatePeriod` class makes this possible. An object of this class can be iterated through with a `foreach` loop, just like an array. Each iteration turns up a new `DateTimeImmutable` object, with all the objects evenly spaced in time.

To create a `DatePeriod` object, you must provide a start and end date, plus a `DateInterval` object defining the rate of iteration. [Listing 31-8](#) shows an example of using this class to automatically list the first seven days of a month.

```
<?php
$today = new DateTimeImmutable();
print 'today: ' . $today->format('l \t\h\e jS \o\f F Y') . PHP_EOL;
$firstOfMonth = $today->modify('first day of this month');
$oneWeekLater = $firstOfMonth->modify('+1 week');

$interval = DateInterval::createFromDateString("1 day");
❶ $period = new DatePeriod($firstOfMonth, $interval, $oneWeekLater);

print '--- first 7 days of current month ---'. PHP_EOL;
❷ foreach ($period as $date) {
    print $date->format('l \t\h\e jS \o\f F Y') . PHP_EOL;
}
```

Listing 31-8: Iterating through a `DatePeriod` object

First, we create and print a `DateTimeImmutable` object for the current day (`$today`). Then we use the `modify()` method to create two more `DateTimeImmutable` objects, `$firstOfMonth` for the first day of the current month, and `$oneWeekLater` for a week after that. These will be the start and end points for the iteration. Next, we create a one-day `DateInterval` object, which we use, along with the start and end points, to create a `DatePeriod` object ❶. The order of arguments is the start date, the interval, and then the end date. Finally, we run a `foreach` loop to iterate through the `DatePeriod` object ❷, printing a formatted string for each date as it comes up. The output should look something like the following:

```
today: Friday the 22nd of November 1968
--- first 7 days of current month ---
Friday the 1st of November 1968
Saturday the 2nd of November 1968
Sunday the 3rd of November 1968
Monday the 4th of November 1968
Tuesday the 5th of November 1968
Wednesday the 6th of November 1968
Thursday the 7th of November 1968
```

Based on a start day of November 22, 1968, the script successfully looped through and displayed the first seven days of that month.

Rather than a start and end date, another way to create `DatePeriod` objects is to give the start date, the interval, and the number of recurrences. This third argument doesn't count the start date itself, so to list the first seven days of a month, the number of recurrences would be 6. The `DatePeriod` constructor also has an optional parameter to exclude the start date by passing a fourth argument of the constant `DatePeriod::EXCLUDE_START_DATE`.

NOTE

At present, `DatePeriod` works only for positive `DateInterval` objects, so loops have to move forward in time.

Time Zones

A *time zone* is a geographical area that observes the same time. These days, all time zones worldwide are defined relative to the time standard of *Coordinated Universal Time (UTC)*. This is the time at the International Reference Meridian (0° longitude), which passes through Greenwich, England. For example, UTC +3 and UTC −2 indicate three hours ahead and two hours behind UTC, respectively. UTC is also nicknamed *Zulu time*, with *Zulu* being the standard codeword for the letter Z in the NATO phonetic alphabet. (The Z is for *zero*.) [Table 31-1](#) lists some example UTC offsets and their associated time zones.

Table 31-1: Example Time Zones

UTC offset	Abbreviation	Common name
UTC +0	GMT	Greenwich Mean Time
UTC +1	BST	British Summer Time
UTC +1	IST	Irish Standard Time
UTC +11	AEDT	Australian Eastern Daylight Time (Tasmania is where my twin brother lives)
UTC −5	EST	Eastern Standard Time
UTC +2	CEST	Central European Summer Time

You can set the default time zone for your system's PHP setup in the *php.ini* configuration file. The time zone itself is given with a *time-zone identifier*, a string that includes a region (such as America, Europe, or Pacific) and a city located in the desired time zone, separated by a forward slash. [Listing 31-9](#), for example, shows I've set up my system to the Europe/Dublin time zone.

```
--snip--
[Date]
; Defines the default timezone used by the date functions
; https://php.net/date.timezone
```



```
date.timezone = Europe/Dublin
--snip--
```

Listing 31-9: An excerpt of a php.ini file setting the default time zone to Europe/Dublin

You use `date.timezone` to set a time-zone identifier for the PHP engine. To verify that your system's time zone has been set correctly, use the `date_default_timezone_get()` function. This returns `Europe/Dublin` for me.

NOTE

You can find a full list of acceptable time-zone identifiers in the PHP documentation at <https://www.php.net/manual/en/timezones.php>. However, avoid using any of the identifiers listed in the Others region, apart from UTC. These identifiers are for backward compatibility only and may be changed in the future.

When you create a `DateTimeImmutable` object in a PHP script, it defaults to the time zone of your system. My preferred way to specify a time zone is to pass a `DateTimeZone` object as a second argument to the `DateTimeImmutable` constructor. Another common method is to append a UTC offset to the end of an ISO 8601 string. For example, adding `+03:00` to the end of the string indicates that the date-time is three hours ahead of UTC. [Listing 31-10](#) illustrates each of these methods.

```
<?php
function prettyPrintDatetime(string $name, DateTimeImmutable
$date)
{
    print '-----' . $name . '-----' . PHP_EOL;
    print $date->format(DATE_ATOM) . ' ' . $date->getTimezon
e()->getName() . PHP_EOL . PHP_EOL;
}

$iceCreamDay = '2009-08-02';
$localDatetime = new DateTimeImmutable($iceCreamDay); ❶
$utcDatetime = new DateTimeImmutable($iceCreamDay, new DateT
```

```

imeZone('UTC')); ❷
$londonDatetime = new DateTimeImmutable($iceCreamDay, new DateTimeZone('Europe/London'));
$parisDatetime = new DateTimeImmutable($iceCreamDay, new DateTimeZone('Europe/Paris'));
$hobartDatetime = new DateTimeImmutable($iceCreamDay, new DateTimeZone('Australia/Hobart'));
$threeHoursAhead = new DateTimeImmutable('2000-01-01T10:00:00 + 03:00'); ❸

print 'local time zone = ' . date_default_timezone_get() . PHP_EOL; ❹
prettyPrintDatetime('local', $localDatetime); ❺
prettyPrintDatetime('UTC', $utcDatetime);
prettyPrintDatetime('London', $londonDatetime);
prettyPrintDatetime('Paris', $parisDatetime);
prettyPrintDatetime('Hobart', $hobartDatetime);
prettyPrintDatetime('+03', $threeHoursAhead);

```

Listing 31-10: *Creating `DateTimeImmutable` objects with different time zones*

We declare a `prettyPrintDatetime()` function to nicely print out a `DateTimeImmutable` object and its time zone, along with a string label passed in as the `$name` parameter. The time zone is accessed using the `DateTimeImmutable` object's `getTimezone()` method, which returns a `DateTimeZone` object. Then we have to call the `DateTimeZone` object's `getName()` method, which returns the name of the time zone as a string.

Next, we declare a series of `DateTimeImmutable` objects with different time zones, all for the date August 2, 2009 (2009-08-02), the first occurrence of National Ice Cream Sandwich Day in the United States. The `$localDatetime` object ❶ holds the date according to the system's default time zone (Europe/Dublin for me, per my *php.ini* file). Since we didn't specify a time, the time will default to midnight.

The `$utcDatetime` object ❷ is set to UTC by passing two arguments to the `DateTimeImmutable` constructor: `$iceCreamDay` to specify the date, and a `DateTimeZone` object set to 'UTC' to specify the time zone. We use this same technique to create objects for the time in London, Paris, and

Hobart. The `$threeHoursAhead` object ❸ is created by appending the UTC offset `+03:00` to the date-time string `2000-01-01T10:00:00` passed to the `DateTimeImmutable` constructor, indicating the time is three hours ahead of UTC.

We print confirmation of the computer system's time-zone setting by using the `date_default_timezone_get()` built-in function ❹. Then we pass our `DateTimeImmutable` objects one at a time to the `prettyPrintDatetime()` function ❺. The output should look something like this:

```
❶ local time zone = Europe/Dublin
-----local-----
2009-08-02T00:00:00 + 01:00 Europe/Dublin

-----UTC-----
2009-08-02T00:00:00 + 00:00 UTC

-----London-----
2009-08-02T00:00:00 + 01:00 Europe/London

-----Paris-----
2009-08-02T00:00:00 + 02:00 Europe/Paris

-----Hobart-----
2009-08-02T00:00:00 + 10:00 Australia/Hobart

-----+03-----
❷ 2000-01-01T10:00:00 + 03:00 +03:00
```

The first data printed is the system's time-zone setting, `Europe/Dublin` for me ❶. Then as each of our `DateTimeImmutable` objects is printed, its time-zone information is visible in the output in two ways: in the UTC offset at the end of the date-time string (for example, `+01:00` for London and Dublin, and `+10:00` for Hobart), and in the separate time-zone string we extract as part of the `prettyPrintDatetime()` function. Notice, however, that when we specify the time zone by using a generic UTC offset like

+03:00 rather than a more specific time-zone identifier, that's how the `DateTimeImmutable` object records the time zone ❷. This is because PHP doesn't know which region and city to associate with the offset. For example, +03:00 could be Europe/Moscow, Asia/Riyadh, or Africa/Mogadishu.

Daylight Saving Time

Around a quarter of all countries worldwide operate a system of daylight saving time: clocks are set forward by one hour in the spring ("spring forward"), then set back by one hour in the autumn ("fall back"). PHP's `DateTimeImmutable` objects automatically account for these changes if the location designated by an object's time-zone identifier observes daylight saving time.

The `format()` method of the `DateTimeImmutable` class has a special value for identifying whether daylight saving time is in effect for that object: an uppercase letter `I`. Calling `format('I')` returns `1` (true) if daylight saving time applies, or `0` (false) if not. [Listing 31-11](#) shows an updated version of the time-zone script from [Listing 31-10](#), with an expanded `prettyPrintDatetime()` function that displays additional information about daylight saving time.

```
<?php
function prettyPrintDatetime(string $name, DateTimeImmutable
$date)
{
    print '-----' . $name . '-----' . PHP_EOL;
    ❶ $isDaylightSaving = $date->format('I');
    if ($isDaylightSaving) {
        $dstString = ' (daylight saving time = TRUE)';
    } else {
        $dstString = ' (daylight saving time = FALSE)';
    }
    print $date->format(DATE_ATOM) . ' ' . $date->getTimezon
e()->getName()
        . $dstString . PHP_EOL . PHP_EOL;
}
```

```

$iceCreamDay = '2009-08-02';
$localDatetime = new DateTimeImmutable($iceCreamDay);
$utcDatetime = new DateTimeImmutable(
    $iceCreamDay, new DateTimeZone('UTC'));
$londonDatetime = new DateTimeImmutable(
    $iceCreamDay, new DateTimeZone('Europe/London'));
--snip--

```

Listing 31-11: An updated `prettyPrintDatetime()` function to output a message about daylight saving time

We call `format('I')` on the `DateTimeImmutable` object passed into the `prettyPrintDatetime()` function, storing the resulting 1 or 0 in the `$isDaylight Saving` variable ❶. Then we use an `if...else` statement to create an appropriate true/false message about daylight saving time based on this variable. All the `DateTimeImmutable` objects have again been created for National Ice Cream Sandwich Day, a useful summertime date in the northern hemisphere for demonstrating whether daylight saving applies in different time zones. Here's the output when this updated script is executed:

```

local time zone = Europe/Dublin
-----local-----
2009-08-02T00:00:00 + 01:00 Europe/Dublin (daylight saving time = FALSE)

-----UTC-----
2009-08-02T00:00:00 + 00:00 UTC (daylight saving time = FALSE)

-----London-----
2009-08-02T00:00:00 + 01:00 Europe/London (daylight saving time = TRUE)

-----Paris-----
2009-08-02T00:00:00 + 02:00 Europe/Paris (daylight saving time = TRUE)

```

```

-----Hobart-----
2009-08-02T00:00:00 + 10:00 Australia/Hobart (daylight savin
g time = FALSE)

-----+03-----
2000-01-01T10:00:00 + 03:00 +03:00 (daylight saving time = F
ALSE)

```

Daylight saving time never applies to UTC, so the UTC line is shown as `FALSE`. The United Kingdom and France observe daylight saving time starting in late March, so London and Paris both show `TRUE`.

Strangely, while the clocks in Ireland do shift forward as well, Dublin is shown to be `FALSE`. This appears to be due to the way the Republic of Ireland's time zone is legally defined as being GMT in the winter and IST (Irish Standard Time) in the summer. By contrast, the United Kingdom and France are defined as being in a *summer* time rather than a *standard* time when daylight saving is in effect (BST, or British Summer Time, for London, and CEST, or Central European Summer Time, for Paris). Therefore, while the `UTC+01:00` offset is correct in the summer for the time zones identified by `Europe/Dublin` and `Europe/London`, one is considered daylight saving time and the other is not by the `format('I')` method.

Epochs and Unix Time

Many computer systems measure time relative to an *epoch*, a fixed point in time that's treated as time 0. For example, Unix systems (including macOS) use the `time_t` format, commonly known as *Unix time*, which represents time in terms of the number of seconds that have elapsed since the beginning (00:00:00) of Thursday, January 1, 1970. [Table 31-2](#) shows a few Unix timestamps and their equivalent ISO 8601 date-times. Notice that timestamps before 1970 are represented as negative values.

Table 31-2: Example Unix Timestamps

Date	Timestamp
1969-12-31 23:59:00	-60
1970-01-01 00:00:00	0

Date	Timestamp
1970-01-01 00:02:00	120
2009-08-02 00:00:00	1249171200

PHP's built-in `time()` function returns the current date and time as a Unix timestamp. While modern PHP programmers typically use `DateTimeImmutable` objects, you may encounter the `time()` function in older code or code from non-object-oriented programmers. Therefore, it's useful to be able to work with code that stores these Unix timestamps. If you have a `DateTimeImmutable` object, you can get its equivalent Unix timestamp by using the object's `getTimestamp()` method. [Listing 31-12](#) shows a script to create objects and print the corresponding timestamps for each row in [Table 31-2](#).

```
<?php
function print_timestamp(string $dateString): void
{
    $date = new DateTimeImmutable($dateString);
    print $date->format('D, F j, Y g.i:s');
    print ' / timestamp = ' . $date->getTimestamp() . PHP_EOL;
}
print_timestamp('1969-12-31T23:59:00');
print_timestamp('1970-01-01T00:00:00');
print_timestamp('1970-01-01T00:02:00');
print_timestamp('2009-08-02T00:00:00');
```

Listing 31-12: Converting `DateTimeImmutable` objects to Unix timestamps

First, we declare a `print_timestamp()` function that takes in a date-time string, creates a `DateTimeImmutable` object for that string, and prints the equivalent timestamp (along with a custom-formatted, human-readable version of the date-time) by using the `getTimestamp()` method. Then we invoke the function four times, once for each row from [Table 31-2](#). Here's the result:

```
Wed, December 31, 1969 11.59:00 / timestamp = -60
Thu, January 1, 1970 12.00:00 / timestamp = 0
Thu, January 1, 1970 12.02:00 / timestamp = 120
Sun, August 2, 2009 12.00:00 / timestamp = 1249171200
```

The reverse of `getTimestamp()` is the `setTimeStamp()` method, which creates a new `DateTimeImmutable` object corresponding to a given Unix timestamp, as shown here:

```
$datetime = (new DateTimeImmutable())->setTimeStamp($timestamp);
```

Notice the extra set of parentheses around `(new DateTimeImmutable())`. This creates a new default `DateTimeImmutable` object, which we then use to call the `setTimeStamp()` method, passing the relevant timestamp in the `$timestamp` variable. This in turn creates another new `DateTimeImmutable` object set to correspond with the timestamp, which we store in the `$datetime` variable. Without the extra parentheses, the statement would look as follows, and the PHP engine wouldn't understand the syntax:

```
// This will not work
$datetime = new DateTimeImmutable()->setTimeStamp($timestamp);
```

When working with code that uses Unix timestamps, I recommend refactoring the code to use `setTimeStamp()` to create an equivalent `DateTimeImmutable` object, then do all the logic with that object. You can then use the `getTimestamp()` method to convert the final result back to a timestamp. Or, better still, refactor all the code to use `DateTimeImmutable` objects with no reference whatsoever to timestamps.

NOTE

Unix timestamps were originally stored using 32-bit integers, a shortsighted scheme. The last timestamp that can be correctly stored in the original 32-bit format will be $+2147483647$ ($2^{31} - 1$), the equivalent of 3:14 AM and 7

seconds on January 19, 2038. Ticking forward one more second would result in an overflow error and a timestamp of -2147483648, or 8:45 PM and 52 seconds on December 13, 1901. Fortunately, most systems have already been upgraded to use 64 bits for Unix timestamps, which postpones the overflow error by 292 billion years.

Date-Time Information in a Web Application

In this section, we'll build a simple web application to synthesize what we've covered so far about working with date-time information in PHP. The application will provide a form for the user to enter an address and a date, and it will display the sunrise and sunset times for that day and location, along with the total duration of daylight for that day as determined by the `DateInterval` class.

As it happens, PHP has a built-in `date_sun_info()` function that reports the sunrise and sunset times (along with other information) for a given date and location. The function requires the location to be specified as latitude and longitude coordinates rather than a street address, however. Our application will therefore also demonstrate how to obtain data from an external API, as we'll rely on OpenStreetMap to convert the address to coordinates. We'll use a popular open source PHP library called Guzzle to communicate with OpenStreetMap. Guzzle provides an HTTP client, allowing code to send and receive HTTP requests. This makes it straightforward to integrate PHP web applications with external web services.

[Figure 31-1](#) features screenshots of the pages we'll create.

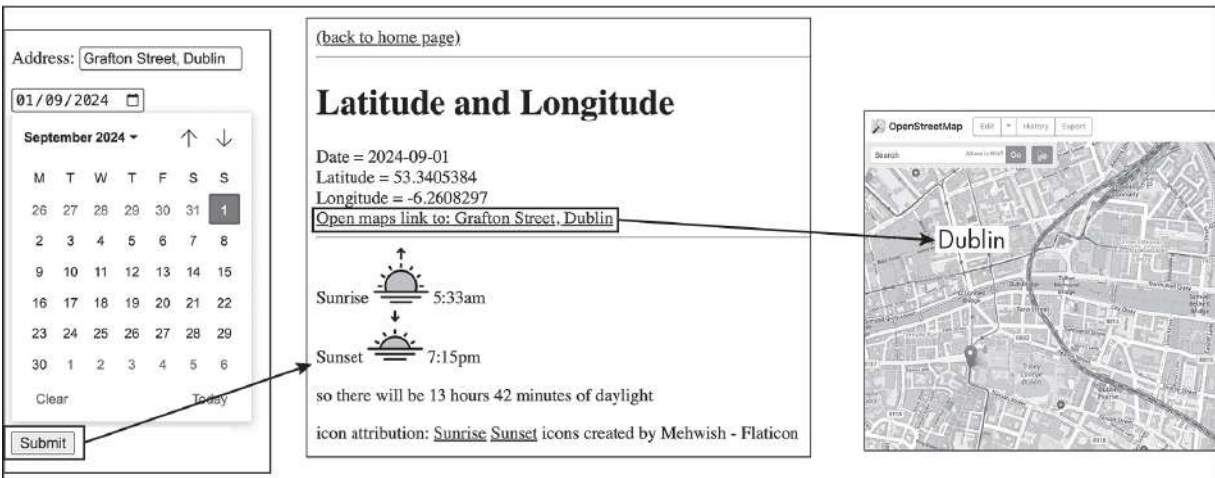


Figure 31-1: Screenshots of the sunrise-sunset web application

The home page allows the user to submit a street address and a date. This leads to a results page showing the calculated information. The results page will also have a link to view the specified location in OpenStreetMap.

To get started, create a new project with the usual *composer.json* file and *public/index.php* script that creates an `Application` object and calls its `run()` method. Then enter `composer require guzzlehttp/guzzle` at the command line to add the third-party Guzzle library to the project. Since we're running Composer and have created the *composer.json* file for our namespace, Composer will also create our namespaced class autoloader at this time.

The Application Class

Now we'll declare the `Application` class for our site in *src/Application.php*. The class will handle requests by either displaying the home page or processing data from the application's web form. [Listing 31-13](#) shows the code.

```
<?php
namespace Mattsmithdev;

class Application
{
    public function run(): void
```

```

{
    $action = filter_input(INPUT_GET, 'action');
    switch ($action) {
        case 'processForm': ❶
            $address = filter_input(INPUT_POST, 'address');
            $date = filter_input(INPUT_POST, 'date');
            if (empty($address) || empty($date)) { ❷
                $this->homepage('you must enter a valid
address and a date');
            } else {
                $this->processForm($date, $address);
            }
            break;

        default: ❸
            $this->homepage();
    }
}

private function homepage(string $errorMessage = ''): void
id ❹
{
    require_once __DIR__ . '/../templates/homepage.php';
}

private function processForm(string $dateString, string
$address): void
{
    try {
        $streetMapper = new StreetMap();
        $latLongArray = $streetMapper->getOpenStreetMapD
ata($address); ❺
        $latitude = $latLongArray['latitude'];
        $longitude = $latLongArray['longitude'];

        $date = new \DateTimeImmutable($dateString);
        $sunData = new SunData($date, $latitude, $longit
ude); ❻
    }
}

```

```

        $sunrise = $sunData->getSunrise()->format('g:i
a');

        $sunset = $sunData->getSunset()->format('g:ia');
        $hoursDaylight = $sunData->getHoursDaylight()->f
ormat("%h hours %i minutes"); ❷

        require_once __DIR__ . '/../templates/result.ph
p'; ❸

    } catch (\Exception) { ❹
        print 'sorry - an error occurred trying to retri
eve data from Open Street Map';
        print '<br>';
        print '<a href="/">home</a>';

    }
}
}

```

Listing 31-13: The *Application* class declaring two routes

As usual, the class's `run()` method uses a `switch` statement to handle incoming requests. First, we declare the case for when `$action` is `'processForm'` ❶. For this case, we attempt to extract the `address` and `date` variables from the `POST` data in the request. If either is empty ❷, we invoke the `homepage()` method, passing in an error message. Otherwise, we pass along the `address` and `date` to the `processForm()` method.

The only other case in the `switch` statement is the default route ❸, which simply invokes the `homepage()` method without any arguments. The `homepage()` method itself ❹ uses `require_once` to display the *homepage.php* template (we're using regular PHP files for templating rather than Twig for simplicity). The method has an `$errorMessage` parameter with a default value of an empty string. This variable will be in the scope of the template for printing.

The meat of this class is the `processForm()` method, which takes in the submitted `address` and `date` as strings and uses them to obtain sunrise and sunset times, along with a total duration of daylight. We first need to convert the `address` into latitude and longitude coordinates. For that, we create a new `StreetMap` object (we'll look at this class shortly) and invoke

its `getOpenStreetMapData()` method ⑤, passing in the `$address` string. The result, in the `$latLongArray` variable, is an array with 'latitude' and 'longitude' keys holding the necessary coordinates, which we extract into separate variables.

Next, we use the date string submitted through the web form to create a corresponding `DateTimeImmutable` object called `$date`. We then create a new `SunData` object (another class we'll look at shortly), passing `$date`, `$latitude`, and `$longitude` as arguments ⑥. The `SunData` object uses the provided information to calculate the sunrise and sunset times and the daylight duration. We extract this data into individual variables via the appropriate getter methods, which are chained with calls to `format()` to convert the date-time information into strings. The `$sunrise` and `$sunset` variables are given in the form 8.35am. The `$hoursDaylight` variable is a `DateInterval` object presented in the form 16 hours 39 minutes, using `%h` for the hours and `%i` for minutes ⑦. With all these variables in scope, we display the *result.php* template ⑧.

All this activity in the `processForm()` method is embedded inside a `try` block. If something goes wrong, such as a failure to connect to OpenStreetMap, the `catch` statement at the end of the method ⑨ displays an error message along with a link back to the home page.

The Supporting Classes

We'll next declare the supporting classes for the application, starting with `StreetMap`, which manages working with the OpenStreetMap web server. Create *src/StreetMap.php* as shown in [Listing 31-14](#).

```
<?php
namespace Mattsmithdev;

use GuzzleHttp\Client;

class StreetMap
{
    private Client $client;

    public function __construct()
```

```

{
    $this->client = new Client([
        'timeout' => 10.0,
        'headers' => [
            'User-Agent' => 'matt smith demo',
            ❶ 'Accept'      => 'application/json',
        ],
        'verify' => true,
    ]);
}

public function getOpenStreetMapData(
    string $address = 'graffon street, dublin, ireland'
): array
{
    $url = $this->buildQueryString($address);
    ❷ $response = $this->client->request('GET', $url);

    if ($response->getStatusCode() == 200) {
        $responseBody = $response->getBody();
        ❸ $jsonData = json_decode($responseBody, true);
        if (empty($jsonData)) {
            throw new \Exception('no JSON data receive
d');
        }
    } else {
        ❹ throw new \Exception('Invalid status code');
    }

    ❺ return [
        'latitude' => $jsonData[0]['lat'],
        'longitude' => $jsonData[0]['lon'],
    ];
}

private function buildQueryString(string $address): stri
ng
{
    ❻ $query = http_build_query([

```

```

        'format'           => 'jsonv2',
        'q'               => $address,
        'addressdetails' => 1,
    ]);

    $url = "https://nominatim.openstreetmap.org/search?
$query";

    return $url;
}
}

```

Listing 31-14: The `StreetMap` class for accessing the OpenStreetMap server

First, the `use` statement allows us to draw on the Guzzle library's `Client` class, which is in the `GuzzleHttp` namespace. This class will manage the details of making an HTTP request to an external site. We then declare the `client` instance variable and initialize it as a `Client` object in the constructor. We provide various Guzzle `Client` parameters, such as the time-out (the amount of time to wait for a response) and the sending agent name ('matt smith demo' is fine for this project). We take care to configure the `Client` to accept JSON data ❶, since that's the format returned by the OpenStreetMap API.

Next, we declare the `getOpenStreetMapData()` method. It takes in an address (I've provided a default value for testing purposes) and uses it to build an appropriate query string via the `buildQueryString()` method. Then it uses the `request()` method of the `Client` object to send the request to the OpenStreetMap API and store the received response ❷. If the response code is valid (200), the received data is decoded into a `$jsonData` array ❸. If either the response code isn't 200 or an empty array is received, we throw an exception to signal to the calling code that there has been a problem getting data from the OpenStreetMap API ❹. If the code gets through the `if...else` statement without throwing an exception, the latitude and longitude are extracted from the received `$jsonData` and returned as an array ❺.

Finally, we declare the `buildQueryString()` method. It uses PHP's built-in `http_build_query()` function to encode the address and other details into an appropriate query string for the Nominatim OpenStreetMap API ❹. We provide the query information to `http_build_query()` as key/value pairs in an array, then attach the encoded query string, in variable `$query`, to the end of a `$url`.

NOTE

For more about the requirements of the Nominatim OpenStreetMap API, see <https://nominatim.org>. Nominatim (Latin for “by name”) is an open source software project that offers searching of OpenStreetMap data. It facilitates both geocoding (location from a given name and address) and reverse geocoding (address from a given location).

Now we'll look at the `SunData` class, which is designed to simplify the process of working with PHP's built-in `date_sun_info()` function. Declare the class in `src/SunData.php` as in [Listing 31-15](#).

```
<?php
namespace Mattsmithdev;

class SunData
{
    private \DateTimeImmutable $sunrise;
    private \DateTimeImmutable $sunset;
    private \DateInterval $hoursDaylight;

    public function __construct(\DateTimeImmutable $date, float $latitude, float $longitude)
    {
        $timestamp = $date->getTimestamp(); ❶
        $data = date_sun_info($timestamp, $latitude, $longitude); ❷

        $this->sunrise = $this->dateFromTimestamp($data['sunrise']);
        $this->sunset = $this->dateFromTimestamp($data['sunset']);
    }
}
```



```

et']);
        $this->hoursDaylight = $this->sunset->diff($this->sunrise); ❸
    }

    private function dateFromTimestamp(int $timestamp): \DateTimeImmutable
    {
        return (new \DateTimeImmutable())->setTimeStamp($timestamp); ❹
    }

    public function getSunrise(): \DateTimeImmutable
    {
        return $this->sunrise;
    }

    public function getSunset(): \DateTimeImmutable
    {
        return $this->sunset;
    }

    public function getHoursDaylight(): \DateInterval
    {
        return $this->hoursDaylight;
    }
}

```

Listing 31-15: The `SunData` class for working with the `date_sun_info()` function

We give the `SunData` class three instance variables: `sunrise` and `sunset` are `DateTimeImmutable` objects for the sunrise and sunset times, and `hoursDaylight` is a `DateInterval` object for the duration of daylight. The `SunData` constructor takes in three arguments: the date (a `DateTimeImmutable` object) and the latitude and longitude of the location of interest. These are the pieces of information the `date_sun_info()` function needs, although the date must be in the form of a Unix timestamp, so the constructor starts by calling `getTime stamp()` to make the conversion ❶.

Then we call `date_sun_info()`, storing the result, an array of information, in the `$data` variable ❷. We extract the sunrise and sunset times from the `$data` array, storing them in the appropriate instance variables. Because `date_sun_info()` returns date-time information as Unix timestamps, we use the `dateFromTimestamp()` helper method to convert back from timestamps to `DateTimeImmutable` objects. (In this method, notice once again that we have to add extra parentheses around the creation of the new `DateTimeImmutable` object before we can call its `setTimestamp()` method ❹.)

For the duration of daylight, we simply take the difference between the sunset and sunrise times ❸. The remainder of the `SunData` class consists of simple getter methods to return the three instance variables.

The Templates

We're now ready to create the templates for the home page (with the web form) and the results page. We'll start with the Home page template in *templates/homepage.php*. [Listing 31-16](#) shows the code.

```
<!doctype html>
<html lang="en">
<head><title>Sun Data</title></head>
<body>
❶ <?php if (!empty($errorMessage)): ?>
    <p style="background-color: pink; padding: 2rem">
        <?= $errorMessage ?>
    </p>
<?php endif; ?>

<form action="/?action=processForm" method="post">
    <p>
        Address:
        <input name="address">
    </p>
    <p>
        ❷ <input name="date" type="date">
    </p>
    <input type="submit">
```

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

Listing 31-16: The form to input an address and date from the user

In the body of the page, we first use the alternative `if` statement syntax to display a pink-styled paragraph containing an error message string, provided the `$errorMessage` variable isn't empty ❶. Then we create a form with an action of `processForm` and fields for the address and date. For the latter, we use an `<input>` element of type `date` ❷, which most web browsers display as a user-friendly calendar date-picker widget, as shown earlier in [Figure 31-1](#).

The second template is for displaying the results to the user. Create `templates/result.php` with the code in [Listing 31-17](#).

```
<!doctype html>
<html lang="en">
<head><title>results</title></head>
<body>
<a href="/">(back to home page)</a> ❶
<hr>

<h1>Latitude and Longitude</h1>
Date = <?= $dateString ?><br>
Latitude = <?= $latitude ?><br>
Longitude = <?= $longitude ?><br>

<a href="http://www.openstreetmap.org/?zoom=17&mlat=<?= $lat
itude ?>&mlon=<?= $longitude ?>"> ❷
    Open maps link to: <?= $address ?>
</a>

<hr>
Sunrise 
<?= $sunrise ?>
<br>
```

```

Sunset 
<?= $sunset ?>
<p>
    so there will be <?= $hoursDaylight ?> of daylight
</p>

<footer> ❸
    icon attribution:
    <a href="https://www.flaticon.com/free-icon/sunrise_3920688" title="sunrise icons">
        Sunrise</a>
    <a href="https://www.flaticon.com/free-icon/sunset_3920799" title="sunset icons">
        Sunset</a>
    icons created by Mehwish - Flaticon
</footer>
</body>
</html>

```

Listing 31-17: The template to present sun data results to the user

In this template, we first offer the user a link back the home page ❶. Then we display the provided date and the latitude and longitude corresponding to the provided address.

Next, we offer a link to view the location in OpenStreetMap, with the values of the `$latitude` and `$longitude` variables inserted into the link for the `mlat` and `m lon` query fields ❷. We then output the sunrise, sunset, and hours of daylight values, along with appropriate images next to the sunrise and sunset times (*sunrise.png* and *sunset.png*, from user Mehwish at <https://www.flaticon.com>).

Links to these images, with acknowledgment to the publisher, are provided in the page's footer element ❸. Download these images and copy them into the *public/images* directory to complete the creation of the web application. Then try running the web server and testing out the application with different dates and addresses.

This project has brought together lots of the concepts from this chapter, demonstrating a practical use for the `DateInterval` class and showing how

to juggle between `DateTimeImmutable` objects and Unix timestamps. It also illustrates the power of open source libraries, in this case showing how Guzzle makes it easy to send requests to external APIs and process the returned JSON data.

MySQL Dates

When working in PHP, using native PHP date and time objects and functions makes sense. When storing temporal data in a database, however, it's best to use the database system's native format. This way, database queries can be performed on the fields, and applications written in other programming languages can also work with the stored database data. It's therefore important to understand the database formats for date-time information and learn how to convert between PHP and the relevant database formats when reading and writing to the database system. In this section, we'll look at how MySQL handles date-time information, but the principles are the same for any DBMS.

MySQL can store dates, date-times, and timestamps, but we'll focus on date-times here. The basic individual date and time formats for MySQL are the same as ISO 8601, with dates in the form `YYYY-MM-DD` and times in the form `HH:MM:SS`. However, MySQL uses a space rather than the letter `T` as the separator between the date and time components, so `1968-11-22T09:05:30` in ISO 8601 format would appear as `1968-11-22 09:05:30` when stored in MySQL. Like PHP, MySQL can add decimal places to the time component to store fractions of seconds, down to microseconds (six decimal places).

To specify that you want a column in a MySQL table to store date-times, declare the column with the `DATETIME` data type. Modern MySQL systems default to zero decimal places for the time component (whole seconds). To include fractions of a second, add the desired number of decimal places in parentheses after the data type. For example, to store date-times down to the microsecond, you'd declare a column of type `DATETIME(6)`.

MySQL stores date-time data as UTC values. Therefore, if the MySQL server is set to a time zone that isn't UTC, it will convert any date-times to UTC for storage and then convert them back from UTC upon retrieval. In

practice, it's common to create UTC `DateTimeImmutable` objects for storage in the database, and to have the web application logic convert retrieved date-times to any other desired time zone.

NOTE

If you're using the `TIMESTAMP` data type in MySQL, be aware that MySQL will automatically convert it to UTC, using the time-zone settings of your MySQL server.

To create a MySQL `datetime` string for insertion into a table from a PHP script, start with a `DateTimeImmutable` object and use its `format()` method, with the format string `'Y-m-d H:i:s'` for whole seconds or `'Y-m-d H:i:s.u'` for fractional seconds. Notice in particular the space between the `d` and the `H`. Likewise, having retrieved a date-time string from MySQL, you can use the `createFromFormat()` static method of the `DateTimeImmutable` class to get an equivalent `DateTimeImmutable` object for that MySQL data.

To demonstrate how to go back and forth between `DateTimeImmutable` objects and MySQL `DATETIME` fields, let's create a project with an `Appointment` entity class and a corresponding `appointment` database table to hold the names and date-times of appointments. Start a new project with the usual `composer.json` file, and enter `composer dump-autoload` at the command line to generate the autoloader. Then create a new MySQL database schema called `date1`, and create an `appointment` table in that schema by using the SQL statement in [Listing 31-18](#). (See “Setting Up the Database Schema” on [page 543](#) to review how to integrate this SQL statement into a PHP script.)

```
CREATE TABLE IF NOT EXISTS appointment (  
    id integer PRIMARY KEY AUTO_INCREMENT,  
    title text,  
    startdatetime datetime(6)  
)
```

Listing 31-18: The SQL to create the `appointment` MySQL database table

The table has an auto-incrementing `id` field for the primary key, a `title` field for a description of the appointment, and a `startdatetime` field for when it begins. We declare the `startdatetime` field to be of type `datetime(6)` to illustrate working with fractions of a second, but note that MySQL's default of zero decimal places would be sufficient for the majority of real-world meeting or appointment applications.

Now we'll declare the `Appointment` class corresponding to this table. Enter the contents of [Listing 31-19](#) into *src/Appointment.php*.

```
<?php
namespace Mattsmithdev;

class Appointment
{
    private int $id;
    private string $title;
    private \DateTimeImmutable $startDateTime; ❶

    public function getId(): int
    {
        return $this->id;
    }

    public function setId(int $id): void
    {
        $this->id = $id;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function setTitle(string $title): void
    {
        $this->title = $title;
    }
}
```

```

    public function getStartDateTime(): \DateTimeImmutable
    {
        return $this->startDateTime;
    }

    public function setStartDateTime(\DateTimeImmutable|string
    $startDateTime): void ❷
    {
        if (is_string($startDateTime)) {
            $startDateTime = \DateTimeImmutable::createFromF
            ormat(
                AppointmentRepository::MYSQL_DATE_FORMAT_STR
                ING, $startDateTime);
        }

        $this->startDateTime = $startDateTime;
    }
}

```

Listing 31-19: The `Appointment` entity class, containing a `DateTimeImmutable` property

The `Appointment` class has `id`, `title`, and `startDateTime` properties to match the columns in the `appointment` table. Notice that the `startDateTime` property is a `DateTimeImmutable` object ❶. We give each property appropriate getter and setter methods. This includes a special setter method for the `startDateTime` property that uses the union type `DateTimeImmutable|string` to allow either a `DateTimeImmutable` object or a string to be provided as an argument ❷.

If the received argument is a string, we convert it to a `DateTimeImmutable` object by using the public constant `MYSQL_DATE_FORMAT_STRING` to help with the formatting. (We'll declare this constant later in the `AppointmentRepository` class.) This mechanism allows the same setter method to work with PHP `DateTimeImmutable` objects as well as the MySQL date-time strings received from the database. The extra logic could be avoided by using an ORM library such as Doctrine, which would seamlessly convert between PHP data types and their database equivalents.

We'll next create an `AppointmentRepository` class with methods to insert a new appointment into the `appointments` table and fetch all the appointments. For simplicity, we'll combine the database connection and repository methods into this one class, but see [Chapter 28](#) for examples of managing the database connection in a separate `Database` class. Create *src/AppointmentRepository.php* containing the code in [Listing 31-20](#).

```
<?php
namespace Mattsmithdev;

class AppointmentRepository
{
    public const MYSQL_DATE_FORMAT_STRING = 'Y-m-d H:i:s.u';
    ❶

    public const MYSQL_DATABASE = 'date1';
    public const MYSQL_HOST = 'localhost:3306';
    public const MYSQL_USER = 'root';
    public const MYSQL_PASS = 'passpass';

    private ?\PDO $connection = NULL;

    public function __construct()
    {
        try {
            $this->connection = new \PDO('mysql:dbname='
                . self::MYSQL_DATABASE . ';host='
                . self::MYSQL_HOST , self::MYSQL_USER, sel
f::MYSQL_PASS
            ); ❷
        } catch (\Exception) {
            print 'sorry - there was a problem connecting to
database ' . self::MYSQL_DATABASE;
        }
    }

    public function insert(Appointment $appointment): int
    {
```

```

        if (NULL == $this->connection) return -1;

        $title = $appointment->getTitle();
        $startDateTime = $appointment->getStartDateTime();
        $dateString = $startDateTime->format(self::MYSQL_DATE_FORMAT_STRING); ❸

        // Prepare SQL
        $sql = 'INSERT INTO appointment (title, startdatetime) VALUES (:title,

:startdatetime)';
        $stmt = $this->connection->prepare($sql);

        // Bind parameters to statement variables
        $stmt->bindParam(':title', $title);
        $stmt->bindParam(':startdatetime', $dateString);

        // Execute statement
        $success = $stmt->execute();

        if ($success) {
            return $this->connection->lastInsertId();
        } else {
            return -1;
        }
    }

    public function findAll(): array
    {
        $sql = 'SELECT * FROM appointment';
        $stmt = $this->connection->prepare($sql);
        $stmt->execute();
        $objects = $stmt->fetchAll(); ❹

        $appointments = [];
        foreach ($objects as $object) {
            $appointment = new Appointment();
            $appointment->setId($object['id']);

```

```

        $appointment->setTitle($object['title']);
        $appointment->setStartDateTime($object['startdate
etime']);
        $appointments[] = $appointment;
    }

    return $appointments;
}
}

```

Listing 31-20: The `AppointmentRepository` class for MySQL database interaction

We declare a public `MYSQL_DATE_FORMAT_STRING` constant holding the string with the necessary formatting for compatibility between MySQL datetimes and PHP `DateTimeImmutable` objects ❶. We then declare more constants for the database credentials (be sure to fill in your own values for these), along with a private `connection` property to hold the PDO database connection object. In the constructor, we create the database connection and store it in the `connection` property ❷, using a `try...catch` structure to handle any problems.

We next declare the `insert()` method, which takes in an `Appointment` object and extracts its `title` and `startDateTime` properties into individual variables. To create the MySQL date string `$dateString`, we pass the `MYSQL_DATE_FORMAT_STRING` constant to the `DateTimeImmutable` object's `format()` method to get the proper string formatting ❸. We then prepare a SQL `INSERT` statement, populate it with values, and execute the statement to add a new row into the `appointment` table.

In the `findAll()` method, we use PDO's `fetchAll()` method to retrieve all the entries from the `appointment` table as an associative array of keys and values ❹. The method then loops through this array, creating an `Appointment` object from each element and adding it to the `$appointments` array, which is then returned.

Finally, we'll create an index script that we can run at the command line to create a few sample `Appointment` objects, add their data to the database, and then retrieve the entries back out of the database as an array of `Appointment` objects. Create `public/index.php` as in [Listing 31-21](#).

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Mattsmithdev\Appointment;
use Mattsmithdev\AppointmentRepository;

$appointmentRepository = new AppointmentRepository();

$appointment = new Appointment();
$appointment->setTitle('get an ice cream sandwich');
$appointment->setStartDateTime(new DateTimeImmutable('2009-08-02T11:00:00.5'));
$appointmentRepository->insert($appointment);

$appointment2 = new Appointment();
$appointment2->setTitle('celebrate birthday');
$appointment2->setStartDateTime(new DateTimeImmutable('2025-11-22T09:05:30.77'));
$appointmentRepository->insert($appointment2);

$appointments = $appointmentRepository->findAll();
foreach ($appointments as $appointment) {
    var_dump($appointment);
}
```

Listing 31-21: The index script /public/index.php testing our MySQL date example

We require the autoloader and declare `use` statements for the `Mattsmithdev` namespaced `AppointmentRepository` and `Appointment` classes. Then we create an `AppointmentRepository` object called `$appointmentRepository`, along with two sample `Appointment` objects, which we insert into the database by using the `$appointmentRepository` object's `insert()` method. Each object is given a time with a fractional second component and is specified using ISO 8601 formatting, including the `T` separator between the date and the time. This formatting doesn't matter, however, since we've written the `AppointmentRepository` class with logic to convert to MySQL date-time formatting. We finish the script by calling the `findAll()` method of the repository object to retrieve all

database rows as an array of `Appointment` objects, which we loop through and pass to `var_dump()`.

Here's the output when the index script is executed:

```
object(Mattsmithdev\Appointment)#9 (3) {
  ["id":"Mattsmithdev\Appointment":private]=>
  int(1)
  ["title":"Mattsmithdev\Appointment":private]=>
  string(25) "get an ice cream sandwich"
  ["startDateTime":"Mattsmithdev\Appointment":private]=>
  object(DateTimeImmutable)#10 (3) {
    ["date"]=>
    string(26) "2009-08-02 11:00:00.500000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(13) "Europe/Dublin"
  }
}
object(Mattsmithdev\Appointment)#11 (3) {
  ["id":"Mattsmithdev\Appointment":private]=>
  int(2)
  ["title":"Mattsmithdev\Appointment":private]=>
  string(18) "celebrate birthday"
  ["startDateTime":"Mattsmithdev\Appointment":private]=>
  object(DateTimeImmutable)#12 (3) {
    ["date"]=>
    string(26) "2025-11-22 09:05:30.770000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(13) "Europe/Dublin"
  }
}
```

Two appointments have been retrieved from the database and output to the console. The first is the 'get an ice cream sandwich' appointment, with a start date-time of 2009-08-02 11:00:00.500000. The second is the

'celebrate birthday' appointment, with a start date-time of 2025-11-22 09:05:30.770000. The date components for both have been stored to six decimal places for fractions of a second. Notice that the time zones are Europe/Dublin, the setting for my PHP setup that is applied by default when new `DateTimeImmutable` objects are created. If the web application was working with dates from different time zones, one solution would be to store the time zone along with the UTC version of each date-time in the database, then convert the retrieved date-times back to that time zone upon retrieval.

Summary

Manipulating dates and times is often a necessary part of developing applications, since date-time information provides useful functionality to users (such as maintaining a calendar) and is valuable for recording data about when actions and requests have occurred. In this chapter, we explored the most useful PHP classes and functions relating to dates and times, including the `DateTimeImmutable` and `DateInterval` classes.

We put these language features to work in a web application that reported sunrise and sunset information, which relied on the Guzzle library to make HTTP requests to an external site. We also looked at how to move date-time information back and forth between PHP scripts and MySQL database tables, converting formats as appropriate.

Exercises

1. Write a script to create (and `var_dump`) `DateTimeImmutable` objects in UTC (Zulu time), Irish Standard Time, and Eastern Standard Time for the following:

```
2025-01-01 10:00:00
```

```
2025-01-02 12:00:00.05
```

2. Write a script to create (and `var_dump`) `DateInterval` objects between 2000-01-01 22:00:00 and the following:

```
2000-01-02 22:00:00
```

```
2010-05-06 00:00:00
```

```
2010-05-06 00:00:30
```

2020-01-01 22:00:00

3. Develop a project to create, store, and retrieve patient meetings with doctors. The project should use a MySQL database for storing the records. Base your project around a `Consultation` entity class containing the following properties:

Patient name (`string`)

Doctor name (`string`)

Consultation date and time (`DateTimeImmutable`)

Duration in minutes (`integer`)

Here's a SQL statement to create a database table for such records:

```
CREATE TABLE IF NOT EXISTS consultation (  
    id integer PRIMARY KEY AUTO_INCREMENT,  
    patient text,  
    doctor text,  
    duration integer,  
    consultationdatetime datetime  
)
```

4. Create a new project to find the hours of daylight in New York and Dublin on the last day of the previous millennium, December 31, 1999.

A

INSTALLING PHP



While it's possible to do PHP development in the cloud, many programmers and students prefer to have PHP installed on their local computer. This appendix goes through the steps to install PHP for macOS, Linux, and Windows computer systems.

macOS

The easiest way to install PHP on macOS is to use Homebrew, a free package manager that greatly simplifies installing and setting up software packages. If you don't already have it, first install Homebrew by visiting <https://brew.sh> and following the instructions there.

Once you have Homebrew installed, enter the following at the command line to install PHP:

```
$ brew install php
```

To verify this has worked, enter the following:

```
$ php -v
```

If PHP has been successfully installed, you should see the latest PHP version number in the output.

You can also use Homebrew to install the Composer dependency manager, which is discussed in [Chapter 20](#):

```
$ brew install composer
```

One last step is to check which, if any, INI configuration file your PHP installation is reading from. Use this command:

```
$ php --ini
```

The output should be something like this:

```
Configuration File (php.ini) Path: /opt/homebrew/etc/php/8.x
Loaded Configuration File:         /opt/homebrew/etc/php/8.x/php.ini
--snip--
```

The first two lines displayed tell you the path and filename of the INI file from which the PHP engine is reading its settings. On my macOS computer, for example, the INI is located at */opt/homebrew/etc/php/8.<x>/php.ini*.

Linux

Many Linux distributions use the Advanced Packaging Tool (APT) package manager for software installation. When using APT at the command line, it's a good idea to first update and upgrade any previously installed packages with the following commands:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

You can then install PHP as follows:

```
$ sudo apt-get install php
$ sudo apt-get install php-cgi
```

This should work for most Linux distributions, such as Ubuntu, but you can find details about installing PHP on a wider range of distributions at <https://www.zend.com/blog/installing-php-linux>.

To verify that the installation has worked, enter the following:

```
$ php -v
```

If PHP has been successfully installed, you should see the latest PHP version number in the output. You should also check which, if any, INI configuration file the PHP installation is reading from with this command:

```
$ php --ini
```

The result should be something like this:

```
Configuration File (php.ini) Path: /etc/php/8.x/cli
Loaded Configuration File:         /etc/php/8.x/cli/php.ini
--snip--
```

The first two lines displayed tell you the path and filename of the INI file from which the PHP engine is reading its settings. On my Ubuntu Linux computer, for example, the INI is located at `/etc/php/8.x/cli/php.ini`.

Windows

The cleanest way to install PHP on a Windows computer is to visit the Downloads page at <https://www.php.net>, click the **Windows Downloads** link for the latest version of PHP listed, and download the ZIP file. Then unzip the contents of that ZIP file to `c:\php`.

NOTE

You don't have to install PHP to c:\php, but I find this to be the easiest location. If you prefer to install PHP elsewhere, make a note of your preferred installation folder path so you can add it to the PATH environment variable, as discussed next.

You now need to add the *c:\php* path to the PATH environment variable for your computer system. In the Quick Launch search box, enter **Environment** and then select **Edit the System Environment Variables** to open the Advanced tab of the System Properties dialog. Next, click the **Environment Variables** button to pull up the Environment Variables dialog. Locate the Path row in the list of system variables and click **Edit**, as shown in [Figure A-1](#).

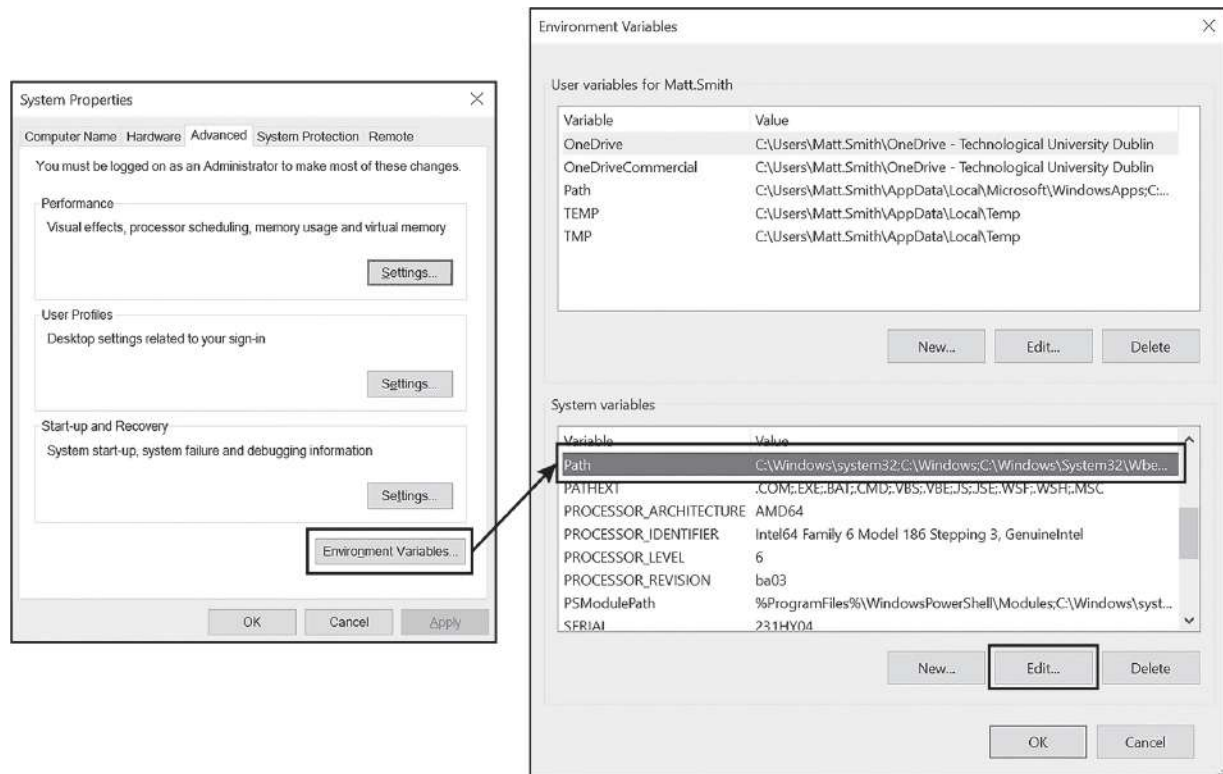


Figure A-1: Accessing the Path row in the Environment Variables dialog

In the next dialog (see [Figure A-2](#)), click **New** and enter the PHP location of *c:\php*. Then keep clicking **OK** to save changes until all the dialogs have closed.

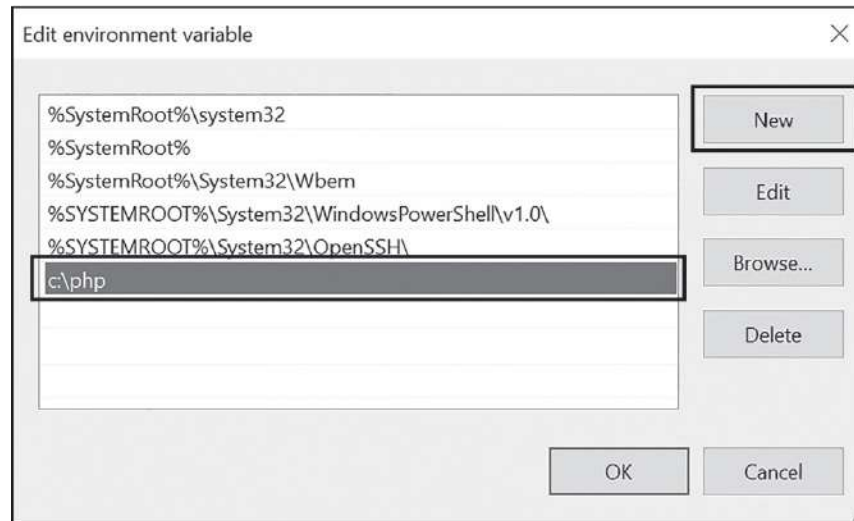


Figure A-2: Adding the `c:\php` path to the `PATH` environment variable

To make sure everything has worked, open up a command line terminal and enter the following:

```
> php -v
```

If PHP has been successfully installed, you should see the latest PHP version number in the output. You should also make sure your installation has an INI configuration file by entering the following at the command line:

```
> php --ini
```

You should see something like this in response:

```
Configuration File (php.ini) Path:  
Loaded Configuration File:          c:\php\php.ini
```

These lines tell you the path and filename of the INI file from which the PHP engine is reading its settings. On my Windows computer, for example, the INI file is located at `c:\php\php.ini`. If no INI file is listed, go to the `c:\php` folder and rename the `php.ini-development` file to `php.ini`. Then run the `php --ini` command again, and you should see this file listed as your loaded configuration file.

For Windows, you should make sure that four commonly used extensions are enabled in the INI file. Once you know where it is, open the file with a text editor and search for `extension=curl`. Then remove any semicolon at the beginning of the lines for these four extensions:

```
extension=curl
extension=pdo_mysql
extension=pdo_sqlite
extension=zip
```

The first and last of these (`curl` and `zip`) will help the Composer tool manage third-party packages for your projects (see [Chapter 20](#)). The middle two (`pdo_mysql` and `pdo_sqlite`) enable the PDO database communication extensions for MySQL and SQLite (see [Chapter 28](#)). If you've made any changes, save the updated text file before closing it.

AMP Installations

While I recommend following the steps outlined in this appendix to install PHP, some developers instead prefer to install a complete AMP stack, which uses a single installer to bundle together the three common components for PHP web application development: a web server such as Apache HTTP Server, a database management system such as MySQL, and the PHP engine. Popular AMP systems include the following:

- XAMPP (for Linux, macOS, and Windows): <https://www.apachefriends.org>
- WampServer (for Windows): <https://wampserver.aviatechno.net>
- MAMP (for macOS and Windows): <https://www.mamp.info>

Visit these websites to learn more about the necessary installation process.

B

DATABASE SETUP



[Part VI](#) of this book outlined how to use PHP to interact with MySQL and SQLite databases. This appendix covers how to make sure these database management systems are set up on your local computer.

MySQL

MySQL is available in various editions. For the purposes of this book, the free MySQL Community Server is sufficient. We'll discuss how to install it for your chosen operating system.

macOS and Windows

To install MySQL Community Server on macOS or Windows, visit <https://dev.mysql.com/downloads/mysql/>. The website should detect the operating system you're using, so you just need to download the latest version of the appropriate installer for your system. For macOS, I recommend one of the DMG Archive files: either the ARM installer for an M-series machine or the x86 installer for an Intel-based machine. For Windows, I suggest the Microsoft Software Installer (MSI).

Once you've downloaded the installer for your system, run it and take the defaults that are offered. The only part of the installation you need to take special care with is when you're asked to enter a password for the root user of the MySQL server. Choose a password you can remember, since you'll need to provide this password in your PHP scripts that communicate with the database server.

Once you've completed the installation process, the MySQL server should be ready to use with your PHP applications. The default installation will configure the server to start up and run in the background each time you restart your system, so you shouldn't need to manually start the MySQL server before using it.

Linux

If you're a Linux user, you'll need to install the PDO and MySQL server extension packages to enable PHP to communicate with MySQL databases using the PDO library. Use the following commands:

```
$ sudo apt-get install php-mysql
$ sudo apt-get install mysql-server
```

The database server should be running once it's installed, which you can check with the following command:

```
$ sudo ss -tap | grep mysql
LISTEN 0      70          127.0.0.1:33060      0.0.0.0:*
users: (("mysqld",pid=21486,fd=21))
LISTEN 0      151         127.0.0.1:mysql      0.0.0.0:*
users: (("mysqld",pid=21486,fd=23))
```

This indicates that the server is active and running on port 33060. If you ever need to restart the MySQL server, you can do so with this command:

```
$ sudo service mysql restart
```

If you wish, you can set a password for the `root` MySQL user as follows (replacing *password* with whatever you prefer):

```
$ sudo mysql
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'password';
mysql> exit
Bye
```

You're now ready to use MySQL databases with your PHP projects.

SQLite

When you install PHP on macOS with Homebrew, SQLite should be enabled by default. On Windows, SQLite will be available as long as the `pdo_sqlite` extension is enabled in your INI file. We discussed how to verify this in [Appendix A](#).

On Linux, use this command to enable PHP to communicate with SQLite databases:

```
$ sudo apt install php-sqlite3
```

As of this writing, version 3 is the latest stable version of SQLite.

Confirming the MySQL and SQLite Extensions

You can check your active PHP database extensions at any time by creating an *index.php* script that calls the `phpinfo()` function. As discussed in [Chapter 1](#), this function prints out a detailed report about your PHP installation. [Listing B-1](#) shows the *index.php* file you need.

```
<?php
phpinfo();
```

Listing B-1: An index.php script to view your PHP settings

Serve this script by entering `php -S localhost:8000` at the command line, then open a browser to *localhost:8000*. Search the resulting page for

PDO to see the list of PDO database extensions. If everything is working, you should see that both MySQL and SQLite are enabled.

C

REPLIT CONFIGURATION



If you've chosen to use the Replit online coding environment to follow along with this book, you'll be able to get started right away using Replit's default PHP settings. As you move through the book, however, you may need to make some changes to make Replit work with more sophisticated tools like the Composer dependency manager and a database management system. This appendix discusses how to reconfigure your Replit projects. The settings we'll discuss apply to both PHP CLI and PHP Web Server projects.

Changing the PHP Version

A new Replit project may not be running the latest version of PHP by default. To find out, enter `php -v` into the Replit command line shell. You should see the PHP version number printed in response. If this isn't the latest version of PHP, you may be able to change the version by editing one of the project's hidden configuration files. First, show the hidden files by

clicking the three vertical dots widget in the left-hand Files column and choosing **Show Hidden Files** (see [Figure C-1](#)).

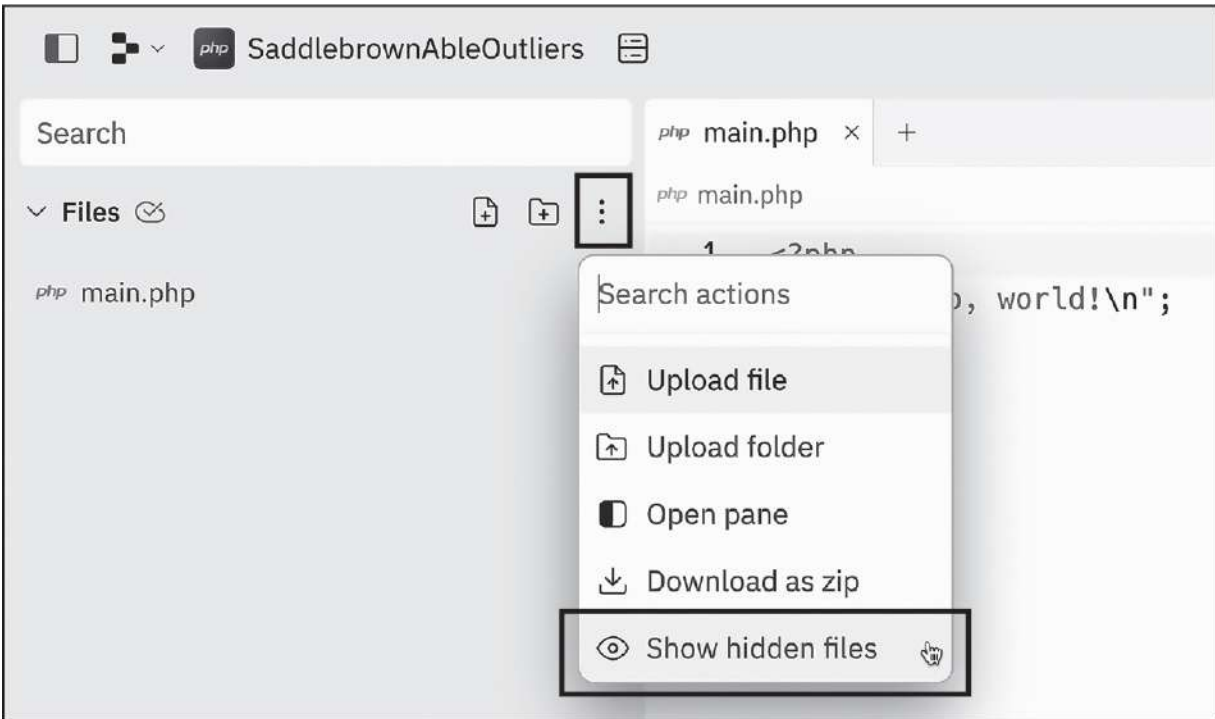


Figure C-1: Showing hidden files for the current Replit project

You should now see a new section in the Files column labeled Config Files and containing two files: `.replit` and `replit.nix`. Select the `replit.nix` file, and you should see its contents in the middle editor column. The contents should look something like [Listing C-1](#).

```
{pkgs}: {  
  deps = [  
    pkgs.php  
  ];  
}
```

Listing C-1: The `replit.nix` configuration file

To change the PHP version, add two digits to the end of `pkgs.php` representing the major and minor version numbers you want (for example,

`pkgs.php82` to use PHP version 8.2.x). Then, if you enter `php -v` at the command line again after a short time, you should see the new version number listed.

This may take some trial and error, as Replit may not be able to work with the absolute latest version of PHP. For example, it can't run PHP 8.3 at the time of this writing, though in the future you should be able to use `pkgs.php83` to run PHP 8.3.x, then `pkgs.php84` for version 8.4.x, and so on.

NOTE

Rather than guessing how long it will take to rebuild the environment after changing a configuration setting, you can close the current shell tab and then open a new one. You won't be shown a command line prompt in the new tab until the new environment has been fully loaded.

Adding the Composer Tool

[Chapter 20](#) introduces the Composer command line tool for dependency management. This tool isn't available by default in Replit PHP projects, but you can easily add it by editing the `replit.nix` configuration file. Make the changes shown in [Listing C-2](#), replacing the `x` after the `8` with the appropriate PHP minor version number, such as `2` for PHP version 8.2.

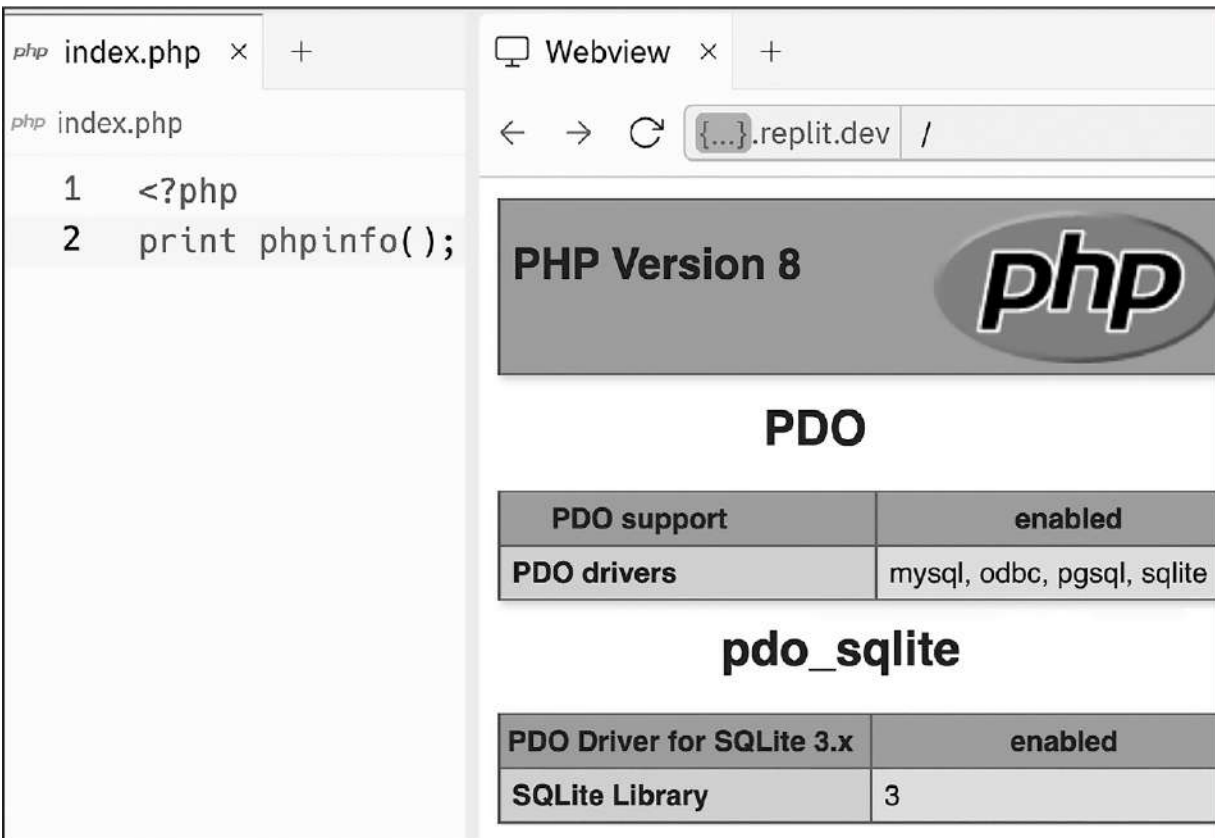
```
{pkgs}: {  
  deps = [  
    pkgs.php8x  
    pkgs.php8xPackages.composer  
  ];  
}
```

Listing C-2: Adding Composer to the `replit.nix` configuration file

After the Replit environment updates, enter `composer` at the command line. If everything is working, you should see a list of all the commands available with the Composer tool.

Using the SQLite Database System

[Part VI](#) introduced database programming with the MySQL and SQLite database systems. If you're following along with Replit, the most straightforward option is to use SQLite, which as of this writing is available by default for all Replit PHP projects. You can verify this by executing the `phpinfo()` function and checking the PDO and `pdo_sqlite` entries, as in [Figure C-2](#).



The screenshot shows a web browser window with the URL `{...}.replit.dev /`. The page displays the output of a PHP script. At the top, it says "PHP Version 8" next to the PHP logo. Below that, the section "PDO" is shown. A table indicates that "PDO support" is "enabled" and lists "PDO drivers" as "mysql, odbc, pgsql, sqlite". Underneath, the section "pdo_sqlite" is shown. Another table indicates that the "PDO Driver for SQLite 3.x" is "enabled" and lists the "SQLite Library" version as "3".

PDO support	enabled
PDO drivers	mysql, odbc, pgsql, sqlite

PDO Driver for SQLite 3.x	enabled
SQLite Library	3

Figure C-2: Confirming PDO SQLite features by printing `phpinfo()`

In the event the default installation of SQLite is removed in the future, you can add it to a Replit project by editing the `replit.nix` configuration file to include the two extra lines shown in [Listing C-3](#). This is how my typical PHP Web Server project `replit.nix` file looked before SQLite was added as a default.

```
{pkgs}: {  
  deps = [  
    pkgs.php8x  
    pkgs.php8xPackages.composer  
    pkgs.php8xExtensions.pdo  
    pkgs.sqlite  
  ];  
}
```

Listing C-3: Enabling SQLite in the replit.nix file

These extra lines add the PDO extension and SQLite to the project. Once again, replace the `x` with the latest minor version number of PHP available.

Serving Pages from the public Directory

When you click the Run button in a PHP Web Server project, Replit defaults to serving all files and folders for the project. As discussed in [Chapter 10](#), however, it's best for security reasons to create a *public* folder for the project and serve only the contents of this folder. While you can always serve from *public* by entering `php -S localhost:8000 -t public` in the shell, you may find it more convenient to change the behavior of the Run button instead. For that, open the hidden *.replit* configuration file and change the first line as follows:

```
run = "php -S 0.0.0.0:8000 -t ./public"
```

If your *index.php* script is located in a *public* folder and you don't make this change, clicking the Run button will trigger a 404 Not Found error, since Replit will be looking for the *index.php* script in the root directory of the project.

INDEX

Symbols

200 OK status code, [179](#)
404 error codes, [179](#), [192](#)
+ (addition operator), [21](#)
& (ampersand), [107](#), [138](#)
&& (AND operator), [39](#), [70](#)
= (assignment operator), [17](#), [22](#), [101](#), [124](#)
\ (backslash escape character), [43](#)
{ } (braces), [47–48](#), [67](#), [85](#), [112](#)
::cases() static method, [495](#)
: (colon), [122](#)
+= (combined assignment and addition operator), [122](#), [124](#)
/= (combined assignment and division operator), [23](#)
*= (combined assignment and exponentiation operator), [23](#)
%= (combined assignment and modulo operator), [23](#)
*= (combined assignment and multiplication operator), [23](#)
-= (combined assignment and subtraction operator), [22](#)
.= (concatenating assignment operator), [45](#)
-- (decrement operator), [23](#)
/ (division operator), [21–23](#)
\$ (dollar sign), [17–18](#), [20](#)
=> (double-arrow operator), [76](#), [135](#), [144](#), [329](#)
{ { } } (double braces in Twig), [399](#)
== (equal operator), [35–36](#), [74–75](#)
\\ (escaped backslash), [46](#)
\" (escaped double quote), [43](#)
\n (escaped newline), [46](#), [52](#), [59](#), [60](#), [170](#)
\' (escaped single quote), [43](#), [59](#)
\t (escaped tab), [46](#), [60](#)
** (exponentiation operator), [21](#)
« » (guillemets, UML notation), [366](#)
=== (identical operator), [36](#), [75](#), [229](#)
++ (increment operator), [23](#), [117–118](#)
?int nullable integer type, [96](#)
% (modulo operator), [21–22](#), [120](#)

`/* ... */` (multiline comment), [16](#)
`*` (multiplication operator), [21](#)
`<>` (not-equal operator), [36](#), [150](#)
`!=` (not-equal operator), [36](#), [150](#)
`!==` (not-identical operator), [36](#), [150](#)
`!` (NOT operator), [36](#), [70–71](#), [122](#)
`??` (null-coalescing operator), [77–78](#), [305](#)
`||` (OR operator), [39](#), [70](#), [72–73](#)
`::` (scope-resolution operator), [372](#), [478](#)
`%20` (space replacement character), [197](#)
`<==>` (spaceship operator), [38](#)
`...` (spread operator), [137](#), [150–151](#)
`[]` (square brackets), [126–127](#)
`{%%}` statements in Twig, [406–407](#)
`.` (string concatenation operator), [35](#), [44](#), [84](#)
`-` (subtraction operator), [21](#)
`?` (ternary operator), [39](#), [76–77](#)
`\u{<hex>}` (Unicode escape), [48](#)
`+` (union operator), [150](#)

A

absolute filepaths, [83](#)
abstract classes, [367](#), [502](#), [505](#), [509](#)
 abstract superclass, [498–499](#), [526](#)
abstraction, [327](#)
`abstract` keyword, [367](#), [437](#)
abstract methods, [497](#), [500–509](#), [526](#)
access control logic, [301](#)
accessor methods, [341–342](#), [363](#)
action controllers, [241](#)
addition, combined assignment and addition operator (`+=`), [122](#), [124](#)
addition operator (`+`), [21](#)
`add()` method (`DateTimeImmutable`), [639](#)
Advanced Packaging Tool (APT), [664](#)
AJAX (Asynchronous JavaScript and XML), [182](#)
aliases, [387](#)
alternative loop syntax, [122–123](#)
 `foreach` loop, [214](#)
AMP (Apache HTTP Server, MySQL, PHP) installations, [667](#)
ampersand (`&`), [107](#), [138](#)
AND operations, [70](#), [71–72](#)
AND operator (`&&`), [39](#), [70](#)
anonymous classes, [338](#)
anonymous functions, [153](#)
antipattern, [491](#)
application programming interface (API), [42](#)

- API key, [473](#)
- architecture, web application, [177](#), [438](#)
- arguments, [55](#), [86](#)
 - named, [102](#)
- arithmetic expressions, [33–34](#)
- arms, [76](#)
- array functions
 - `array()`, [127](#)
 - `array_flip()`, [141](#)
 - `array_is_list()`, [132](#), [144](#)
 - `array_key_last()`, [132](#)
 - `array_map()`, [141](#), [153](#)
 - `array_pop()`, [130](#), [149](#)
 - `array_push()`, [129](#)
 - `array_rand()`, [141](#)
 - `array_slice()`, [141](#)
 - `array_sum()`, [137](#)
 - `array_walk()`, [141](#), [153](#)
- arrays, [28](#), [111](#), [119](#), [125](#)
 - accessing keys and values, [126](#), [134](#)
 - appending an element, [127](#)
 - array-based form validation logic, [235–237](#)
 - checkboxes as, [218–220](#)
 - combining, [150](#)
 - comparing, [150](#)
 - copies, [137–139](#)
 - data type, [137](#)
 - default mapping, [126](#)
 - destructuring into variables, [152](#)
 - elements of, [126](#)
 - empty, [128](#)
 - ID-indexed, [282](#)
 - imploding, [135](#)
 - integer keys, [126](#)
 - negative, [139](#)
 - nonsequential, [144](#)
 - unique, [128](#)
 - key/value pairs, [144](#)
 - looping through, [133](#)
 - managing multiple validation errors with, [227](#)
 - multidimensional, [147–148](#), [542](#)
 - non-integer keys, [129](#)
 - operators, [150–151](#)
 - references, [137–139](#)
 - simple, [125](#)
 - sophisticated, [143](#)
 - as stacks, [131](#)

- string keys, [146](#)
- superglobal, [204](#), [265](#)
- arrow functions, [154](#)
- ASCII (American Standard Code for Information Interchange), [59](#)
 - ASCII art, [90](#)
- assignment operator (=), [17](#), [22](#), [101](#), [124](#)
- Asynchronous JavaScript and XML (AJAX), [182](#)
- ATOM constant, [634](#)
- augmenting inherited behavior, [371–373](#)
- authentication, [301–316](#), [533](#)
 - tokens, [265](#)
- authorization, [301](#), [303](#), [316](#)
- auto-incrementing, [579](#)
- autoloaders, [360](#), [388–391](#)
 - class namespaces, [388](#)
- autoload property, [389](#)

B

- backed enums, [494](#)
- backslash escape character (\), [43](#)
 - escaped (\\), [46](#)
- base template, Twig, [408](#)
- block (Twig keyword), [416–419](#)
- block() function, [419](#)
- blocks of code, [15](#), [416](#)
- bool data type, [28](#)
- Boolean values, [21](#), [33](#)
 - expressions, [66](#), [70](#)
 - flags, [114](#), [116](#)
 - variables, [72](#)
- Bootstrap CSS, [209](#), [304](#), [581](#)
- bootstrap script, [616–17](#)
- braces ({}), [47–48](#), [67](#), [85](#), [112](#)
 - double (Twig output), [399](#)
- break statement, [74](#), [116](#)
- browser
 - developer tools, [180](#), [203–204](#), [208](#)
 - sessions, [262–263](#)
- brute-force techniques, [608](#)
- bubbling, [454–456](#), [467](#)
- buffers, output, [187](#)
- built-in classes, [382](#)
- built-in web server, [190](#)

C

- caching, [303](#), [488](#), [509](#), [616](#)

- callback functions, [153](#)
- call-stack bubbling, [454–456](#), [467](#)
- camel case
 - lower, [19–20](#), [85](#), [216](#)
 - Pascal, [338](#)
 - upper, [338](#)
- Cannot instantiate abstract class error message, [368](#)
- capitalization, [19](#), [37](#)
- Cascading Style Sheets (CSS), [181](#), [190](#), [209](#), [277](#), [291](#), [304](#), [309](#)
- cases, [19–20](#), [37](#), [53–54](#), [74](#), [338](#)
 - sensitivity, [19](#), [57](#)
- `::cases()` static method, [495](#)
- casting, [27](#), [30](#), [39](#)
- catching exceptions, [446–447](#)
- `catch` statements, [442](#)
- channels, [465–466](#)
- checkboxes, [216–220](#)
- child templates, [408](#), [415](#)
- `chmod()` function, [163](#)
- choice statements, [66](#)
- classes, [19](#), [327](#), [329](#), [334](#), [338–339](#)
 - abstract, [367](#), [502](#), [505](#), [509](#)
 - anonymous, [338](#)
 - built-in, [382](#)
 - constants, [480](#)
 - custom exception, [441](#), [451–453](#)
 - declaring, [337–348](#), [374](#)
 - default behavior, [346](#)
 - entity, [535](#), [619](#)
 - enum, [491](#)
 - helper, [622](#)
 - hierarchy, [360](#), [436](#), [504](#)
 - interchangeability, [526](#)
 - log handler, [465](#), [472](#)
 - members of, [329](#)
 - model, [538](#), [542](#), [596](#), [601](#)
 - names, [338](#)
 - repository, [556](#), [570](#), [595](#), [622](#)
 - utility, [482](#)
- client/server communication, [177](#), [196–204](#)
- code blocks, [15](#), [416](#)
- code reuse, [522](#)
- collections, [111](#), [351](#)
- colon (:), [122](#)
- combined assignment operators
 - addition (`+=`), [122](#), [124](#)
 - division (`/=`), [23](#)

- exponentiation (`**=`), [23](#)
- modulo (`%=`), [23](#)
- multiplication (`*=`), [23](#)
- subtraction (`-=`), [22](#)
- command line interface (CLI), [4](#), [6](#), [9](#)
- comma-separated values (CSV), [126](#), [166](#)
- comments, [15–16](#)
- comparative expressions, [35](#)
- comparison operators, [35–37](#)
- compiled programming languages, [6](#)
- Composer, [381](#), [386](#), [398](#), [673](#)
 - autoloader, [389](#)
 - package-dependency features, [391](#)
- composer.json* configuration file, [386](#), [464](#)
- compound data types, [125](#)
- computational efficiency, [596](#)
- concatenating assignment operator (`.=`), [45](#)
- concatenation, [45](#), [126](#), [542](#)
- concatenation, string operator (`.`), [35](#), [44](#), [84](#)
- conditional loops, [112](#)
- conditional statements, [39](#), [65](#)
- confirmation dialog, [570](#)
- consoles, [5](#)
- constants, [20](#)
 - calculated, [481](#)
 - class, [480](#)
 - magic, [83](#), [340](#), [346](#)
 - undefined, [18](#)
- constant-time function, [612](#)
- `__construct()` method, [346](#)
- constructor methods, [346–348](#), [433](#)
 - constructor property promotion, [348](#)
- `continue` keyword, [119](#)
- contracts, [497](#)
- controllers, [240–241](#)
 - action, [241](#)
 - classes, [428](#)
 - multiple, [430–435](#)
 - front, [185](#), [333](#), [428–430](#)
 - Application class as, [409](#)
 - creating, [284](#)
 - updating, [320](#)
 - model-view-controller, [177](#), [189](#), [240](#), [396](#), [535](#)
- cookies, [263–264](#)
 - time out, [270](#)
- Coordinated Universal Time (UTC), [641](#)
- `count_chars()` function, [56](#)

- counter variables, [117](#)
- `count()` function, [131](#)
- `createFromDateString()` method, [639](#)
- `createFromFormat()` method, [635](#)
- creating objects, [339](#)
- credentials, database, [598](#)
 - securing, [615](#)
- CRUD (create, read, update, delete) database operations, [569](#), [585](#), [596](#)
- curly brackets (`{}`), [47](#), [67](#), [85](#), [112](#)
 - double, [399](#)
- current page link, [249](#), [251](#), [305](#), [420](#), [434](#)
- custom exception classes, [441](#), [451–453](#)
- custom form validation logic, [226](#)
- custom methods, [353](#)

D

data

- filtering, [205](#)
- sorting, [39](#)
- source name, [545–546](#)
- structures, [299](#)
- types, [27–28](#), [75](#), [85](#), [88](#), [125](#), [137](#), [492](#)
- validation, [233](#), [255](#)

databases, [531](#)

- abstracting away lower-level work, [596](#)
- columns, [620](#)
- connections, [533](#), [581](#), [598](#)
 - concurrent, [534](#)
- credentials, [596](#), [598](#)
- database-driven applications, [529](#), [543](#), [569](#)
- drivers, [533](#)
- images, [282](#), [460](#)
- integrity of, [532](#)
- management system (DBMS), [533](#)
- programming, [541](#)
- queries, [535](#)
- records in, [532](#)
- relational, [532](#)
- schema, [533](#), [607](#)
- security, [595](#), [615](#)
- tables, [532](#)

date classes

- `DateInterval`, [639](#)
- `DatePeriod`, [640](#)
- `DateTimeImmutable`, [633–641](#)

date functions

- `date_default_timezone_get()`, [643](#)

- `date_sun_info()`, [647](#)
 - `date.timezone`, [642](#)
- dates, [631](#)
 - manipulating, [637](#)
- date-time, [632–640](#)
 - letter codes in, [635](#)
- DATETIME MySQL data type, [656](#)
- DateTime object, [636](#)
- daylight saving time, [631](#), [644](#)
- debugging, [15](#), [446](#)
- decimal place formatting, [100](#), [281](#), [288](#), [292](#), [434](#), [484](#), [564](#), [576](#), [589](#)
- declaring
 - abstract classes, [367](#)
 - abstract methods, [501](#)
 - array keys, [144](#)
 - arrays, [126–127](#)
 - classes, [337–348](#)
 - final, [374](#)
 - enumerations, [492](#)
 - exceptions, [452](#)
 - final methods, [375](#)
 - functions, [82](#), [84–88](#)
 - interfaces, [505](#)
 - static methods, [522](#)
 - traits, [522](#)
- decrement operator (--), [23](#)
- decryption, [608](#)
- default
 - class behavior, [346](#)
 - method implementation, [500](#)
 - routing, [185](#)
 - value, [101](#)
 - visibility, [342](#)
- default keyword, [74](#)
- define() function, [21](#)
- delimiters, [49–50](#)
- dependency management, [391](#)
- deprecation messages, [88–89](#)
- design patterns, [240](#)
- developer tools, browser, [180](#), [203–204](#), [208](#)
- die() function, [234](#)
- diff() method, [639](#)
- __DIR__ constant, [83](#), [340](#)
- directories, [157](#)
 - confirming existence, [161](#)
 - creating, [161](#)
 - recursive creation, [162](#)

- deleting, [165](#)
- paths, [83](#)
 - array of, [169](#)
- permissions, [163](#)
- renaming, [165](#)
- division, combined assignment and division operator (`/=`), [23](#)
- division operator (`/`), [21–23](#)
- Doctrine ORM library, [595](#), [615](#)
 - bootstrap script, [616–17](#)
- dollar sign (`$`), [17–18](#), [20](#)
- don't repeat yourself (DRY) principle, [82](#)
- dotenv files, [598](#)
- Dotenv object, [617](#)
- dot notation, [403](#)
- double-arrow operator (`=>`), [76](#), [135](#), [144](#), [329](#)
- double keyword, [29](#)
- double-precision format, [29](#)
- double-quoted strings, [41](#), [46–48](#)
 - object access, [341](#)
- do...while loops, [111](#), [113](#)
- `DriverManager::getConnection()` method, [619](#)
- drop tables, SQL, [607](#)
- DSN (data source name), [545–546](#)
- dynamic web servers, [183](#)

E

- echo command, [5–7](#), [10](#)
- efficiency, computational, [596](#)
- elements, appending, [127](#)
- elseif statement, [68](#)
- else statement, [67](#)
- `{% else %}` statement in Twig, [407](#)
- emojis, [48](#)
- empty arrays, [128](#)
- `empty()` function, [93](#), [227](#)
- encapsulation, [331](#)
- encryption, [608](#)
- endblock statements, [416](#)
- endfor statements, [123](#)
- endif statements, [69](#)
- entity classes, [535](#), [619](#)
- EntityManager class, [617](#)
- entity-relationship (ER) model, [535–536](#), [544](#)
- enumerations (enums), [475](#), [491–492](#)
 - backed, [494](#)
 - classes, [491](#)
 - value, [494](#)

- `.env` file, [598](#)
- Environment Variables dialog, [665](#)
- EOT (end of text) delimiter, [49](#)
- epochs, [646](#)
- equality tests, [36](#), [74](#)
- equal operator (`==`), [35–36](#), [74–75](#)
- equal sign (`=`), [17](#)
- ER (entity-relationship) model, [535–536](#), [544](#)
 - diagram, [536](#)
- `error_log()` function, [462](#)
- `error()` method, [465](#)
- errors
 - codes, [178–179](#), [192](#)
 - fatal, [87–88](#), [350](#), [454](#)
 - handling, [441](#)
 - messages, [160](#), [368](#), [575](#), [587](#), [611](#)
 - multiple validation, [227](#)
 - pages, [243](#), [256](#)
- escaped backslash (`\\`), [46](#)
- escaped character (`\`), [43](#)
 - date-time formatting, [635](#)
- escaped double quote (`\"`), [43](#)
- escaped newline (`\n`), [46](#), [52](#), [59](#), [60](#), [170](#)
- escaped single quote (`\'`), [43](#), [59](#)
- escaped tab (`\t`), [46](#) [60](#)
- escape sequences, [46–48](#), [50](#)
 - Unicode, [48](#)
- evaluation of expressions, [17](#)
- `Exception` object, [442](#), [471](#), [619](#)
- exceptions, [441–457](#), [460](#)
 - bubbling, [454–456](#), [467](#)
 - built-in, [449–451](#)
 - catching, [446–447](#)
 - custom, [441](#), [451–453](#)
 - finally statements, [447–448](#)
 - logging, [469](#)
 - multiple classes, [449–454](#)
 - subclasses, [456](#)
 - throwing, [442–446](#)
 - uncaught, [442–446](#)
- exclusive-OR (XOR) operations, [70](#), [73](#)
- execution, halting, [442](#)
- exponentiation, combined assignment and exponentiation operator (`**=`), [23](#)
- exponentiation operator (`**`), [21](#)
- expressions, [17](#), [21–24](#), [32–35](#)
 - Boolean, [66](#), [70](#)
 - comparative, [35](#)

- evaluation of, [17](#)
- string, [30](#), [35](#), [44](#), [52](#), [350](#)
- extends keyword, [359](#), [438](#)
- Twig, [418](#)
- eXtensible Markup Language (XML), [173](#), [599](#)

F

- fatal errors, [87–88](#), [350](#), [454](#)
- favicons, [267](#)

- FILE_APPEND option, [164](#)

file functions

- `fclose()` function, [167](#)
- `feof()` function, [168](#)
- `fgetc()` function, [168](#)
- `fgetcsv()` function, [173](#)
- `fgets()` function, [168](#)
- `file_exists()` function, [160](#)
- `file()` function, [166](#)
- `file_get_contents()` function, [158](#), [166](#)
- `file_put_contents()` function, [163](#), [166](#)
- `filesize()` function, [159](#)
- `fopen()` function, [167](#)
- `fputcsv()` function, [173](#)
- `fread()` function, [167](#)
- `fseek()` function, [168](#)
- `ftell()` function, [168](#)

- FILE_IGNORE_NEW_LINES flag, [166](#)

- filename pattern strings, [170](#)

- file not found message, [160](#)

files, [157](#)

- array of paths, [169](#)
- bytestreams, [167](#)
- closing, [168](#)
- confirming existence, [159](#)
- deleting, [164](#)
- end of, [168](#)
- extensions, [10](#)
- permissions, [163](#)
- reading, [158](#)
 - into an array, [166](#)
- renaming, [165](#)
- size, [159](#)
- text, [158](#), [177](#), [190](#), [387](#), [599](#)
 - naming, [5](#)
 - writing to, [163–164](#)
- touching, [161](#)
- `.txt`, [157](#)

- file servers, [182](#)
- `FILE_SKIP_EMPTY_LINES` flag, [166](#)
- filesystem pointers, [167](#)
- `FILTER_DEFAULT` value, [220](#)
- `filter_has_var()` function, [210](#), [217](#)
- `filter_input()` function, [201](#), [203–204](#), [226](#), [243](#)
- `FILTER_REQUIRE_ARRAY` argument, [220](#), [223](#)
- `FILTER_SANITIZE_SPECIAL_CHARS` argument, [205](#)
- final declarations, [374–376](#)
- finally statements, [447–448](#)
- `findstr` command, [463](#)
- Flaticon (website), [655](#)
- float data type, [28–29](#)
- floating-point numbers, [27](#), [29](#)
- flow of control, [333](#)
- foreach loops, [111](#), [134–136](#), [145](#)
- foreign keys, [532](#)
 - relationships in Doctrine, [624](#)
- for loops, [111](#), [117–122](#), [133](#)
 - last iteration, handling the, [120–122](#)
- form actions
 - avoiding double form submissions, [590](#)
 - default, [234](#)
 - processing data, [201](#)
 - validation, [225–226](#), [231–233](#), [235](#)
- `format()` method, [634–635](#)
- for statements, Twig, [406](#)
- front controllers, [185](#), [333](#), [428–430](#)
 - Application class as, [409](#)
 - creating, [284](#)
 - updating, [320](#)
- fully qualified names, [548](#)
 - namespaced classes, [383](#)
- functions, [10–11](#), [17](#), [21](#), [81](#), [85](#). *See also* names of individual functions
 - anonymous, [153](#)
 - callback, [153](#)
 - calling, [82](#), [86](#)
 - declaring, [82](#), [84–88](#)
 - helper, [297](#)
 - higher-order, [153](#)
 - moving website logic into, [245](#)
 - parameters, [85–86](#)
 - return type, [85](#)
 - scope of, [107](#)
 - signature, [85](#), [97](#)
 - variable number of arguments, [136](#)
 - for website logic, [245](#)

G

- generalization, [332](#), [358](#)
- geocoding, [652](#)
- `$_GET` array, [204](#)
- GET HTTP method, [178–181](#)
 - method forms, [200–202](#)
 - requests, [197](#)
- `getInstance()` function, [488](#)
- «*get/set*» annotations, [366](#)
- getter methods, [342](#), [354](#), [363](#), [476](#)
 - Twig, [403](#)
- `getTimestamp()` function, [646](#)
- `getTimezone()` function, [643](#)
- `getType()` function, [29](#)
- Git, [599](#)
- .gitignore* files, [615](#)
- global variables, [82](#)
- global visibility, [491](#)
- `glob()` function, [169](#)
- Glyphicon stars, [280](#)
- greater-than operator (`>`), [35](#), [37](#), [119](#), [188](#)
- greater-than-or-equal-to (`>=`) operator, [37](#)
- `grep` function, [463](#)
- guillemets, UML notation (`«` `»`), [366](#)
- `GuzzleHttp` namespace, [651](#)
- Guzzle library, [648](#)

H

- halting execution, [442](#)
- hardcoded
 - database credentials, [596](#)
 - references, [521](#)
 - values, [90](#)
- hashing, [596](#), [608–609](#)
- haystack strings, [55](#)
- HEAD HTTP method, [179](#)
- “Hello, world!” script, [5–6](#), [15](#)
- helper
 - classes, [622](#)
 - functions, [297](#)
 - methods, [591](#)
 - scripts, [606](#)
- heredocs, [49–52](#)
 - operator, [49](#)
 - strings, [41](#), [49](#), [163](#)
 - indentation in, [50](#)

- hexadecimal code, [48](#)
- hidden variables, [577](#)
- hiding information, [331](#)
- higher-order functions, [153](#)
- highlighting the current page, [249](#), [251](#), [305](#), [420](#), [434](#)
- hit counters, [266](#)
- Homebrew package manager, [663](#)
- .htaccess* files, [264](#)
- HTML (HyperText Markup Language), [7](#), [49](#)
 - template text, [244](#)
- HTTP (HyperText Transfer Protocol), [20](#), [178](#)
 - cookies, [263–264](#)
 - headers, [181](#)
 - method, [204](#)
 - requests, [178](#), [195](#), [197](#)
 - responses, [178](#)
 - status codes, [179](#)
 - Secure (HTTPS), [181](#)
- `http_build_query()` function, [652](#)
- hyperlinks
 - dynamically created, [213](#)
 - encoding data, [211](#)

I

- IDE (integrated development environment), [8](#)
- identical code, [435](#)
- identical operator (`===`), [36](#), [75](#), [229](#)
- identifiers, [16](#)
- identity, [36](#), [75](#), [150](#)
- ID-indexed arrays, [282](#)
- `id` property, [603](#)
- `if...else` statements, [67](#)
 - `if...elseif`, [68](#)
 - `if...elseif...else`, [69](#)
 - nested, [68](#)
- `if` statements, [66](#)
 - alternative syntax, [69](#)
 - in Twig, [406](#)
- images
 - databases, [282](#), [460](#)
 - logo, [306](#)
- `implode()` function, [122](#), [135](#), [220](#)
- `include` command, [83](#)
 - Twig, [414](#)
- increment operator (`++`), [23](#), [117–118](#)
- indentation in heredoc strings, [50](#)
- indexing, zero-based, [54](#)

- index.php* file, [7](#), [11–13](#)
- infinite loops, [123](#)
- information disclosure, minimum, [314](#)
- information hiding, [331](#)
- inheritance, [331](#), [357](#), [427](#), [607](#)
 - augmenting behavior, [371–373](#)
 - multiple, [503–504](#)
 - OOP, [435](#)
 - hierarchies, [497](#)
 - Twig, [415](#)
- INI file, [664](#)
- `INPUT_GET` argument, [201](#)
- `INPUT_POST` argument, [203](#)
- input prompts, [112](#)
- `INSERT` statement, [660](#)
- installing PHP, [663–667](#)
 - extensions, [667](#)
- instances, [328](#), [339](#)
 - instantiation, [339](#), [479](#), [498](#)
- instance-level
 - members, [508](#)
 - properties, [478](#)
- `insteadof` keyword, [525](#)
- `int` data type, [28](#)
- integers, [27–28](#)
- integrated development environment (IDE), [8](#)
- interactive mode, [28](#)
- interfaces, [497](#), [502](#)
 - extension of multiple, [509](#)
 - implementing, [506–507](#)
- interpreters, [6](#)
- `?int` nullable integer type, [96](#)
- `intval()` function, [229](#)
- `InvalidArgumentException` class, [449](#)
- `is_datatype` functions
 - `is_bool()` function, [31](#)
 - `is_dir()` function, [161](#)
 - `is_float()` function, [31](#)
 - `is_int()` function, [31](#)
 - `is_null()` function, [31](#)
 - `is_numeric()` function, [32](#), [228](#)
- ISO 8601 standard, [632](#)
- `isset()` function, [132](#), [204](#), [305](#)
- `is_string()` function, [31](#)
- iterator (`$i`) variable, [117](#)

J

JavaScript Object Notation (JSON), [171](#), [387](#), [514](#), [655](#)

JSON-formatted string, [172](#)

`json_decode()` function, [172](#)

`json_encode()` function, [172](#)

K

keys

accessing, [126](#), [134](#)

API, [473](#)

array, [125](#)

databases, [532](#)

foreign, [532](#), [624](#)

integer, [126](#)

negative, [139](#)

nonsequential, [144](#)

unique, [128](#)

non-integer, [129](#)

primary, [532](#)

string, [146](#)

key-value mapping, [126](#)

L

last iteration, handling the, [120–122](#)

`lcfirst()` function, [54](#)

leading numeric strings, [34](#), [38](#)

Lerdorf, Rasmus, [632](#)

less-than (<) operator, [37](#), [119](#)

less-than-or-equal-to (<=) operator, [37](#), [119](#)

Linux, [664](#)

Liskov substitution principle (LSP), [370–371](#)

lists, [132](#), [144](#)

multiple-selection, [221](#)

single-selection, [220](#)

literal values, [17](#), [21](#)

load balancing, [534](#)

localhost, [12](#)

local variables, [87](#), [107](#)

logfiles, [462–463](#), [467](#)

logging, [446](#), [459–460](#), [472–473](#)

cloud, [472](#)

interface, [464–465](#), [509](#)

`Logger` class, [464](#), [466–474](#), [485–491](#), [509](#)

log handler classes, [465](#), [472](#)

logical comparison operators, [39](#), [70–73](#), [122](#)

login

authentication tokens, [265](#)

- forms, [301](#)
- storing data with sessions, [316](#)
- username, displaying, [321](#)
- verifying hashed passwords at, [608–609](#)
- logo images, [306](#)
- logout feature, [319–321](#)
- LOG_WARNING constant, [463](#)
- lollipop notation, [504–505](#)
- loops, [111](#)
 - alternative syntax, [122–123](#)
 - through arrays, [133](#)
 - conditional, [112](#)
 - do...while, [111](#), [113](#)
 - for, [111](#), [117–122](#), [133](#)
 - foreach, [111](#), [134–136](#), [145](#)
 - infinite, [123](#)
 - Twig, [433](#)
- loose coupling, [526](#)
- lower camel case, [19–20](#), [85](#), [216](#)
- lowercase letters, [37](#), [53–54](#)
- low-level code, [596](#)
- ltrim() function, [59](#)

M

- macOS, [663](#)
- magic constants, [83](#), [340](#), [346](#)
- magic methods, [346](#)
- many-to-one relationships, [626](#)
- mapping, key-value, [126](#)
- match statements, [75–76](#), [96–99](#)
- members of a class, [329](#)
- messages
 - deprecation, [88–89](#)
 - error, [160](#), [368](#), [575](#), [587](#), [611](#)
 - between objects, [327–328](#)
 - pop-up confirmation, [577](#)
 - warning, [17](#)
- metadata, [596](#), [619–620](#)
 - tags, [619](#)
- methods, [329](#), [339](#)
 - abstract, [497](#), [500–509](#), [526](#)
 - accessor, [341–342](#), [363](#)
 - constructor, [346–348](#), [433](#)
 - custom, [353](#)
 - declaring final, [375](#)
 - default implementation, [500](#)
 - helper, [591](#)

- magic, [346](#)
- overriding, [333](#), [357](#), [368–376](#), [498](#)
- signatures, [370](#), [497](#), [527](#)
- static, [480](#)
- minimum information disclosure, [314](#)
- mixins, [522](#)
- `mkdir()` function, [161](#)
- models, [240–241](#)
 - classes, [538](#), [542](#), [596](#), [601](#)
- model-view-controller (MVC) architecture, [177](#), [189](#), [240](#), [396](#), [535](#)
- `modify()` function, [636](#)
- modulo, combined assignment and modulo operator (`%=`), [23](#)
- modulo operator (`%`), [21–22](#), [120](#)
- Monolog library, [464](#)
- multidimensional arrays, [147–148](#), [542](#)
- multiline comment (`/* . . */`), [16](#)
- multiple attribute, [221](#)
- multiple controller classes, [430–435](#)
- multiple exception classes, [449–454](#)
- multiple inheritance, [503–504](#)
- multiple interfaces, extension of, [509](#)
- multiple levels of inheritance, [361](#)
- multiple return types, [95](#)
- multiple-selection lists, [221](#)
- multiple validation errors, [227](#)
- multiplication operator (`*`), [21](#)
- MVC (model-view-controller architecture), [177](#), [189](#), [240](#), [396](#), [535](#)
- MySQL, [533](#), [542](#), [667](#), [669](#)
 - dates, [655](#)
 - server, [599](#)
- `MYSQL_DATE_FORMAT_STRING` constant, [658](#)

N

- `\n` (newline character), [5](#), [10](#), [14–15](#), [42–43](#), [46](#), [158](#)
- named arguments, [102](#)
- namespaces, [381–384](#)
 - fully qualified names, [383](#)
 - referencing, [384–385](#)
 - root, [382](#), [444](#)
- name/value pairs, [198](#)
- naming collision, [381](#)
- navigation bars, [396](#)
- needle strings, [55](#)
- `new` keyword, [339](#), [346](#)
- Nominatim, [652](#)
- non-numeric characters, [32–34](#)
- non-numeric strings, [35](#)

- not-equal operator (`!=`), [36](#), [150](#)
- not-equal operator (`<>`), [36](#), [150](#)
- not-identical operator (`!==`), [36](#), [150](#)
- NOT operator (`!`), [36](#), [70–71](#), [122](#)
- nowdoc strings, [41](#), [52–53](#)
- nullable
 - parameters, [97](#)
 - types, [95–98](#)
- null-coalescing operator (`??`), [77–78](#), [305](#)
- NULL type, [28](#), [30–31](#), [37](#), [77–78](#)
 - connection, [573](#)
 - referring to missing objects, [351–352](#)
 - returning, [91](#)
- `number_format` filter, [434](#), [564–565](#), [576](#), [589](#)
- `number_format()` function, [100](#), [281](#), [288](#), [292](#), [481](#), [484](#)
- numeric
 - characters, [32–34](#)
 - comparisons, [36](#)
 - strings, [34](#), [38](#)

O

- object fetch mode, [542](#), [548](#)
- object operator (`->`), [45](#), [329](#), [339–340](#)
- object-oriented
 - PHP, [325](#), [427](#)
 - programming, [19–20](#), [327](#), [427](#)
 - inheritance, [435](#), [497](#)
 - web applications, [402](#), [438](#)
- object-relational mapping (ORM), [595–596](#), [615](#)
- objects, [28](#), [31](#), [327](#), [339](#)
 - resource, [167](#)
- online coding environments, [4](#)
- open source projects, [598](#)
- OpenStreetMap, [648](#)
- operands, [21](#), [32](#)
- operators, [21](#)
 - arithmetic, [21–22](#), [33](#)
 - array, [150–151](#)
 - assignment, [17](#), [101](#), [124](#)
 - combined arithmetic assignment, [22–23](#)
 - comparison, [35](#)
 - concatenating assignment, [45](#)
 - decrement, [23](#)
 - equal, [35–36](#)
 - identical, [36](#), [75](#)
 - increment, [23](#)
 - logical AND, [39](#), [70](#), [71–72](#)

- logical NOT, [36](#), [70–71](#), [122](#)
- logical OR, [39](#)
- null-coalescing, [77–78](#), [305](#)
- order of precedence, [22](#)
- string concatenation, [44](#)
- ternary, [39](#), [76–77](#)
- unary, [23](#)
- optional parameters, [100](#)
- ORM (object-relational mapping), [595–596](#), [615](#)
- OR operations, [70](#)
- OR operator (`||`), [39](#), [70](#), [72–73](#)
- output buffers, [187](#)
- overriding
 - methods, [333](#), [357](#), [368–376](#), [498](#)
 - preventing, [374](#)
 - Twig, [417](#), [434](#)

P

- Packagist (website), [392](#)
- padding, [61](#)
- page-generation logic, [430](#)
- page header HTML templates, [287](#)
- parameters, [85–86](#)
 - nullable, [97](#)
 - optional, [100](#)
 - skipped, [104](#)
- parentheses (`()`), [22](#), [71](#)
- `parent` keyword, [371](#)
- parsed strings, [35](#), [46](#)
- partial templates, [287](#), [414](#)
- Pascal case, [338](#)
- pass-by-reference approach, [105](#), [130](#)
- pass-by-value approach, [105](#)
- `password_hash()` function, [608](#)
- passwords, [112](#), [601](#)
 - hashing, [596](#)
 - verifying at login, [608–609](#)
 - text fields, [302](#)
- `password_verify()` function, [609](#)
- PATH environment variable, [665](#)
- `PATH_TO_TEMPLATES` constant, [400](#)
- `pdo-crud-for-free-repositories` library, [598](#)
- permissions, files and directories, [163](#)
- PHP
 - Data Objects, [541](#), [670](#), [675](#)
 - engine, [6](#), [11](#), [15](#)
 - Extension Community Library, [509](#)

- extensions, [667](#)
- Hypertext Preprocessor, [xxvi](#)
- installing, [663–667](#)
- libraries, [392](#)
- object-oriented, [325](#), [427](#)
- Standards Recommendations, [81](#)
 - PSR-3, [464](#)
 - PSR-4, [388–389](#)
 - PSR-6, [510](#)
 - PSR-16, [510](#)
- tags, [187](#)
- `PHP_EOL` constant, [44](#), [47](#)
- `phpinfo()` function, [11](#)
- php.ini* file, [264](#), [462](#), [533](#), [642](#)
- plus sign (+), [21](#)
- popping (stack), [131](#)
- pop-up confirmation messages, [577](#)
- ports, [12](#)
- `$_POST` array, [204](#)
- postback scripts, [230](#)
- PostgreSQL, [629](#)
- POST HTTP method, [178](#), [291](#), [571](#)
 - method forms, [202–204](#)
 - requests, [197](#)
- post-increment values, [24](#)
- post-redirect-get (PRG) pattern, [590](#)
- power operator (**), [21](#)
- precedence, [22](#), [70–71](#)
- pre-increment values, [24](#)
- prepared statements, [542](#), [545](#)
- primary keys, [532](#)
- `print_r()` function, [150](#)
- print statements, [10](#), [14](#)
- `print_timestamp()` function, [646](#)
- private constructors, [488](#)
- private keyword, [335](#), [362](#)
- private properties, [341](#)
- procedural programming, [327](#)
- process IDs, [463](#)
- processing form data, [201](#)
- project structures
 - directory, [190](#)
 - file, [304](#)
 - secure, [245](#)
- properties of a class, [329](#), [338–339](#)
- protected keyword, [342](#), [362](#), [436](#)
- pseudo-variables, [343](#)

PSR (PHP Standards Recommendations), [81](#)

PSR-3, [464](#)

PSR-4, [388–389](#)

PSR-6, [510](#)

PSR-16, [510](#)

`public` keyword, [335](#), [338](#), [362](#)

public properties, [338–339](#), [341](#)

pushing (stack), [131](#)

Q

qualified names, [384](#)

query-string variables, [197–198](#)

hardcoded IDs, [207](#)

in hyperlinks, [206](#)

quotation marks, [14](#), [21](#), [30](#), [88](#)

double (`""`), [6](#), [41](#), [50](#)

escaped, [43](#)

single (`' '`), [41–43](#)

R

radio buttons, [215](#)

read-evaluate-print loop (REPL), [4](#)

`readline()` function, [113](#)

records in databases, [532](#)

redirects, [580](#), [590](#)

reference, pass-by-reference approach, [105](#), [130](#)

reference operator (`&`), [138](#)

references, [31](#), [329](#), [339](#), [351](#)

to arrays, [137–139](#)

hardcoded, [521](#)

referencing namespaces, [384–385](#)

reflection, [598](#)

relational databases, [532](#)

management system, [533](#)

relationships

databases, [532](#)

ER model, [536](#)

between objects, [330](#)

relative paths, [83](#)

`rename()` function, [165](#)

`render()` function, [397](#)

Replit, [673](#)

replit.nix file, [674](#)

repository classes, [556](#), [570](#), [595](#), [622](#)

`REQUEST_METHOD` keyword, [232](#)

request-response cycle, HTTP, [178](#)

- requests for comments (RFC), [89](#)
 - RFC 5424 levels, [460](#)
- require command, [83](#)
- require_once command, [82–84](#)
- resource-expensive operations, [488](#)
- resource objects, [167](#)
- return statements, [86](#), [91](#)
- return types, [85](#), [347](#)
 - multiple, [95](#)
- reusability, [82](#)
- reverse geocoding, [652](#)
- rewind() function, [168](#)
- rmdir() function, [165](#)
- root namespace, [382](#), [444](#)
- rounding down, [40](#)
- routing, [184–185](#)
- rtrim() function, [59](#)
- runtime notices, [88](#)

S

- sanitization, [205](#)
- scalable web applications, [239](#)
- scalar data types, [28](#), [39](#), [147](#), [150](#)
- schema, database, [533](#), [607](#)
- scope
 - of functions, [107](#)
 - of variables, [87](#), [247](#)
- scope-resolution operator (: :), [372](#), [478](#)
- scripts, [5–6](#), [15](#), [388](#)
 - bootstrap, [623](#)
 - helper, [606](#)
 - postback, [230](#)
- security, [301](#)
 - best practices, [608](#)
 - credentials, [189](#)
 - database, [595](#), [615](#)
 - of project folder structures, [245](#)
 - Twig, [399](#)
 - vulnerability, [446](#)
- self:: prefix, [401](#), [478](#)
- semicolons, [6](#)
- sensitivity, case, [19](#), [57](#)
- sentinel values, [478](#)
- separating display and logic files, [242](#)
- \$_SERVER array, [232](#)
- server-based database management systems (DBMS), [533](#)
- servers, [195](#)

- client communication, [177](#), [196–204](#)
- file, [182](#)
- `$_SESSION` array, [265](#), [275](#), [317](#), [591](#)
- `session.auto_start` configuration setting, [264](#)
- `session_destroy()` function, [270](#)
- `session_id()` function, [264](#)
- sessions, [261–272](#), [301](#)
 - destroying, [270](#)
 - IDs, [262](#)
 - shopping carts, [275](#)
 - storing login data with, [316](#)
 - time out, [263](#)
- `session_start()` function, [264](#), [298](#), [593](#)
- `setDate()` function, [638](#)
- setter methods, [335](#), [342](#), [354](#), [363](#), [476](#)
- `setTime()` function, [638](#)
- `setTimeStamp()` function, [647](#), [652–653](#)
- severity levels, [463](#), [474](#)
 - logging, [460](#)
- shared header templates, [304](#)
- sharing static resources, [485](#)
- shopping carts, [13](#), [16](#), [211](#), [275](#), [397](#)
 - subtotal calculation in, [288–290](#)
- short echo tags, [188](#)
- side effects, [82](#)
- signatures of methods, [370](#), [497](#), [527](#)
- simple arrays, [125](#)
- `SimpleXMLElement` class, [173](#)
- `simplexml_load_file()` function, [173](#)
- single-quoted strings, [41–43](#)
 - nowdoc strings, [52](#)
- single-selection lists, [220](#)
- singleton pattern, [488](#)
- `sizeof()` function, [132](#)
- skipped parameters, [104](#)
- snake case, [19–20](#), [85](#)
- snapshots of databases, [460](#)
- software architecture, [430](#)
- sophisticated arrays, [143](#)
- `sort()` function, [141](#)
- sorting data, [39](#)
- space replacement character (`%20`), [197](#)
- spaceship operator (`<=>`), [38](#)
- spread operator (`...`), [137](#), [150–151](#)
- SQL (Structured Query Language), [49](#), [534](#), [570](#)
 - drop tables, [607](#)
 - injection, [542](#)

- statements, [534](#)
- SQLite, [533](#), [542](#), [667](#), [675](#)
 - installation, [671](#)
- square brackets (`[]`), [126–127](#)
- `src` directory, [245](#), [338](#)
- stacks, [131](#)
- stack trace, [446](#)
- Standard PHP Library (SPL), [449](#)
- statements, [6](#), [13](#), [24](#)
 - branch, [68](#)
 - choice, [66](#)
 - conditional, [39](#), [65](#)
 - group, [67](#)
 - prepared, [542](#), [545](#)
- static, [475](#)
 - content, [183](#)
 - members, [367](#), [476](#)
 - methods and properties, [480](#)
- status codes, [179](#)
- stereotyping, [366](#)
- sticky forms, [230](#)
- `str_contains()` function, [72](#)
- `StreamHandler` class, [464–465](#)
- `str_getcsv()` function, [173](#)
- string concatenation operator (`.`), [35](#), [44](#), [84](#)
- `string` data type, [90](#)
- string literal values, [21](#)
- strings, [6](#), [28](#), [41](#)
 - array of characters, [139](#)
 - comparison, [36](#)
 - functions, [612](#)
 - concatenation, [45](#), [126](#), [542](#)
 - double-quoted, [41](#), [46–48](#)
 - expression, [42](#)
 - functions, [41](#), [53](#)
 - heredoc, [41](#), [49](#), [163](#)
 - keys, [146](#)
 - negative integer, [139](#)
 - needle and haystack, [55](#)
 - non-numeric, [35](#)
 - nowdoc, [41](#), [52–53](#)
 - numeric, [34](#), [38](#)
 - parsed, [35](#), [46](#)
 - replacement, [58](#)
 - searching, [54](#)
 - single-quoted, [41](#), [43](#)
 - unparsed, [52](#)

- `stristr()` function, [57](#)
- `strlen()` function, [55](#), [73](#), [112](#), [228](#)
- `STR_PAD_BOTH` constant, [61–62](#)
- `str_pad()` function, [61](#), [90](#)
- `STR_PAD_LEFT` constant, [61–62](#)
- `strpos()` function, [55](#)
- `str_repeat()` function, [61](#)
- `str_replace()` function, [58](#)
- `str_split()` function, [140](#)
- `strtolower()` function, [53](#)
- `strtoupper()` function, [53](#)
- structures, data, [299](#)
- subclasses, [331](#), [357](#), [451](#)
 - exceptions, [456](#)
- subclassing, preventing, [374](#)
- `sub()` method (`DateTimeImmutable`), [639](#)
- submit buttons, [200](#), [208](#)
- `substr_count()` function, [55](#)
- `substr()` function, [56](#)
- substrings, [54–56](#)
- `substr_replace()` function, [58](#)
- subtraction, combined assignment and subtraction operator (`--`), [22](#)
- subtraction operator (`-`), [21](#)
- superclasses, [331](#), [357](#), [435](#)
 - abstract, [498–499](#), [526](#)
- superglobal arrays, [204](#), [265](#)
- switch statements, [73](#)
- Symfony web framework, [386](#)
- `syslog()` function, [460](#), [462](#)

T

- `\t` (tab escape character), [83](#)
- tables, database, [532](#)
- tabs, [42](#)
- tags, [5](#), [619](#)
 - PHP, [187](#)
 - short echo, [188](#)
- templating, [185](#), [395](#)
 - library, [396](#)
 - script, [281](#)
 - text, [13–14](#), [244](#)
- ternary operator (`?`), [39](#), [76–77](#)
- textarea input, [253](#)
- text box form input, [200](#)
- text/html content type, [181](#)
- third-party libraries, [382](#), [390](#)

- `$this` keyword, [20](#), [335](#), [343](#)
- throwing exceptions, [442–446](#)
- `throw` statements, [442](#)
- `time()` function, [646](#)
- times, [631](#)
 - daylight saving, [631](#), [644](#)
 - manipulating, [637](#)
 - offsets, [633](#)
- `timestamp()` function, [653](#)
- timestamps, [40](#), [463](#), [656](#)
 - logging, [473](#)
- `time_t` format, [646](#)
- time zones, [631](#), [641](#)
 - identifiers, [642](#)
- timing attacks, [612](#)
- `__toString()` method, [346](#), [349](#)
- `touch()` function, [161](#)
- traits, [497](#), [521](#)
 - resolving conflicts, [525](#)
- `trim()` function, [59](#)
- `try...catch` statements, [446](#)
- Twig, [395](#)
 - Composer, [398](#)
 - control structures, [406](#)
 - inheritance, [415](#)
 - loops, [433](#)
 - security, [399](#)
 - templates, [408](#)
 - variables, [397](#)
- type casting, [27](#), [39](#)
- type conversions, [34](#)
- `TypeError` message, [35](#), [88](#)
- type juggling, [27](#), [30](#), [32–39](#), [89–90](#)
 - comparative, [35–39](#)
 - logical, [39](#)
 - numeric, [33–35](#)
 - string, [35](#)
- types, data, [27–28](#), [75](#), [85](#), [88](#), [492](#)

U

- `\u{<hex>}` Unicode escape sequence, [48](#)
- Ubuntu, [665](#)
- `ucfirst()` function, [54](#)
- `ucwords()` function, [54](#)
- unary operator, [23](#)
- undefined
 - constants, [18](#)

- variables, [17](#)
- Undefined property warning, [364](#)
- Unicode characters, [48](#)
- Unified Modeling Language (UML), [339](#)
 - diagram, [365](#)
- union operator (+), [150](#)
- union types, [98](#)
- Unix time, [646](#)
- `unlink()` function, [165](#)
- unparsed strings, [52](#)
- `unset()` function, [31](#), [132](#), [149](#), [269](#)
- unsettling a variable, [40](#)
- upper camel case, [338](#)
- uppercase letters, [53–54](#)
- username/password login forms, [608](#)
 - displaying logged-in username, [321](#)
- use statements, [384](#)
- `usort()` function, [141](#)
- UTC (Coordinated Universal Time), [641](#)
- utility classes, [482](#)

V

- validation, [205](#), [231](#), [233](#), [255](#)
 - errors, [227](#)
 - form data, [225–226](#), [233](#)
 - array-based, [235–237](#)
- value-backed enums (`->value`), [494](#)
- `var_dump()` function, [30](#), [607](#)
- variable multiplication operator (`*=`), [23](#)
- variables, [16–21](#)
 - Boolean, [72](#)
 - counter, [117](#)
 - destructuring arrays into, [152](#)
 - global, [82](#)
 - hidden, [577](#)
 - local, [87](#), [107](#)
 - names, [47](#)
 - pseudo, [343](#)
 - query-string, [197–198](#)
 - scope of, [87](#), [247](#)
 - Twig, [397](#)
 - undefined, [17](#)
 - unsettling, [40](#)
- vendor folders, [390](#)
- view, [240](#)
 - model-view-controller architecture, [189](#), [396](#)
- virtual attributes, [354](#)

virtual machines, [4](#)
visibility
 default, [342](#)
 global, [491](#)
 keywords, [362](#)
void return type, [90](#)

W

warning() function, [465](#)
warnings, [88](#), [364](#)
 messages, [17](#)
web applications
 architecture of, [438](#)
 multipage, [249](#)
 object-oriented, [402](#), [438](#)
 scalable, [239](#)
web forms, [195](#)
 client/server communications for, [196–203](#)
 encoding data, [211–214](#)
 filtering, [203–206](#)
 input types, [214–223](#)
 mixed variables, [207–208](#)
 multiple submit buttons, [208–211](#)
 noneditable data, [206–207](#)
web servers, [4](#), [7–8](#), [11–12](#)
 built-in, [190](#)
 dynamic, [183](#)
 installing a different, [192](#)
 public directory, [676](#)
while loops, [111–112](#)
whitespace, [33](#), [42](#), [188](#)
 trimming, [59](#)
Windows, [665–667](#)

X

XML (eXtensible Markup Language), [173](#), [599](#)
XOR (exclusive-OR) operations, [70](#), [73](#)

Y

YAML (YAML Ain't Markup Language), [173](#), [599](#)
YAML functions
 yaml_emit(), [173](#)
 yaml_emit_file(), [173](#)
 yaml_parse(), [173](#)
 yaml_parse_file(), [173](#)

Z

zero, numeric value of, [228](#)

zero-based indexing, [54](#)

Zulu time, [641](#)