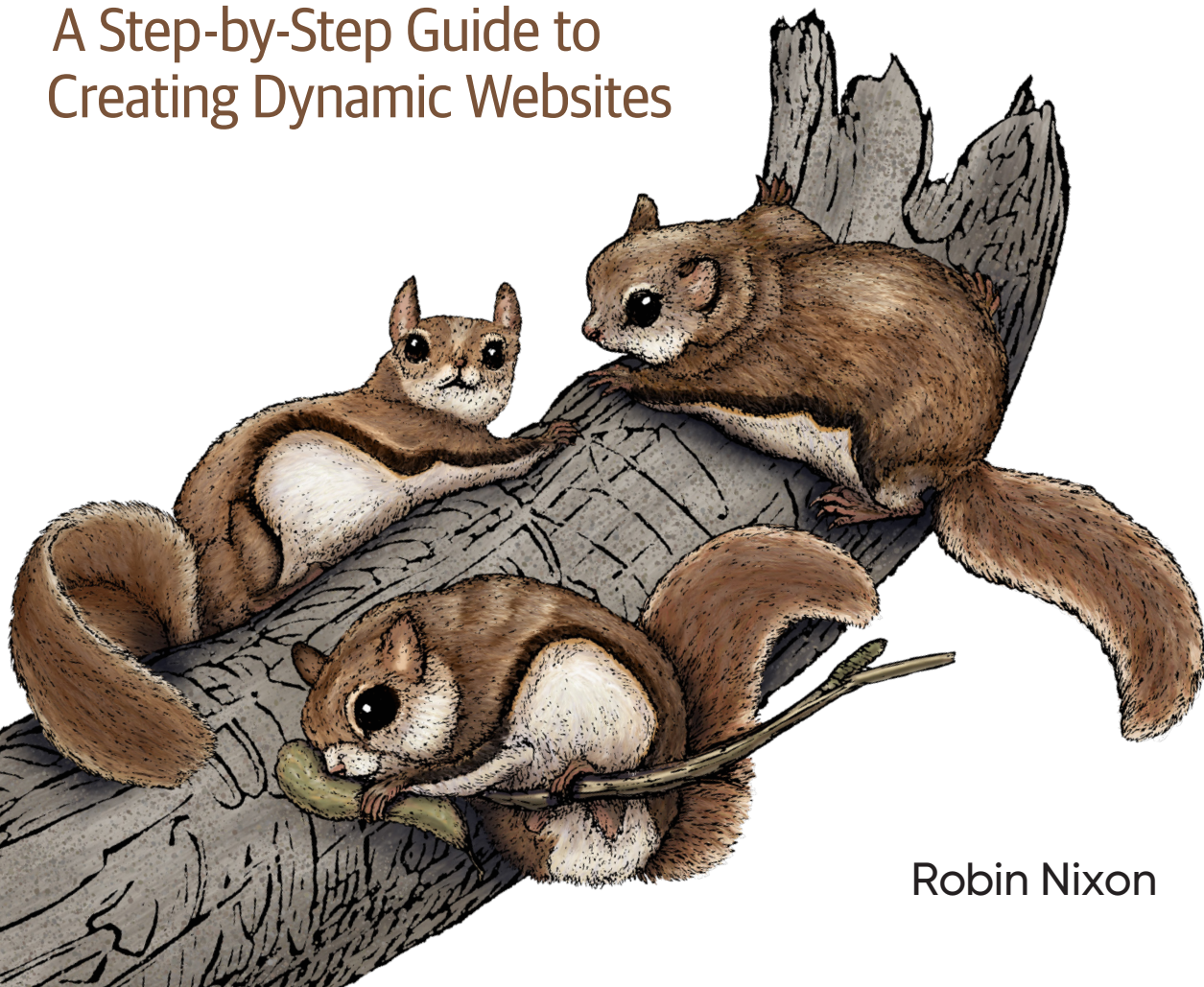


O'REILLY®

7th Edition

Learning PHP, MySQL & JavaScript

A Step-by-Step Guide to
Creating Dynamic Websites



Robin Nixon

"This is a great beginner's book that introduces several crucial web developer languages. It's a quick-paced, easy-to-read, information-packed book that will soon have you creating dynamically driven websites, including a basic social networking site."

Albert Wiersch
Developer of CSE HTML Validator

Learning PHP, MySQL & JavaScript

Build interactive, data-driven websites with the potent combination of open source technologies and web standards, even if you have only basic HTML knowledge. With the latest edition of this popular hands-on guide, you'll tackle dynamic web programming using the most recent versions of today's core technologies: PHP, MySQL, JavaScript, CSS, HTML5, jQuery, Node.js, and the powerful React library.

Web designers will learn how to use these technologies together while picking up valuable web programming practices along the way, including how to optimize websites for mobile devices. You'll put everything together to build a fully functional social networking site suitable for both desktop and mobile browsers.

- Explore MySQL from database structure to complex queries
- Use the MySQL PDO extension, PHP's improved MySQL interface
- Create dynamic PHP web pages that tailor themselves to the user
- Manage cookies and sessions and maintain a high level of security
- Use Ajax calls for background browser-server communication
- Style your web pages by acquiring CSS skills
- Reformat your websites into mobile web apps
- Learn to use enhanced CSS features, such as CSS Grid and Flexbox

Robin Nixon has worked with and written about computers since the early 1980s. One of the websites he developed presented the world's first radio station licensed by the music copyright holders. He also developed the first known pop-up windows to enable people to surf while listening. Earlier in his career, Robin worked full time for one of Britain's main IT magazine publishers, where he held several roles including editorial, promotions, and cover disc editing.

WEB DEVELOPMENT

US \$59.99 CAN \$74.99

ISBN: 978-1-098-15235-2



5 5 9 9 9

O'REILLY®

SEVENTH EDITION

Learning PHP, MySQL & JavaScript

*A Step-by-Step Guide
to Creating Dynamic Websites*

Robin Nixon

O'REILLY®

Learning PHP, MySQL & JavaScript

by Robin Nixon

Copyright © 2025 Robin Nixon. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editors: Amanda Quinn & Louise Corrigan

Development Editors: Rita Fernando & Michele Cronin

Production Editor: Elizabeth Faerm

Copyeditor: Piper Editorial Consulting, LLC

Proofreader: Kim Cofer

Indexer: Sue Klefsstad

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

January 2025: Seventh Edition

Revision History for the Seventh Edition

2025-01-10: First Release

See oreilly.com/catalog/errata.csp?isbn=0636920912620 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning PHP, MySQL & JavaScript*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15235-2

[LSI]

This book is dedicated to all the readers who have contributed ideas, suggestions, errata, and otherwise generally helped keep its contents fresh and relevant throughout seven editions since 2009. I salute your dedication to PHP and the associated technologies covered in this book—please always keep your feedback coming.

Table of Contents

Preface.....	xix
1. Introduction to Dynamic Web Content.....	1
HTTP and HTML: Berners-Lee's Basics	2
The Request/Response Procedure	3
The Benefits of PHP, MySQL, JavaScript, CSS, and HTML	5
MariaDB: The MySQL Clone	6
Using PHP	7
Using MySQL	8
Using JavaScript	9
Using CSS	10
And Then There's HTML5	11
The Apache Web Server	12
Node.js: An Alternative to Apache	13
About Open Source	14
Bringing It All Together	14
Questions	16
2. Setting Up a Development Server.....	17
What Is a WAMP, MAMP, or LAMP?	18
Installing AMPPS on Windows	18
Testing the Installation	23
Accessing the Document Root (Windows)	25
Alternative WAMPs	26
Installing AMPPS on macOS	27
Installing a LAMP on Linux	29
Working Remotely	29
Logging In	29

Transferring Files	30
Using a Code Editor	31
Questions	32
3. Introduction to PHP.....	33
Incorporating PHP Within HTML	33
This Book's Examples	35
The Structure of PHP	35
Using Comments	35
Basic Syntax	36
Variables	37
Operators	42
Variable Assignment	46
Multiline Strings	49
Variable Typing	52
Constants	53
Predefined Constants	53
The Difference Between the echo and print Commands	54
Functions	55
Variable Scope	56
Questions	62
4. Expressions and Control Flow in PHP.....	63
Expressions	63
TRUE or FALSE?	64
Literals and Variables	65
Operators	66
Operator Precedence	67
Associativity	69
Relational Operators	70
Conditionals	74
The if Statement	75
The else Statement	76
The elseif Statement	78
The switch Statement	79
The ? (or Ternary) Operator	82
Looping	83
while Loops	84
do...while Loops	86
for Loops	86
Breaking Out of a Loop	88

The continue Statement	89
Implicit and Explicit Casting	90
PHP Modularization	91
Questions	92
5. PHP Functions and Objects.....	93
PHP Functions	94
Defining a Function	96
Returning a Value	96
Returning an Array	98
Returning Global Variables	98
Recap of Variable Scope	99
Including and Requiring Files	99
The include Statement	100
Using include_once	100
Using require and require_once	101
PHP Version Compatibility	101
PHP Objects	102
Terminology	102
Declaring a Class	104
Creating an Object	105
Accessing Objects	105
Cloning Objects	106
Constructors	108
Destructors	108
Writing Methods	109
Declaring Properties	110
Static Methods	110
Declaring Constants	111
Property and Method Scope	112
Static Properties	113
Inheritance	114
Questions	118
6. PHP Arrays.....	119
Basic Access	119
Numerically Indexed Arrays	119
Associative Arrays	121
Assignment Using the array Keyword	122
The foreach...as Loop	123
Multidimensional Arrays	125

Using Array Functions	129
is_array	129
count	129
sort	129
shuffle	130
explode	130
compact	131
reset	132
end	133
Questions	133
7. Practical PHP.....	135
Using printf	135
Precision Setting	137
String Padding	138
Using sprintf	139
Date and Time Functions	140
Date Constants	142
Using checkdate	143
File Handling	143
Checking Whether a File Exists	143
Creating a File	144
Reading from Files	145
Copying Files	146
Moving a File	147
Deleting a File	147
Updating Files	148
Locking Files for Multiple Accesses	149
Reading an Entire File	151
Uploading Files	152
System Calls	157
Questions	159
8. Introduction to MySQL.....	161
MySQL Basics	161
Key Database Terms	162
Accessing MySQL via the Command Line	162
Starting the Command-Line Interface	163
Using the Command-Line Interface	167
MySQL Commands	168
Data Types	173

Indexes	184
Creating an Index	184
Querying a MySQL Database	190
Joining Tables	201
Using Logical Operators	204
MySQL Functions	205
Accessing MySQL via phpMyAdmin	205
Questions	207
9. Mastering MySQL.....	209
Database Design	209
Primary Keys: The Keys to Relational Databases	210
Normalization	211
First Normal Form	212
Second Normal Form	214
Third Normal Form	217
When Not to Use Normalization	219
Relationships	219
One-to-One	220
One-to-Many	220
Many-to-Many	221
Databases and Privacy	223
Transactions	223
Transaction Storage Engines	223
Using START TRANSACTION	225
Using COMMIT	225
Using ROLLBACK	225
Using EXPLAIN	226
Backing Up and Restoring	228
Using mysqldump	228
Creating a Backup File	230
Restoring from a Backup File	232
Dumping Data in CSV Format	233
Planning Your Backups	233
Questions	234
10. Accessing MySQL Using PHP.....	235
Querying a MySQL Database with PHP	235
The Process	235
Creating a Login File	236
Connecting to a MySQL Database	238

Building and Executing a Query	239
Fetching a Result	239
Fetching a Row While Specifying the Style	241
Closing a Connection	242
A Practical Example	243
The \$_POST Array	246
Deleting a Record	247
Displaying the Form	247
Querying the Database	248
Running the Program	249
Practical MySQL	250
Creating a Table	250
Describing a Table	251
Dropping a Table	252
Adding Data	253
Retrieving Data	254
Updating Data	255
Deleting Data	255
Using AUTO_INCREMENT	256
Performing Additional Queries	257
Preventing Hacking Attempts	259
Steps You Can Take	260
Using Placeholders	261
Preventing JavaScript Injection into HTML	264
Questions	266
11. Form Handling.....	267
Building Forms	267
Retrieving Submitted Data	269
Default Values	270
Input Types	271
Sanitizing Input	283
An Example Program	284
Questions	287
12. Cookies, Sessions, and Authentication.....	289
Using Cookies in PHP	289
Setting a Cookie	291
Accessing a Cookie	292
Destroying a Cookie	292
HTTP Authentication	292

Storing Usernames and Passwords	296
An Example Program	298
Using Sessions	301
Starting a Session	302
Ending a Session	304
Setting a Timeout	305
Session Security	306
Questions	309
13. Exploring JavaScript.....	311
Outputting the Results	312
Using console.log	312
Using alert	312
Writing into Elements	312
Using document.write	312
JavaScript and HTML Text	313
Using Scripts Within a Document Head	315
Including JavaScript Files	315
Debugging JavaScript Errors	316
Using Comments	316
Semicolons	317
Variables	317
String Variables	318
Numeric Variables	318
Arrays	318
Operators	319
Arithmetic Operators	319
Assignment Operators	320
Comparison Operators	320
Logical Operators	321
Incrementing, Decrementing, and Shorthand Assignment	321
String Concatenation	321
Escape Characters	322
Variable Typing	322
Functions	324
Global Variables	324
Local Variables	325
Using let	326
Using const	327
The Document Object Model	328
Another Use for the \$ Symbol	330

Using the DOM	330
Questions	332
14. Expressions and Control Flow in JavaScript.....	333
Expressions	333
Literals and Variables	334
Operators	335
Operator Precedence	336
Associativity	337
Relational Operators	337
Using onerror	342
Using try...catch	343
Conditionals	344
The if Statement	344
The else Statement	344
The switch Statement	345
The ? Operator	347
Looping	347
while Loops	348
do...while Loops	348
for Loops	349
Breaking Out of a Loop	350
The continue Statement	351
Explicit Casting	351
Questions	352
15. JavaScript Functions, Objects, and Arrays.....	353
JavaScript Functions	353
Defining a Function	353
Returning a Value	356
Returning an Array	358
JavaScript Objects	359
Declaring a Class	359
Creating an Instance	360
Accessing Objects	360
Static Methods and Properties	361
The Legacy Objects Simulated with Functions	361
JavaScript Arrays	363
Arrays	363
Associative Arrays	364
Multidimensional Arrays	365

Using Array Methods	366
Anonymous Functions	372
Arrow Functions	373
Questions	373
16. JavaScript and PHP Validation and Error Handling.	375
Validating User Input with JavaScript	375
The validate.html Document (Part 1)	376
The validate.html Document (Part 2)	379
Regular Expressions	383
Matching Through Metacharacters	383
Wildcard Matching	384
Grouping Through Parentheses	385
Character Classes	386
Indicating a Range	386
Negation	386
Some More Complicated Examples	387
Summary of Metacharacters	389
General Modifiers	391
Using Regular Expressions in JavaScript	391
Using Regular Expressions in PHP	392
Redisplaying a Form After PHP Validation	393
Questions	400
17. Using Asynchronous Communication.	403
The Fetch API	403
Your First Asynchronous Program	404
The Server Half of the Asynchronous Process	406
Cross-Origin Resource Sharing (CORS)	407
Using GET Instead of POST	409
Sending JSON Requests	410
Using XMLHttpRequest	412
Using Frameworks for Asynchronous Communication	413
Questions	413
18. Advanced CSS.	415
Attribute Selectors	416
The ^= Operator	417
The \$= Operator	417
The *= Operator	417
The box-sizing Property	417

CSS Backgrounds	418
The background-clip Property	419
The background-origin Property	421
The background-size Property	421
Using the auto Value	422
Multiple Backgrounds	422
CSS Borders	425
The border-color Property	425
The border-radius Property	426
Box Shadows	428
Element Overflow	429
Multicolumn Layout	429
Colors and Opacity	431
HSL Colors	431
HSLA Colors	432
RGB Colors	432
RGBA Colors	433
The opacity Property	433
Text Effects	433
The text-shadow Property	433
The text-overflow Property	434
The word-wrap Property	435
Web Fonts	435
Google Web Fonts	436
Transformations	438
Transitions	440
Properties to Transition	440
Transition Duration	441
Transition Delay	441
Transition Timing	441
Shorthand Syntax	442
Flexbox	444
Flex Items	445
Flow Direction	445
Justifying Content	447
Aligning Items	449
Aligning Content	450
Resizing Items	451
Flex Wrap	451
Order	453
Item Gaps	453

CSS Grid	454
Grid Container	454
Grid Columns and Rows	455
Grid Flow	456
Placing Grid Items	457
Grid Gaps	458
Alignment	459
Questions	460
19. Accessing CSS from JavaScript.	463
Revisiting the getElementById Function	463
The byId Function	464
The style Function	464
The by Function	465
Including the Functions	465
Accessing CSS Properties from JavaScript	466
Some Common Properties	466
Other Properties	468
Inline JavaScript	469
The this Keyword	470
Attaching Events to Objects in a Script	470
Attaching to Other Events	471
Adding New Elements	472
Removing Elements	474
Alternatives to Adding and Removing Elements	474
Time-based Events	475
Using setTimeout	475
Using setInterval	476
Using Time-Based Events for Animation	478
Questions	480
20. Introduction to React.	483
What Is the Point of React Anyway?	484
Accessing the React Files	485
Including babel.js	486
Our First React Project	487
Using a Class Instead of a Function	488
Pure and Impure Code: A Golden Rule	489
Using Both a Class and a Function	490
Props and Components	490
The Differences Between Using a Class and a Function	492

React State and Life Cycle	492
Events in React	495
Inline JSX Conditional Statements	497
Using Lists and Keys	498
Unique Keys	499
Handling Forms	501
Using Text Input	501
Using textarea	503
Using select	504
React Native	506
Questions	506
21. Introduction to Node.js.....	509
Installing Node.js on Windows	510
Installing Node.js on macOS	516
Installing Node.js on Linux	519
Getting Started with Node.js	520
Building a Functioning Web Server	522
Working with Modules	525
Built-in Modules	526
Installing Modules with npm	526
Accessing MySQL	527
Further Information	529
Questions	530
22. Bringing It All Together.....	531
Designing a Social Networking App	532
Online Repository	532
functions.php	532
header.php	535
setup.php	537
index.php	539
signup.php	540
Checking for Username Availability	541
Logging In	541
checkuser.php	543
login.php	544
profile.php	547
Adding the “About Me” Text	547
Adding a Profile Image	547
Processing the Image	548

Displaying the Current Profile	548
members.php	551
Viewing a User's Profile	552
Adding and Dropping Friends	552
Listing All Members	552
friends.php	555
messages.php	558
logout.php	562
styles.css	562
javascript.js	566
Questions	566
A. Solutions to the Chapter Questions.	567
Index.	587

Preface

The combination of PHP and MySQL is the most convenient approach to dynamic, database-driven web design, holding its own in the face of challenges from some other integrated frameworks that are harder to learn. Due to its open source roots, it is free to implement and is therefore an extremely popular option for web development.

Any would-be developer on a Unix/Linux or even a Windows platform will need to master these technologies. And, combined with the partner technologies of JavaScript, React, CSS, and HTML5, you will be able to create websites of the caliber of industry standards like Facebook, Reddit, TikTok, and Gmail.

Audience

This book is for people who wish to learn how to create effective and dynamic websites. This may include webmasters or graphic designers who have already mastered creating static websites, or a CMS such as WordPress but wish to take their skills to the next level, as well as high school and college students, recent graduates, and self-taught individuals.

In fact, anyone ready to learn the fundamentals behind responsive web design will obtain a thorough grounding in the core technologies of PHP, MySQL, JavaScript, CSS, and HTML5, and you'll learn the basics of the React library and how to use Node.js to support backend development using JavaScript.

Assumptions This Book Makes

This book assumes that you have a basic understanding of HTML and can at least put together a simple, static website but does not assume that you have any prior knowledge of PHP, MySQL, JavaScript, and CSS—although if you do, your progress through the book will be even quicker.

Organization of This Book

The chapters in this book are written in a specific order, first introducing all of the core technologies it covers and then walking you through their installation on a web development server so that you will be ready to work through the examples.

In the first section, you will gain a grounding in the PHP programming language, covering the basics of syntax, arrays, functions, and object-oriented programming.

Then, with PHP under your belt, you will move on to an introduction to the MySQL database system, where you will learn everything from how MySQL databases are structured to how to generate complex queries.

After that, you will learn how you can combine PHP and MySQL to start creating your own dynamic web pages by integrating forms and other HTML features. You will then get down to the nitty-gritty practical aspects of PHP and MySQL development by learning a variety of useful functions and how to manage cookies and sessions, as well as how to maintain a high level of security.

In the next few chapters, you will gain a thorough grounding in JavaScript, from simple functions and event handling to accessing the Document Object Model, in-browser validation, and error handling. You'll also get a comprehensive primer on using the popular React library for JavaScript.

With an understanding of all three of these core technologies, you will then learn how to make behind-the-scenes Ajax calls and turn your websites into highly dynamic environments.

Next, you'll learn all about using CSS to dynamically style and lay out your web pages, before discovering how the React libraries can make your development job a great deal easier, and how you can use Node.js instead of PHP and the Apache web server to write your backend code in JavaScript. Finally you'll put together everything you've learned in a complete set of programs that together constitute a fully functional social networking website.

Along the way, you'll find plenty of advice on good programming practices and tips that can help you find and solve hard-to-detect programming errors. There are also plenty of links to websites containing further details on the topics covered.

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text

Indicates menu titles, options, and buttons.

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities. Also used for database, table, and column names.

Constant width

Indicates commands and command-line options, variables and other code elements, HTML tags, the contents of files, as well as user input.

Constant width bold

Shows program output and is used to highlight sections of code that are discussed in the text.

Constant width italic

Shows text that should be replaced with user-supplied values.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

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

- [Supplemental Chapter 1, “Introduction to CSS”](#)
- [Supplemental Chapter 2, “Introduction to jQuery”](#)
- [Supplemental Chapter 3, “Introduction to jQuery Mobile”](#)
- [Supplemental Chapter 4, “Introduction to HTML5”](#)

- Supplemental Chapter 5, “The HTML5 Canvas”
- Supplemental Chapter 6, “HTML5 Audio and Video”
- Supplemental Chapter 7, “Other HTML5 Features”
- Supplemental Chapter 8, “What’s New in PHP 8 and MySQL 8”

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Learning PHP, MySQL & JavaScript*, 7th Edition by Robin Nixon (O’Reilly). Copyright 2025 Robin Nixon, 978-1-098-15235-2.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O’Reilly Online Learning



For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *<https://oreil.ly/learning-php-mysql-js-7e>*.

For news and information about our books and courses, visit *<https://oreilly.com>*.

Find us on LinkedIn: *<https://linkedin.com/company/oreilly-media>*.

Watch us on YouTube: *<https://youtube.com/oreillymedia>*.

Acknowledgments

I would like to thank Senior Content Acquisitions Editor Amanda Quinn, Content Development Editors Rita Fernando and Michele Cronin, and everyone who worked so hard on this book, including Michal Špaček and David Mackey for their comprehensive technical reviews, Michal Špaček for his excellent help during production, Elizabeth Faerm for overseeing production, Beth Richards for copy editing, Kim Cofer for proof-reading, Sue Klefstad for creating the index, Karen Montgomery for the original sugar glider front cover design, Susan Brown for the latest book cover, my original editor, Andy Oram, for overseeing the first five editions, and everyone else too numerous to name who submitted errata and offered suggestions for this new edition.

Introduction to Dynamic Web Content

The World Wide Web is a constantly evolving network that has already traveled far beyond its conception in the early 1990s, when it was created to solve a specific problem. State-of-the-art experiments at CERN (the European Laboratory for Particle Physics, now best known as the operator of the Large Hadron Collider) were producing incredible amounts of data—so much that the data was proving unwieldy to distribute to the participating scientists, who were spread out across the world.

At this time, the internet was already in place, connecting several hundred thousand computers, so Tim Berners-Lee (a CERN fellow) devised a method of navigating between them using a hyperlinking framework, which came to be known as Hypertext Transfer Protocol, or HTTP. He also created a markup language called Hypertext Markup Language, or HTML. To bring these together, he wrote the first web browser and web server.



The Advent of Web 1.0

Web 1.0 was given its name only when the term Web 2.0 was coined. During the 1.0 era, most users were content consumers, and although there were some personal web pages, there were no social networks. Guestbooks were used instead of comment sections. Some sites had already used databases but server resources and bandwidth were very limited. Navigation and layout in Web 1.0 was managed with simple buttons and graphics, while interaction was very limited.

Today we take these simple tools for granted, but back then, the concept was revolutionary. The most connectivity experienced by at-home modem users at that time was dialing up and connecting to a bulletin board where you could communicate and swap data only with other users of that service. Consequently, you needed to

be a member of many bulletin board systems in order to effectively communicate electronically with your colleagues and friends.

But Berners-Lee changed all that in one fell swoop, and by the mid-1990s, three major graphical web browsers were competing for the attention of five million users. It soon became obvious, though, that something was missing. Yes, pages of text and graphics with hyperlinks to take you to other pages was a brilliant concept, but the results didn't reflect the instantaneous potential of computers and the internet to meet the particular needs of each user with dynamically changing content. Using the web was a very dry, plain experience, even if we did have scrolling text and animated GIFs!

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



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

HTTP and HTML: Berners-Lee's Basics

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

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

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

The Request/Response Procedure

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

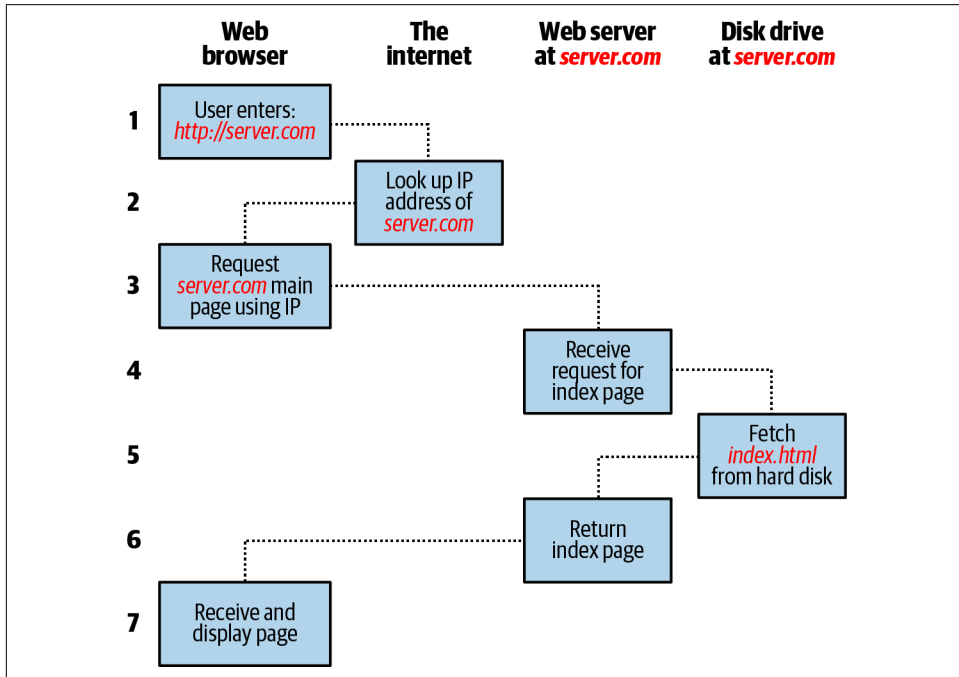


Figure 1-1. The basic client/server request/response sequence

The steps in the request and response sequence are:

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

For an average web page, this process also takes place once for each object within the page such as a graphic, an embedded video, or a CSS stylesheet.

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

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

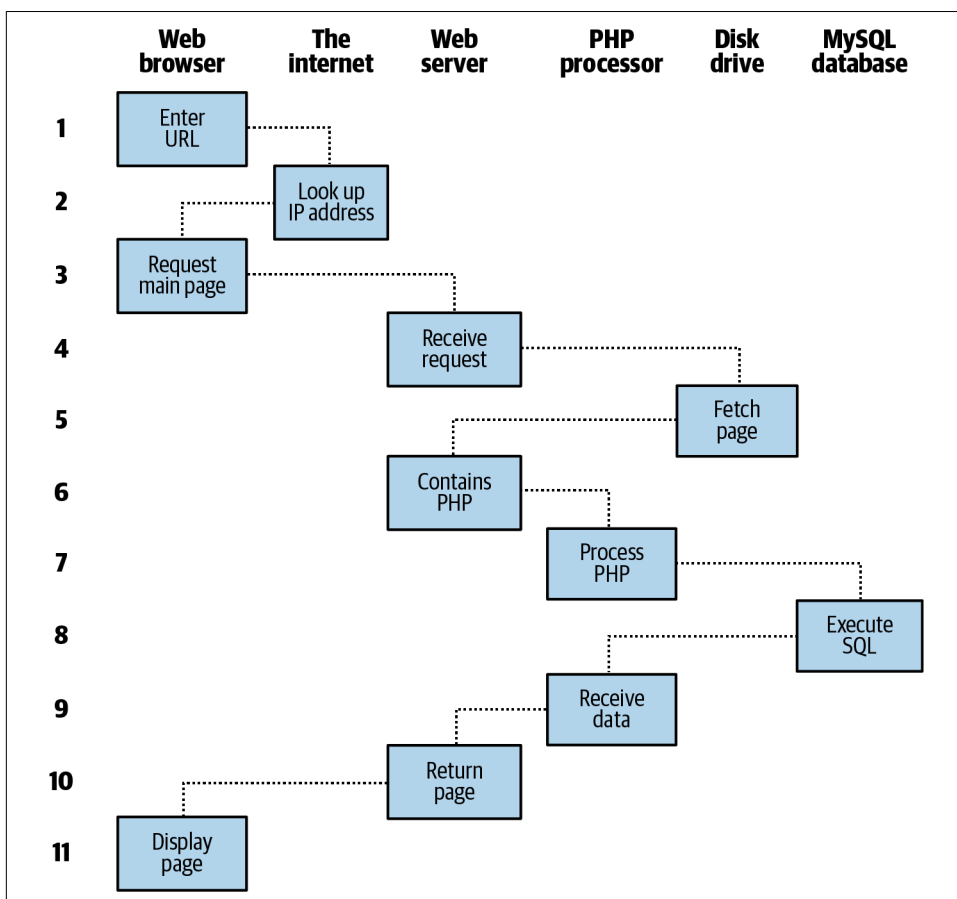


Figure 1-2. A dynamic client/server request/response sequence

The steps in the dynamic sequence are:

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

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

The HTML pages returned to the browser in each example may contain JavaScript, which will be interpreted locally by the client, and which could initiate another request.

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

At the start of this chapter, I introduced the world of Web 1.0, but it wasn't long before the rush was on to create Web 1.1, with the development of such browser enhancements as Java, JavaScript, Flash, and ActiveX. On the server side, progress was being made on the Common Gateway Interface (CGI) using scripting languages such as Perl (an alternative to the PHP language) and *server-side scripting*—inserting the contents of one file (or the output of running a local program) into another one dynamically.

Once the dust had settled, three main technologies stood head and shoulders above the others. Although Perl was still a popular scripting language with a strong following, PHP's simplicity and built-in links to the MySQL database program had earned it more than double the number of users. And JavaScript, which had become an essential part of the equation for dynamically manipulating HTML, now took on the

even more muscular task of handling the client side of asynchronous communication (exchanging data between a client and server after a web page has loaded). Using asynchronous communication, web pages perform data handling and send requests to web servers in the background—without the web user being aware that this is going on.

No doubt the symbiotic nature and the open source licenses of PHP and MySQL helped propel them both forward, but what attracted developers to them in the first place? The simple answer is the ease with which you can use them to quickly create dynamic elements on websites. MySQL is a fast and powerful yet easy-to-use database system that offers just about anything a website would need to find and serve up data to browsers.

And when you bring JavaScript and CSS into the mix, you have a recipe for building highly dynamic and interactive websites—especially as there is now a wide range of sophisticated frameworks of JavaScript functions you can call on to speed up web development. These include the well-known jQuery, which until recently was one of the most common ways programmers accessed asynchronous communication features.

The more recent React JavaScript library has also been growing quickly in popularity, and is now one of the most widely downloaded and implemented frameworks, so much so that at the time of writing the Indeed job site lists many more positions for React developers than for jQuery.

React provides state-of-the-art functionality for building complex UI interactions that communicate with the server in real time with JavaScript-driven pages. It lets you create components that are the building blocks of the React application.

A **React component** can be anything in your web application. It can be as simple as a Button, Text, Label, or Grid, or even as complex as a Login widget or a popup modal with control buttons. React also supports server rendering of its components using tools like **Next.js**. You can even use React in your existing apps (it was designed with this in mind). You can change a small part of your existing application by using React, and if that change works, then you can start converting your whole application over to React.js. However, other frameworks such as Vue.js may be more suitable for this sort of iterative implementation.

MariaDB: The MySQL Clone

After Oracle (the database management corporation) purchased Sun Microsystems (the owners of MySQL), the community became wary that MySQL might not remain fully open source, so MariaDB was forked from it to keep it free under the GNU GPL, the software license that guarantees users the freedom to run, study, share, and modify the software. Development of MariaDB is led by some of the original

developers of MySQL, and it retains exceedingly close compatibility with MySQL. Therefore, you may well encounter MariaDB on some servers in place of MySQL—but but not to worry, everything in this book works equally well on both MySQL and MariaDB. For all intents and purposes, you can swap one with the other and notice no difference.

Fortunately, many of the initial fears appear to have been allayed as MySQL remains open source, with Oracle simply charging for support and for editions that provide additional features such as geo-replication and automatic scaling. However, unlike MariaDB, MySQL is no longer community driven, so knowing that MariaDB will always be there if needed will reassure many developers and likely ensure that MySQL itself will remain open source.

Using PHP

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

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

Here's the latest news.

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

The final output of the two parts looks like this:

Today is Wednesday. Here's the latest news.

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

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

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



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

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

Using MySQL

Of course, there's not much point in being able to change HTML output dynamically unless you also have a means to track the information users provide to your website as they use it. In the early days of the web, many sites used “flat” text files to store data such as usernames and passwords. But this approach could cause problems if the file wasn't correctly locked against corruption from multiple simultaneous accesses. Also, a flat file can get only so big before it becomes unwieldy to manage—not to mention the difficulty of trying to merge files and perform complex searches in a reasonable time.

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

The highest level of MySQL structure is a database, within which you can have one or more tables that contain your data. This is similar to let's say an Excel spreadsheet file that consists of multiple sheets: the spreadsheet file can be viewed as a database and the individual sheets as tables.

Let's suppose you are working on a table called *users*, within which you have created columns for *surname*, *firstname*, and *email*, and you now wish to add another user. One command you might use to do this is:

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

You will previously have issued other commands to create the database and table and to set up all the correct fields, but the SQL INSERT command here shows how simple it can be to add new data to a database.

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

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


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

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

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

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

Using JavaScript

JavaScript was created to enable scripting access to all the elements of an HTML document. In other words, it provides a means for dynamic user interaction such as checking email address validity in input forms and displaying prompts such as "Did you really mean that?" (although it cannot be relied upon for security, which should always be performed on the web server).

Combined with CSS (see ["Using CSS" on page 10](#)), JavaScript is the power behind dynamic web pages that change in front of your eyes rather than when a new page is returned by the server.

However, JavaScript used to be tricky to use, due to the way the language was initially designed and to some major differences in how different browsers have chosen to implement it. This came about when some manufacturers tried to put additional functionality into their browsers at the expense of compatibility with their rivals.

Thankfully, the language evolves, and the browser developers have mostly come to their senses, realizing the need for full compatibility with one another, so it is less necessary these days to have to optimize your code for different browsers.

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

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

This code snippet tells the web browser to interpret everything within the <script> tags as JavaScript, which the browser does by writing the text Today is to the current

document, along with the date, using the JavaScript function `Date`. The result will look something like this:

Today is Wed Jan 01 2025 01:23:45



Walking Before Running

The `document.write` function is deliberately being used here in the way it was originally intended, for the sake of simplicity in very small code snippets. However, there are better ways to write into web pages and for issuing feedback while debugging, all of which will be revealed at the right times in this book, as well as explanations for when and why the other options will work better for you.

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

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

Using CSS

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

By the way, the numbering standard for CSS releases (such as CSS2 or CSS3) has now been dropped, so Cascading Style Sheets are now referred to as simply CSS, but various submodules have their own numbering such as CSS Selectors Level 4 and CSS Images Level 3.

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

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

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

The many different ways you can lay out CSS rules are discussed in [Supplemental Chapter 1, “Introduction to CSS”](#), and you can also include them directly within tags or save a set of rules to an external file to be loaded in separately. This flexibility not only lets you style your HTML precisely but can also (for example) provide built-in hover functionality to animate objects as the mouse passes over them. You will also learn how to access all of an element’s CSS properties from JavaScript as well as HTML.

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

And Then There’s HTML5

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

To clear all this up and take the internet beyond Web 2.0 and into its next iteration, a new standard for HTML was created to address all these shortcomings: *HTML5*. Its development began as long ago as 2004, when the first draft was drawn up by the Mozilla Foundation and Opera Software, developers of two popular web browsers. Today, the HTML5 standard is maintained by WHATWG (Web Hypertext Application Technology Working Group) and is officially called HTML Living Standard.

It’s a never-ending cycle of development, though, and more functionality is sure to be built into it over time. Some of the best features in HTML5 for handling and displaying media include the `<audio>`, `<video>`, and `<canvas>` elements, which add sound, video, and advanced graphics. Everything you need to know about these and all other aspects of HTML5 is covered in detail starting in the PDF of [Supplemental Chapter 4, “Introduction to HTML5”](#), available in the book’s [GitHub repository](#).



One of the little things I like about the HTML5 specification is that XHTML syntax is no longer required for self-closing elements. In the past, you could display a line break using the `<br>` element. Then, to ensure future compatibility with XHTML (the planned replacement for HTML that never happened), this was changed to `<br />`, in which a closing `/` character was added (since all elements were expected to include a closing tag featuring this character). But now things have gone full circle, and you can use either version of these types of elements. In this book I have reverted to the former style of `<br>`, `<hr>`, and so on, as this is also what the HTML standard now recommends. Do note, however, that frameworks such as React use an extension to JavaScript called JSX, which *does* require the preceding `/` character, and where such examples occur in this book, the preceding `/` is used.

The Apache Web Server

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

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

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

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

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

Node.js: An Alternative to Apache

In 2009 developer Ryan Dahl was dissatisfied with Apache and its difficulties with handling large numbers of concurrent connections, and came up with a solution he called Node.js, which uses Google's V8 JavaScript engine to allow developers to use JavaScript for server-side scripting. Shortly after, a package manager was introduced for the Node.js environment called *npm*, which made it easier for programmers to publish and share source code of Node.js packages, simplifying installation, updating, and uninstallation of packages.

As of 2024 Node.js has reached version 22.6.0 and has become a fully mainstream alternative to the Apache web server. This book's new edition would be remiss to not detail its benefits and provide enough information to get you up and running with it, if you choose. You might make that choice, for the three reasons discussed next.

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

It allows developers to use JavaScript on both the frontend and backend, making it a full-stack development environment. This eliminates the need to switch between different programming languages, enabling better code reusability and streamlining the development process. Yes, that means you won't have to keep up-to-date with PHP if you make the switch, and indeed Node.js will not be able to run your PHP scripts. However, a rather complex app can still use both Node.js and Apache with PHP each for different parts or tasks.

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

As you will learn, there are many other solid reasons for using Node.js, but just these few are already highly persuasive. PHP remains a very important language prevalent across the internet, is actively developed, has active communities and is often used together with other languages and environments such as Node.js.

About Open Source

The technologies in this book are open source: anyone is allowed to read and change the code. Whether this status is the reason these technologies are so popular has often been debated, but PHP, MySQL, and Apache *are* the three most commonly used tools in their categories. What can be said definitively, though, is that their being open source means that they have been developed in the community by teams of programmers writing the features they themselves want and need, with the original code available for all to see and change. Bugs can be found quickly, and security breaches can be prevented before they happen.

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

Bringing It All Together

The real beauty of PHP, MySQL, JavaScript, CSS, and HTML is the wonderful way they all work together to produce dynamic web content: PHP handles all the main work on the web server, MySQL manages all the data, and the combination of CSS and JavaScript looks after web page presentation. JavaScript can also talk with your PHP code on the web server whenever it needs to update something (either on the server or on the web page). And with powerful HTML features like the canvas, audio and video, and geolocation, you can make your web pages highly dynamic, interactive, and multimedia-packed.

Without using program code, let's summarize the contents of this chapter by looking at the process of combining some of these technologies into an everyday asynchronous communication feature that many websites use: checking whether a desired username already exists on the site when a user is signing up for a new account. A good example of this can be seen with Gmail (see [Figure 1-3](#)).

The screenshot shows the Google sign-in interface. At the top is the Google logo. Below it, the heading "How you'll sign in" is centered. Underneath, the text "Create a Gmail address for signing in to your Google Account" is displayed. A form for creating a Gmail address is shown, with a red border around the input fields. The "Username" field contains "arthurjohnson" and the email domain "@gmail.com" is shown to the right. Below the form, a red error message with an exclamation mark icon states: "That username is taken. Try another." Below this, the text "Available: aj8245778" is shown. A blue "Next" button is located at the bottom right of the form area.

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

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

1. The server outputs the HTML to create the web form, which asks for the necessary details, such as username, first name, last name, and email address.
2. At the same time, the server attaches some JavaScript to the HTML to monitor the username input box and check for two things: whether some text has been typed into it, and whether the input has been deselected because the user has clicked another input box or tabbed away.
3. Once the text has been entered and the field deselected, in the background the JavaScript code passes the username that was entered back to a software on the web server and awaits a response.
4. The web server looks up the username and replies to the JavaScript about whether that name has been taken.
5. The JavaScript then places an indication next to the username input box to show whether the name is available to the user—perhaps a green checkmark or a red cross graphic, along with some text.

6. If the username is not available and the user still submits the form, the JavaScript interrupts the submission and reemphasizes (perhaps with a larger graphic and/or an alert box) that the user needs to choose another username.
7. Optionally, an improved version of this process could look at the username requested by the user and suggest an alternative that is currently available.

All of this takes place quietly in the background and makes for a comfortable and seamless user experience. Without asynchronous communication, the entire form would have to be submitted to the server, which would then send back HTML, highlighting any mistakes. It would be a workable solution but nowhere near as tidy or pleasurable as on-the-fly form field processing.

Asynchronous communication can be used for a lot more than simple input verification and processing, though; we'll explore many additional things that you can do with it later in this book.

In this chapter, you have read an introduction to the core technologies of PHP, MySQL, JavaScript, CSS, and HTML (as well as Apache) and have learned how they work together. In [Chapter 2](#), we'll look at how you can install your own web development server on which to practice everything that you will be learning. Now, as in all this book's chapters, I recommend you see whether you can answer the following questions to check that you have absorbed its contents.

Questions

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

See [“Chapter 1 Answers” on page 567](#) in the [Appendix](#) for the answers to these questions.

Setting Up a Development Server

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

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

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

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

In this chapter, we cover just the server side of the web experience, as described in [Chapter 1](#). But to test the results of your work—particularly when we start using JavaScript, CSS, and HTML later in this book—you should ideally have an instance of every major web browser running on some system convenient to you. Sometimes, testing on two different browsers may be sufficient but whenever possible, the list of browsers should include at least Mozilla Firefox, Safari, and Google Chrome.

Even though there are multiple other browsers based on the Google Chromium browser there may still be minor differences in their implementation that make it worthwhile testing your code on all possible browsers before final release. You may

need all these once you have a product ready to release, just to ensure everything runs as expected on all browsers and platforms.

If you plan to ensure that your sites look good on mobile devices too, you should try to arrange access to a wide range of iOS and Android devices, and services like BrowserStack will help you with that. Browser developer tools also offer mobile device emulation to help you verify the site is responsive and viewable on those smaller screens.

What Is a WAMP, MAMP, or LAMP?

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

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

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

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



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

Installing AMPPS on Windows

There are several available WAMP servers, each offering slightly different configurations. Different editions of this book have recommended different WAMP products according to which seems to offer the best features and appears the most reliable at the time. Currently AMPPS looks like the best option (although you could choose other alternatives if you preferred and still be able to follow the examples in this

book). You can download AMPPS by clicking the download button on the [website's home page](#). (There are also Mac and Linux versions available; see “[Installing AMPPS on macOS](#)” on page 27 and “[Installing a LAMP on Linux](#)” on page 29.)

I recommend that you always download the latest stable release (as I write this, it's 4.4, the installer for which is about 46 MB in size). The various Windows, macOS, and Linux installers are listed on the download page.

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

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

C:\Program Files\Ampps

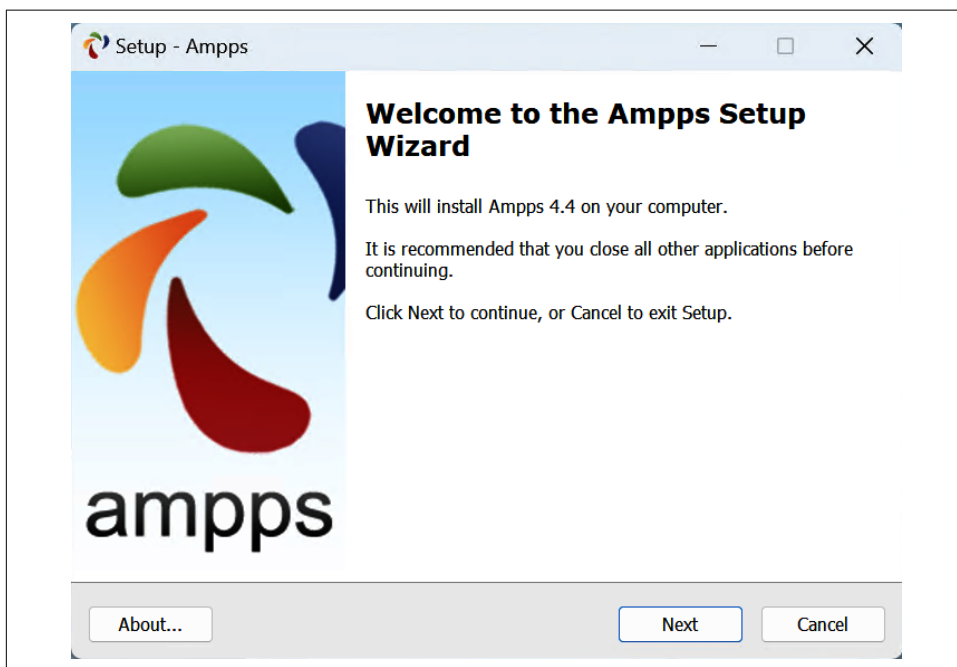


Figure 2-1. The opening window of the installer



During the lifetime of this edition, some of the screens and options shown in the following walk-through may change. If so, just use your common sense to proceed as similarly as possible to the sequence of actions described.

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

Once you have decided where to install AMPPS, click Next, decide where shortcuts should be saved (the default shown is usually just fine), and click Next again to choose which icons you wish to install, as shown in [Figure 2-2](#). On the screen that follows, click the Install button to start the process.

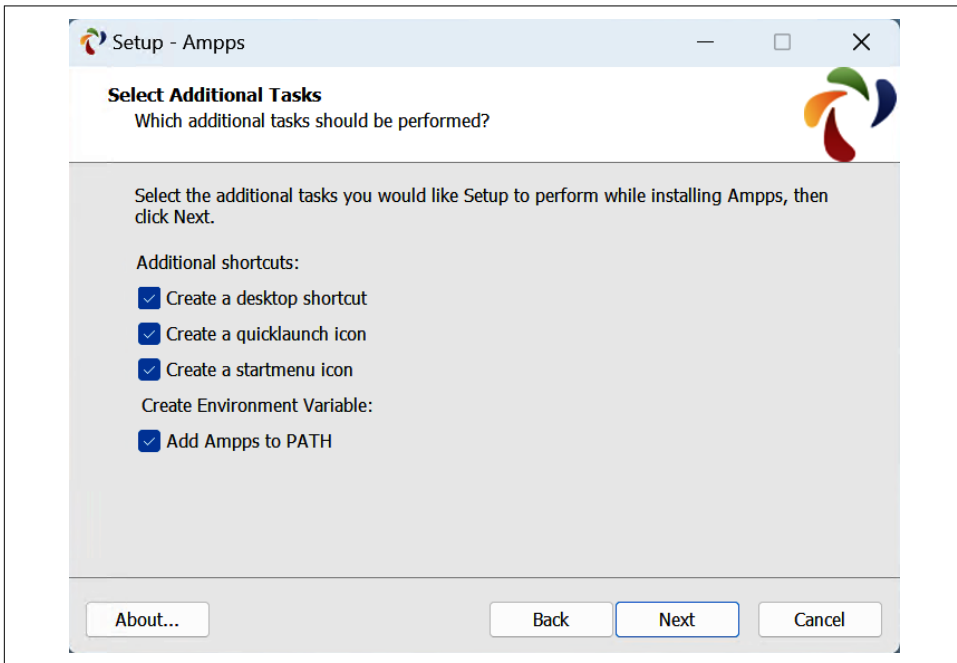


Figure 2-2. Choose which icons to install

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

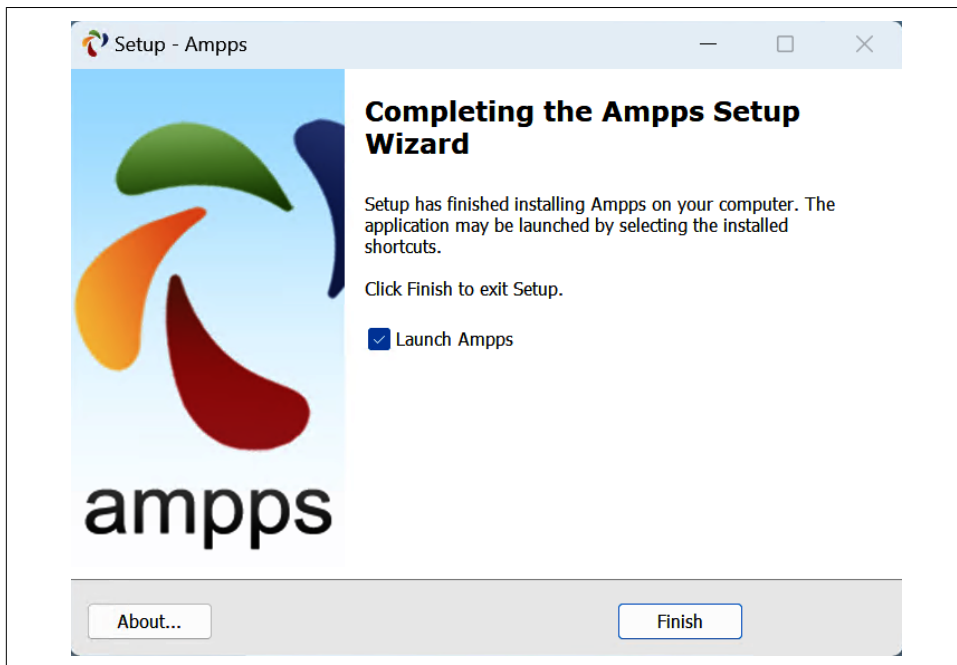


Figure 2-3. AMPPS is now installed

The final thing you must do is install Microsoft Visual C++ Redistributable, if you haven't already. A window will pop up to prompt you, as shown in [Figure 2-4](#). Click Install to start the installation and if you already have it you will be told whether you need to reinstall it, which you can skip.

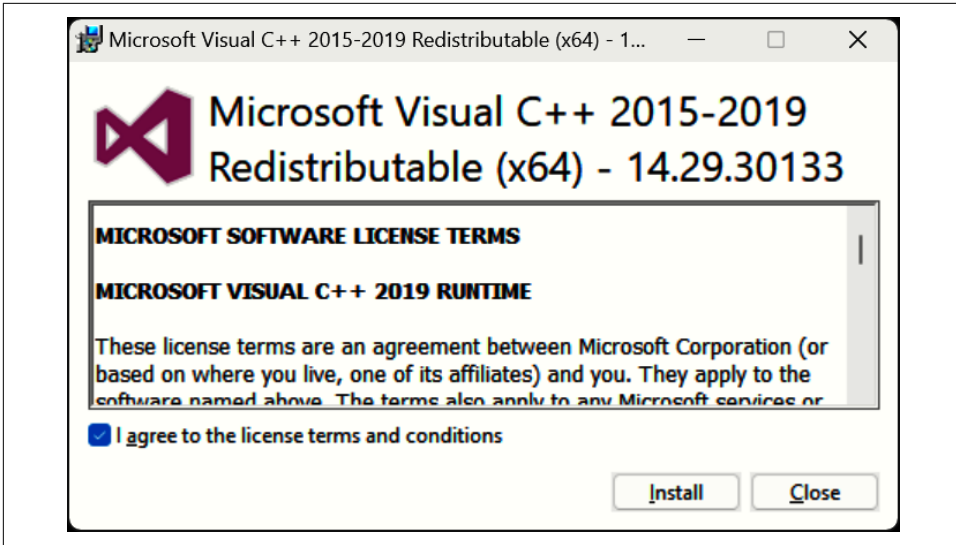


Figure 2-4. Install the Visual C++ Redistributable if you don't already have it

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

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

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

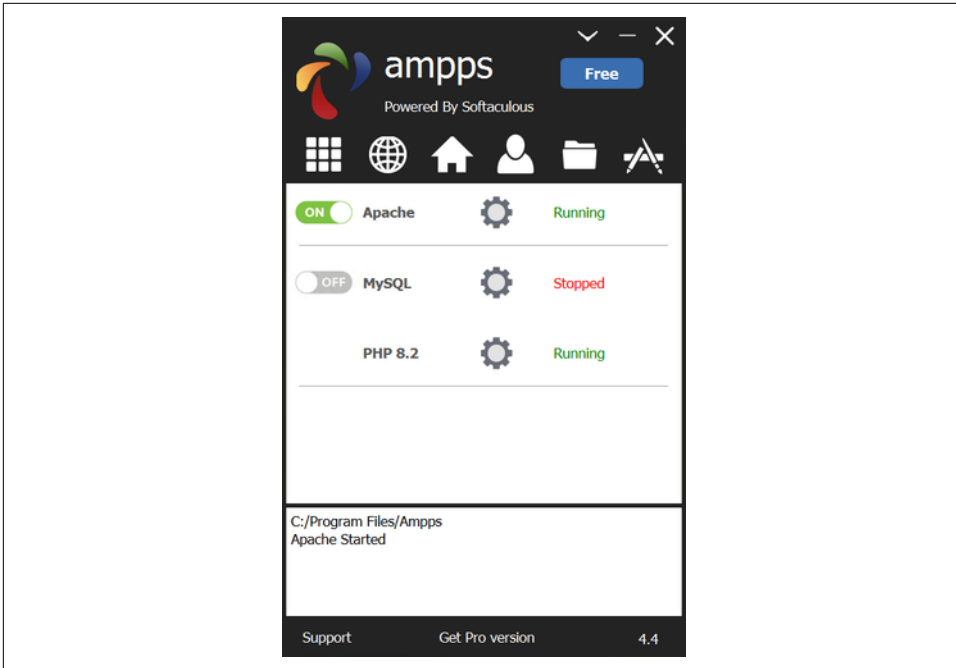


Figure 2-5. The AMPPS control window

You may notice that the default version of PHP in AMPPS is 8.2. If you wish to try other versions for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window and then select Change PHP Version; a new menu will appear from which you can choose to install a different version.

Testing the Installation

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

`http://localhost`

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

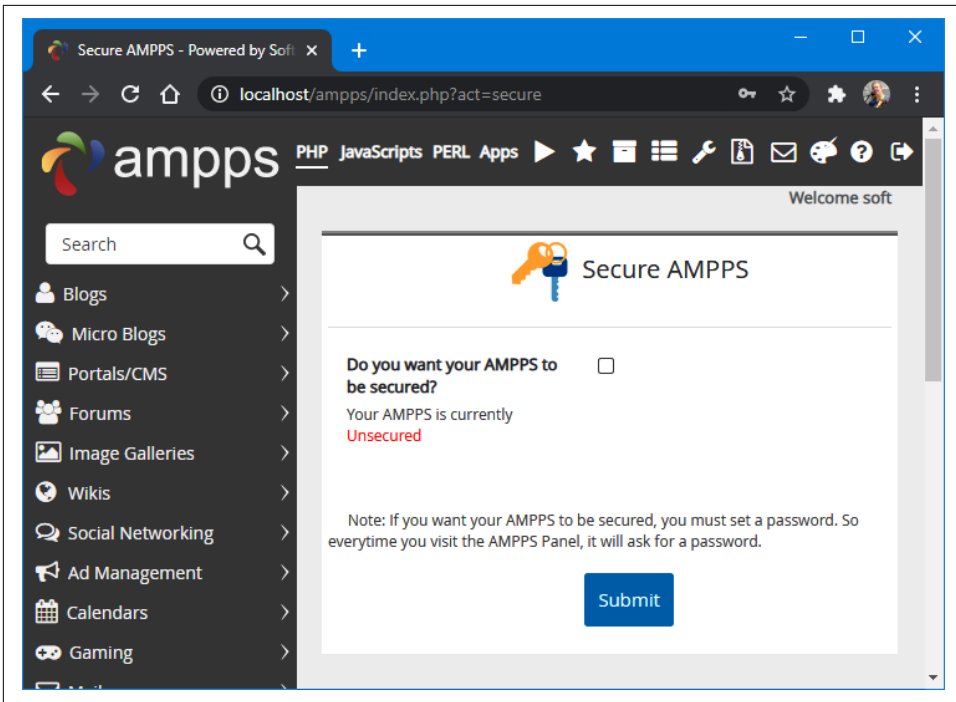


Figure 2-6. The initial security setup screen

Once this has been completed you will be taken to the main control page at <http://localhost/ampps/>. From here you can configure and control all aspects of the AMPPS stack, so note this for future reference or set a bookmark in your browser.

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

```
http://localhost
```

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

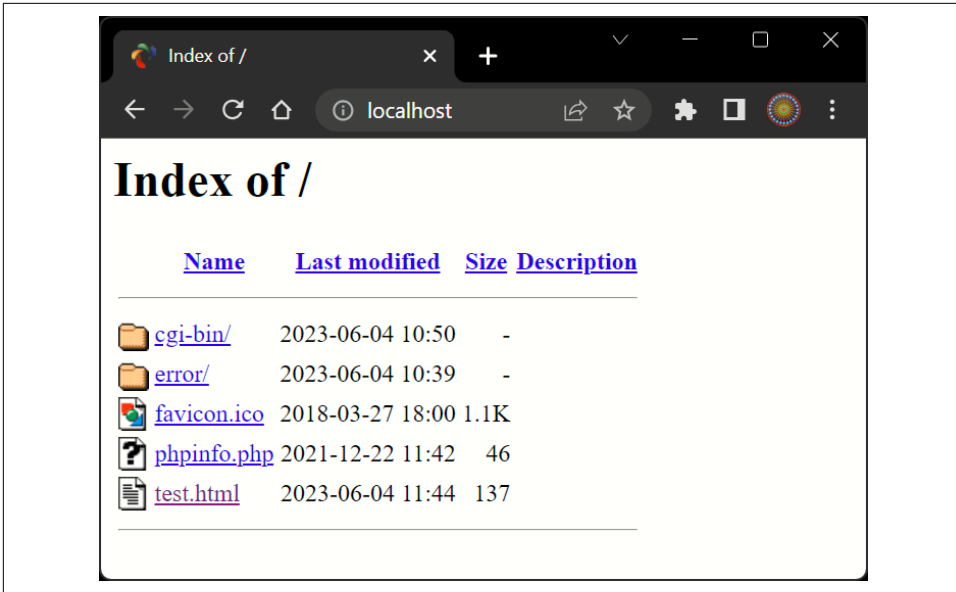


Figure 2-7. Viewing the document root

Accessing the Document Root (Windows)

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

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

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

To ensure that you have everything correctly configured, you should now create the obligatory “Hello World” file. So, create a small HTML file along the following lines using a plain-text editor such as Windows Notepad (which will work just fine, although better suited applications called code editors are discussed later in this chapter):

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

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

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

`http://localhost/test.html`

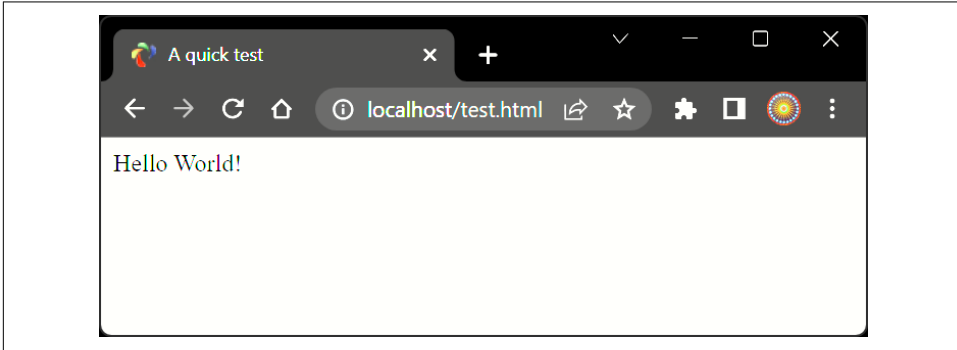


Figure 2-8. Your first web page

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

Alternative WAMPs

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

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

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

- [EasyPHP](#)
- [XAMPP](#)
- [WAMPServer](#)



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

Installing AMPPS on macOS

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

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

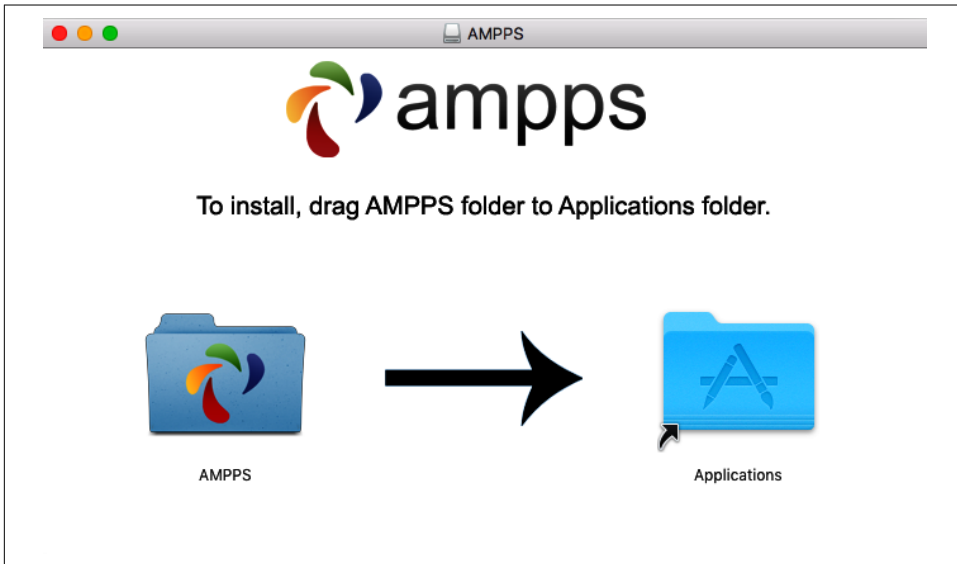


Figure 2-9. Drag the AMPPS folder to Applications

Open your *Applications* folder as usual, and double-click the AMPPS program. If your security settings prevent the file being opened, hold down the Control key and click the icon once. A new window will pop up asking if you are sure you wish to open it. Click Open to do so. When the app starts, you may have to enter your macOS password to proceed.

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



You may notice that the default version of PHP in AMPPS is 8.2. If you wish to try a different version for any reason, click the Options button (nine white boxes in a square) within the AMPPS control window, then select Change PHP. A new menu will appear in which you can choose to install other versions of PHP.

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

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

To ensure that you have everything correctly configured, you should now create the obligatory “Hello World” file. So, create a small HTML file along the following lines using the TextEdit program (which will work just fine, although better suited applications called code editors are discussed later in this chapter):

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

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

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

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



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

Installing a LAMP on Linux

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

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

```
http://localhost
```

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

Working Remotely

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

Accessing MySQL remotely may not be easy either. You should use the secure SSH protocol to log in to your server to manually create databases and set permissions from the command line. Your web hosting company will advise you on how best to do this and provide you with any password it has set for your MySQL access (as well as, of course, for getting into the server in the first place).

Logging In

I recommend that Windows users should install a program such as **PuTTY** for SSH access (SSH is much more secure than the Telnet protocol). Although modern Windows come with SSH preinstalled, PuTTY's user interface may be a bit easier to use especially if you're a beginner.

On a Mac, you already have SSH available as well. Just select the *Applications* folder, followed by *Utilities*, and then launch Terminal. In the Terminal window, log in to a server using SSH like this:

```
ssh mylogin@server.com
```

where *server.com* is the name of the server you wish to log in to and *mylogin* is the username you will log in under. You will then be prompted for the correct password for that username and, if you enter it correctly, you will be logged in.

Transferring Files

To transfer files to and from your web server, you will need a file transfer program that implements an FTPS or SFTP protocol, to ensure proper security on your web server. If you go searching the web for a good client, you'll find so many that it could take you quite a while to locate one with all the right features for you.



Don't Use FTP

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

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

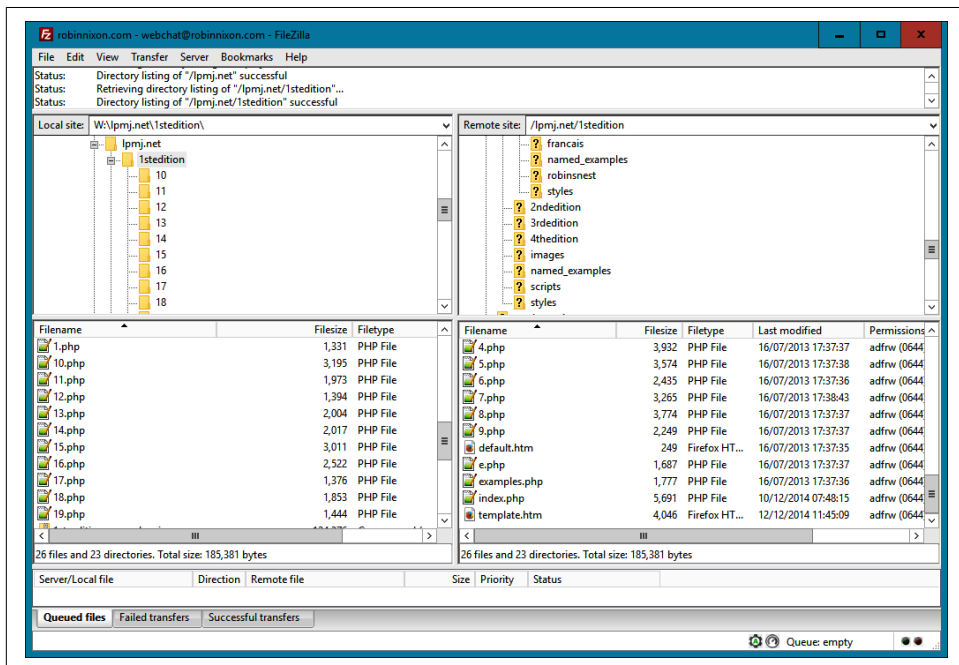


Figure 2-10. FileZilla is a full-featured SFTP program

Another well-known tool is **WinSCP** which, despite its name, also supports SFTP and FTP. Of course, if you already have an FTPS or SFTP program, all the better—stick with what you know.

Using a Code Editor

Although a plain-text editor works for editing HTML, PHP, and JavaScript, there have been some tremendous improvements in dedicated code editors, which now incorporate very handy features such as colored syntax highlighting. Today's program editors are smart and can show your syntax errors before you even run a program. Once you've used a modern editor, you'll wonder how you ever managed without one.

There are a number of good programs available, but I have settled on Visual Studio Code (VSC) from Microsoft because it's powerful; runs on all of Windows, Mac, and Linux; and is free (see [Figure 2-11](#)). It is also a comprehensive developing environment and is becoming ever more standard in the industry.

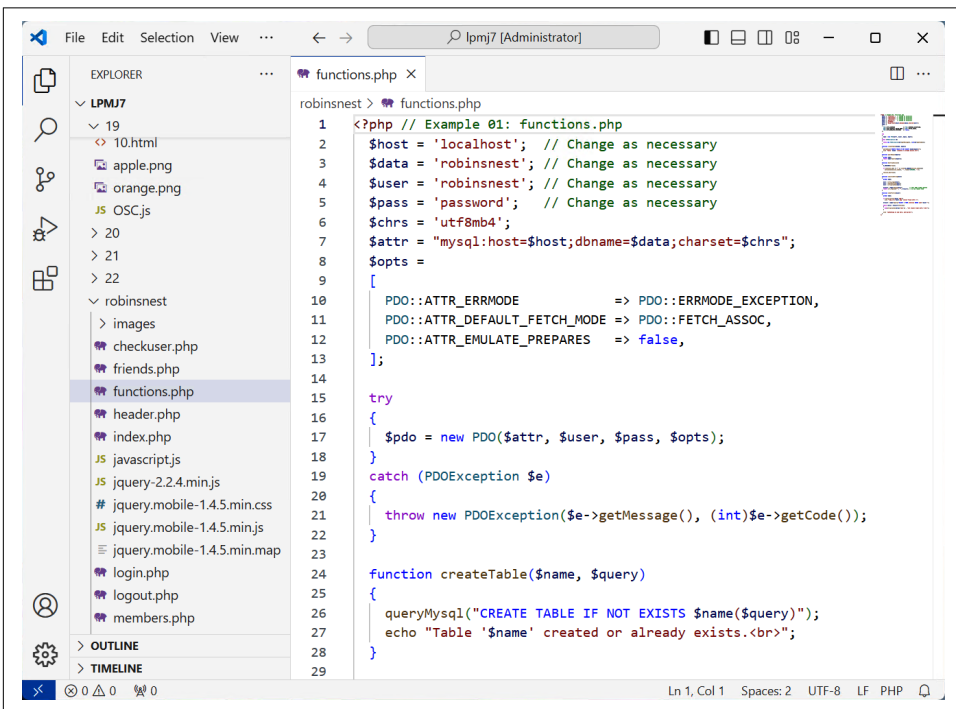


Figure 2-11. Program editors (like Visual Studio Code) are superior to plain-text editors

As you can see in [Figure 2-11](#), VSC highlights the syntax appropriately, using colors to help clarify what's going on. What's more, you can place the cursor next to brackets or braces, and it will highlight the matching ones so that you can check whether you have too many or too few. In fact, VSC does a lot more in addition, which you will discover and enjoy as you use it. You can download a copy from the [Visual Studio website](#).

Again, if you have a different preferred program editor, use that; it's always a good idea to use programs you're already familiar with. However, you will be hard pressed to find something better than the now industry standard VSC, and you should know how to use this product as many job positions will require it.

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

Questions

1. What is the difference between a WAMP, a MAMP, and a LAMP?
2. What is the purpose of an SFTP program?
3. Name the main disadvantage of working on a remote web server.
4. Why is it better to use a code editor instead of a plain-text editor?

See “[Chapter 2 Answers](#)” on [page 568](#) in the [Appendix](#) for the answers to these questions.

Introduction to PHP

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

In production, your web pages will be a combination of HTML, CSS, JavaScript, PHP, and SQL. Furthermore, each page can lead to other pages to provide users with ways to click through links and fill out forms.

We can avoid all that complexity while learning each language, though. Let’s focus, for now, on just writing PHP code and making sure that you get the output you expect—or at least that you understand the output you actually get!

Incorporating PHP Within HTML

By default, PHP documents end with the extension *.php*. When a web server encounters this extension in a requested file, it automatically passes it to the PHP processor. Of course, web servers are highly configurable, and some web developers choose to force files ending with *.htm* or *.html* to also get parsed by the PHP processor, usually because they want to hide their use of PHP.

Your PHP program is responsible for passing back a clean file suitable for display in a web browser. At its very simplest, a PHP document will output only HTML. To prove this, you can take any normal HTML document and save it as a PHP document (for example, saving *index.html* as *index.php*), and it will display identically to the original (as long as the file is being served with Apache and not directly from your filesystem).

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

```
<?php
```

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

```
?>
```

A small PHP “Hello World” program might look like [Example 3-1](#).

Example 3-1. Invoking PHP

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

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

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

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

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

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

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



If you have only PHP code in a file, you may omit the closing `?>`. This can be a good practice, as it will ensure that you have no excess whitespace leaking from your PHP files. It is especially important when you're writing and including object-oriented code; otherwise, a trailing newline character inserted after the closing part may be sent to the browser when it's not expected.

This Book's Examples

To save you the time it would take to type them all in, you can find all the examples from this book in the [repo at GitHub](#).

In addition to listing all the examples by chapter and example number, some of the examples may require explicit filenames, in which case copies of the example(s) are also saved using the filename(s) in the same folder (such as the upcoming [Example 3-4](#), which should be saved as `test1.php`).

The Structure of PHP

We're going to cover a lot of ground in this section, and I recommend that you work your way through it carefully, as it lays the foundation for everything else in this book. As always, there are some useful questions at the end of the chapter that you can use to test how much you've learned.

Using Comments

There are two ways to add comments to your PHP code. The first turns a single line into a comment by preceding it with a pair of forward slashes:

```
// This is a comment
```

This version of the comment feature is a great way to temporarily remove a line of code from a program that is giving you errors. For example, you could use such a comment to hide a debugging line of code until you need it, like this:

```
// echo "X equals $x";
```

You can also use this type of comment directly after a line of code to describe its action, like this:

```
$x += 10; // Move 10 pixels for visual separation
```



Single-line # Comments

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

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

Example 3-2. A multiline comment

```
<?php
/* This is a section
   of multiline comments
   which will not be
   interpreted */
?>
```

You can use the `/*` and `*/` pairs of characters to open and close comments almost anywhere you like inside your code. Most programmers use this construct to temporarily comment out entire sections of code that do not work or that, for one reason or another, they do not wish to be interpreted.



A common error is to use `/*` and `*/` to comment out a large section of code that already contains a commented-out section that uses those characters. You can't nest comments this way; the PHP interpreter won't know where a comment ends and will display an error message. However, if you use an editor or IDE with syntax highlighting, this type of error is easier to spot.

Basic Syntax

PHP is quite a simple language with roots in C and Perl (if you have ever come across these), yet it looks more like Java. It is also very flexible, but you need to learn a few rules about its syntax and structure.

Semicolons

You may have noticed in the previous examples that the PHP commands ended with a semicolon, like this:

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

One of the most common causes of errors you will encounter with PHP is forgetting this semicolon. This causes PHP to treat multiple statements like one statement, which it is unable to understand, prompting it to produce a Parse error message.

The \$ symbol

The \$ symbol is used in many different ways by different programming languages. For example, in the BASIC language, it was used to terminate variable names to denote them as strings.

In PHP, however, you must place a \$ in front of *all* variables. This is required to make the PHP parser faster, as it instantly knows whenever it comes across a variable. Whether your variables are numbers, strings, or arrays, they should all look something like those in [Example 3-3](#).

Example 3-3. Three different types of variable assignment

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

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

Variables

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

String variables

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

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

The quotation marks indicate that “Fred Smith” is a *string* of characters. You must enclose each string in either quotation marks or apostrophes (single quotes), although there is a subtle difference between the two types of quote, as explained later. When you want to see what's in the box, you open it, take out the piece of

paper, and read it. In PHP, doing so looks like this (which displays the contents of the variable):

```
echo $username;
```

Or you can assign it to another variable (photocopy the paper and place the copy in another matchbox), like this:

```
$current_user = $username;
```

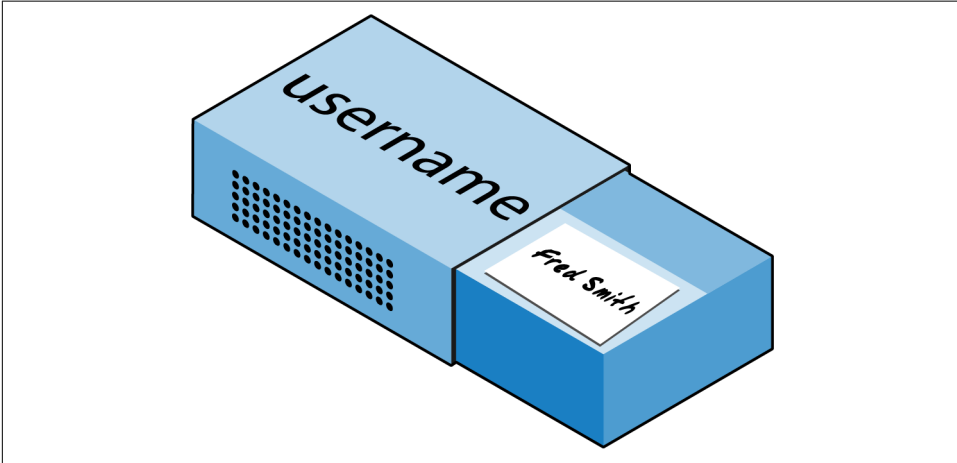


Figure 3-1. You can think of variables as matchboxes containing items

Let's bring all these variables together to form a complete program, as in [Example 3-4](#).

Example 3-4. Your first PHP program

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

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

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



In the unlikely event that during the installation of your web server (as detailed in [Chapter 2](#)) you changed the port assigned to the server to anything other than 80, then you must place that port number within the URL in this and all other examples in this book. So, for example, if you changed the port to 8080, the preceding URL would become this:

```
http://localhost:8080/test1.php
```

I won't mention this again, so just remember to use the port number (if required) when trying examples or writing your own code.

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

Numeric variables

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

```
$count = 17;
```

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

```
$count = 17.5;
```

If you want to use the number in PHP, you can assign the value of `$count` to another variable or perhaps just echo it to the web browser. Either of those would be the equivalent to opening the matchbox and counting the beads.

Arrays

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

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

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

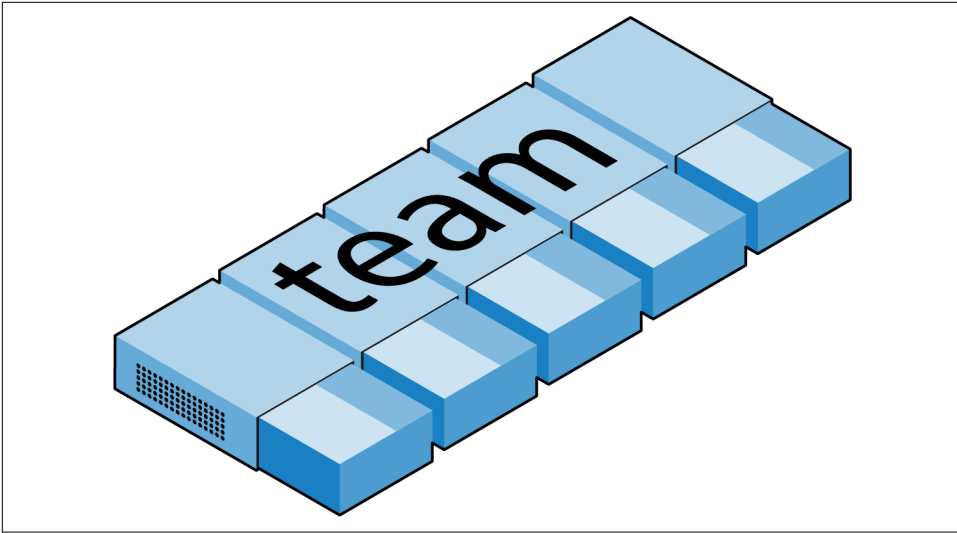


Figure 3-2. An array is like several matchboxes glued together

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

```
array();
```

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



Short Array Syntax

An alternative short array syntax uses [...] instead of the array(...) construct. The previous array could also be written as:

```
$team = ['Bill', 'Mary', 'Mike', 'Chris', 'Anne'];
```

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

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

The reason the previous statement has the number 3, not 4, is that the first element of a PHP array is actually the zeroth element, so the player numbers will therefore be 0 through 4.

Two-dimensional arrays

There's a lot more you can do with arrays. For example, instead of being single-dimensional lines of matchboxes, they can be two-dimensional matrixes or have even more dimensions.

As an example of a two-dimensional array, say we want to keep track of a game of tic-tac-toe, which requires a data structure of nine cells arranged in a 3×3 square. To represent this with matchboxes, imagine nine of them glued to one another in a matrix of three rows by three columns using an array named `$oxo` (see [Figure 3-3](#)).

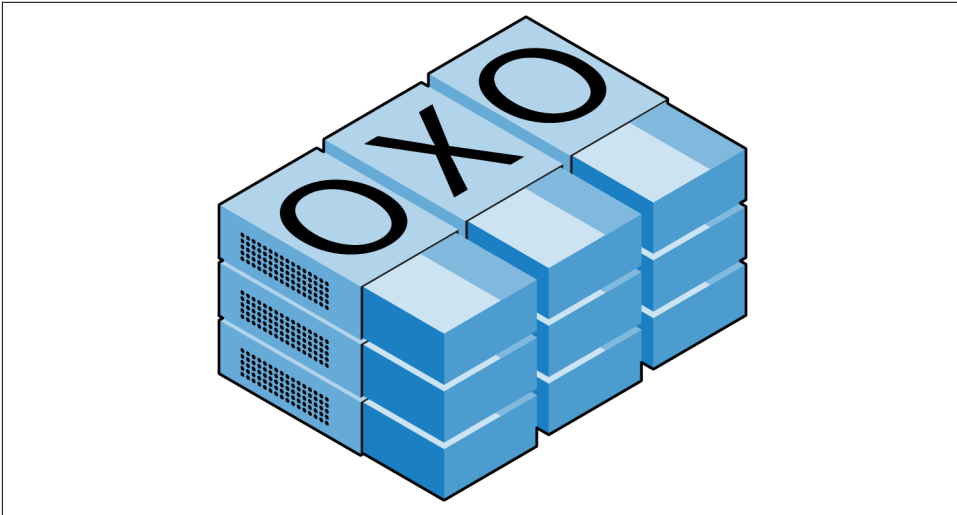


Figure 3-3. A multidimensional array simulated with matchboxes

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

Example 3-5. Defining a two-dimensional array

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

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

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

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



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

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

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

Variable-naming rules

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

- Variable names, after the dollar sign, must start with a letter of the alphabet or the _ (underscore) character.
- Variable names can contain only the characters a–z, A–Z, 0–9, and _ (underscore).
- Variable names may not contain spaces. If a variable name must comprise more than one word, separate the words with the _ (underscore) character (e.g., \$user_name).
- Variable names are case-sensitive. The variable \$High_Score is not the same as the variable \$high_score.



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

Operators

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

```
echo 6 + 2;
```

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

Arithmetic operators

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

Table 3-1. Arithmetic operators

Operator	Description	Example
+	Addition	<code>\$j + 1</code>
-	Subtraction	<code>\$j - 6</code>
*	Multiplication	<code>\$j * 11</code>
/	Division	<code>\$j / 4</code>
%	Modulus (the remainder after a division is performed)	<code>\$j % 9</code>
++	Increment	<code>++\$j</code>
--	Decrement	<code>--\$j</code>
**	Exponentiation (or power)	<code>\$j**2</code>

Assignment operators

Assignment operators assign values to variables. They start with the very simple `=` and move on to `+=`, `-=`, and so on (see [Table 3-2](#)). The operator `+=` adds the value on the right side to the variable on the left, instead of totally replacing the value on the left. Thus, if `$count` starts with the value 5, the statement:

```
$count += 1;
```

sets `$count` to 6, just like the more familiar assignment statement:

```
$count = $count + 1;
```

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

```
$a = "Hello";  
$a .= "."; // Equivalent to writing $a = $a . ".";  
echo $a; // The output is Hello.
```

The `%=` operator assigns the modulus:

```
$number = 12;  
$number %= 10; // Equivalent to writing $number = $number % 10;  
echo $number; // Echoes 2, the remainder of 12 divided by 10.
```

Table 3-2. Assignment operators

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

Comparison operators

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

Table 3-3. Comparison operators

Operator	Description	Example
<code>==</code>	Is equal to	<code>\$j == 4</code>
<code>!=</code>	Is not equal to	<code>\$j != 21</code>
<code>></code>	Is greater than	<code>\$j > 3</code>
<code><</code>	Is less than	<code>\$j < 100</code>
<code>>=</code>	Is greater than or equal to	<code>\$j >= 15</code>
<code><=</code>	Is less than or equal to	<code>\$j <= 8</code>
<code><></code>	Is not equal to	<code>\$j <> 23</code>
<code>===</code>	Is identical to	<code>\$j === "987"</code>
<code>!==</code>	Is not identical to	<code>\$j !== "1.2e3"</code>

Note the difference between `=` and `==`. The first is an assignment operator, and the second is a comparison operator. Also remember that `==` compares the two values for being equivalent, while `===` requires them to be identical. Even advanced programmers can sometimes confuse the use of these when coding hurriedly, so be careful.

Logical operators

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

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

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

As the previous example shows, you generally use a logical operator to combine the results of two of the comparison operators shown in “Comparison operators” on page 44. A logical operator can also be input to another logical operator: “If the time is later than 12 p.m. and earlier than 2 p.m., or if the smell of a roast is permeating the hallway and there are plates on the table.” As a rule, if something has a TRUE or FALSE value, it can be input to a logical operator. A logical operator takes two true or false inputs and produces a true or false result.

Table 3-4 shows the logical operators according to precedence, which is discussed in Chapter 4.

Table 3-4. Logical operators

Operator	Description	Example
&&	<i>And</i>	<code>\$j == 3 && \$k == 2</code>
and	Low-precedence <i>and</i>	<code>\$j == 3 and \$k == 2</code>
	<i>Or</i>	<code>\$j < 5 \$j > 10</code>
or	Low-precedence <i>or</i>	<code>\$j < 5 or \$j > 10</code>
!	<i>Not</i>	<code>! (\$j == \$k)</code>
xor	<i>Exclusive or</i>	<code>\$j xor \$k</code>

Note that `&&` is usually interchangeable with `and`; the same is true for `||` and `or`. However, because `and` and `or` have a lower precedence, you should avoid using them except when they are the only option, as in the following statement, which *must* use the `or` operator (`||` cannot be used to force a second statement to execute if the first fails):

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

Operator precedence

Operator precedence determines how particular expressions are grouped together. The concept is also used in common math as illustrated by the following statement:

`5 + 2 * 3`

The result is 11, because multiplication has a higher precedence than addition, so the result is computed as *2 multiplied by 3*, which equals to 6, *plus 5*. It can be rewritten as:

`5 + (2 * 3)`

The parentheses in this case are optional but they help to illustrate the precedence. In other cases, parentheses can be used to change or force precedence, for example the result of the following statement is 21:

```
(5 + 2) * 3
```

You'll learn more about operator precedence in [Chapter 4](#).

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

```
$ammonia = true;
$bleach = false;
$safe = ($ammonia xor $bleach);
echo $safe; // output 1 which is true
```

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

Variable Assignment

The syntax to assign a value to a variable is always `$variable = value`. Or, to reassign the value to another variable, it is `$other_variable = $variable`, remembering to preface variable names with `$` symbols in PHP.

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

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

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

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

Variable incrementing and decrementing

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

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

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

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

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

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

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

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

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

String concatenation

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

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

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

```
You have 5 messages.
```

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

```
$bulletin = "This is a test of the broadcast system.";
$newsflash = "Houston, we have a problem.";
$bulletin .= " " . $newsflash;
echo $bulletin;
```

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

String types

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

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

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

On the other hand, when you want to include the value of a variable inside a string, you do so by using double-quoted strings. You can wrap the variable name in curly braces `{` and `}` to explicitly specify the end of the variable name:

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

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

Escaping characters

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

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

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

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

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

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

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

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


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

Multiline Strings

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

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

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

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

    - $author.";
?>
```

Example 3-7. A multiline string assignment

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

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

    - $author.";
?>
```

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

Example 3-8. Alternative multiline `echo` statement

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

    echo <<<_END
    Debugging is twice as hard as writing the code in the first place.
    Therefore, if you write the code as cleverly as possible, you are,
```

by definition, not smart enough to debug it.

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

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

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



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

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

Example 3-9. A multiline string variable assignment

```
<?php  
$author = "Scott Adams";  
  
$out = <<<_END  
Normal people believe that if it ain't broke, don't fix it.  
Engineers believe that if it ain't broke, it doesn't have enough  
features yet.  
  
- $author.  
_END;  
echo $out;  
?>
```

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

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

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

Using a nowdoc

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

Example 3-10. A nowdoc multiline assignment

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

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

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

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



Laying out text over multiple lines is usually just a convenience to make your PHP code easier to read, because once it is displayed in a web page, HTML formatting rules take over and whitespace is suppressed (but in a heredoc, `$author` in our example will still be replaced with the variable's value, unlike in a nowdoc).

So, for example, if you load these multiline output examples into a browser, they will *not* display over several lines, because all browsers treat newlines just like spaces. However, if you use the browser's View Source feature, you will find that the newlines are correctly placed and that PHP preserved the line breaks.

Variable Typing

PHP is a loosely typed language. This means variables do not have to be declared before they are used and PHP always converts variables to the type required by their context when they are accessed.

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

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

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

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

The same goes for turning a string into a number, and so on. In [Example 3-12](#), the variable `$pi` is set to a string value, which is then automatically turned into a floating-point number in the third line by the equation for calculating a circle's area, which outputs the value 78.5398175.

Example 3-12. Automatically converting a string to a number

```
<?php
$pi      = "3.1415927";
$radius = 5;
echo $pi * ($radius * $radius);
?>
```

In practice, what this means is that you don't have to worry too much about your variable types, although it is possible for type declarations to be added to function arguments, return values, and (as of PHP 7.4.0) class properties, ensuring that the value is of the specified type at call time; otherwise, a **TypeError** is thrown.

Assuming type declarations are not being used, just assign them values that make sense to you, and PHP will convert them if necessary. Then, when you want to retrieve values, just ask for them—for example, with an `echo` statement, but do

remember that sometimes automatic conversions do not operate quite as you might expect.

If type declarations are being used to make the code behave more predictably, you can change the type of the variable by prefixing it with the chosen type in parentheses, like this:

```
$string = (string)$number;  
$number = (int)$string;  
$boolean = (bool)$integer;
```

Sometimes, it may not be clear at the first sight how the type conversion (sometimes called *type casting*) will go and what will be the result. The PHP manual has all the [conversion rules nicely documented](#).

Constants

Constants are similar to variables, holding information to be accessed later, except that they are what they sound like—constant. In other words, once you have defined a constant, its value is set for the remainder of the program and cannot be altered.

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

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

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

```
$directory = ROOT_LOCATION;
```

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



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

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

Predefined Constants

PHP comes ready-made with dozens of predefined constants that you won't generally use as a beginner. However, there are a few—known as the *magic constants*—that you will find useful. The names of the magic constants always have two underscores at the beginning and two at the end so that you won't accidentally try to name one of your

own constants with a name that is already taken. These are detailed in [Table 3-5](#). The concepts referred to in the table will be introduced in future chapters.

Table 3-5. PHP’s magic constants

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

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

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

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

The Difference Between the echo and print Commands

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

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

By and large, the echo command will be a tad faster than print, because it doesn’t set a return value. On the other hand, because it isn’t implemented like a function, echo cannot be used as part of a more complex expression, whereas print can. Here’s an example to output whether the value of a variable is TRUE or FALSE using print—something you could not perform in the same manner with echo, because it would display a Parse error message as the ternary operator expects an expression that returns a value and while for example echo "TRUE" doesn’t, print "TRUE" returns 1:

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

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

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

Functions

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

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

To create a function, declare it as shown in [Example 3-13](#).

Example 3-13. A simple function declaration

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

This function returns a date in the format *Sunday May 2nd 2027*. Any number of parameters can be passed between the initial parentheses; we have chosen to accept just one. The curly braces enclose all the code that is executed when you later call the function. Note that the first letter within the date function call in this example is a lowercase letter `L`, not to be confused with the number `1`.

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

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

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

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

which passes to `longdate` the current time less the number of seconds since 17 days ago (17 days $\times$ 24 hours $\times$ 60 minutes $\times$ 60 seconds).

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

Variable Scope

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

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

Local variables

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

One set of local variables is the list of arguments to a function. In “[Functions](#)” on [page 55](#), we defined a function that accepted a parameter named `$timestamp`. This is meaningful only in the body of the function; you can't get or set its value outside the function.

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

Example 3-14. An expanded version of the `longdate` function

```
<?php
function longdate($timestamp)
{
    $temp = date("l F jS Y", $timestamp);
    return "The date is $temp";
}
?>
```

Here we have assigned the value returned by the `date` function to the temporary variable `$temp`, which is then inserted into the string returned by the function. As soon as the function returns, the `$temp` variable and its contents disappear, as if they had never been used at all.

To see the effects of variable scope making an outside variable invisible inside a function, let's look at some similar code in [Example 3-15](#). Here `$temp` has been created *before* we call the `longdate` function.

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

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

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

However, because `$temp` was neither created within the `longdate` function nor passed to it as a parameter, `longdate` cannot access it. Therefore, this code snippet outputs only the date, not the preceding text. In fact, depending on how PHP is configured, it may first display the error message `Notice: Undefined variable: temp`, something you don't want your users to see. The reason for this is, by default, variables created within a function are local to that function, and variables created outside of any functions can be accessed only by nonfunction code.

Some ways to repair [Example 3-15](#) appear in [Examples 3-16](#) and [3-17](#).

Example 3-16. Rewriting to refer to `$temp` within its local scope fixes the problem

```
<?php
$temp = "The date is ";
echo $temp . longdate(time());

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

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

Example 3-17. An alternative solution: passing `$temp` as an argument

```
<?php
$temp = "The date is ";
echo longdate($temp, time());
```

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

The solution in [Example 3-17](#) passes `$temp` to the `longdate` function as an extra argument. `longdate` reads it into a temporary variable that it creates called `$text` and outputs the desired result.



Forgetting the scope of a variable is a common programming error, so remembering how variable scope works will help you debug some quite obscure problems. Suffice it to say that unless you have declared a variable otherwise, its scope is limited to being local: either to the current function or to the code outside of any functions, depending on whether it was first created or accessed inside or outside a function.

Global variables

In some cases you need a variable to have *global* scope, because you want all your code to be able to access it. Also, some data may be large and complex, and you don't want to keep passing it as arguments to functions.

To access variables from global scope, add the keyword `global`. Let's assume that you have a way of logging your users in to your website and want all your code to know whether it is interacting with a logged-in user or a guest. One way to do this is to use the `global` keyword before a variable, such as `$IS_LOGGED_IN`:

```
global $IS_LOGGED_IN;
```

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

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

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



I generally adopt the convention of making all variable names that require global access uppercase (just as it's recommended that constants should be uppercase, except constants are not prefixed with `$`) so that I can see at a glance the scope of a variable.

Static variables

In “[Local variables](#)” on page 56, I mentioned that the value of a local variable is wiped out when the function ends. If a function runs many times, it starts with a fresh copy of the variable, and the previous setting has no effect.

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

Example 3-18. A function using a static variable

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

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

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

If you plan to use static variables, you should note that you cannot assign the result of an expression in their definitions. They can be initialized only with predetermined values (see [Example 3-19](#)). Generally, however, like global variables, static variables make functions less deterministic, meaning that the function can have a different output given the same input, and they are best avoided in preference for functions with no side effects like changing a static variable.

Example 3-19. Allowed and disallowed static variable declarations

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

Superglobal variables

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

These superglobals contain lots of useful information about the currently running program and its environment (see Table 3-6). They are structured as associative arrays, a topic discussed in Chapter 6.

Table 3-6. PHP’s superglobal variables

Superglobal name	Contents
\$GLOBALS	All variables that are currently defined in the global scope of the script. The variable names are the keys of the array.
\$_SERVER	Information such as headers, paths, and locations of scripts. The entries in this array are created by the web server, and there is no guarantee that every web server will provide any or all of these.
\$_GET	Variables passed to the current script via the HTTP GET method.
\$_POST	Variables passed to the current script via the HTTP POST method.
\$_FILES	Items uploaded to the current script via the HTTP POST method.
\$_COOKIE	Variables passed to the current script via HTTP cookies.
\$_SESSION	Session variables available to the current script.
\$_REQUEST	Contents of information passed from the browser; by default, \$_GET, \$_POST, and \$_COOKIE.
\$_ENV	Variables passed to the current script via the environment method.

All of the superglobals (except for \$GLOBALS) are named with a single initial underscore and only capital letters; therefore, you should avoid naming your own variables in this manner to avoid potential confusion.

To illustrate how you use them, let’s look at a common example. Among the many nuggets of information supplied by superglobal variables is the URL of the page that referred the user to the current web page. This referring page information can be accessed like this:

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

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



Difference between Superglobals and Constants

Superglobals are regular variables that have the full scope of a program during runtime, are visible and usable everywhere, whereas constants, while also visible inside functions, have no scope whatsoever, because they are accessed early on at compile time and “baked” as fixed values into the runtime code before any scope is created.

Superglobals and security

A word of caution is in order before you start using superglobal variables, because they are often used by hackers trying to find exploits to break into your website. What they do is load up `$_POST`, `$_GET`, or other superglobals with malicious code, such as Unix or MySQL commands that can damage or display sensitive data if you naively access them.

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

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

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



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

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

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

Questions

1. What tag is used to invoke PHP to start interpreting program code? And what is the short form of the tag?
2. What are the two types of comment tags?
3. Which character must be placed at the end of every PHP statement?
4. Which symbol is used to preface all PHP variables?
5. What can a variable store?
6. What is the difference between `$variable = 1`, `$variable == 1`, and `$variable === 1`?
7. Why is an underscore allowed in variable names (`$current_user`), whereas hyphens are not (`$current-user`)?
8. Are variable names case-sensitive?
9. Can you use spaces in variable names?
10. How do you convert one variable type to another (say, a string to a number)?
11. What is the difference between `++$j` and `$j++`?
12. Are the operators `&&` and `and` interchangeable?
13. How can you create a multiline echo or assignment?
14. Can you redefine a constant?
15. How do you escape a quotation mark?
16. What is the difference between the `echo` and `print` commands?
17. What is the purpose of functions?
18. How can you make a variable accessible to all parts of a PHP program?
19. If you generate data within a function, what are a couple of ways to convey the data to the rest of the program?
20. What is the result of combining a string with a number?

See “[Chapter 3 Answers](#)” on page 568 in the [Appendix](#) for the answers to these questions.

Expressions and Control Flow in PHP

Chapter 3 introduced several topics in passing that this chapter covers more fully, such as making choices (branching) and creating complex expressions. In **Chapter 3**, I wanted to focus on the most basic syntax and operations in PHP, but I couldn't avoid touching on more advanced topics. Now I can fill in the background that you need to use these powerful PHP features properly.

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

Expressions

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

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

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

Which in PHP would be:

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

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

TRUE or FALSE?

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



Note that I am using uppercase letters for the names TRUE and FALSE. This is because they are predefined constants in PHP. You can use the lowercase versions if you prefer, as they are also predefined.

PHP doesn't actually print the predefined constants if you ask it to do so as in [Example 4-1](#). For each line, the example prints out a letter followed by a colon and a predefined constant. Only strings can be printed in PHP, and conversion rules have been defined for other types, like numbers or Boolean values. When converting to string, PHP arbitrarily assigns a string value of "1" to TRUE, so 1 is displayed after a: when the example runs. Even more mysteriously, the line starting with b: doesn't print 0 as you may expect. That's because during the conversion, FALSE is converted to an empty string "". The constant FALSE is different than NULL, another predefined constant that denotes nothing, even though both are converted to an empty string when printed.

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

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

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

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

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

Example 4-2. Four simple Boolean expressions

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


The output from this code is:

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

By the way, in some languages FALSE may be defined as 0 or even -1, so it's worth checking on its definition in each language you use. Luckily, you normally don't have to worry about what TRUE and FALSE look like internally.

Literals and Variables

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

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

Example 4-3. Literals and variables

```
$myname = "Brian";
$myage  = 37;

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



About Literals and Non-literals

The difference between a literal and a non-literal is that if you assign a literal to a variable, the variable will have the same value as the literal, even if you assign it repeatedly. However, if you assign a function output to a variable, for example, the function can have a different output each time it is called, making the variable content non-literal.

And, as you'd expect, you see a return value from all of these in the following output:

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

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

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

Example 4-4. An expression and a statement

```
$days_to_new_year = 366 - $day_number; // Expression

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

Operators

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

Table 4-1. PHP operator types

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

Each operator takes a different number of operands:

- *Unary* operators, such as incrementing (`$a++`) or negation (`!$a`), take a single operand.
- *Binary* operators, which represent the bulk of PHP operators (including addition, subtraction, multiplication, and division), take two operands.

- The one *ternary* operator, which takes the form `expr ? x : y`, requires three operands. It's a terse, single-line `if` statement that returns `x` if `expr` is `TRUE` and `y` if `expr` is `FALSE`.



About Execution Operators

Although the execution operators are powerful, you should avoid using them unless strictly necessary and you know precisely what you are doing, because you could potentially expose a massive vulnerability in your project.

Operator Precedence

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

Example 4-5. Three equivalent expressions

```
1 + 2 + 3 - 4 + 5    // 7
2 - 4 + 5 + 3 + 1    // 7
5 + 2 - 4 + 1 + 3    // 7
```

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

Example 4-6. Three expressions that are also equivalent

```
1 * 2 * 3 / 4 * 5    // 7.5
2 / 4 * 5 * 3 * 1    // 7.5
5 * 2 / 4 * 1 * 3    // 7.5
```

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

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

```
1 + 2 * 3 - 4 * 5    // Without precedence would be 25
2 - 4 * 5 * 3 + 1    // Without precedence would be -29
5 + 2 - 4 + 1 * 3    // Without precedence would be 12
```

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

Example 4-8. Three expressions showing implied parentheses

```
1 + (2 * 3) - (4 * 5) // With precedence: -13
2 - (4 * 5 * 3) + 1   // With precedence: -57
5 + 2 - 4 + (1 * 3)   // With precedence: 6
```

PHP evaluates the subexpressions within parentheses first to derive the semi-completed expressions in [Example 4-9](#).

Example 4-9. After evaluating the subexpressions in parentheses

```
1 + (6) - (20) // With precedence: -13
2 - (60) + 1   // With precedence: -57
5 + 2 - 4 + (3) // With precedence: 6
```

The final results of these expressions are -13, -57, and 6, respectively (quite different from the results of 25, -29, and 12 had there been no operator precedence).

Of course, you can override the default operator precedence by inserting your own parentheses and forcing whatever order you want (see [Example 4-10](#)).

Example 4-10. Forcing left-to-right evaluation

```
((1 + 2) * 3 - 4) * 5 // With forced precedence: 25
(2 - 4) * 5 * 3 + 1   // With forced precedence: -29
(5 + 2 - 4 + 1) * 3   // With forced precedence: 12
```

With parentheses correctly inserted, we now see the values 25, -29, and 12, respectively.

[Table 4-2](#) lists PHP's operators in order of precedence from high to low.

Table 4-2. Precedence of PHP operators (high to low)

Operator(s)	Type
()	Parentheses
++ --	Increment/decrement
!	Logical
* / %	Arithmetic
+ - .	Arithmetic and string

Operator(s)	Type
<< >>	Bitwise
< <= > >= <>	Comparison
== != === !==	Comparison
&	Bitwise (and references)
^	Bitwise
	Bitwise
&&	Logical
	Logical
? :	Ternary
= += -= *= /= . = %= &= != ^= <<= >>=	Assignment
and	Logical
xor	Logical
or	Logical

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

Associativity

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

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

Table 4-3. Operator associativity

Operator	Description	Associativity
< <= > >= != === !== <>	Comparison	None
!	Logical NOT	Right
~	Bitwise NOT	Right
++ --	Increment and decrement	Right
(int)	Cast to an integer	Right
(double) (float) (real)	Cast to a floating-point number	Right
(string)	Cast to a string	Right
(array)	Cast to an array	Right
(object)	Cast to an object	Right

Operator	Description	Associativity
@	Inhibit error reporting	Right
= += -= *= /=	Assignment	Right
.= %= &= = ^= <=> >>=	Assignment	Right
+	Addition and unary plus	Left
-	Subtraction and negation	Left
*	Multiplication	Left
/	Division	Left
%	Modulus	Left
.	String concatenation	Left
<< >> & ^	Bitwise	Left
?:	Ternary	Left
&& and or xor	Logical	Left
,	Separator	Left

For example, let's look at the assignment operator in [Example 4-11](#), where three variables are all set to the value 0.

Example 4-11. A multiple-assignment statement

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

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



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

Relational Operators

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

Equality operators

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

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

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

As you see, by returning either `TRUE` or `FALSE`, the equality operator enables you to test for conditions using, for example, an `if` statement. But that's not the whole story, because PHP is a loosely typed language. If the two operands of an equality expression are of different types, PHP will convert them to whatever type makes the best sense to it. The *identity* operator, which consists of three equals signs in a row, can be used to compare items without doing conversion.

For example, any strings composed entirely of numbers will be converted to numbers whenever compared with a number. In [Example 4-13](#), `$a` and `$b` are two different strings, and we might therefore expect neither of the `if` statements to output a result.

Example 4-13. The equality and identity operators

```
$a = "1000";  
$b = "+1000";  
  
if ($a == $b) echo "1";  
if ($a === $b) echo "2";
```

However, if you run the example, you will see that it outputs the number 1, which means that the first `if` statement evaluated to `TRUE`. This is because both strings were first converted to numbers, and 1000 is the same numerical value as +1000. In contrast, the second `if` statement uses the identity operator, so it compares `$a` and `$b` as strings, sees that they are different, and thus doesn't output anything.

As with forcing operator precedence, whenever you have any doubt about how PHP will convert operand types, you can use the identity operator to turn this behavior off. And although the equality operator `==` looks useful in the previous example, it's the identity operator `===` that's recommended for comparisons especially in any new code to reduce risk of unintended behavior from type conversions.

In the same way that you can use the equality operator to test for operands being equal, you can test for them *not* being equal using `!=`, the inequality operator. In [Example 4-14](#), which is a rewrite of [Example 4-13](#), the equality and identity operators have been replaced with their inverses.

Example 4-14. The inequality and not-identical operators

```
$a = "1000";  
$b = "+1000";  
  
if ($a != $b) echo "1";  
if ($a !== $b) echo "2";
```

And, as you might expect, the first `if` statement does not output the number 1, because the code is asking whether `$a` and `$b` are *not* equal to each other numerically.

Instead, this code outputs the number 2, because the second `if` statement is asking whether `$a` and `$b` are *not* identical to each other in their actual string type, and the answer is TRUE; they are not the same.

Comparison operators

Using comparison operators, you can test for more than just equality and inequality. PHP also gives you `>` (is greater than), `<` (is less than), `>=` (is greater than or equal to), and `<=` (is less than or equal to) to play with. [Example 4-15](#) shows these in use.

Example 4-15. The four comparison operators

```
$a = 2; $b = 3;  
  
if ($a > $b) echo "$a is greater than $b<br>";  
if ($a < $b) echo "$a is less than $b<br>";  
if ($a >= $b) echo "$a is greater than or equal to $b<br>";  
if ($a <= $b) echo "$a is less than or equal to $b<br>";
```

In this example, where `$a` is 2 and `$b` is 3, the output is:

```
2 is less than 3  
2 is less than or equal to 3
```

Try this example yourself, altering the values of `$a` and `$b`, to see the results. Try setting them to the same value and see what happens.

Logical operators

Logical operators produce true or false results and therefore are also known as *Boolean operators*. There are four of them (see [Table 4-4](#)).

Table 4-4. The logical operators

Logical operator	Description
AND	TRUE if both operands are TRUE
OR	TRUE if either operand is TRUE
XOR	TRUE if one of the two operands is TRUE
!	TRUE if operand is FALSE, or FALSE if operand is TRUE (<i>NOT operator</i>)

You can see these operators used in [Example 4-16](#). Note that the `!` symbol is required by PHP in place of NOT. Furthermore, the operators can be lowercase or uppercase (as in the case of `or` in this example, which is in lowercase rather than uppercase).

Example 4-16. The logical operators in use

```
$a = 1; $b = 0;

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

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



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

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

Example 4-17. A statement using the OR operator

```
if ($finished == 1 OR getnext() == 1) exit;
```

If you need `getnext` to be called at each `if` statement, you could rewrite the code as in [Example 4-18](#).

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

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

In this case, the code executes the `getnext` function and stores the value returned in `$gn` before executing the `if` statement. While this has now ensured that `getnext` is called as required, if `$finished` equals 1 then the value in `$gn` will not be tested due to the `or` operator.



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

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

Table 4-5. All possible PHP logical expressions

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

Conditionals

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

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

The if Statement

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

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

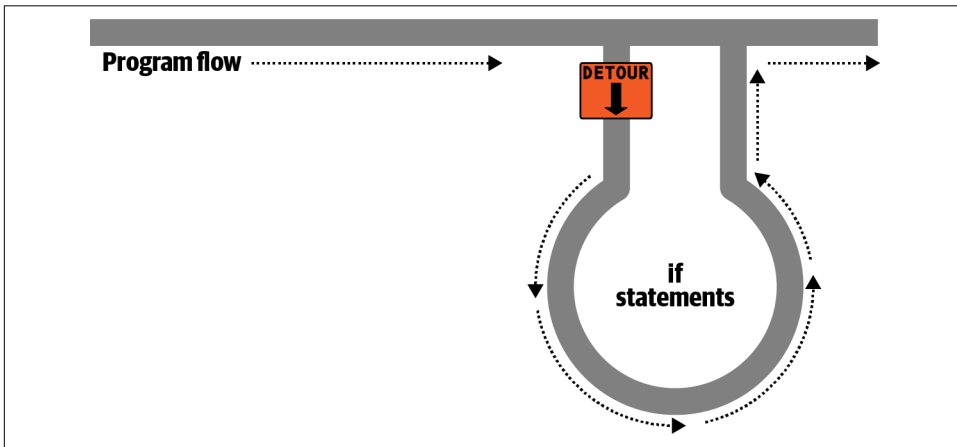


Figure 4-1. Program flow is like a single-lane highway

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

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



A notorious security vulnerability known as the “goto fail” bug haunted the Secure Sockets Layer (SSL) code in Apple’s products for many years because a programmer had forgotten the curly braces around an `if` statement, causing a function to sometimes report a successful connection when that was not the case. This allowed a malicious attacker to get a secure certificate accepted when it should have been rejected. If in doubt, place braces around your `if` statements, although for brevity and clarity, and where the code is small and the intention clear and obvious, some examples in this book do omit the braces for single statements.

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

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

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

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

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

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

The else Statement

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

With an `if...else` statement, the first conditional statement is executed if the condition is `TRUE`. But if it’s `FALSE`, the second one is executed. One of the two choices *must* be executed. Under no circumstance can both (or neither) be executed. **Example 4-20** shows the use of the `if...else` structure.

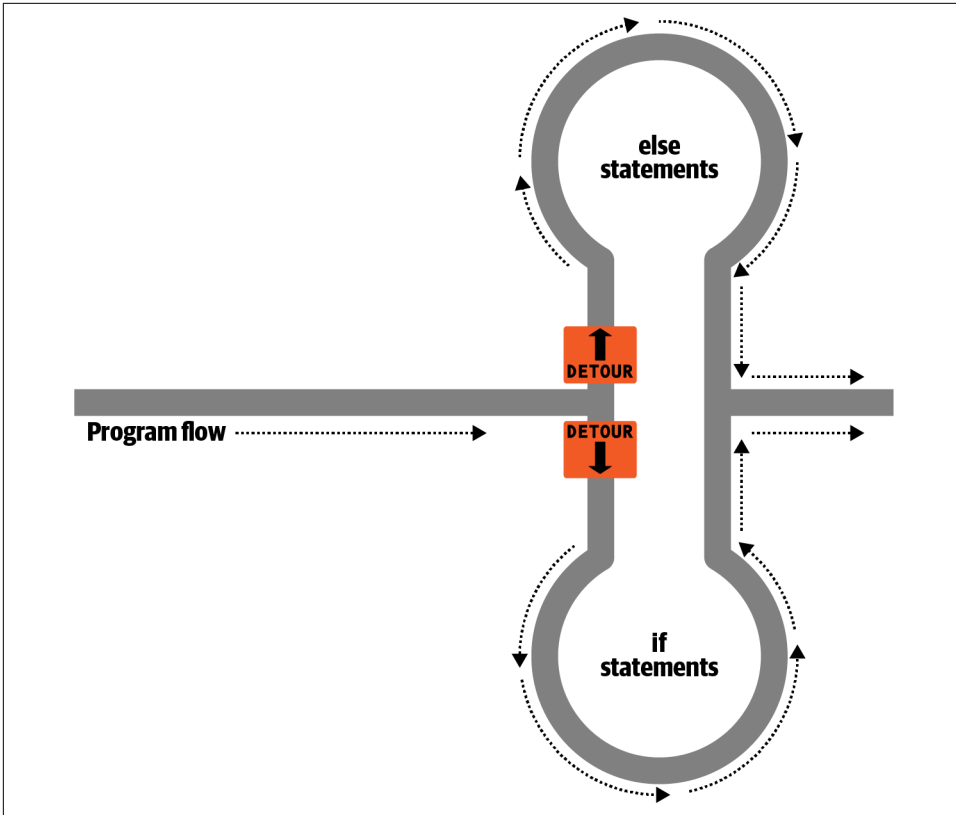


Figure 4-2. Highway now has an *if* detour and an *else* detour

Example 4-20. An *if...else* statement with curly braces

```
if ($bank_balance < 100)
{
    $money      = 1000;
    $bank_balance += $money;
}
else
{
    $savings    += 50;
    $bank_balance -= 50;
}
```

In this example, if you've ascertained that you have \$100 or more in the bank, the *else* statement is executed, placing some of this money into your savings account.

As with the `if` statements, curly braces are always recommended for the `else` statement as well. First, they make the code easier to understand. Second, they let you easily add more statements to the branch later.

The `elseif` Statement

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

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

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

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

Although I'm starting to stretch the metaphor a bit, you can imagine this as a multiway set of detours (see [Figure 4-3](#)).



An `else` statement closes either an `if...else` or an `if...elseif...else` statement. You can leave out a final `else` if it is not required, but you cannot have one before an `elseif`; you also cannot have an `elseif` before an `if` statement.

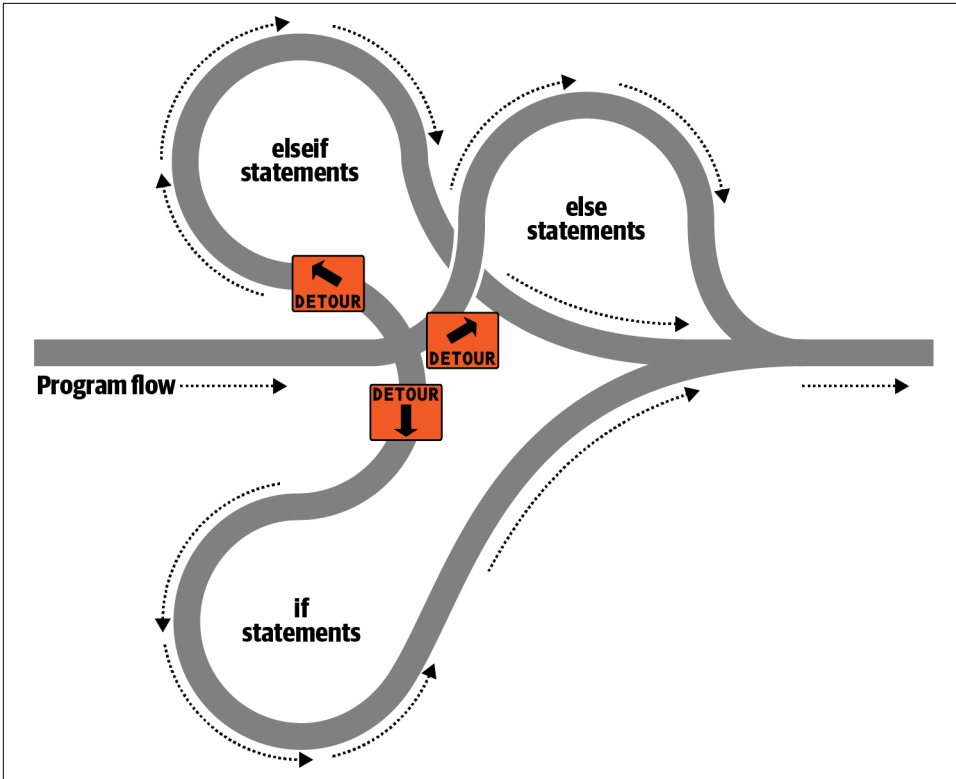


Figure 4-3. The highway with if, elseif, and else detours

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

The switch Statement

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

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

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

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

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

If we use a switch statement, the code might look like [Example 4-23](#).

Example 4-23. A switch statement

```
switch ($page)
{
    case "Home":
        echo "You selected Home";
        break;
    case "About":
        echo "You selected About";
        break;
    case "News":
        echo "You selected News";
        break;
    case "Login":
        echo "You selected Login";
        break;
    case "Links":
        echo "You selected Links";
        break;
}
```

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



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

Breaking out

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

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

Default action

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

Example 4-24. A default statement to add to [Example 4-23](#)

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

This replicates the effect of the `else` statement in [Example 4-22](#).

Although a `break` command is not required here because the default is the final substatement and program flow will automatically continue to the closing curly brace, should you decide to move the `default` statement higher up (not generally recommended practice), it would definitely need a `break` command to prevent program flow from dropping into the following statements. Generally, the safest practice is to always include the `break` command.

Alternative syntax

If you prefer, you can replace the first curly brace in a `switch` statement with a single colon and the final curly brace with an `endswitch` command, as in [Example 4-25](#). However, this approach is not commonly used and is mentioned here only in case you encounter it in third-party code.

Example 4-25. Alternate switch statement syntax

```
switch ($page):  
    case "Home":  
        echo "You selected Home";  
        break;  
  
    // etc  
  
    case "Links":  
        echo "You selected Links";  
        break;  
endswitch;
```

The ? (or Ternary) Operator

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

We briefly discussed this in [Chapter 3](#) about the difference between the `print` and `echo` statements as an example of an operator type that works well with `print` but not `echo`.

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

Example 4-26. Using the ? operator

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

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

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

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

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

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

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

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

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

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

Looping

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

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

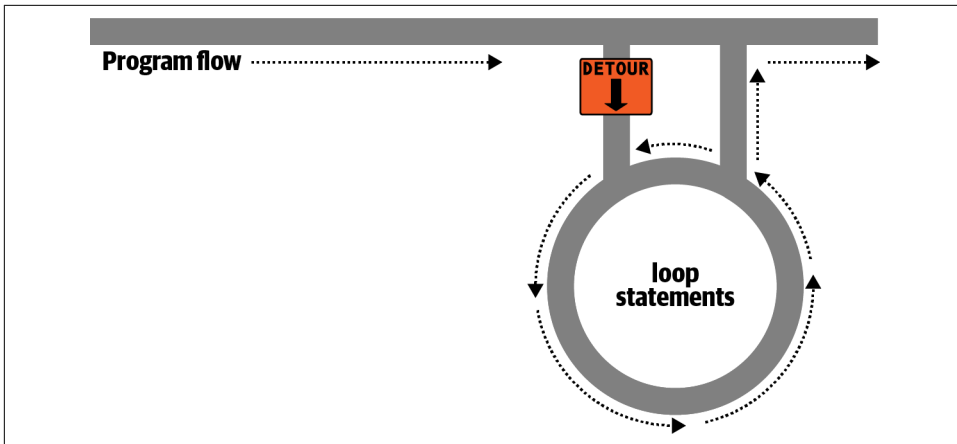


Figure 4-4. Imagining a loop as part of a program highway layout

while Loops

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

Example 4-28. A `while` loop

```
$fuel = 10;

while ($fuel > 1)
{
    // Keep driving...
    echo "There's enough fuel";
}
```

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



As with `if` statements, curly braces are required to hold the statements inside the `while` statements, unless there's only one.

For another example of a `while` loop that displays the 12 times table, see [Example 4-29](#).

Example 4-29. A while loop to print the 12 times table

```
$count = 1;

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

Here the variable `$count` is initialized to a value of 1, and then a `while` loop starts with the comparative expression `$count <= 12`. This loop will continue executing until the variable is greater than 12. The output from this code is:

```
1 times 12 is 12
2 times 12 is 24
3 times 12 is 36
and so on...
```

Inside the loop, a string is printed along with the value of `$count` multiplied by 12. For neatness, this is followed with a `<br>` tag to force a new line. Then `$count` is incremented, ready for the final curly brace that tells PHP to return to the start of the loop.

At this point, `$count` is again tested to see whether it is greater than 12. It isn't, but it now has the value 2, and after another 11 times around the loop, it will have the value 13. When that happens, the code within the `while` loop is skipped and execution passes to the code following the loop, which, in this case, is the end of the program.

If the `++$count` statement (which equally could have been `$count++`) had not been there, this loop would be like the first one in this section. It would never end, and only the result of `1 * 12` would be printed over and over.

But there is a much neater way to write this loop. Take a look at [Example 4-30](#).

Example 4-30. A shortened version of [Example 4-29](#)

```
$count = 0;

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

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

is entered. If you keep the initialization at 1, only results between 2 and 12 will be output.



Multiline Statements

In the preceding example, no braces are placed around the code following the `while` statement, and therefore only that line will be executed. If you wish to add more commands at this point then ensure curly braces are used to surround them.

do...while Loops

A slight variation of the `while` loop is the `do...while` loop, used when you want a block of code to be executed at least once and made conditional only after that.

Example 4-31 shows a modified version of the code for the 12 times table that uses such a loop.

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

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

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

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

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

```
$count = 1;

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

for Loops

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

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

```

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

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

Breaking Out of a Loop

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

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

```

$fp = fopen("text.txt", 'wb');

for ($j = 0 ; $j < 100 ; ++$j)
{
    $written = fwrite($fp, "data");

    if ($written == FALSE) break;
}

fclose($fp);

```

This is the most complicated piece of code that you have seen so far, but you're ready for it. We'll look into the file-handling commands in [Chapter 7](#), but for now all you need to know is that the first line opens the file `text.txt` for writing in binary mode

and then returns a pointer to the file in the variable `$fp`, which is used later to refer to the open file.

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

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

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

```
break 2;
```

The continue Statement

The `continue` statement is a little like a `break` statement, except that it instructs PHP to stop processing the current iteration of the loop and move right to its next iteration. So, instead of breaking out of the whole loop, PHP exits only the current iteration.

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

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

```
$j = 11;

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

    if ($j == 0) continue;

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

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

Implicit and Explicit Casting

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

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

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

```
$a = 56;  
$b = 12;  
$c = $a / $b;  
  
echo $c;
```

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

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

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

You can explicitly cast variables and literals to the types shown in [Table 4-6](#).

Table 4-6. PHP’s cast types

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



PHP also has built-in functions that do the same thing. For example, to obtain an integer value, you could use the `intval` function. But those functions often do more than just casting; for example, the `intval` function supports specifying the base for the conversion. Usually, simple casting as mentioned previously is enough to change the type.

PHP Modularization

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

But although it is possible to build an entire website this way, it's not recommended, because your source code will grow and grow and start to become unwieldy, as it has to account for every possible action a user could take.

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

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

As long as you have created a way to track your user through your website by means of cookies or session variables (both of which we'll look at more closely in later chapters), you can split up your website into sensible sections of PHP code, each one self-contained, and therefore treat yourself to a much easier future, developing each new feature and maintaining old ones. If you have a team, different people can work on different modules so that each programmer needs to learn just one part of the program thoroughly. In [Chapter 5](#), you'll learn to use functions and objects to create reusable code components; but before that, let's repeat what you've learned in this chapter.

Questions

1. When printing data that contains TRUE and FALSE constants, what's displayed instead of those two constants and why?
2. What are the simplest two forms of expressions?
3. What is the difference between unary, binary, and ternary operators?
4. What is the best way to force your own operator precedence?
5. What is meant by *operator associativity*?
6. When would you use the === (identity) operator?
7. Name the three conditional statement types.
8. What command can you use to skip the current iteration of a loop and move on to the next one?
9. What's the difference between the for loop and the while loop?
10. How do if and while statements interpret conditional expressions of different data types?

See “Chapter 4 Answers” on page 570 in the [Appendix](#) for the answers to these questions.

PHP Functions and Objects

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

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

In PHP functions have many advantages over contiguous, inline code. For example, they:

- Involve less typing
- Reduce syntax and other programming errors
- Decrease the loading time of program files
- Accept arguments and can therefore be used for general as well as specific cases
- Are easier to write tests for

Object-oriented programming takes this concept a step further. A *class* is like a template that allows you to create *objects*, which encapsulate one or more functions and the data they use.

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



It is now highly unusual (and definitely not recommended) to use any version of PHP lower than 5.4. Therefore, this chapter assumes this release is the bare minimum version you will be working with, and I would strongly recommend you stick with a minimum of version 8.2 (the version supplied with AMPPS as described in [Chapter 2](#)). Should you need to use a different version such as the newer 8.x releases, you can install one from the AMPPS control panel, as described in [Chapter 2](#).

PHP Functions

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

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

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

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

```
phpinfo();
```

The result of calling this function can be seen in [Figure 5-1](#).



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

PHP Version 7.4.33	
System	Windows NT ZENBOOK 10.0 build 22631 (Windows 10) AMD64
Build Date	Nov 2 2022 15:57:50
Compiler	Visual C++ 2017
Architecture	x64
Configure Command	cscript /nologo /e:javascript configure.js "--enable-snapshot-build" "--enable-debug-pack" "--with-pdo-oci=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--with-oci8-12c=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--enable-object-out-dir=.\obj\" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	no value
Loaded Configuration File	C:\Program Files\Ampps\php74\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20190902
PHP Extension	20190902
Zend Extension	320190902
Zend Extension Build	API320190902,TS,VC15
PHP Extension Build	API20190902,TS,VC15
Debug Build	no
Thread Safety	enabled
Thread API	Windows Threads
Zend Signal Handling	disabled
Zend Memory Manager	enabled

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

Some of the built-in functions that use one or more arguments appear in [Example 5-1](#).

Example 5-1. Three string functions

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

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

Hello world. Hip Hip HOORAY!

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

Defining a Function

The general syntax for a function is:

```
function function_name(parameter, parameter 2...)
{
    // Statements
}
```

The first line of the syntax indicates:

- A definition starts with the word `function`.
- A name follows, which must start with a letter or underscore, followed by any number of letters, numbers, or underscores; the same requirements are used for variable naming.
- The parentheses are required.
- One or more parameters, separated by commas, are optional.

Function names are case-insensitive, so all of the following strings can refer to the `print` function: `PRINT`, `Print`, and `PrInt`, but whichever style you or your supervisor sets up should be adhered to for consistency.

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

Returning a Value

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

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

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

The output of this experiment is:

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

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


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

The output is:

Any # of letters and punctuation you want

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

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

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

The expression `5-8` is evaluated first, and the output is `-3`. (As you saw in [Chapter 4](#), PHP converts the result to a string in order to display it.) If the expression contains a function, that function is evaluated as well:

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

PHP is doing several things in executing that short statement:

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

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

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

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

Any # of letters and punctuation you want

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

Example 5-2. Cleaning up a full name

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

```

    return $n1 . ' ' . $n2 . ' ' . $n3;
}

echo fix_names('WILLIAM', 'henry', 'gatES');
?>

```

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

William Henry Gates

Returning an Array

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

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

Example 5-3. Returning multiple values in an array

```

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

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

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

```

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

Returning Global Variables

Another way, although not recommended, to give a function access to an externally created variable that is not passed as an argument is by declaring it to have global access from within the function. The `global` keyword followed by the variable name gives every part of your code full access to it (see [Example 5-4](#)).

Example 5-4. Returning values in global variables

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

function fix_names()
{
    global $a1; $a1 = ucfirst(strtolower($a1));
    global $a2; $a2 = ucfirst(strtolower($a2));
    global $a3; $a3 = ucfirst(strtolower($a3));
}

echo $a1 . ' ' . $a2 . ' ' . $a3 . '<br>';
fix_names();
echo $a1 . ' ' . $a2 . ' ' . $a3;
?>
```

Now you don't have to pass parameters to the function, and it doesn't have to accept them. Once declared, these variables retain global access and are available to the rest of your program, including its functions. You may spot this approach in legacy code but is strongly not recommended for any new development as it introduces a hidden side effect to the `fix_names` function that is not clear. It also makes testing the function much harder and safe renaming of the variables almost impossible.

Recap of Variable Scope

A quick reminder of what you know from [Chapter 3](#):

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

Including and Requiring Files

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

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

The include Statement

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

Example 5-5. Including a PHP file

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

// Your code goes here
?>
```

Using include_once

Each time you issue the `include` directive, it includes the requested file again, even if you've already inserted it. For instance, suppose that *library.php* contains a lot of useful functions, so you include it in your file, but you also include another library that includes *library.php*. Through nesting, you've inadvertently included *library.php* twice. This will produce error messages, because you're trying to define the same constant or function multiple times. So, you should use `include_once` instead (see [Example 5-6](#)).

Example 5-6. Including a PHP file only once

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

// Your code goes here
?>
```

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



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

Using require and require_once

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

When it is absolutely essential to include a file, `require` it. For the same reasons I gave for using `include_once`, I recommend that you stick with `require_once` whenever you need to `require` a file (see [Example 5-7](#)).

Example 5-7. Requiring a PHP file only once

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

// Your code goes here
?>
```

PHP Version Compatibility

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

[Example 5-8](#) checks for `array_combine`, a function specific to only some versions of PHP.

Example 5-8. Checking for a function's existence

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

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

PHP Objects

In much the same way that functions represent a huge increase in programming power over the early days of computing, *object-oriented programming* (OOP) takes the use of functions in a different direction.

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

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

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

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

Terminology

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

The data associated with an object is called its *properties*; the functions it uses are called *methods*. In defining a class, you supply the names of its properties and the code for its methods. See [Figure 5-2](#) for a jukebox metaphor for an object. Think of the CDs that it holds in the carousel as its properties; the method of playing them is to press buttons on the front panel. There is also a slot for inserting coins (the method used to activate the object) and a laser disc reader (the method used to retrieve the music, or properties, from the CDs), or software to download and play an online file.

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



Figure 5-2. A jukebox: a great example of a self-contained object

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

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

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

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

Declaring a Class

Before you can use an object, you must define a class with the `class` keyword. Class definitions contain the class name (which is case-insensitive), its properties, and its methods. **Example 5-9** defines the class `User` with two properties, which are `$name` and `$password` (indicated by the `public` keyword—see “**Property and Method Scope**” on page 112). It also creates a new instance (called `$object`) of this class.

Example 5-9. Declaring a class and examining an object

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

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

$object = new User;
print_r($object);
?>
```

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

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

However, a browser compresses all the whitespace, so the output in a browser is slightly harder to read (although you can always display output within `<pre>` and `</pre>` tags to display all the whitespace):

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

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

Creating an Object

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

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

On the first line, we create an instance of the `User` class and assign it to a variable called `$object`. In the second line, we provide arguments to the class *constructor*, a special method explained later in the chapter, when we create the instance of the `User` class and assign the instance to the variable `$temp`.

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

Accessing Objects

Let's add a few lines to [Example 5-9](#) and check the results. [Example 5-10](#) extends the previous code by setting object properties and calling a method.

Example 5-10. Creating and interacting with an object

```
<?php  
$object = new User;  
print_r($object); echo "<br>";  
  
$object->name      = "Joe";  
$object->password  = "mypass";  
print_r($object); echo "<br>";  
  
$object->save_user();  
  
class User  
{  
    public $name, $password;  
  
    function save_user()  
    {  
        echo "Save User code goes here";  
    }  
}  
?>
```

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

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

When looked at using a browser's View Source facility, the output from [Example 5-10](#) is:

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

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

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



You can place functions and class definitions anywhere in your code, before or after statements that use them. Generally, though, it is considered good practice to place them in their own files, or in shorter pieces of code where extra files are not required, toward the end of a file.

Cloning Objects

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

In other words, making object assignments does not copy objects in their entirety; only a new reference to an existing object is created. You'll see how this works in [Example 5-11](#), where we define a very simple `User` class with no methods and only the property name.

Example 5-11. Copying an object

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

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

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

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

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

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

To avoid this confusion, you can use the `clone` operator, which creates a new instance of the class and copies the property values from the original instance to the new instance. [Example 5-12](#) illustrates this usage.

Example 5-12. Cloning an object

```
<?php
class User
{
    public $name;
}

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

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

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

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

Constructors

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

To do this you use the function name `__construct` (that is, `construct` preceded by two underscore characters), as in [Example 5-13](#). The constructor in the example takes two arguments `$name` and `$password` and initializes two properties `name` and `password`. A special variable `$this` is used to set the current object's properties: the `name` property declared as `public` `$name` is accessed as `$this->name` in the class methods including the constructor.

Example 5-13. Creating a constructor method

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

    function __construct($name, $password)
    {
        $this->name = $name;
        $this->password = $password;
    }
}
?>
```

Destructors

You also have the ability to create *destructor* methods, useful for when code has made the last reference to an object or when a script reaches the end. [Example 5-14](#) shows how to create a destructor method. The destructor can do clean-up such as releasing a connection to a database or some other resource that you reserved within the object. Because you reserved the resource within the object, you have to release it here, or it will stick around indefinitely. Many system-wide problems are caused by programs reserving resources and forgetting to release them.

Example 5-14. Creating a destructor method

```
<?php
class User
{
```

```
function __destruct()
{
    // Destructor code goes here
}
?>
```

Writing Methods

As you have seen, declaring a method is similar to declaring a function, but there are a few differences. For example, method names beginning with a double underscore (__) are reserved (for example, __construct and __destruct).

You also have access to a special variable called \$this, which can be used to access the current object's properties. To see how it works, see [Example 5-15](#), which contains a different method from the User class definition called get_password.

Example 5-15. Using the variable \$this in a method

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

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

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

Here's how you would use the class defined in [Example 5-15](#):

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

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

This code prints the password secret.

Declaring Properties

It is not necessary to explicitly declare properties within classes, as they can be implicitly defined when first used, but this technique has been deprecated since PHP 8.2. To illustrate this, in [Example 5-16](#) the class `User` has no properties and no methods but is legal code.

Example 5-16. Defining a property implicitly

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

echo $object1->name;

class User {}
?>
```

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

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

Static Methods

You can define a method as `static`, which means that it is called on a class, not on an object. A static method has no access to any object properties and is created and accessed as in [Example 5-17](#).

Example 5-17. Creating and accessing a static method

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

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

Note how we call the class itself, along with the static method, using a double colon (also known as the *scope resolution* operator), not `->`. Static functions are useful for performing actions relating to the class itself but not to specific instances of the class. You can see another example of a static method in [Example 5-18](#).



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

Declaring Constants

In the same way that you can create a global constant with the `define` function, you can define constants inside classes. The generally accepted practice is to use uppercase letters to make them stand out, as in [Example 5-18](#).

Example 5-18. Defining constants within a class

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

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

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

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

Outside of the class, you can access the constant directly by the class name:

```
print_r(Translate::GERMAN);
```

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

Property and Method Scope

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

public

Public members can be referenced anywhere, including by other classes and instances of the object. This is the default when variables are declared with the `var` or `public` keywords, or when a variable is implicitly declared the first time it is used.

The keywords `var` and `public` are interchangeable because, although deprecated, `var` is retained for compatibility with previous versions of PHP. Methods are assumed to be `public` by default.

protected

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

private

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

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

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

Example 5-19 illustrates the use of these keywords.

Example 5-19. Changing property and method scope

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

    private function admin() // Private method
    {
        // Admin code goes here
    }
}
```



```

    }
}
?>

```

Static Properties

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

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

As shown briefly in [Example 5-17](#), declaring members of a class `static` makes them accessible without an instantiation of the class. A property declared `static` cannot be directly accessed within an instance of a class, but a static method can.

[Example 5-20](#) defines a class called `Test` with a static property and a public method.

Example 5-20. Defining a class with a static property

```

<?php
$temp = new Test();

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

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

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

```

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

```

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

```

This example shows that the property `$static_property` could be directly referenced from the class itself via the double colon operator in Test A. Also, Test B could obtain

its value by calling the `get_sp` method of the object `$temp`, created from class `Test`. But Test C failed, because the static property `$static_property` was not accessible to the object `$temp`.

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

Inheritance

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



Use Inheritance with Caution

Inheritance should be approached with caution and used sparingly. If overused, it can make testing, refactoring, and reasoning more difficult. A common error is to create a class (for example a `Mailer` class) that extends the `Database` class because the `Mailer` class needs the `Database` class. A better approach is *class composition*, where the `Mailer` class uses, but does not extend, multiple other classes, like the `Database` class, the `Email` class, and the `User` class. Objects created from the other classes can be passed to the `Mailer` object as constructor arguments or can be created in the constructor. Consider inheritance a more advanced design pattern with some downsides to factor in when deciding whether to use it.

In [Example 5-21](#), the class `Subscriber` is declared a subclass of `User` by means of the `extends` keyword.

Example 5-21. Inheriting and extending a class

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

class User
{
    public $name, $password;

    function save_user()
```

```

    {
        echo "Save User code goes here";
    }
}

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

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

```

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

```

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

```

The parent keyword

If you write a method in a subclass with the same name as one in its parent class, its statements will override those of the parent class. Sometimes this is not the behavior you want, and you need to access the parent's method. To do this, you can use the parent operator, as in [Example 5-22](#).

Example 5-22. Overriding a method and using the parent operator

```

<?php
$object = new Son;
$object->test();
$object->test2();

class Dad
{
    function test()
    {
        echo "[Class Dad] I am your Father<br>";
    }
}

```

```

}

class Son extends Dad
{
    function test()
    {
        echo "[Class Son] I am Luke<br>";
    }

    function test2()
    {
        parent::test();
    }
}
?>

```

This code creates a class called Dad and a subclass called Son that inherits its properties and methods and then overrides the method `test`. Therefore, when line 2 calls the method `test`, the new method is executed. The only way to execute the overridden `test` method in the Dad class is to use the parent operator, as shown in function `test2` of class Son. The code outputs this:

```

[Class Son] I am Luke
[Class Dad] I am your Father

```

If you wish to ensure that your code calls a method from the current class, you can use the `self` keyword, like this:

```

self::method();

```

Using `self` to call static methods is very common, but it is rarely used to call the object ones. The difference between using `self` and `$this` to call an object method is that if the method would be overridden in a subclass and you'd call it using the `self` keyword in the parent class, then the method from the parent class would be called, not the overridden one, unlike when you'd use `$this` to call it.

Subclass constructors

When you extend a class and declare your own constructor, you should be aware that PHP will not automatically call the constructor method of the parent class. If you want to be certain that all initialization code is executed, subclasses should always call the parent constructors, as in [Example 5-23](#).

Example 5-23. Calling the parent class constructor

```

<?php
$object = new Tiger();

echo "Tigers have...<br>";

```

```

echo "Fur: " . $object->fur . "<br>";
echo "Stripes: " . $object->stripes;

class Wildcat
{
    public $fur; // Wildcats have fur

    function __construct()
    {
        $this->fur = "TRUE";
    }
}

class Tiger extends Wildcat
{
    public $stripes; // Tigers have stripes

    function __construct()
    {
        parent::__construct(); // Call parent constructor first
        $this->stripes = "TRUE";
    }
}
?>

```

This example takes advantage of inheritance in the typical manner. The `Wildcat` class has created the property `$fur`, which we'd like to reuse, so we create the `Tiger` class to inherit `$fur` and additionally create another property, `$stripes`. To verify that both constructors have been called, the program outputs:

```

Tigers have...
Fur: TRUE
Stripes: TRUE

```

Final methods

When you wish to prevent a subclass from overriding a superclass method, you can use the `final` keyword. [Example 5-24](#) shows how.

Example 5-24. Creating a final method

```

<?php
class User
{
    final function copyright()
    {
        echo "This class was written by Joe Smith";
    }
}
?>

```

If you tried to override the `copyright` method in a subclass of the `User` class, you'd get an error message saying you cannot override the `final` method.

Private methods, except for the constructor, cannot use the `final` keyword. It would make little sense as they are never overridden by other classes.

Once you have digested the contents of this chapter, you should have a strong feel for what PHP can do for you. You should be able to use functions with ease and, if you wish, write object-oriented code. In [Chapter 6](#), we'll complete our initial exploration of PHP by looking at the workings of PHP arrays, but first test your understanding of this chapter using the following questions.

Questions

1. What is the main benefit of using a function?
2. How many values can a function return?
3. What is the difference between accessing a variable by name and by reference?
4. What is the meaning of *scope* in PHP?
5. How can you incorporate one PHP file within another?
6. How is an object different from a function?
7. How do you create a new object in PHP?
8. What syntax would you use to create a subclass from an existing one?
9. How can you cause an object to be initialized when you create it?
10. Why is it a good idea to explicitly declare properties within a class?

See [“Chapter 5 Answers” on page 571](#) in the [Appendix](#) for the answers to these questions.

PHP Arrays

In [Chapter 3](#), I gave a very brief introduction to PHP's arrays, just enough for a little taste of their power. In this chapter, I'll show you many more things you can do with arrays, some of which—if you have ever used a strongly typed language such as C—may surprise you with their elegance and simplicity.

Not only do arrays remove the tedium of writing code to deal with complicated data structures, but they also provide numerous ways to access data while remaining amazingly fast.

Basic Access

We've already looked at arrays as if they were clusters of matchboxes glued together. Another way to think of an array is like a string of beads, with the beads representing variables that can be numbers, strings, or even other arrays. They are like bead strings because each element has its own location and (with the exception of the first and last ones) each has other elements on either side.

Some arrays are referenced by numeric indexes; others allow alphanumeric identifiers. Built-in functions let you sort them, add or remove sections, and walk through them to handle each item through a special kind of loop. By placing one or more arrays inside another, you can create arrays of two, three, or any number of dimensions.

Numerically Indexed Arrays

Let's assume that you've been tasked with creating a simple website for a local office supply company and you're currently working on the section devoted to paper. One way to manage the various items of stock in this category would be to place them in a numeric array. You can see the simplest way of doing so in [Example 6-1](#).

Example 6-1. Adding items to an array

```
<?php
$paper[] = "Copier";
$paper[] = "Inkjet";
$paper[] = "Laser";
$paper[] = "Photo";

print_r($paper);
?>
```

In this example, each time you assign a value to the array `$paper`, the first empty location within that array is used to store the value, and a pointer internal to PHP is incremented to point to the next free location, ready for future insertions. The familiar `print_r` function (which prints out the contents of a variable, array, or object) is used to verify that the array has been correctly populated. It prints out the following:

```
Array
(
    [0] => Copier
    [1] => Inkjet
    [2] => Laser
    [3] => Photo
)
```

The previous code also could have been written as shown in [Example 6-2](#), where the exact location of each item within the array is specified. But, as you can see, that approach requires extra typing and makes your code harder to maintain if you want to insert supplies into or remove them from the array. So, unless you wish to specify a different order, it's better to let PHP handle the actual location numbers.

Example 6-2. Adding items to an array using explicit locations

```
<?php
$paper[0] = "Copier";
$paper[1] = "Inkjet";
$paper[2] = "Laser";
$paper[3] = "Photo";

print_r($paper);
?>
```

The output from these examples is identical, but you are not likely to use `print_r` in a developed website, so [Example 6-3](#) shows how you might print out the various types of paper the website offers using a `for` loop.

Example 6-3. Adding items to an array and retrieving them

```
<?php
    $paper[] = "Copier";
    $paper[] = "Inkjet";
    $paper[] = "Laser";
    $paper[] = "Photo";

    for ($j = 0 ; $j < 4 ; ++$j)
        echo "$j: {$paper[$j]}<br>";
?>
```

This example prints out:

```
0: Copier
1: Inkjet
2: Laser
3: Photo
```

So far, you've seen a couple of ways you can add items to an array and one way of referencing them. PHP offers many more, which I'll get to shortly. But first, we'll look at another type of array.

Associative Arrays

Keeping track of array elements by numeric index works just fine but can require extra work in terms of remembering which number refers to which product. It can also make code hard for other programmers to follow.

This is where associative arrays come in. Using them, you can reference the items in an array by name rather than by number. [Example 6-4](#) expands on the previous code by giving each element in the array an identifying name and a longer, more explanatory string value.

Example 6-4. Adding items to an associative array and retrieving them

```
<?php
    $paper['copier'] = "Copier & Multipurpose";
    $paper['inkjet'] = "Inkjet Printer";
    $paper['laser'] = "Laser Printer";
    $paper['photo'] = "Photographic Paper";

    echo $paper['laser'];
?>
```

In place of a number (which doesn't convey any useful information, aside from the position of the item in the array), each item now has a unique name that you can use to reference it elsewhere, as with the echo statement, which simply prints out Laser

Printer. The names (copier, inkjet, and so on) are called *indexes* or *keys*, and the items assigned to them (such as Laser Printer) are called *values*.

This very powerful PHP feature is often used when you are extracting information from XML and HTML. For example, an HTML parser such as those used by a search engine could place all the elements of a web page into an associative array whose names reflect the page's structure:

```
$html['title'] = "My web page";  
$html['body'] = "... body of web page ...";
```

The program would also break down all the links found within a page into another array, and all the headings and subheadings into another. When you use associative rather than numeric arrays, the code to refer to all of these items is easy to write and debug.



PHP's associative arrays are similar to maps, dictionaries, or objects in other languages.

Assignment Using the array Keyword

So far, you've seen how to assign values to arrays by adding new items one at a time. Whether you specify keys, specify numeric identifiers, or let PHP assign numeric identifiers implicitly, this is a long-winded approach. A more compact and faster assignment method uses the `array` keyword. **Example 6-5** shows both a numeric and an associative array assigned using this method.

Example 6-5. Adding items to an array using the array keyword

```
<?php  
$p1 = array("Copier", "Inkjet", "Laser", "Photo");  
  
echo "p1 element: " . $p1[2] . "<br>";  
  
$p2 = array('copier' => "Copier & Multipurpose",  
            'inkjet' => "Inkjet Printer",  
            'laser' => "Laser Printer",  
            'photo' => "Photographic Paper");  
  
echo "p2 element: " . $p2['inkjet'] . "<br>";  
?>
```

The first half of this snippet assigns the old, shortened product descriptions to the array `$p1`. There are four items, so they will occupy slots 0 through 3. Therefore, the `echo` statement prints out:

```
p1 element: Laser
```

The second half assigns associative identifiers and accompanying longer product descriptions to the array `$p2` using the format *key => value*. The use of `=>` is similar to the regular `=` assignment operator, except that you are assigning a value to an *index* and not to a *variable*. The index is then linked with that value, unless it is assigned a new value. The `echo` command therefore prints out:

```
p2 element: Inkjet Printer
```

You can verify that `$p1` and `$p2` are different types of array, because both of the following commands, when appended to the code, will cause an `Undefined index` or `Undefined offset` error, as the array identifier for each is incorrect:

```
echo $p1['inkjet']; // Undefined index
echo $p2[3];        // Undefined offset
```

The foreach...as Loop

The creators of PHP have gone to great lengths to make the language easy to use. So, not content with the loop structures already provided, they added another one especially for arrays: the `foreach...as` loop. Using it, you can step through all the items in an array, one at a time, and do something with them.

The process starts with the first item and ends with the last one, so you don't even have to know how many items are in an array. [Example 6-6](#) shows how `foreach...as` can be used to rewrite [Example 6-3](#).

Example 6-6. Walking through a numeric array using `foreach...as`

```
<?php
$paper = array("Copier", "Inkjet", "Laser", "Photo");
$j = 0;

foreach($paper as $item)
{
    echo "$j: $item<br>";
    ++$j;
}
?>
```

When PHP encounters a `foreach` statement, it takes the first item of the array and places it in the variable following the `as` keyword; each time control flow returns to the `foreach`, the next array element is placed in the `as` keyword. In this case, the variable `$item` is set to each of the four values in turn in the array `$paper`. Once all values have been used, execution of the loop ends. The output from this code is exactly the same as in [Example 6-3](#).

Now let's see how `foreach` works with an associative array in [Example 6-7](#), which is a rewrite of the second half of [Example 6-5](#).

Example 6-7. Walking through an associative array using `foreach...as`

```
<?php
    $paper = array('copier' => "Copier & Multipurpose",
                  'inkjet'  => "Inkjet Printer",
                  'laser'   => "Laser Printer",
                  'photo'   => "Photographic Paper");

    foreach($paper as $item => $description)
        echo "$item: $description<br>";
?>
```

Remember that associative arrays do not require numeric indexes, so the variable `$j` is not used in this example. Instead, each item of the array `$paper` is fed into the key/value pair of variables `$item` and `$description`, from which they are printed out. The displayed result of this code is:

```
copier: Copier & Multipurpose
inkjet: Inkjet Printer
laser: Laser Printer
photo: Photographic Paper
```

There are some alternative ways to walk through an associative array. I once used the `list` and `each` functions, but each has since been removed from PHP. Luckily, it is possible to write a replacement function, which I have named `myEach`, to be used with the `list` function in conjunction with a `while` loop, as in [Example 6-8](#).

Example 6-8. Walking through an associative array using `myEach` and `list`

```
<?php
    $paper = array('copier' => "Copier & Multipurpose",
                  'inkjet'  => "Inkjet Printer",
                  'laser'   => "Laser Printer",
                  'photo'   => "Photographic Paper");

    while (list($item, $description) = myEach($paper))
        echo "$item: $description<br>";
```

```

function myEach(&$array) // Replacement for deprecated 'each' function
{
    $key    = key($array);
    $result = ($key === null) ? false :
                [$key, current($array), 'key', 'value' =>
                    current($array)];
    next($array);
    return $result;
}
?>

```

In this example, a while loop is set up and will continue looping until the `myEach` function returns a value of `FALSE`. The `myEach` function acts like `foreach` in that it returns an array containing a key/value pair from the array `$paper` and then moves its built-in pointer to the next pair in that array. When there are no more pairs to return, `myEach` returns `FALSE`.

Unlike with `foreach`, calling `myEach` modifies the internal array pointer because it uses `next`, which is something you should be aware of. You can spot the difference when you add `print_r(current($paper));` after both the `foreach` and `while` loops. Using `foreach` to walk through arrays is more common as it doesn't have this side effect and is also marginally faster.

The `list` function takes an array as its argument (in this case, the key/value pair returned by the function `myEach`) and then assigns the values of the array to the variables listed within parentheses.

You can see how `list` works a little more clearly in [Example 6-9](#), where an array is created out of the two strings `Alice` and `Bob` and then passed to the `list` function, which assigns those strings as values to the variables `$a` and `$b`.

Example 6-9. Using the `list` function

```

<?php
list($a, $b) = array('Alice', 'Bob');
echo "a=$a b=$b";
?>

```

The output from this code is:

```
a=Alice b=Bob
```

Multidimensional Arrays

A simple design feature in PHP's array syntax makes it possible to create arrays of more than one dimension. In fact, they can be as many dimensions as you like (although it's a rare application that goes beyond three).

That feature is the ability to include an entire array as a part of another one and to be able to keep doing so, just like the old rhyme by Augustus De Morgan, the British mathematician and logician: “Big fleas have little fleas upon their backs to bite ’em. Little fleas have lesser fleas and so ad infinitum.”

Let’s look at how this works by extending the associative array in the previous example; see [Example 6-10](#).

Example 6-10. Creating a multidimensional associative array

```
<?php
$products = array(

    'paper' => array(

        'copier' => "Copier & Multipurpose",
        'inkjet' => "Inkjet Printer",
        'laser'  => "Laser Printer",
        'photo'  => "Photographic Paper"),

    'pens' => array(

        'ball'   => "Ball Point",
        'hilite' => "Highlighters",
        'marker' => "Markers"),

    'misc' => array(

        'tape'   => "Sticky Tape",
        'glue'   => "Adhesives",
        'clips'  => "Paperclips"
    )
);

echo "<pre>";

foreach($products as $section => $items)
    foreach($items as $key => $value)
        echo "$section:\t$key\t($value)<br>";

echo "</pre>";
?>
```

To make things clearer now that the code is starting to grow, I’ve renamed some of the elements. For example, because the previous array `$paper` is now just a subsection of a larger array, the main array is now called `$products`. Within this array, there are three items—paper, pens, and misc—each of which contains another array with key/value pairs.

If necessary, these subarrays could contain even further arrays. For example, under `ball` there might be many different types and colors of ballpoint pens available in the online store. But for now, I've restricted the code to a depth of just two.

Once the array data has been assigned, I use a pair of nested `foreach...as` loops to print out the various values. The outer loop extracts the main sections from the top level of the array, and the inner loop extracts the key/value pairs for the categories within each section.

As long as you remember that each level of the array works the same way (it's a key/value pair), you can easily write code to access any element at any level.

The echo statement uses the PHP escape character `\t`, which outputs a tab. Although tabs are not normally significant to the web browser, I let them be used for layout through the `<pre>...</pre>` tags, which tell the web browser to format the text as preformatted and monospaced, and *not* to ignore whitespace characters such as tabs and line feeds. The output from this code looks like this:

paper: copier (Copier & Multipurpose)
 paper: inkjet (Inkjet Printer)
 paper: laser (Laser Printer)
 paper: photo (Photographic Paper)
 pens: ball (Ball Point)
 pens: hilite (Highlighters)
 pens: marker (Markers)
 misc: tape (Sticky Tape)
 misc: glue (Adhesives)
 misc: clips (Paperclips)

You can directly access a particular element of the array by using square brackets:

```
echo $products['misc']['glue'];
```

This outputs the value `Adhesives`.

You can also create numeric multidimensional arrays that are accessed directly by indexes rather than by alphanumeric identifiers. **Example 6-11** creates the board for a chess game with the pieces in their starting positions.

Example 6-11. Creating a multidimensional numeric array

```
<?php
$schessboard = array(
    array('r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'),
    array('p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'),
    array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    array('p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'),
    array('r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'),
);
```

```

    array('R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R')
);

echo "<pre>";

foreach($chessboard as $row)
{
    foreach ($row as $piece)
        echo "$piece ";

    echo "<br>";
}

echo "</pre>";
?>

```

In this example, the lowercase letters represent black pieces, and the uppercase white. The key is `r` = rook, `n` = knight, `b` = bishop, `k` = king, `q` = queen, and `p` = pawn. Again, a pair of nested `foreach...as` loops walks through the array and displays its contents. The outer loop processes each row into the variable `$row`, which itself is an array, because the `$chessboard` array uses a subarray for each row. This loop has two statements within it, so curly braces enclose them.

The inner loop then processes each square in a row, outputting the character (`$piece`) stored in it, followed by a space (to square up the printout). This loop has a single statement, so curly braces are not required to enclose it. The `<pre>` and `</pre>` tags ensure that the output displays correctly, like this:

```

r n b q k b n r
p p p p p p p

```

```

P P P P P P P
R N B Q K B N R

```

You can also directly access any element within this array by using square brackets:

```
echo $chessboard[7][3];
```

This statement outputs the uppercase letter `Q`, the eighth element down and the fourth along (remember that array indexes start at 0, not 1).

Using Array Functions

You've already seen the `list` and `each` functions, but PHP comes with numerous other functions for handling arrays. You can find the full list in the [documentation](#). However, some of these functions are so fundamental that it's worth taking the time to discuss them here.

`is_array`

Arrays and variables share the same namespace (as arrays are types of variable). This means you cannot have a string variable called `$fred` and an array also called `$fred`. If you're in doubt and your code needs to check whether a variable is an array, you can use the `is_array` function, like this:

```
echo is_array($fred) ? "Is an array" : "Is not an array";
```

Note that if `$fred` has not yet been assigned a value, an `Undefined variable` message will be generated.

`count`

Although the `each` function and `foreach...as` loop structure are excellent ways to walk through an array's contents, sometimes you need to know exactly how many elements are in your array, particularly if you will be referencing them directly. To count all the elements in the top level of an array, use a command such as:

```
echo count($fred);
```

Should you wish to know how many elements altogether are in a multidimensional array, you can use a statement such as:

```
echo count($fred, 1);
```

The second parameter is optional and sets the mode to use. It should be either `0` to limit counting to only the top level or `1` to force recursive counting of all subarray elements.

`sort`

Sorting is so common that PHP provides a built-in function for it. In its simplest form, you would use it like this (to sort items normally: the default):

```
sort($fred);
```

It is important to remember that, unlike some other functions, `sort` will act directly on the supplied array rather than returning a new array of sorted elements. It returns `TRUE` on success and `FALSE` on error and also supports a few flags—the main two flags

you might wish to use force items to be sorted either numerically or as strings, like this:

```
sort($fred, SORT_NUMERIC);
sort($fred, SORT_STRING);
```

You can also sort an array in reverse order using the `rsort` function, like this:

```
rsort($fred, SORT_NUMERIC);
rsort($fred, SORT_STRING);
```

shuffle

There may be times when you need the elements of an array to be put in random order, such as when you're creating a game of playing cards:

```
shuffle($cards);
```

Like `sort`, `shuffle` acts directly on the supplied array and returns `TRUE` on success or `FALSE` on error.

explode

`explode` is a very useful function; you can take a string containing several items separated by a single character (or string of characters) and then place each of these items into an array. One handy example is to split up a sentence into an array containing all its words, as in [Example 6-12](#).

Example 6-12. Exploding a string into an array using spaces

```
<?php
$temp = explode(' ', "This is a sentence with seven words");
print_r($temp);
?>
```

This example prints out the following (on a single line when viewed in a browser):

```
Array
(
    [0] => This
    [1] => is
    [2] => a
    [3] => sentence
    [4] => with
    [5] => seven
    [6] => words
)
```

The first parameter, the delimiter, need not be a space or even a single character. [Example 6-13](#) shows a slight variation.

*Example 6-13. Exploding a string delimited with *** into an array*

```
<?php
$temp = explode('***', "A***sentence***with***asterisks");
print_r($temp);
?>
```

The code in [Example 6-13](#) prints this:

```
Array
(
    [0] => A
    [1] => sentence
    [2] => with
    [3] => asterisks
)
```

compact

At times you may want to use `compact`, the inverse of `extract`, to create an array from variables and their values. [Example 6-14](#) shows how to use this function by passing variable names without the preceding `$` characters.

Example 6-14. Using the `compact` function

```
<?php
$name      = "Doctor";
$sname     = "Who";
$planet    = "Gallifrey";
$system    = "Gridlock";
$constellation = "Kasterborous";

$contact = compact('fname', 'sname', 'planet',
                  'system', 'constellation');

print_r($contact);
?>
```

The result of running [Example 6-14](#) is:

```
Array
(
    [fname] => Doctor
    [sname] => Who
    [planet] => Gallifrey
    [system] => Gridlock
    [constellation] => Kasterborous
)
```

Note how `compact` requires the variable names to be supplied in quotes, not preceded by a `$` symbol. This is because `compact` is looking for a list of variable names, not their values.

Another use of this function is for debugging, when you wish to quickly view several variables and their values, as in [Example 6-15](#).

Example 6-15. Using `compact` to help with debugging

```
<?php
$j      = 23;
$temp   = "Hello";
$address = "1 Old Street";
$age    = 61;

print_r(compact(explode(' ', 'j temp address age')));
?>
```

This works by using the `explode` function to extract all the words from the string into an array, which is then passed to the `compact` function, which in turn returns an array to `print_r`, which finally shows its contents.

If you copy and paste the `print_r` line of code, you only need to alter the variables named there for a quick printout of a group of variables' values. In this example, the output is:

```
Array
(
    [j] => 23
    [temp] => Hello
    [address] => 1 Old Street
    [age] => 61
)
```

reset

When calling the next function (as seen in the `myEach` function earlier in the chapter), PHP's internal array pointer, which makes a note of which element of the array it should return next, advances one place forward. If your code ever needs to return to the start of an array, you can issue `reset`, which also returns the value of that element. Examples of how to use this function are:

```
reset($fred);           // Throw away return value
$item = reset($fred);    // Keep first element of the array in $item
```

It's important to note that the `foreach...as` construct does not modify the internal array pointer, so if you're using it to walk through an array, you don't need to care about the pointer, or resetting it.

end

As with `reset`, you can move PHP's internal array pointer to the final element in an array using the `end` function, which also returns the value of the element and can be used as in these examples:

```
end($fred);  
$item = end($fred);
```

This chapter concludes your basic introduction to PHP, and you should now be able to write quite complex programs using the skills you have learned. In [Chapter 7](#), we'll look at using PHP for common, practical tasks, but before you go, test your understanding of PHP arrays by answering these questions.

Questions

1. What is the difference between a numeric and an associative array?
2. What is the main benefit of the `array` keyword?
3. What are some alternative ways of walking through an associative array as compared to `foreach`?
4. How can you create a multidimensional array?
5. How can you determine the number of elements in an array?
6. What is the purpose of the `explode` function?
7. How can you set PHP's internal pointer into an array back to the first element of the array?

See [“Chapter 6 Answers” on page 572](#) in the [Appendix](#) for the answers to these questions.

The previous chapters discussed and illustrated the elements of the PHP language. This chapter builds on your new programming skills to teach you how to perform some common but important practical tasks. You will learn the best ways to handle strings in order to achieve clear and concise code that displays in web browsers exactly how you want it to, including advanced date and time management. You'll also learn how to create and modify files, including those uploaded by users.

Using printf

You've already seen the `print` and `echo` functions, which simply output text to the browser. But a much more powerful function, `printf`, controls the format of the output by letting you put special formatting characters in a string. For each formatting character, `printf` expects you to pass an argument that it will display using that format. For instance, the following example uses the `%d` conversion specifier to display the value 3 in decimal:

```
printf("There are %d items in your basket", 3);
```

If you replace the `%d` with `%b`, the value 3 will be displayed in binary (11). [Table 7-1](#) shows the conversion specifiers supported.

Table 7-1. The printf conversion specifiers

Specifier	Conversion action on argument <code>arg</code>	Example (for an <code>arg</code> of 123)
%	Display a % character (no <code>arg</code> required)	%
b	Display <code>arg</code> as a binary integer	1111011
c	Display ASCII character for <code>arg</code>	{
d	Display <code>arg</code> as a signed decimal integer	123
e	Display <code>arg</code> using scientific notation	1.23000e+2

Specifier	Conversion action on argument <code>arg</code>	Example (for an <code>arg</code> of 123)
f	Display <code>arg</code> as floating point	123.000000
o	Display <code>arg</code> as an octal integer	173
s	Display <code>arg</code> as a string	123
u	Display <code>arg</code> as an unsigned decimal	123
x	Display <code>arg</code> in lowercase hexadecimal	7b
X	Display <code>arg</code> in uppercase hexadecimal	7B

If you need a percent sign in the output, just use a double percent sign (%%). The following code will print "The rate is 5 %":

```
printf("The rate is %d %%", 5);
```

You can have as many specifiers as you like in a `printf` function, as long as you pass a matching number of arguments and as long as each specifier is prefaced by a % symbol. Therefore, the following code is valid and will output "My name is Simon. I'm 33 years old, which is 21 in hexadecimal":

```
printf("My name is %. I'm %d years old, which is %X in hexadecimal",
      'Simon', 33, 33);
```

If you leave out any arguments, you will receive a parse error informing you that a right bracket,), was unexpectedly encountered or that there are too few arguments.

A more practical example of `printf` sets colors in HTML using decimal values. For example, suppose you know you want a color that has a triplet value of 65 red, 127 green, and 245 blue but don't want to convert this to hexadecimal yourself. Here's a simple solution:

```
printf("<span style='color:##X%X%X'>Hello</span>", 65, 127, 245);
```

Check the format of the color specification between the apostrophes (') carefully. First comes the pound, or hash, sign (#) expected by the color specification. Then come three %X format specifiers, one for each of your numbers. The resulting output from this command is:

```
<span style='color:#417FF5'>Hello</span>
```

Usually, you'll find it convenient to use variables or expressions as arguments to `printf`. For instance, if you stored values for your colors in the three variables `$r`, `$g`, and `$b`, you could create a darker color with these simple mathematical expressions (as long as `$r`, `$g`, and `$b` are greater than 19):

```
printf("<span style='color:##X%X%X'>Hello</span>", $r-20, $g-20, $b-20);
```


Precision Setting

Not only can you specify a conversion type, but you also can set the precision of the displayed result. For example, amounts of currency are usually displayed with only two digits of precision. However, after a calculation, a value may have a greater precision than this, such as $123.42 / 12$, which results in 10.285. To ensure that such values are displayed with only two digits of precision, you can insert the string ".2" between the % symbol and the conversion specifier:

```
printf("The result is: $.2f", 123.42 / 12);
```

The output from this command is:

The result is \$10.29

But you actually have even more control, because you also can specify whether to pad output with either zeros or spaces by prefacing the specifier with certain values.

Example 7-1 shows four possible combinations.

Example 7-1. Precision setting

```
<?php
echo "<pre>"; // Enables viewing of the spaces

// Pad to 15 spaces
printf("The result is $%15f\n", 123.42 / 12);

// Pad to 15 spaces, fill with zeros
printf("The result is $%015f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision
printf("The result is $%15.2f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision, fill with zeros
printf("The result is $%015.2f\n", 123.42 / 12);

// Pad to 15 spaces, 2 decimal places precision, fill with # symbol
printf("The result is $%#15.2f\n", 123.42 / 12);
?>
```

The output from this example looks like this:

```
The result is $      10.285000
The result is $000000010.285000
The result is $      10.29
The result is $0000000000010.29
The result is $#####10.29
```

The way it works is simple if you go from right to left (see [Table 7-2](#)). Notice that:

- The rightmost character is the conversion specifier: in this case, `f` for floating point.
- Just before the conversion specifier, if there is a period and a number together, then the precision of the output is specified as the value of the number.
- Regardless of whether there's a precision specifier, if there is a number, then that represents the number of characters to which the output should be padded. In the previous example, this is 15 characters. It represents the total minimum width of the output, not the number of pad characters to add. If the output is already equal to or greater than the padding length, then this argument is ignored.
- The leftmost parameter allowed after the `%` symbol is a `0`, which is ignored unless a padding value has been set, in which case the output is padded with zeros instead of spaces. If a pad character other than zero or a space is required, you can use any one of your choice as long as you preface it with a single quotation mark, like this: `'#'`.
- On the left is the `%` symbol, which starts the conversion.

Table 7-2. Conversion specifier components

Start conversion	Pad character	Number of pad characters	Display precision	Conversion specifier	Example
%		15		f	10.285000
%	0	15	.2	f	0000000000010.29
%	'#'	15	.4	f	#####10.2850

String Padding

You can also pad strings to required lengths (as you can with numbers), select different padding characters, and even choose between left and right justification. [Example 7-2](#) shows various examples.

Example 7-2. String padding

```
<?php
echo "<pre>"; // Enables viewing of the spaces

$h = 'Rasmus';

printf("[%s]\n",          $h); // Standard string output
printf("[%12s]\n",        $h); // Right justify with spaces to width 12
printf("[% -12s]\n",      $h); // Left justify with spaces
printf("[%012s]\n",       $h); // Pad with zeros
```

```
printf("[%'#12s]\n\n", $h); // Use the custom padding character '#'

$d = 'Rasmus Lerdorf';      // The original creator of PHP

printf("[%12.8s]\n", $d); // Right justify, cutoff of 8 characters
printf("[%~12.12s]\n", $d); // Left justify, cutoff of 12 characters
printf("[%~ '@12.10s]\n", $d); // Left justify, pad '@', cutoff of 10 chars
?>
```

Note how for purposes of web page layout, I've used the `<pre>` HTML tag to preserve all the spaces and the `\n` newline character after each of the lines to be displayed. The output from this example is:

```
[Rasmus]
[   Rasmus]
[Rasmus  ]
[000000Rasmus]
[#####Rasmus]

[   Rasmus L]
[Rasmus Lerdo]
[Rasmus Ler@@]
```

When you specify a padding value, strings of a length equal to or greater than that value will be ignored and preserved entirely, *unless* a cutoff value is given that shortens the strings back to less than the padding value.

Table 7-3 shows the components available to string conversion specifiers.

Table 7-3. String conversion specifier components

Start conversion	Left/right justify	Padding character	Number of pad characters	Cutoff	Conversion specifier	Example (using "Rasmus")
%					s	[Rasmus]
%	-		10		s	[Rasmus]
%		'#	8	.4	s	[#####Rasm]

Using sprintf

Often, you don't want to output the result of a conversion but need it to use elsewhere in your code. This is where the `sprintf` function (which stands for string print) comes in. With it, you can send the output to another variable rather than to the browser.

You might use it to make a conversion, as in the following example, which returns the hexadecimal string value for the RGB color group 65, 127, 245 in `$hexstring`:

```
$hexstring = sprintf("%X%X%X", 65, 127, 245);
```

Or you can store output in a variable for other use or display:

```
$out = sprintf("The result is: %.2f", 123.42 / 12);  
echo $out;
```

Date and Time Functions

To keep track of the date and time, PHP uses standard Unix timestamps, which are simply the number of seconds since the start of January 1, 1970 (sometimes referred to as the *Unix epoch*). To determine the current timestamp, you can use the `time` function:

```
echo time();
```

Because the value is stored as seconds, to obtain the timestamp for this time next week, you would use the following, which adds the result of 7 days $\times$ 24 hours $\times$ 60 minutes $\times$ 60 seconds to the returned value:

```
echo time() + 7 * 24 * 60 * 60;
```

If you wish to create a timestamp for a given date, you can use the `mktime` function. Its output is the timestamp 1827619200 for the first second of the first minute of the first hour of the first day of December in 2027:

```
echo mktime(0, 0, 0, 12, 1, 2027);
```

The parameters to pass are, in order from left to right:

- The number of the hour (0–23)
- The number of the minute (0–59)
- The number of seconds (0–59)
- The number of the month (1–12)
- The number of the day (1–31)
- The year (1970–2038, or 1901–2038 with PHP 5.1.0+ on 32-bit signed systems)

The Y2K38 Bug

You may ask why you are limited to 1970 through 2038. Well, it's because the original developers of Unix chose the start of 1970 as the earliest date that any programmer would ever need to reference!

Luckily, as of version 5.1.0, PHP supports systems using a signed 32-bit integer for the timestamp, allowing dates from 1901 to 2038. However, that introduces a problem even worse than the original one, because the Unix designers also decided that nobody would still be using Unix after about 70 years or so and therefore believed they could get away with storing the timestamp as a 32-bit value—which will run out on January 19, 2038!

This will create what has come to be known as the Y2K38 bug (much like the millennium bug, which was caused by storing years as two-digit values and also had to be fixed). PHP introduced the `DateTime` class in version 5.2 to overcome this issue, but it will work only on 64-bit architecture, which most computers will be these days (but do check before you use it).

To display the date, use the `date` function, which supports a plethora of formatting options enabling you to display the date any way you wish. The format is:

```
date($format, $timestamp);
```

The parameter `$format` should be a string containing formatting specifiers as detailed in [Table 7-4](#), and `$timestamp` should be a Unix timestamp. For the complete list of specifiers, please see the [documentation](#).

The following command will output the current date and time in the format "Monday February 17th, 2027 - 1:38pm":

```
echo date("l F jS, Y - g:ia", time());
```

Table 7-4. The major date function format specifiers

Format	Description	Returned value
Day specifiers		
d	Day of month, two digits, with leading zeros	01 to 31
D	Day of the week, three letters	Mon to Sun
j	Day of month, no leading zeros	1 to 31
l	Day of week, full names	Sunday to Saturday
N	Day of week, numeric, Monday to Sunday	1 to 7
S	Suffix for day of month (useful with specifier j)	st, nd, rd, or th
w	Day of week, numeric, Sunday to Saturday	0 to 6
z	Day of year	0 to 365
Week specifier		
W	Week number of year	01 to 52
Month specifiers		
F	Month name	January to December
m	Month number with leading zeros	01 to 12
M	Month name, three letters	Jan to Dec
n	Month number, no leading zeros	1 to 12
t	Number of days in given month	28 to 31
Year specifiers		
L	Leap year	1 = Yes, 0 = No
y	Year, 2 digits	00 to 99
Y	Year, 4 digits	0000 to 9999

Format	Description	Returned value
Time specifiers		
a	Before or after midday, lowercase	am or pm
A	Before or after midday, uppercase	AM or PM
g	Hour of day, 12-hour format, no leading zeros	1 to 12
G	Hour of day, 24-hour format, no leading zeros	0 to 23
h	Hour of day, 12-hour format, with leading zeros	01 to 12
H	Hour of day, 24-hour format, with leading zeros	00 to 23
i	Minutes, with leading zeros	00 to 59
s	Seconds, with leading zeros	00 to 59
Timezone specifiers		
e	Timezone identifier	For example UTC or Europe/Prague
O	Difference to GMT, no colon between hours and minutes	For example +0200
P	Difference to GMT, with colon	For example +02:00
T	Timezone abbreviation, if known, or the GMT offset	For example CEST or GMT-0500

Date Constants

You can use a number of constants with the `date` command to return the date in specific formats. For example, `date(DATE_RSS)` returns the current date and time in the valid format for an RSS feed. Some of the more commonly used constants are:

DATE_ATOM

This is the format for Atom feeds. The PHP format is `"Y-m-d\TH:i:sP"`, and example output is `"2025-05-15T12:00:00+00:00"`. `DATE_RFC3339` uses the same format.

DATE_COOKIE

This is the format for cookies set from a web server or JavaScript. The PHP format is `"l, d-M-y H:i:s T"`, and example output is `"Thursday, 15-May-25 12:00:00 UTC"`.

DATE_RSS

This is the format for RSS feeds. The PHP format is `"D, d M Y H:i:s O"`, and example output is `"Thu, 15 May 2025 12:00:00 +0000"`.

DATE_W3C

This is the format defined by the World Wide Web Consortium for use in World Wide Web-related standards. The PHP format is `"Y-m-d\TH:i:sP"`, and example output is `"2025-05-15T12:00:00+00:00"`. It is the same format as `DATE_ATOM` and `DATE_RFC3339`.

You can find the complete list in the [documentation](#).

Using checkdate

You've seen how to display a valid date in a variety of formats. But how can you check whether a user has submitted a valid date to your program? The answer is to pass the month, day, and year to the `checkdate` function, which returns a value of `TRUE` if the date is valid or `FALSE` if it is not.

For example, if September 31 of any year is input, it will always be an invalid date. **Example 7-3** shows code that you could use for this. As it stands, it will find the given date invalid.

Example 7-3. Checking for the validity of a date

```
<?php
$month = 9;    // September (only has 30 days)
$day   = 31;   // 31st
$year  = 2025;

if (checkdate($month, $day, $year)) echo "Date is valid";
else echo "Date is invalid";
?>
```

File Handling

Powerful as it is, MySQL, discussed later in the book, is not the only (or necessarily the best) way to store all data on a web server. Sometimes it can be quicker and more convenient to directly access files on the hard disk. Cases in which you might need to do this are when modifying images such as uploaded user avatars or with a logfile that you wish to process.

First, though, a note about file naming: if you are writing code that might be used on various PHP installations, there is no way of knowing whether these systems are case-sensitive. For example, Windows and macOS filenames are not case-sensitive (unless the file format has been specifically changed to be case-sensitive), but Linux and Unix filenames are. Therefore, you should always assume that the system is case-sensitive and stick to a convention such as all-lowercase filenames.

Checking Whether a File Exists

To determine whether a file already exists, you can use the `file_exists` function, which returns either `TRUE` or `FALSE` and is used like this:

```
if (file_exists("testfile.txt")) echo "File exists";
```

Creating a File

At this point, *testfile.txt* doesn't exist, so let's create it and write a few lines to it. Type **Example 7-4** and save it as *testfile.php*.

Example 7-4. Creating a simple text file

```
<?php // testfile.php
    $fh = fopen("testfile.txt", 'w') or die("Failed to create file");

    $text = <<<_END
Line 1
Line 2
Line 3
    _END;

    fwrite($fh, $text) or die("Could not write to file");
    fclose($fh);
    echo "File 'testfile.txt' written successfully";
?>
```

Should a program call the `die` function, the open file will be automatically closed as part of terminating the program.

When you run this in a browser, all being well, you will receive the message File 'testfile.txt' written successfully. If you receive an error message, your hard disk may be full or, more likely, you may not have permission to create or write to the file, in which case you should modify the attributes of the destination folder according to your operating system. Otherwise, the file *testfile.txt* should now be residing in the same folder in which you saved the *testfile.php* program. Try opening the file in a text or program editor—the contents will look like this:

```
Line 1
Line 2
Line 3
```

This simple example shows the sequence that all file handling takes:

1. Always start by opening the file. You do this through a call to `fopen`.
2. Then you can call other functions; here we write to the file (`fwrite`), but you can also read from an existing file (`fread` or `fgets`) and do other things.
3. Finish by closing the file (`fclose`). Although the program does this for you when it ends, you should clean up by closing the file when you're finished.

Every open file requires a file resource so that PHP can access and manage it. The preceding example sets the variable `$fh` (which I chose to stand for *file handle*) to the value returned by the `fopen` function. Thereafter, each file-handling function that

accesses the opened file, such as `fwrite` or `fclose`, must be passed `$fh` as a parameter to identify the file being accessed. Don't worry about the content of the `$fh` variable; it's a number PHP uses to refer to internal information about the file—you just pass the variable to other functions.

Upon failure, `FALSE` will be returned by `fopen`. The previous example shows a simple way to capture and respond to the failure: it calls the `die` function to end the program and give the user an error message. A web application would never abort in this crude way (you would create a web page with an error message instead), but this is fine for our testing purposes.

Notice the second parameter to the `fopen` call. It is simply the character `w`, which tells the function to open the file for writing. The function creates the file if it doesn't already exist. Be careful when playing around with these functions: if the file already exists, the `w` mode parameter causes the `fopen` call to delete the old contents (even if you don't write anything new!).

There are several different mode parameters that can be used here, as detailed in [Table 7-5](#). The modes that include a `+` symbol are further explained in [“Updating Files” on page 148](#).

Table 7-5. The supported fopen modes

Mode	Action	Description
'r'	Read from file's beginning	Open for reading only; place the file pointer at the beginning of the file. Return <code>FALSE</code> if the file doesn't already exist.
'r+'	Read from file's beginning and allow writing	Open for reading and writing; place the file pointer at the beginning of the file. Return <code>FALSE</code> if the file doesn't already exist.
'w'	Write from file's beginning and truncate file	Open for writing only; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.
'w+'	Write from file's beginning, truncate file, and allow reading	Open for reading and writing; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.
'a'	Append to file's end	Open for writing only; place the file pointer at the end of the file. If the file doesn't exist, attempt to create it.
'a+'	Append to file's end and allow reading	Open for reading and writing; place the file pointer at the end of the file. If the file doesn't exist, attempt to create it.

Reading from Files

The easiest way to read from a text file is to grab a whole line through `fgets` (think of the final `s` as standing for *string*), as in [Example 7-5](#).

Example 7-5. Reading a file with *fgets*

```
<?php
    $fh = fopen("testfile.txt", 'r') or
        die("File does not exist or you lack permission to open it");

    $line = fgets($fh);
    fclose($fh);
    echo $line;
?>
```

If you created the file as shown in [Example 7-4](#), you'll get the first line:

Line 1

You can retrieve multiple lines or portions of lines through the *fread* function, as in [Example 7-6](#).

Example 7-6. Reading a file with *fread*

```
<?php
    $fh = fopen("testfile.txt", 'r') or
        die("File does not exist or you lack permission to open it");

    $text = fread($fh, 3);
    fclose($fh);
    echo $text;
?>
```

I've requested three bytes in the *fread* call, so the program displays this:

Lin

The *fread* function is commonly used with binary data. If you use it on text data that spans more than one line, remember to count newline characters.

Copying Files

Let's try out the PHP *copy* function to create a clone of *testfile.txt*. Type [Example 7-7](#), save it as *copyfile.php*, and then call up the program in your browser.

Example 7-7. Copying a file

```
<?php // copyfile.php
    copy('testfile.txt', 'testfile2.txt') or die("Could not copy file");
    echo "File successfully copied to 'testfile2.txt'";
?>
```

If you check your folder again, you'll see it contains the new file *testfile2.txt*. By the way, if you don't want your programs to exit on a failed copy attempt, you could

try the alternate syntax in [Example 7-8](#). This uses the ! (NOT) operator as a quick-and-easy shorthand. Placed in front of an expression, it applies the NOT operator, so the equivalent statement here in English would begin “If not able to copy...”

Example 7-8. Alternate syntax for copying a file

```
<?php // copyfile2.php
if (!copy('testfile.txt', 'testfile2.txt')) echo "Could not copy file";
else echo "File successfully copied to 'testfile2.txt'";
?>
```

Moving a File

To move a file, rename it with the `rename` function, as in [Example 7-9](#).

Example 7-9. Moving a file

```
<?php // movefile.php
if (!rename('testfile2.txt', 'testfile2.new'))
    echo "Could not rename file";
else echo "File successfully renamed to 'testfile2.new'";
?>
```

You can use the `rename` function on directories, too. To avoid any warning messages if the original file doesn’t exist, you can call the `file_exists` function first to check.

Deleting a File

Deleting a file is just a matter of using the `unlink` function to remove it from the filesystem, as in [Example 7-10](#).

Example 7-10. Deleting a file

```
<?php // deletefile.php
if (!unlink('testfile2.new')) echo "Could not delete file";
else echo "File 'testfile2.new' successfully deleted";
?>
```



Whenever you directly access files on your hard disk, you must always ensure that it is impossible for your filesystem to be compromised. For example, if you are deleting a file based on user input, you must make absolutely certain it is a file that can be safely deleted and that the user is allowed to delete it.

As with moving a file, a warning message will be displayed if the file doesn't exist, which you can avoid by using `file_exists` to first check for its existence before calling `unlink`.

Updating Files

Often, you will want to add more data to a saved file, which you can do in many ways. You can use one of the append write modes (see [Table 7-5](#)), or you can simply open a file for reading and writing with one of the other modes that supports writing, and move the file pointer to the correct place within the file that you wish to write to or read from.

The *file pointer* is the position within a file at which the next file access will take place, whether it's a read or a write. It is not the same as the *file handle* (as stored in the variable `$fh` in [Example 7-4](#)), which contains details about the file being accessed.

You can see this in action by typing [Example 7-11](#) and saving it as *update.php*. Then call it up in your browser.

Example 7-11. Updating a file

```
<?php // update.php
$fh  = fopen("testfile.txt", 'r+') or die("Failed to open file");
$text = fgets($fh);

fseek($fh, 0, SEEK_END);
fwrite($fh, "\n$text") or die("Could not write to file");
fclose($fh);

echo "File 'testfile.txt' successfully updated";
?>
```

This program opens *testfile.txt*, as created in [Example 7-4](#), for both reading and writing by setting the mode with `'r+'`, which puts the file pointer right at the start. It then uses the `fgets` function to read in a single line from the file (up to the first line feed). After that, the `fseek` function is called to move the file pointer right to the file end, at which point the line of text that was extracted from the start of the file (stored in `$text`) is then appended to the file's end (preceded by a `\n` line feed) and the file is closed. The resulting file now looks like this:

```
Line 1
Line 2
Line 3
Line 1
```

The first line has successfully been copied and then appended to the file's end.

As used here, in addition to the `$fh` file handle, the `fseek` function was passed two other parameters, `0` and `SEEK_END`. `SEEK_END` tells the function to move the file pointer to the end of the file, and `0` tells it how many positions it should then be moved backward from that point. In the case of [Example 7-11](#), a value of `0` is used because the pointer is required to remain at the file's end.

Two other seek options available to the `fseek` function are: `SEEK_SET` and `SEEK_CUR`. The `SEEK_SET` option tells the function to set the file pointer to the exact position given by the preceding parameter. Thus, the following example moves the file pointer to position 18:

```
fseek($fh, 18, SEEK_SET);
```

`SEEK_CUR` sets the file pointer to the current position *plus* the value of the given offset. Therefore, if the file pointer is currently at position 18, the following call will move it to position 23:

```
fseek($fh, 5, SEEK_CUR);
```

Locking Files for Multiple Accesses

Web programs are often called by many users at the same time. If more than one person tries to write to a file simultaneously, it can become corrupted. And if one person writes to it while another is reading from it, the file is all right, but the person reading it can get odd results. To handle simultaneous users, you must use the file-locking `flock` function. This function queues up all other requests to access a file until your program releases the lock. So, whenever your programs use write access on files that may be accessed concurrently by multiple users, you should also add file locking to them, as in [Example 7-12](#), which is an updated version of [Example 7-11](#).

Example 7-12. Updating a file with file locking

```
<?php
$fh  = fopen("testfile.txt", 'r+') or die("Failed to open file");
$text = fgets($fh);

if (flock($fh, LOCK_EX))
{
    fseek($fh, 0, SEEK_END);
    fwrite($fh, "$text") or die("Could not write to file");
    flock($fh, LOCK_UN);
}

fclose($fh);
echo "File 'testfile.txt' successfully updated";
?>
```

There is a trick to file locking to preserve the best possible response time for your website visitors: perform it directly before a change you make to a file, and then unlock it immediately afterward. Having a file locked for any longer than this will slow your application unnecessarily. This is why the calls to `flock` in [Example 7-12](#) are directly before and after the `fwrite` call.

The first call to `flock` sets an exclusive file lock on the file referred to by `$fh` using the `LOCK_EX` parameter:

```
flock($fh, LOCK_EX);
```

From this point, no other processes can write to (or even read from) the file until you release the lock by using the `LOCK_UN` parameter, like this:

```
flock($fh, LOCK_UN);
```

As soon as the lock is released, other processes are again allowed access to the file. This is one reason you should reseek to the point you wish to access in a file each time you need to read or write data—another process could have changed the file since the last access.

However, did you notice that the call to request an exclusive lock is nested as part of an `if` statement? This is because `flock` is not supported on all systems; thus, it is wise to check whether you successfully secured a lock, just in case one could not be obtained.

Something else you must consider is that `flock` is what is known as an *advisory* lock. This means that it locks out only other processes that call the function. If you have any code that goes right in and modifies files without implementing `flock` file locking, it will always override the locking and could wreak havoc on your files.

By the way, implementing file locking and then accidentally leaving it out in one section of code can lead to an extremely hard-to-locate bug.



`flock` will not work on NFS and many other networked filesystems. Also, when using a multithreaded server like ISAPI, you may not be able to rely on `flock` to protect files against other PHP scripts running in parallel threads of the same server instance. Additionally, `flock` is not supported on any system using the old FAT filesystem, such as older versions of Windows, although you are unlikely to come across such systems (hopefully).

If in doubt, try making a quick lock on a test file at the start of a program to see whether you can obtain a lock on the file. Don't forget to unlock it (and maybe delete it if not needed) after checking.

Also remember that any call to the `die` function automatically unlocks a lock and closes the file as part of ending the program.

Reading an Entire File

A handy function for reading in an entire file without having to use file handles is `file_get_contents`, as shown in [Example 7-13](#).

Example 7-13. Using `file_get_contents`

```
<?php
echo "<pre>"; // Enables display of line feeds
echo file_get_contents("testfile.txt");
echo "</pre>"; // Terminates <pre> tag
?>
```

But the function is actually a lot more useful, because you also can use it to fetch a file from a server across the internet, as in [Example 7-14](#), which requests the HTML from the O'Reilly home page and then displays it as if the user had surfed to the page itself. The result will be similar to [Figure 7-1](#) (at the time of writing).

Example 7-14. Grabbing the O'Reilly home page

```
<?php
echo file_get_contents("http://oreilly.com");
?>
```

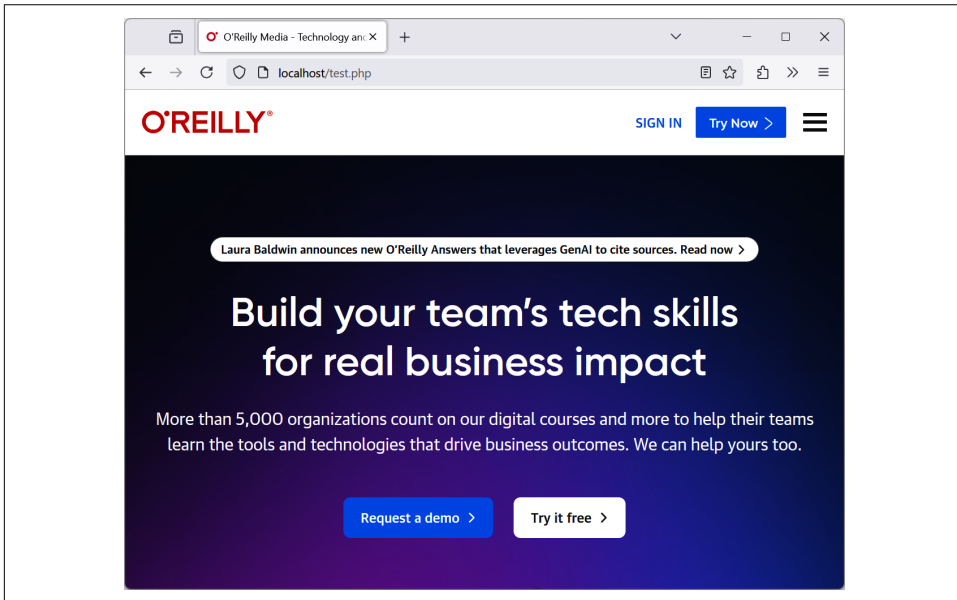


Figure 7-1. O'Reilly home page grabbed with `file_get_contents`

Uploading Files

Uploading files to a web server is a subject that seems daunting to many people, but it actually is very straightforward. All you need to do to upload a file from a form is choose a special type of encoding called `multipart/form-data`, and your browser will handle the rest. To see how this works, type the program in [Example 7-15](#) and save it as *upload.php*. When you run it, you'll see a form in your browser that lets you upload a file of your choice.

*Example 7-15. Image uploader *upload.php**

```
<?php // upload.php
echo <<<_END
    <html><head><title>PHP Form Upload</title></head><body>
    <form method='post' action='upload.php' enctype='multipart/form-data'>
    Select File: <input type='file' name='filename' size='10'>
    <input type='submit' value='Upload'>
    </form>
_END;

if ($_FILES)
{
    $name = $_FILES['filename']['name'];
    move_uploaded_file($_FILES['filename']['tmp_name'], $name);
    echo "Uploaded image '$name'<br><img src='$name'>";
}

echo "</body></html>";
?>
```

Let's examine this program a section at a time. The first line of the multiline `echo` statement starts an HTML document, displays the title, and then starts the document's body.

Next we come to the form, which selects the POST method of form submission, sets the target for posted data to the program *upload.php* (the program itself), and tells the web browser that the data posted should be encoded via the content type of `multipart/form-data`, the mime type used for file uploads.

With the form set up, the next lines display the prompt `Select File:` and then request two inputs. The first request is for a file; it uses an input type of `file`, a name of `filename`, and an input field with a width of 10 characters. The second requested input is a submit button given the label `Upload` (which replaces the default button text of `submit query`). And then the form is closed.

This short program shows a common technique in web programming in which a single program is called twice: once when the user first visits a page (which is an HTTP GET method request) and again when the user clicks the submit button

(an HTTP POST method request that offers some extras over a GET request, for example, the file uploads).

The PHP code to receive the uploaded data is fairly simple, because information about all uploaded files is placed into the associative system array `$_FILES`. Therefore, a quick check to see whether `$_FILES` contains anything is sufficient to determine whether the user has uploaded a file. This is done with the statement `if ($_FILES)`.

The first time the user visits the page (using a GET method request), before uploading a file, `$_FILES` is empty, so the program skips this block of code. When the user uploads a file (a POST method request), the program runs again and discovers an element in the `$_FILES` array.

Once the program realizes that a file was uploaded, the actual name, as read from the uploading computer, is retrieved and placed into the variable `$name`. Now all that's needed is to move the uploaded file from the temporary location in which PHP stored it to a more permanent one. We do this using the built-in `move_uploaded_file` function, passing it the original name of the file, with which it is saved to the current directory.



If you run this program and receive a warning message such as `Permission denied for the move_uploaded_file function call`, then you may not have the correct permissions set for the folder the program is running in.

Finally, the uploaded image is displayed within an `IMG` tag, and the result should look like [Figure 7-2](#).

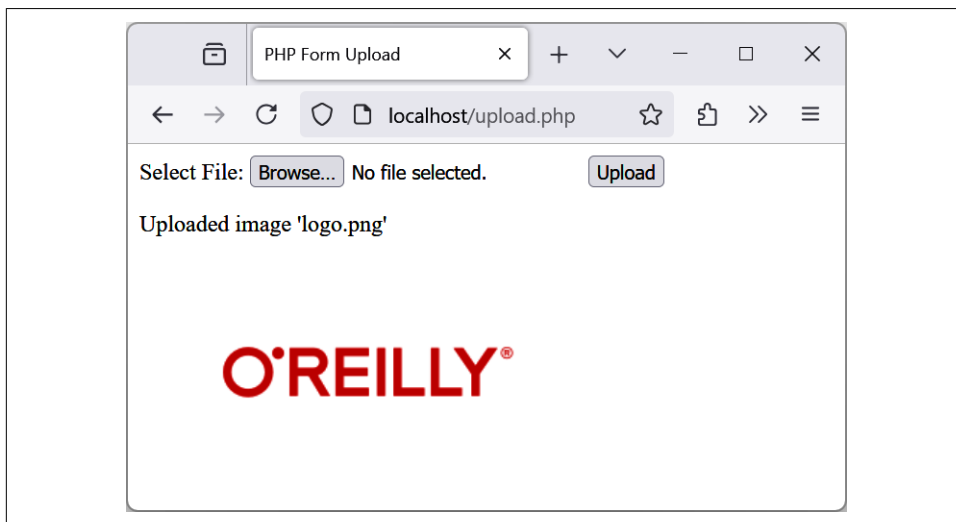


Figure 7-2. Uploading an image as form data

Using \$_FILES

Five things are stored in the `$_FILES` array when a file is uploaded, as shown in [Table 7-6](#) (where *file* is the file upload field name supplied by the submitting form).

Table 7-6. The contents of the `$_FILES` array

Array element	Contents
<code>\$_FILES['file']['name']</code>	The name of the uploaded file (e.g., <i>smiley.jpg</i>)
<code>\$_FILES['file']['type']</code>	The content type of the file (e.g., <i>image/jpeg</i>)
<code>\$_FILES['file']['size']</code>	The file's size in bytes
<code>\$_FILES['file']['tmp_name']</code>	The name of the temporary file stored on the server
<code>\$_FILES['file']['error']</code>	The error code resulting from the file upload

Content types used to be known as *MIME* (Multipurpose Internet Mail Extension) types, but because their use later expanded to the whole internet, now they are often called *internet media types*. [Table 7-7](#) shows some of the more frequently used types that turn up in `$_FILES['file']['type']`.

Table 7-7. Some common internet media content types

application/pdf	image/gif	multipart/form-data	text/xml
application/zip	image/jpeg	text/css	video/mpeg
audio/mpeg	image/png	text/html	video/mp4
audio/x-wav	application/json	text/plain	audio/webm

Validation

It's important to stress here that form data validation is of the utmost importance, due to the possibility of users attempting to hack into your server.

In addition to maliciously formed input data, you also have to check whether a file was actually received and, if so, whether the right type of data was sent.

Taking all these things into account, [Example 7-16](#), *upload2.php*, is a more secure rewrite of *upload.php*.

Example 7-16. A more secure version of *upload.php*

```
<?php // upload2.php
echo <<<_END
    <html><head><title>PHP Form Upload</title></head><body>
    <form method='post' action='upload2.php' enctype='multipart/form-data'>
    Select a JPG, GIF or PNG File:
    <input type='file' name='filename' size='10'>
    <input type='submit' value='Upload'></form>
    _END;
```

```

if ($_FILES)
{
    $name = $_FILES['filename']['name'];

    switch($_FILES['filename']['type'])
    {
        case 'image/jpeg': $ext = 'jpg'; break;
        case 'image/gif':  $ext = 'gif'; break;
        case 'image/png':  $ext = 'png'; break;
        default:           $ext = '';   break;
    }
    if ($ext)
    {
        $n = "image.$ext";
        move_uploaded_file($_FILES['filename']['tmp_name'], $n);
        echo "Uploaded image '$name' as '$n':<br>";
        echo "<img src='$n'>";
    }
    else echo "'$name' is not an accepted image file";
}
else echo "No image has been uploaded";

echo "</body></html>";
?>

```

The non-HTML section of code has been expanded from the half-dozen lines of [Example 7-15](#) to more than 20 lines, starting at `if ($_FILES)`.

As with the previous version, this `if` line checks whether any data was actually posted, but there is now a matching `else` near the bottom of the program that echoes a message to the screen when nothing has been uploaded.

Within the `if` statement, the variable `$name` is assigned the value of the filename as retrieved from the uploading computer (just as before), but this time we won't rely on the user having sent us valid data. Instead, a `switch` statement checks the uploaded content type against the four types of image this program supports. If a match is made, the variable `$ext` is set to the three-letter file extension for that type. Should no match be found, the file uploaded was not of an accepted type, and the variable `$ext` is set to the empty string `""`.



In this example the file type still comes from the browser and can be modified or changed by the user uploading the file. In this instance such user manipulation is not of concern as the files are only being treated as images. But if the file ever could be executable, you should not rely on information you have not ascertained to be absolutely correct.

The next section of code then checks the variable `$ext` to see whether it contains a string and, if so, creates a new filename called `$n` with the base name *image* and the extension stored in `$ext`. This means the program has full control over the file type of the file to be created, as it can be only one of *image.jpg*, *image.gif*, *image.png*, or *image.tif*.

Safe in the knowledge that the program has not been compromised, the rest of the PHP code is much the same as in the previous version. It moves the uploaded temporary image to its new location and then displays it while also displaying the old and new image names.



Don't worry about having to delete the temporary file that PHP creates during the upload process, because if the file has not been moved or renamed, it will be automatically removed when the program exits.

After the `if` statement, there is a matching `else`, which is executed only if an unsupported image type was uploaded (in which case it displays an appropriate error message).

When you write your own file-uploading routines, I strongly advise you to use a similar approach and have prechosen names and locations for uploaded files. That way, no attempts to add pathnames and other malicious data to the variables you use can get through. If this means more than one user could end up having a file uploaded with the same name, you could prefix such files with their user's usernames, or save them to individually created folders for each user.

But if you must use a supplied filename, you should sanitize it by allowing only alphanumeric characters and the period, which you can do with the following command, using a regular expression (see [Chapter 17](#)) to perform a search and replace on `$name`:

```
$name = preg_replace("/[^A-Za-z0-9.]/", "", $name);
```

This leaves only the characters A–Z, a–z, 0–9 and periods in the string `$name`, and strips out everything else.

Even better, to ensure that your program will work on all systems, regardless of whether they are case-sensitive or case-insensitive, you should use the following command instead, which changes all uppercase characters to lowercase at the same time:

```
$name = strtolower(preg_replace("/[^A-Za-z0-9.]/", "", $name));
```



Sometimes you may encounter the media type of `image/pjpeg`, which indicates a progressive JPEG, but you can safely add this to your code as an alias of `image/jpeg`, like this:

```
case 'image/pjpeg':  
case 'image/jpeg': $ext = 'jpg'; break;
```

System Calls

Sometimes PHP will not have the function you need to perform a certain action, but the operating system it is running on may. In such cases, you can use the `exec` system call to do the job.

For example, to quickly view the contents of the current directory, you can use a program such as [Example 7-17](#). If you are on a Windows system, it will run as is using the Windows `dir` command. On Linux, Unix, or macOS, comment out or remove the first line and uncomment the second to use the `ls` system command. You may wish to type this program, save it as `exec.php`, and call it up in your browser.

Example 7-17. Executing a system command

```
<?php // exec.php  
$cmd = "dir"; // Windows, Linux  
// $cmd = "ls"; // Linux, Unix & Mac  
  
exec(escapeshellcmd($cmd), $output, $status);  
  
if ($status) echo "Exec command failed";  
else  
{  
    echo "<pre>";  
    foreach($output as $line) echo htmlspecialchars("$line\n");  
    echo "</pre>";  
}  
?>
```

The `htmlspecialchars` function is called to turn any special characters returned by the system into ones that HTML can understand and properly display, not only neatening the output (as for example the `<` symbol is replaced with the entity `<`;) but also providing a very important security measure preventing HTML injection attacks. Although such attacks are less of a concern when listing a directory contents, you should make it a habit to always secure your output.

Depending on the system you are using, the result of running this program will look something like this (from a Windows `dir` command):

```
Volume in drive C is Hard Disk
Volume Serial Number is DC63-0E29

Directory of C:\Program Files (x86)\Amps\www

11/04/2025  11:58    <DIR>          .
11/04/2025  11:58    <DIR>          ..
28/01/2025  16:45    <DIR>          7th_edition_examples
08/01/2025  10:34    <DIR>          cgi-bin
08/01/2025  10:34    <DIR>          error
29/01/2025  16:18             1,150 favicon.ico
               1 File(s)             1,150 bytes
               5 Dir(s)  1,611,387,486,208 bytes free
```

`exec` takes three arguments:

- The command itself (in the previous case, `$cmd`)
- An array in which the system will put the output from the command (in the previous case, `$output`)
- A variable to contain the returned status of the call (which, in the previous case, is `$status`)

If you wish, you can omit the `$output` and `$status` parameters, but you will not know the output created by the call or even whether it completed successfully.

You should also note the use of the `escapeshellcmd` function. It is a good habit to always use this when issuing an `exec` call, because it sanitizes the command string, preventing the execution of arbitrary commands should you supply user input to the call.



The system call functions are typically disabled on shared web hosts, as they pose a security risk. You should always try to solve your problems within PHP if you can and go to the system directly only if it is necessary. Also, going to the system is relatively slow, and you need to code two implementations if your application is expected to run on both Windows and Linux/Unix systems.

Now that you have mastered programming in PHP, the following chapter will introduce the MySQL database, with which you can process all the data a website could ever need to handle. First, though, test your knowledge on the practical PHP tips in this chapter with the following questions.

Questions

1. Which `printf` conversion specifier would you use to display a floating-point number?
2. What `printf` statement could be used to take the input string "Happy Birthday" and output the string "**Happy"?
3. To send the output from `printf` to a variable instead of to a browser, what alternative function would you use?
4. How would you create a Unix timestamp for 7:11 a.m. on May 2, 2025?
5. Which file access mode would you use with `fopen` to open a file in write and read mode, with the file truncated and the file pointer at the start?
6. What is the PHP command for deleting the file *file.txt*?
7. Which PHP function is used to read in an entire file in one go, even from across the web?
8. Which PHP superglobal variable holds the details on uploaded files?
9. Which PHP function enables the running of system commands?
10. Which function is used to turn any special characters returned by the system into ones that HTML can understand and properly display?

See “Chapter 7 Answers” on page 572 in the [Appendix](#) for the answers to these questions.

Introduction to MySQL

With well over 10 million installations, MySQL is probably the most popular database management system for web servers. Developed in the mid-1990s, it's now a mature technology that powers many of today's most-visited internet destinations.

One reason for its success is that, like PHP, it's free to use. But it's also extremely powerful and exceptionally fast. MySQL is also highly scalable, which means that it can grow with your website; the latest [benchmarks are kept up-to-date online](#).

MySQL Basics

A *database* is a structured collection of records or data stored in a computer system and organized in such a way that it can be quickly searched and information rapidly retrieved.

The *SQL* in MySQL stands for *Structured Query Language*. This language is loosely based on English and also used in other databases such as Oracle and Microsoft SQL Server. It is designed to allow simple requests from a database via commands such as:

```
SELECT title FROM publications WHERE author = 'Charles Dickens';
```

A MySQL database contains one or more *tables*, each of which contains *records* or *rows*. Within these rows are various *columns* or *fields* that contain the data itself. [Table 8-1](#) shows the contents of an example database of five publications, detailing the author, title, type, and year of publication.

Table 8-1. Example of a simple database

Author	Title	Type	Year
Mark Twain	The Adventures of Tom Sawyer	Fiction	1876
Jane Austen	Pride and Prejudice	Fiction	1811
Charles Darwin	On the Origin of Species	Nonfiction	1856
Charles Dickens	The Old Curiosity Shop	Fiction	1841
William Shakespeare	Romeo and Juliet	Play	1594

Each row in the table is the same as a row in a MySQL table, a column in the table corresponds to a column in MySQL, and each element within a row is the same as a MySQL field.

To uniquely identify this database, I'll refer to it as the *publications* database in the examples that follow. And, as you will have observed, all these publications are considered to be classics of literature, so I'll call the table within the database that holds the details *classics*.

Key Database Terms

The main terms you need to acquaint yourself with for now are:

Database

The overall container for a collection of data

Table

A subcontainer within a database that stores the actual data

Row

A single record within a table, which may contain several fields

Column

The name of a field within a row

Note that I'm not trying to reproduce the precise terminology used about relational databases but just to provide simple, everyday terms to help you quickly grasp basic concepts and get started with a database.

Accessing MySQL via the Command Line

There are three main ways you can interact with MySQL: using a command line, via a web interface such as phpMyAdmin, and through a programming language like PHP. We'll start doing the third option in [Chapter 10](#), but for now, let's look at the first two.



Graphical User Interfaces

You can also use a visual or graphical tool like the MySQL Workbench, or DBeaver and a MySQL addon is also available for favorite IDEs including Visual Studio Code and PhpStorm.

Starting the Command-Line Interface

The following sections describe relevant instructions for Windows, macOS, and Linux.

Windows users

If you installed AMPPS (as explained in [Chapter 2](#)) in the usual way, you will be able to access the MySQL executable from the following directory:

```
C:\Program Files\Ampps\mysql\bin
```



If you installed AMPPS in any other place, you will need to use that directory instead, such as the following for 32-bit installations of AMPPS:

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

By default, the initial AMPPS MySQL user is *root*, and it will have a default password of *mysql*. So, to enter MySQL's command-line interface, select Start→Run, enter CMD into the Run box, and press Return. This will call up a Windows command prompt. From there, enter the following (making any appropriate changes as just discussed):

```
cd C:\Program Files\Ampps\mysql\bin"
mysql -u root -pmysql
```

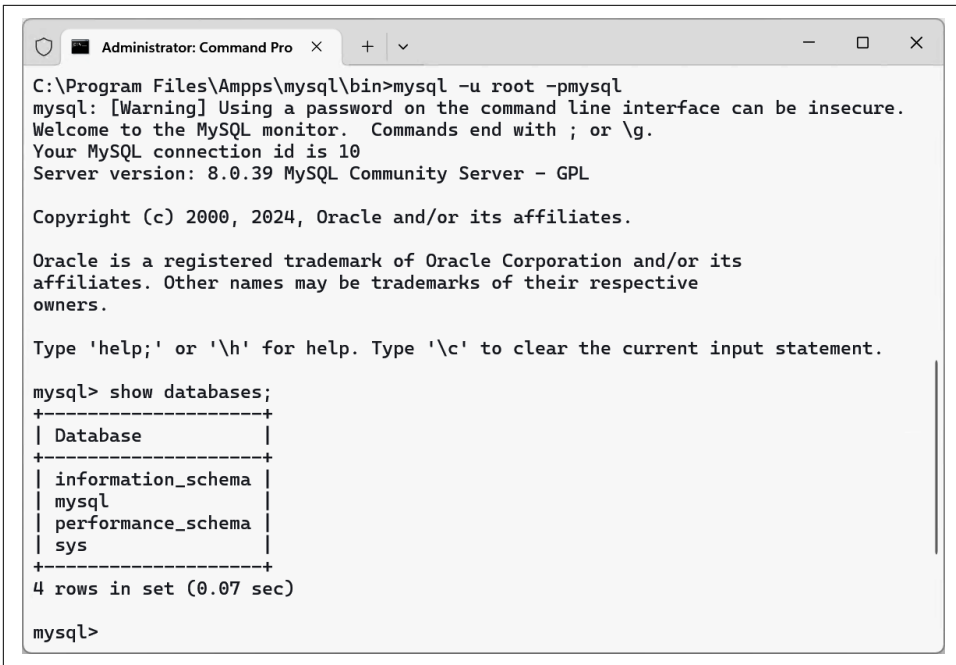
The first command changes to the MySQL directory, and the second tells MySQL to log you in as user *root*, with the password *mysql*. You will now be logged in to MySQL and can start entering commands.

If you are using Windows PowerShell (rather than a command prompt), it will not load commands from the current location as you must explicitly specify where to load a program from, in which case you would, instead, enter the following (note the preceding *./* before the *mysql* command):

```
cd C:\Program Files\Ampps\mysql\bin"
./mysql -u root -pmysql
```

To be sure everything is working as it should be, enter the following; the results should be similar to [Figure 8-1](#):

```
SHOW databases;
```



```
C:\Program Files\Ampps\mysql\bin>mysql -u root -pmysql
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 8.0.39 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| performance_schema |
| sys       |
+-----+
4 rows in set (0.07 sec)

mysql>
```

Figure 8-1. Accessing MySQL from a Windows command prompt

You are now ready to move to the next section, “Using the Command-Line Interface” on page 167.

macOS users

To proceed with this chapter, you should have installed AMPPS as detailed in [Chapter 2](#). You also should have the web server running and the MySQL server started.

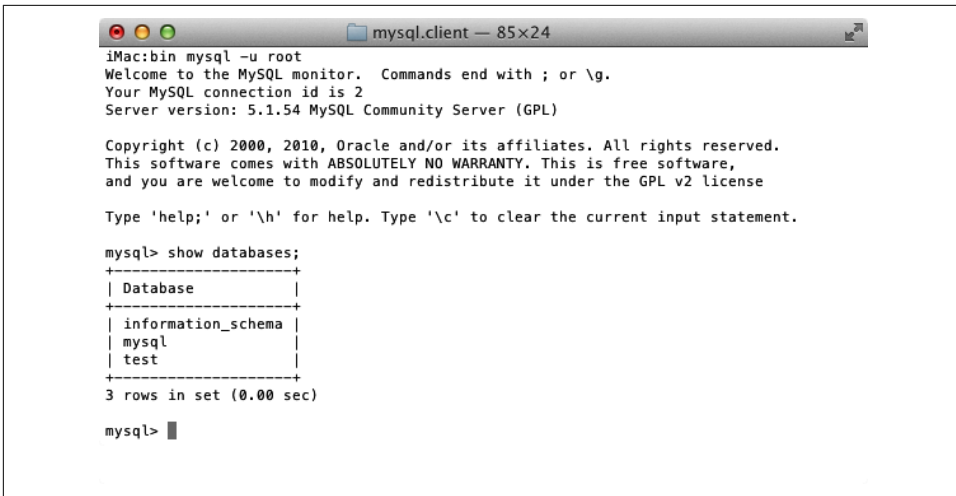
To enter the MySQL command-line interface, start the Terminal program (which should be available in Finder→Utilities). Then call up the MySQL program, which will have been installed in the directory `/Applications/ampps/mysql/bin`.

By default, the initial AMPPS MySQL user is *root*, and it will have a password of *mysql*. So, to start the program, type:

```
/Applications/ampps/mysql/bin/mysql -u root -pmysql
```

This command tells MySQL to log you in as user *root* using the password *mysql*. To verify that all is well, type the following ([Figure 8-2](#) should be the result):

```
SHOW databases;
```



```
iMac:bin mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.1.54 MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| test       |
+-----+
3 rows in set (0.00 sec)

mysql> █
```

Figure 8-2. Accessing MySQL from the macOS Terminal program

If you receive an error such as Can't connect to local MySQL server through socket, you may need to first start the MySQL server as described in [Chapter 2](#).

You should now be ready to move to the next section, “Using the Command-Line Interface” on page 167.

Linux users

On a system running a Unix-like operating system such as Linux, you may already have PHP and MySQL installed and running, and be able to enter the examples in this and the following chapters. First, you should type the following to log in to your MySQL system:

```
mysql -u root -p
```

This tells MySQL to log you in as the user *root* and to request your password. If you have a password, enter it; otherwise, just press Return.

Once you are logged in, type the following to test the program—you should see something like [Figure 8-3](#) in response:

```
SHOW databases;
```

```

robnix:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1264
Server version: 8.0.39-0ubuntu0.22.04.1 (Ubuntu)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.02 sec)

mysql>

```

Figure 8-3. Accessing MySQL using Linux

If this procedure fails at any point, please refer to [Chapter 2](#) to ensure that you have MySQL properly installed. Otherwise, you should now be ready to move to the next section, [“Using the Command-Line Interface” on page 167](#).

MySQL on a remote server

If you are accessing MySQL on a remote server, it will probably be a Linux/FreeBSD/Unix type of box, and you will connect to it via the secure SSH protocol. Once in there, you might find that things are a little different, depending on how the system administrator has set up the server, especially if it’s a shared hosting server. Therefore, you need to ensure that you have been given access to MySQL and that you have your username and password. Then you can type the following, where *username* is the name supplied:

```
mysql -u username -p
```

Enter your password when prompted. Then enter the following command, which should result in something like [Figure 8-3](#):

```
SHOW databases;
```

Other databases already may be created, and the *test* database may not be there.

Bear in mind also that system administrators have ultimate control over everything and that you can encounter some unexpected setups. For example, you may find

that you are required to preface all database names that you create with a unique identifying string to ensure that your names do not conflict with those of databases created by other users.

Therefore, if you have any problems, talk with your system administrator, who will get you sorted out. Just let the sysadmin know that you need a username and password. You should also ask for the ability to create new databases or, at a minimum, to have at least one database created for you ready to use. You can then create all the tables you require within that database.

Using the Command-Line Interface

From here on, it makes no difference whether you are using Windows, macOS, or Linux to access MySQL directly, as all the commands used (and errors you may receive) are identical.

The semicolon

Let's start with the basics. Did you notice the semicolon (;) at the end of the `SHOW databases;` command that you typed? The semicolon is used by MySQL to separate or end commands. If you forget to enter it, MySQL will issue a prompt and wait for you to do so. The required semicolon was made part of the syntax to let you enter multiline commands, which can be convenient because some commands get quite long. It also allows you to issue more than one command at a time by placing a semicolon after each one. The interpreter gets them all in a batch when you press the Enter (or Return) key and executes them in order.



It's very common to receive a MySQL prompt instead of the results of your command; it means that you forgot the final semicolon. Just enter the semicolon and press the Enter key, and you'll get what you want.

There are six different prompts that MySQL may present you with (see [Table 8-2](#)), so you will always know where you are during a multiline input.

Table 8-2. MySQL's six command prompts

MySQL prompt	Meaning
<code>mysql></code>	Ready and waiting for a command
<code>-></code>	Waiting for the next line of a command
<code>'></code>	Waiting for the next line of a string started with a single quote
<code>"></code>	Waiting for the next line of a string started with a double quote
<code>`></code>	Waiting for the next line of a string started with a backtick
<code>/*></code>	Waiting for the next line of a comment started with <code>/*</code>

Canceling a command

If you are partway through entering a command and decide you don't wish to execute it, you can enter `\c` and press Return. This is handy if you are within a set of multiline statements or simply to save you backspacing a lot. [Example 8-1](#) shows how to use the command.

Example 8-1. Canceling a line of input

```
meaningless gibberish \c
```

When you type that line, MySQL will ignore everything you typed and issue a new prompt. Without the `\c`, it would have displayed an error message. Be careful, though: if you have opened a string or comment, close it first before using the `\c` or MySQL will think the `\c` is just part of the string. [Example 8-2](#) shows the right way to do this.

Example 8-2. Canceling input from inside a string

```
this is "meaningless gibberish" \c
```

Also note that using `\c` after a semicolon will not cancel the preceding command, as it is then a new statement.

MySQL Commands

You've already seen the `SHOW` command, which lists tables, databases, and many other items. The commands you'll use most often are listed in [Table 8-3](#).

Table 8-3. Common MySQL commands

Command	Action
ALTER	Alter a database or table
BACKUP	Back up a table
\c	Cancel input
CREATE	Create a database, table, or index
DELETE	Delete a row from a table
DESCRIBE	Describe a table's columns
DROP	Delete a database or table
EXIT (Ctrl-C)	Exit (on some systems)
GRANT	Change user privileges
HELP (\h, \?)	Display help
INSERT	Insert data
LOCK	Lock table(s)

Command	Action
QUIT (\q)	Same as EXIT
RENAME	Rename a table
SHOW	List details about database(s), table(s), column(s), or server status
SOURCE	Execute a file
STATUS (\s)	Display the current status
TRUNCATE	Empty a table
UNLOCK	Unlock table(s)
UPDATE	Update an existing record
USE	Use a database

I'll cover most of these as we proceed, but first, you need to remember a couple of points about MySQL commands:

- SQL commands and keywords are case-insensitive. CREATE, create, and CrEaTe all mean the same thing. However, for the sake of clarity, you may prefer to use uppercase.
- Table names are case-sensitive on Linux and macOS but case-insensitive on Windows. For the sake of portability, you should always choose a case and stick to it. The recommended style is to use lowercase for table names.

Creating a database

If you are working on a remote server and have only a single user account and access to a single database that was created for you, move on to “[Creating a table](#)” on page 171. Otherwise, get the ball rolling by issuing the following command to create a new database called *publications*:

```
CREATE DATABASE publications;
```

A successful command will return a message that doesn't mean much yet—Query OK, 1 row affected (0.00 sec)—but will make sense soon. Now that you've created the database, you want to work with it, so issue the following command:

```
USE publications;
```

You should now see the message Database changed, and you will be set to proceed with the following examples.

Creating users

Now that you've seen how to use MySQL and create your first database, it's time to look at how you create users, as you probably won't want to grant your PHP scripts root access to MySQL—it could cause a real headache should you get hacked.

To create a user, issue the `CREATE USER` command, which takes the following form (don't type this in; it's not an actual working command):

```
CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';  
GRANT PRIVILEGES ON database.object TO 'username'@'hostname';
```

This should all look pretty straightforward, with the possible exception of the *database.object* part, which refers to the database itself and the objects it contains, such as tables (see [Table 8-4](#)).

Table 8-4. Example parameters for the `GRANT` command

Arguments	Meaning
<code>*.*</code>	All databases and all their objects
<code>database.*</code>	Only the database called <i>database</i> and all its objects
<code>database.object</code>	Only the database called <i>database</i> and its object called <i>object</i>

So, let's create a user who can access just the new *publications* database and all its objects, by entering the following commands (replacing the username *jim* and also the password *password* with ones of your choosing):

```
CREATE USER 'jim'@'localhost' IDENTIFIED BY 'password';  
GRANT ALL ON publications.* TO 'jim'@'localhost';
```

This allows the user *jim@localhost* full access to the *publications* database using the password *password*. You can test whether this step has worked by entering `quit` to exit and then rerunning MySQL the way you did before, but instead of logging in as root, log in with whatever username you created. See [Table 8-5](#) for the correct command for your operating system. Modify it as necessary if the *mysql* client program is installed in a different directory on your system.

Table 8-5. Starting MySQL and logging in as *jim@localhost*

OS	Example command
Windows	<code>C:\Program Files\Ampps\mysql\bin\mysql" -u jim -p</code>
macOS	<code>/Applications/ampps/mysql/bin/mysql -u jim -p</code>
Linux	<code>mysql -u jim -p</code>

All you have to do now is enter your password when prompted, and you will be logged in.

If you choose to, you can place your password immediately following the `-p` (without any spaces) to avoid having to enter it when prompted, but this is considered poor practice because if other people are logged in to your system, there may be ways for them to look at the command you entered and find out your password.



You can grant only privileges that you already have, and you must also have the privilege to issue GRANT commands. There are a whole range of privileges you can choose to grant if you are not granting all privileges. For further details on the GRANT command and the REVOKE command, which can remove privileges once granted, see the [documentation](#). Also, be aware that if you create a new user but do not specify an IDENTIFIED BY clause, the user will have no password, a situation that is very insecure and should be avoided.

Creating a table

At this point, you should be logged in to MySQL with ALL privileges granted for the database *publications* (or a database that was created for you), so you're ready to create your first table. Make sure the correct database is in use by typing the following (replacing *publications* with the name of your database if it is different):

```
USE publications;
```

Now enter the command in [Example 8-3](#) one line at a time.

Example 8-3. Creating a table called classics

```
CREATE TABLE classics (  
  author VARCHAR(128),  
  title VARCHAR(128),  
  type VARCHAR(16),  
  year CHAR(4)) ENGINE InnoDB;
```



The final two words in this command require a little explanation. MySQL can process queries in many different ways internally, and these different ways are supported by different *engines*. From version 5.6 onward *InnoDB* is the default storage engine for MySQL, and we use it here because it supports FULLTEXT searches. As long as you have a relatively up-to-date version of MySQL, you can omit the ENGINE InnoDB section of the command when creating a table, but I have kept it in for now to emphasize that this is the engine being used.

InnoDB is generally more efficient and the recommended option. If you installed the AMPPS stack as detailed in [Chapter 2](#), you should have at least version 5.6.35 of MySQL.



You could also issue the previous command on a single line, like this:

```
CREATE TABLE classics (author VARCHAR(128), title
VARCHAR(128), type VARCHAR(16), year CHAR(4)) ENGINE
InnoDB;
```

But MySQL commands can be long and complicated, so I recommend using the format shown in [Example 8-3](#) until you are comfortable with longer ones.

MySQL should then issue the response `Query OK, 0 rows affected`, along with how long it took to execute the command. If you see an error message instead, check your syntax carefully. Every parenthesis and comma counts, and typing errors are easy to make.

To check whether your new table has been created, type:

```
DESCRIBE classics;
```

All being well, you will see the sequence of commands and responses shown in [Example 8-4](#), where you should particularly note the table format displayed.

Example 8-4. A MySQL session: creating and checking a new table

```
mysql> USE publications;
Database changed
mysql> CREATE TABLE classics (
-> author VARCHAR(128),
-> title VARCHAR(128),
-> type VARCHAR(16),
-> year CHAR(4)) ENGINE InnoDB;
Query OK, 0 rows affected (0.03 sec)

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| author | varchar(128) | YES  |     | NULL    |       |
| title  | varchar(128) | YES  |     | NULL    |       |
| type   | varchar(16)  | YES  |     | NULL    |       |
| year   | char(4)       | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

The DESCRIBE command is an invaluable debugging aid when you need to ensure that you have correctly created a MySQL table. You can also use it to remind yourself about a table's field or column names and the types of data in each one. Let's look at each of the headings:

Field

The name of each field or column within a table

Type

The type of data being stored in the field

Null

Whether the field is allowed to contain a value of NULL

Key

What type of key, if any, has been applied (*keys* or *indexes* in MySQL are quick ways to look up and search for data)

Default

The default value that will be assigned to the field if no value is specified when a new row is created

Extra

Additional information, such as whether a field is set to auto-increment

Data Types

In **Example 8-3**, you may have noticed that three of the table's fields were given the data type of VARCHAR, and one was given the type CHAR. The term VARCHAR stands for *VARiable length CHARacter string*, and the command takes a numeric value that tells MySQL the maximum length allowed for a string stored in this field.

Both CHAR and VARCHAR accept text strings and impose a limit on the size of the field. The difference is that every string in a CHAR field has the specified size. If you put in a smaller string, it is padded with spaces. A VARCHAR field does not pad the text; it lets the size of the field vary to fit the text that is inserted. But VARCHAR requires a small amount of overhead to keep track of the size of each value. So, CHAR is slightly more efficient if the sizes are similar in all records, whereas VARCHAR is more efficient if sizes can vary a lot and get large. In addition, the overhead causes access to VARCHAR data to be slightly slower than to CHAR data, but in most use cases that's not a concern as the performance difference will not be noticeable.

Another feature of character and text columns, important for today's global web reach, is *character sets*. These assign particular binary values to particular characters. The character set you use for English is obviously different from the one you'd use for Russian. You can assign the character set to a character or text column when you create it.

VARCHAR is useful in our example, because it can accommodate author names and titles of different lengths while helping MySQL plan the size of the database and perform lookups and searches more easily. Just be aware that if you ever attempt

to assign a string value longer than the length allowed, it will be truncated to the maximum length declared in the table definition.

The year field, however, has predictable values, so instead of VARCHAR we use the more efficient CHAR(4) data type. The parameter of 4 allows for 4 bytes of data, supporting all years from -999 to 9999; a byte comprises 8 bits and can have the values 00000000 through 11111111, which are 0 to 255 in decimal.

You could, of course, just store two-digit values for the year, but if your data is still going to be needed in the following century, or may otherwise wrap around, it will have to be sanitized first. Think of the “millennium bug” that would have caused dates beginning on January 1, 2000, to be treated as 1900 on many of the world’s biggest computer installations.



I didn’t use the YEAR data type in the *classics* table because it supports only the years 0000 and 1901 through 2155. This is because MySQL stores the year in a single byte for reasons of efficiency, but it means that only 256 years are available, and the publication years of the titles in the *classics* table are well before 1901. I used the CHAR type instead, but another option is to use either the INT or the SMALLINT data type.

The CHAR data type

Table 8-6 lists the CHAR data types. Both types offer a parameter that sets the maximum (or exact) length of the string allowed in the field. As the table shows, each type has a built-in maximum number of bytes it can occupy.

Table 8-6. MySQL’s CHAR data types

Data type	Bytes used	Examples
CHAR(<i>n</i>)	Exactly <i>n</i> (≤ 255)	CHAR(5) “Hello” uses 5 bytes CHAR(57) “Goodbye” uses 57 bytes
VARCHAR(<i>n</i>)	Up to <i>n</i> (≤ 65535)	VARCHAR(7) “Hello” uses 5 bytes VARCHAR(100) “Goodbye” uses 7 bytes

The BINARY data type

The BINARY data types (see **Table 8-7**) store strings of bytes that do not have an associated character set. For example, you might use the BINARY data type to store a GIF image.

Table 8-7. MySQL's *BINARY* data types

Data type	Bytes used	Examples
<code>BINARY(<i>n</i>)</code>	Exactly <i>n</i> (≤ 255)	As <code>CHAR</code> but contains binary data
<code>VARBINARY(<i>n</i>)</code>	Up to <i>n</i> (≤ 65535)	As <code>VARCHAR</code> but contains binary data

The TEXT data types

Character data can also be stored in one of the TEXT fields. The differences between these fields and VARCHAR fields are small:

- TEXT fields cannot have default values.
- MySQL indexes only the first *n* characters of a TEXT column (you specify *n* when you create the index).

What this means is that VARCHAR is the better and faster data type to use if you need to search the entire contents of a field. If you will never search more than a certain number of leading characters in a field, use a TEXT data type (see [Table 8-8](#)).

Table 8-8. MySQL's *TEXT* data types

Data type	Bytes used	Attributes
<code>TINYTEXT(<i>n</i>)</code>	Up to <i>n</i> (≤ 255)	Treated as a string with a character set
<code>TEXT(<i>n</i>)</code>	Up to <i>n</i> (≤ 65535)	Treated as a string with a character set
<code>MEDIUMTEXT(<i>n</i>)</code>	Up to <i>n</i> ($\leq 1.67\text{e} + 7$)	Treated as a string with a character set
<code>LONGTEXT(<i>n</i>)</code>	Up to <i>n</i> ($\leq 4.29\text{e} + 9$)	Treated as a string with a character set

The data types that have smaller maximums are also more efficient; therefore, you should use the one with the smallest yet reasonable maximum that you know is enough for any string you will be storing in the field.

The BLOB data types

The term BLOB stands for *Binary Large Object*, and therefore, as you would think, the BLOB data type is most useful for binary data in excess of 65,536 bytes. The main other difference between the BLOB and BINARY data types is that BLOBs cannot have default values. The BLOB data types are listed in [Table 8-9](#).

Table 8-9. MySQL's BLOB data types

Data type	Bytes used	Attributes
TINYBLOB(<i>n</i>)	Up to <i>n</i> (≤ 255)	Treated as binary data—no character set
BLOB(<i>n</i>)	Up to <i>n</i> (≤ 65535)	Treated as binary data—no character set
MEDIUMBLOB(<i>n</i>)	Up to <i>n</i> ($\leq 1.67\text{e} + 7$)	Treated as binary data—no character set
LONGBLOB(<i>n</i>)	Up to <i>n</i> ($\leq 4.29\text{e} + 9$)	Treated as binary data—no character set

Numeric data types

MySQL supports various numeric data types, from a single byte up to double-precision floating-point numbers. Although the most memory that a numeric field can use up is 8 bytes, you are well advised to choose the smallest data type that will adequately handle the largest value you expect. This will help keep your databases small and quickly accessible.

Table 8-10 lists the numeric data types supported by MySQL and the ranges of values they can contain. In case you are not acquainted with the terms, a *signed number* is one with a possible range from a minus value, through 0, to a positive one. An *unsigned number* has a value ranging from 0 to a positive one. They can both hold the same number of values; just picture a signed number as being shifted halfway to the left so that half its values are negative and half are positive. Note that floating-point values (of any precision) may only be signed.

Table 8-10. MySQL's numeric data types

Data type	Bytes used	Minimum value		Maximum value	
		Signed	Unsigned	Signed	Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	$-8.38\text{e} + 6$	0	$8.38\text{e} + 6$	$1.67\text{e} + 7$
INT / INTEGER	4	$-2.15\text{e} + 9$	0	$2.15\text{e} + 9$	$4.29\text{e} + 9$
BIGINT	8	$-9.22\text{e} + 18$	0	$9.22\text{e} + 18$	$1.84\text{e} + 19$
FLOAT	4	$-3.40\text{e} + 38$	<i>n/a</i>	$3.4\text{e} + 38$	<i>n/a</i>
DOUBLE / REAL	8	$-1.80\text{e} + 308$	<i>n/a</i>	$1.80\text{e} + 308$	<i>n/a</i>

To specify whether a data type is unsigned, use the UNSIGNED qualifier. The following example creates a table called *tablename* with a field in it called *fieldname* of the data type UNSIGNED INTEGER:

```
CREATE TABLE tablename (fieldname INT UNSIGNED);
```


When creating a numeric field, you can also pass an optional number as a parameter, like this:

```
CREATE TABLE tablename (fieldname INT(4));
```

But you must remember that, unlike with the `BINARY` and `CHAR` data types, this parameter does not indicate the number of bytes of storage to use. It may seem counterintuitive, but what the number actually represents is the display width of the data in the field when it is retrieved. It is commonly used with the `ZEROFILL` qualifier, like this:

```
CREATE TABLE tablename (fieldname INT(4) ZEROFILL);
```

What this does is cause any numbers with a width of less than four characters to be padded with one or more zeros, sufficient to make the display width of the field four characters long. When a field is already of the specified width or greater, no padding takes place.

DATE and TIME types

The main remaining data types supported by MySQL relate to the date and time and can be seen in [Table 8-11](#).

Table 8-11. MySQL's DATE and TIME data types

Data type	Time/date format
DATETIME	'0000-00-00 00:00:00'
DATE	'0000-00-00'
TIMESTAMP	'0000-00-00 00:00:00'
TIME	'00:00:00'
YEAR	0000 (Only years 0000 and 1901–2155)

The `DATETIME` and `TIMESTAMP` data types display the same way. The main difference is that `TIMESTAMP` has a very narrow range (from the years 1970 through 2037), whereas `DATETIME` will hold just about any date you're likely to specify, unless you're interested in ancient history or science fiction.

`TIMESTAMP` is useful, however, because you can let MySQL set the value for you. If you don't specify the value when adding a row, the current time is automatically inserted. You can also have MySQL update a `TIMESTAMP` column each time you change a row.

The AUTO_INCREMENT attribute

Sometimes you need to ensure that every row in your database is guaranteed to be unique. You could do this in your program by carefully checking the data you enter and making sure there is at least one value that differs in any two rows, but this approach is error-prone and works only in certain circumstances. In the *classics* table,

for instance, an author may appear multiple times. Likewise, the year of publication will also be frequently duplicated, and so on. It would be hard to guarantee that you have no duplicate rows. It is also difficult to guarantee unique rows when multiple scripts can insert rows into the same table in parallel.

The general solution is to use an extra column just for this purpose. In a while, we'll look at using a publication's ISBN (International Standard Book Number), but first I'd like to introduce the `AUTO_INCREMENT` data attribute.

As its name implies, a column given this attribute will set the value of its contents to that of the column entry in the previously inserted row, plus 1. [Example 8-5](#) shows how to add a new column called *id* to the table *classics* with auto-incrementing.

Example 8-5. Adding the auto-incrementing column id

```
ALTER TABLE classics ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT KEY;
```

This is your introduction to the `ALTER` command, which is very similar to `CREATE`. `ALTER` operates on an existing table and can add, change, or delete columns. Our example adds a column named *id* with the following characteristics:

`INT UNSIGNED`

Makes the column take an integer large enough for us to store more than 4 billion records in the table.

`NOT NULL`

Ensures that every column has a value. Many programmers use `NULL` in a field to indicate that it doesn't have any value. But that would allow duplicates, which would violate the whole reason for this column's existence, so we disallow `NULL` values.

`AUTO_INCREMENT`

Causes MySQL to set a unique value for this column in every row, as described earlier. We don't really have control over the value that this column will take in each row, but we don't care: all we care about is that we are guaranteed a unique value.

`KEY`

An auto-increment column is useful as a key, because you will tend to search for rows based on this column. This will be explained in [“Indexes” on page 184](#).

Each entry in the column *id* will now have a unique number, with the first starting at 1 and the others counting upward from there. And whenever a new row is inserted, its *id* column will automatically be given the next number in the sequence.

Rather than applying the column retroactively, you could have included it by issuing the CREATE command in a slightly different format. In that case, the command in [Example 8-3](#) would be replaced with [Example 8-6](#). Check the final line in particular.

Example 8-6. Adding the auto-incrementing id column at table creation

```
CREATE TABLE classics (  
  author VARCHAR(128),  
  title VARCHAR(128),  
  type VARCHAR(16),  
  year CHAR(4),  
  id INT UNSIGNED NOT NULL AUTO_INCREMENT KEY) ENGINE InnoDB;
```

If you wish to check whether the column has been added, use the following command to view the table's columns and data types:

```
DESCRIBE classics;
```

Now that we've finished with it, the *id* column is no longer needed, so if you created it using [Example 8-5](#), you should now remove the column using the command in [Example 8-7](#).

Example 8-7. Removing the id column

```
ALTER TABLE classics DROP id;
```

Adding data to a table

To add data to a table, use the INSERT command. Let's see this in action by populating the table *classics* with the data from [Table 8-1](#), using one form of the INSERT command repeatedly ([Example 8-8](#)).

Example 8-8. Populating the classics table

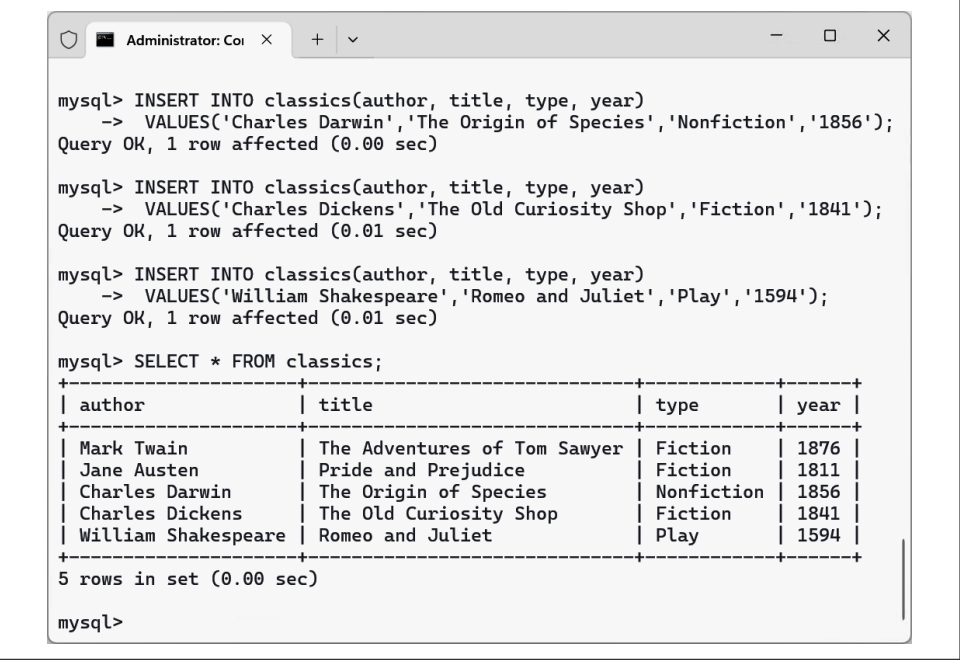
```
INSERT INTO classics(author, title, type, year)  
  VALUES('Mark Twain','The Adventures of Tom Sawyer','Fiction','1876');  
INSERT INTO classics(author, title, type, year)  
  VALUES('Jane Austen','Pride and Prejudice','Fiction','1811');  
INSERT INTO classics(author, title, type, year)  
  VALUES('Charles Darwin','The Origin of Species','Nonfiction','1856');  
INSERT INTO classics(author, title, type, year)  
  VALUES('Charles Dickens','The Old Curiosity Shop','Fiction','1841');  
INSERT INTO classics(author, title, type, year)  
  VALUES('William Shakespeare','Romeo and Juliet','Play','1594');
```

After every second line, you should see a Query OK message. Once all lines have been entered, type the following command, which will display the table's contents. The result should look like [Figure 8-4](#):

```
SELECT * FROM classics;
```

Don't worry about the SELECT command for now—we'll come to it in “[Querying a MySQL Database](#)” on page 190. Suffice it to say that, as typed, it will display all the data you just entered.

Also, don't worry if you see the returned results in a different order; this is normal because the order is unspecified at this point. Later in this chapter we will learn how to use ORDER BY to choose the order in which we want results to be returned, but for now, they can appear in any order.



```
mysql> INSERT INTO classics(author, title, type, year)
-> VALUES('Charles Darwin','The Origin of Species','Nonfiction','1856');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO classics(author, title, type, year)
-> VALUES('Charles Dickens','The Old Curiosity Shop','Fiction','1841');
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO classics(author, title, type, year)
-> VALUES('William Shakespeare','Romeo and Juliet','Play','1594');
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM classics;
+-----+-----+-----+-----+
| author | title | type | year |
+-----+-----+-----+-----+
| Mark Twain | The Adventures of Tom Sawyer | Fiction | 1876 |
| Jane Austen | Pride and Prejudice | Fiction | 1811 |
| Charles Darwin | The Origin of Species | Nonfiction | 1856 |
| Charles Dickens | The Old Curiosity Shop | Fiction | 1841 |
| William Shakespeare | Romeo and Juliet | Play | 1594 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Figure 8-4. Populating the classics table and viewing its contents

Let's go back and look at how we used the INSERT command. The first part, INSERT INTO classics, tells MySQL where to insert the following data. Then, within parentheses, the four column names are listed—*author*, *title*, *type*, and *year*—all separated by commas. This tells MySQL that these are the fields into which the data is to be inserted.

The second line of each INSERT command contains the keyword VALUES followed by four strings within parentheses, separated by commas. This supplies MySQL with the four values to be inserted into the four columns previously specified. (As always, my choice of where to break the lines was arbitrary.)

Each item of data will be inserted into the corresponding column, in a one-to-one correspondence. If you accidentally listed the columns in a different order from the data, the data would go into the wrong columns. Also, the number of columns must match the number of data items. (There are safer ways of using INSERT, which we'll see soon.)

Renaming a table

Renaming a table, like any other change to the structure or meta-information about a table, is achieved via the ALTER command. So, for example, to change the name of the table *classics* to *pre1900*, you would use this command:

```
ALTER TABLE classics RENAME pre1900;
```

If you tried the command, you should revert the table name by entering the following so that later examples in this chapter will work as printed:

```
ALTER TABLE pre1900 RENAME classics;
```

Changing the data type of a column

Changing a column's data type also uses the ALTER command, this time in conjunction with the MODIFY keyword. To change the data type of the column *year* from CHAR(4) to SMALLINT (which requires only 2 bytes of storage and so will save disk space), enter:

```
ALTER TABLE classics MODIFY year SMALLINT;
```

When you do this, if the conversion of data type makes sense to MySQL, it will automatically change the data while keeping the meaning. In this case, it will change each string to a comparable integer, so long as the string is recognizable as referring to an integer.

Adding a new column

Let's suppose that you have created a table and populated it with plenty of data, only to discover you need an additional column. Not to worry. Here's how to add the new column *pages*, which will be used to store the number of pages in a publication:

```
ALTER TABLE classics ADD pages SMALLINT UNSIGNED;
```

This adds the new column with the name *pages* using the UNSIGNED SMALLINT data type, sufficient to hold a value of up to 65,535—hopefully that's more than enough for any book ever published!

If you ask MySQL to describe the updated table by using the DESCRIBE command, as follows, you will see the change has been made (see [Figure 8-5](#)):

```
DESCRIBE classics;
```

```
Administrator: Command Pro x + - □ x
+-----+-----+-----+-----+-----+-----+
| Field | Type           | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| author | varchar(128)    | YES  |     | NULL    |       |
| title  | varchar(128)    | YES  |     | NULL    |       |
| type   | varchar(16)     | YES  |     | NULL    |       |
| year   | smallint        | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> ALTER TABLE classics ADD pages SMALLINT UNSIGNED;
Query OK, 0 rows affected (0.02 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+-----+
| Field | Type           | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| author | varchar(128)    | YES  |     | NULL    |       |
| title  | varchar(128)    | YES  |     | NULL    |       |
| type   | varchar(16)     | YES  |     | NULL    |       |
| year   | smallint        | YES  |     | NULL    |       |
| pages  | smallint unsigned | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> |
```

Figure 8-5. Adding the new pages column and viewing the table

Renaming a column

Looking again at [Figure 8-5](#), you may decide that having a column named *type* is confusing, because that is the name used by MySQL to identify data types. Again, no problem—let’s change its name to *category*, like this:

```
ALTER TABLE classics CHANGE type category VARCHAR(16);
```

Note the addition of `VARCHAR(16)` on the end of this command. That’s because the `CHANGE` keyword requires the data type to be specified, even if you don’t intend to change it, and `VARCHAR(16)` was the data type specified when that column was initially created as *type*.

Removing a column

Actually, upon reflection, you might decide that the page count column *pages* isn’t all that useful for this particular database, so here’s how to remove that column by using the `DROP` keyword:

```
ALTER TABLE classics DROP pages;
```



Remember that DROP is irreversible. You should always use it with caution, because you could inadvertently delete entire tables (and even databases) if you are not careful!

Deleting a table

Deleting a table is very easy indeed. But because I don't want you to have to reenter all the data for the *classics* table, let's quickly create a new table, verify its existence, and then delete it. You can do this by typing the commands in [Example 8-9](#). The result of these four commands should look like [Figure 8-6](#).

Example 8-9. Creating, viewing, and deleting a table

```
CREATE TABLE disposable(trash INT);
DESCRIBE disposable;
DROP TABLE disposable;
SHOW tables;
```

```
Administrator: Command Pro x + - □ ×

1 row in set (0.00 sec)

mysql>
mysql> CREATE TABLE disposable(trash INT);
Query OK, 0 rows affected (0.03 sec)

mysql> DESCRIBE disposable;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| trash | int  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> DROP TABLE disposable;
Query OK, 0 rows affected (0.02 sec)

mysql> SHOW tables;
+-----+
| Tables_in_publications |
+-----+
| classics                |
+-----+
1 row in set (0.00 sec)

mysql> |
```

Figure 8-6. Creating, viewing, and deleting a table

Indexes

As things stand, the table *classics* works and can be searched without problem by MySQL—until it grows to more than a couple of hundred rows. At that point, database accesses will get slower and slower with every new row added, because MySQL has to search through every row whenever a query is issued. This is like searching through every book in a library whenever you need to look something up.

Of course, you don't have to search libraries that way, because they have either a card index system or, most likely, a database of their own. And the same goes for MySQL, because at the expense of a slight overhead in memory and disk space, you can create a “card index” for a table that MySQL will use to conduct lightning-fast searches.

Creating an Index

The way to achieve fast searches is to add an *index*, either when creating a table or at any time afterward. But the decision is not so simple. For example, there are different index types, such as a regular INDEX, a PRIMARY KEY, or a FULLTEXT index. Also, you must decide which columns require an index, a judgment that requires you to predict whether you will be searching any of the data in each column. Indexes can get more complicated too, because you can combine multiple columns in one index. And even when you've decided that, you still have the option of reducing index size by limiting the amount of each column to be indexed.

If we imagine the searches that might be made on the *classics* table, it becomes apparent that all of the columns may need to be searched. However, if the *pages* column created in “Adding a new column” on page 181 had not been deleted, it would probably not have needed an index, as most people would be unlikely to search for books by the number of pages they have. Anyway, go ahead and add an index to each of the columns, using the commands in [Example 8-10](#).

Example 8-10. Adding indexes to the classics table

```
ALTER TABLE classics ADD INDEX(author(20));
ALTER TABLE classics ADD INDEX(title(20));
ALTER TABLE classics ADD INDEX(category(4));
ALTER TABLE classics ADD INDEX(year);
DESCRIBE classics;
```

The first two commands create indexes on the *author* and *title* columns, limiting each index to only the first 20 characters. For instance, when MySQL indexes the following title:

The Adventures of Tom Sawyer

it will actually store in the index only the first 20 characters:

This is done to minimize the size of the index and to optimize database access speed. I chose 20 because it's likely to be sufficient to ensure uniqueness for most strings in these columns. If MySQL finds two indexes with the same contents, it will have to waste time going to the table itself and checking the column that was indexed to find out which rows really matched.

With the *category* column, currently only the first character is required to identify a string as unique (F for Fiction, N for Nonfiction, and P for Play), but I chose an index of four characters to allow for future categories that may share the first three characters. You can also reindex this column later, when you have a more complete set of categories. And finally, I set no limit to the *year* column's index, because it has a clearly defined length of four characters.

The results of issuing these commands (and a DESCRIBE command to confirm that they worked) can be seen in [Figure 8-7](#), which shows the key MUL for each column. This key means that multiple occurrences of a value may occur within that column, which is exactly what we want, as authors may appear many times, the same book title could be used by multiple authors, and so on.

```

Administrator: Command Pro
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE classics ADD INDEX(title(20));
Query OK, 0 rows affected (0.04 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE classics ADD INDEX(category(4));
Query OK, 0 rows affected (0.05 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE classics ADD INDEX(year);
Query OK, 0 rows affected (0.04 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| author | varchar(128) | YES  | MUL | NULL    |       |
| title  | varchar(128) | YES  | MUL | NULL    |       |
| category | varchar(16) | YES  | MUL | NULL    |       |
| year   | smallint    | YES  | MUL | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>

```

Figure 8-7. Adding indexes to the classics table

Using CREATE INDEX

An alternative to using ALTER TABLE to add an index is to use the CREATE INDEX command. They are equivalent, except that CREATE INDEX cannot be used for creating a PRIMARY KEY (see “Primary keys” on page 186). The format of this command is shown in the second line of Example 8-11.

Example 8-11. These two commands are equivalent

```
ALTER TABLE classics ADD INDEX(author(20));  
CREATE INDEX author ON classics (author(20));
```

Adding indexes when creating tables

You don’t have to wait until after creating a table to add indexes. In fact, doing so can be time-consuming, as adding an index to a large table can take a very long time. In fact, you can start planning your tables on paper or using a diagram tool before you write even a single SQL query. Therefore, let’s look at a command that creates the table *classics* with indexes already in place.

Example 8-12 is a reworking of Example 8-3 in which the indexes are created at the same time as the table. Note that to incorporate the modifications made in this chapter, this version uses the new column name *category* instead of *type* and sets the data type of *year* to SMALLINT instead of CHAR(4). If you want to try it out without first deleting your current *classics* table, change the word *classics* in line 1 to something else like *classics1*, and then drop *classics1* after you have finished with it.

Example 8-12. Creating the table classics with indexes

```
CREATE TABLE classics (  
  author VARCHAR(128),  
  title VARCHAR(128),  
  category VARCHAR(16),  
  year SMALLINT,  
  INDEX(author(20)),  
  INDEX(title(20)),  
  INDEX(category(4)),  
  INDEX(year)) ENGINE InnoDB;
```

Primary keys

So far, you’ve created the table *classics* and ensured that MySQL can search it quickly by adding indexes, but there’s still something missing. All the publications in the table can be searched, but there is no single unique key for each publication to enable instant accessing of a row. The importance of having a key with a unique value for each row will come up when we start to combine data from different tables.

“The `AUTO_INCREMENT` attribute” on page 177 briefly introduced the idea of a primary key when creating the auto-incrementing column `id`, which could have been used as a primary key for this table. However, I wanted to reserve that task for a more appropriate column: the internationally recognized ISBN.

So let’s go ahead and create a new column for this key. Now, bearing in mind that ISBNs are 13 characters long, you might think that the following command would do the job:

```
ALTER TABLE classics ADD isbn CHAR(13) PRIMARY KEY;
```

But it doesn’t. If you try it, you’ll get an error similar to Duplicate entry for key 1. The reason is that the table is already populated with some data, and this command is trying to add a column with the value NULL to each row, which is not allowed, as all values must be unique in any column having a primary key index. If there were no data already in the table, this command would work just fine, as would adding the primary key index upon table creation.

In our current situation, we have to create the new column without an index, populate it with data, and then add the index retrospectively using the commands in [Example 8-13](#). Luckily, each of the years is unique in the current set of data, so we can use the `year` column to identify each row for updating. Note that this example uses the UPDATE command and WHERE keyword, which are explained in more detail in “Querying a MySQL Database” on page 190.

Example 8-13. Populating the isbn column with data and using a primary key

```
ALTER TABLE classics ADD isbn CHAR(13);
UPDATE classics SET isbn='9781598184891' WHERE year='1876';
UPDATE classics SET isbn='9780582506206' WHERE year='1811';
UPDATE classics SET isbn='9780517123201' WHERE year='1856';
UPDATE classics SET isbn='9780099533474' WHERE year='1841';
UPDATE classics SET isbn='9780192814968' WHERE year='1594';
ALTER TABLE classics ADD PRIMARY KEY(isbn);
DESCRIBE classics;
```

Once you have typed these commands, the results should look like [Figure 8-8](#). Note that the keywords PRIMARY KEY replace the keyword INDEX in the ALTER TABLE syntax (compare Examples 8-10 and 8-13).

```

mysql> UPDATE classics SET isbn='9780099533474' WHERE year='1841';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> UPDATE classics SET isbn='9780192814968' WHERE year='1594';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> ALTER TABLE classics ADD PRIMARY KEY(isbn);
Query OK, 0 rows affected (0.11 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> DESCRIBE classics;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| author | varchar(128) | YES | MUL | NULL | |
| title | varchar(128) | YES | MUL | NULL | |
| category | varchar(16) | YES | MUL | NULL | |
| year | smallint | YES | MUL | NULL | |
| isbn | char(13) | NO | PRI | NULL | |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>

```

Figure 8-8. Retrospectively adding a primary key to the *classics* table

To have created a primary key when the table *classics* was created, you could have used the commands in [Example 8-14](#). Again, rename *classics* in line 1 to something else if you wish to try this example, and then delete the test table afterward.

*Example 8-14. Creating the table *classics* with a primary key*

```

CREATE TABLE classics (
  author VARCHAR(128),
  title VARCHAR(128),
  category VARCHAR(16),
  year SMALLINT,
  isbn CHAR(13),
  INDEX(author(20)),
  INDEX(title(20)),
  INDEX(category(4)),
  INDEX(year),
  PRIMARY KEY (isbn)) ENGINE InnoDB;

```

Creating a FULLTEXT index

Unlike a regular index, MySQL's FULLTEXT allows super-fast searches of entire columns of text. It stores every word in every data string in a special index that you can search using “natural language,” in a similar manner to using a search engine.



It's not strictly true that MySQL stores *all* the words in a FULLTEXT index, because it has a built-in list of more than 500 words that it chooses to ignore because they are so common that they aren't very helpful for searching anyway—so-called *stopwords*. This list includes *the*, *as*, *is*, *of*, and so on. The list helps MySQL run much more quickly when performing a FULLTEXT search and keeps database sizes down.

Here are some things that you should know about FULLTEXT indexes:

- Since MySQL 5.6, InnoDB tables can use FULLTEXT indexes, but prior to that FULLTEXT indexes could be used only with MyISAM tables. If you need to convert a table to MyISAM, you can usually use the MySQL command `ALTER TABLE tablename ENGINE = MyISAM;`.
- FULLTEXT indexes can be created for CHAR, VARCHAR, and TEXT columns only.
- A FULLTEXT index definition can be given in the CREATE TABLE statement when a table is created or added later using ALTER TABLE (or CREATE INDEX).
- For large data sets, it is *much* faster to load your data into a table that has no FULLTEXT index and then create the index to avoid constant index updates.

To create a FULLTEXT index, apply it to one or more records, as in [Example 8-15](#), which adds a FULLTEXT index to the pair of columns *author* and *title* in the *classics* table (this index is in addition to the ones already created and does not affect them).

Example 8-15. Adding a FULLTEXT index to the table classics

```
ALTER TABLE classics ADD FULLTEXT(author,title);
```

You can now perform FULLTEXT searches across this pair of columns. This feature could really come into its own if you could now add the entire text of these publications to the database (particularly as they're out of copyright protection) and they would be fully searchable. See “[MATCH...AGAINST](#)” on [page 196](#) for a description of searches using FULLTEXT.



If you find that MySQL is running slower than you think it should be when accessing your database, the problem is usually related to your indexes. Either you don't have an index where you need one or the indexes are not optimally designed. Tweaking a table's indexes will often solve such a problem. Performance is beyond the scope of this book, but in [Chapter 9](#) I'll give you a few tips so you know what to look for.

Querying a MySQL Database

So far, we've created a MySQL database and tables, populated them with data, and added indexes to make them fast to search. Now it's time to look at how these searches are performed and the various commands and qualifiers available.

SELECT

As you saw in [Figure 8-4](#), the `SELECT` command is used to extract data from a table. In that section, I used its simplest form to select all data and display it—something you will never want to do on anything but the smallest tables, because all the data will scroll by at an unreadable pace. Alternatively, on Unix/Linux computers, you can tell MySQL to page output a screen at a time by issuing this command:

```
pager less;
```

This pipes output to the `less` program. To restore standard output and turn paging off, you can issue this command:

```
nopager;
```

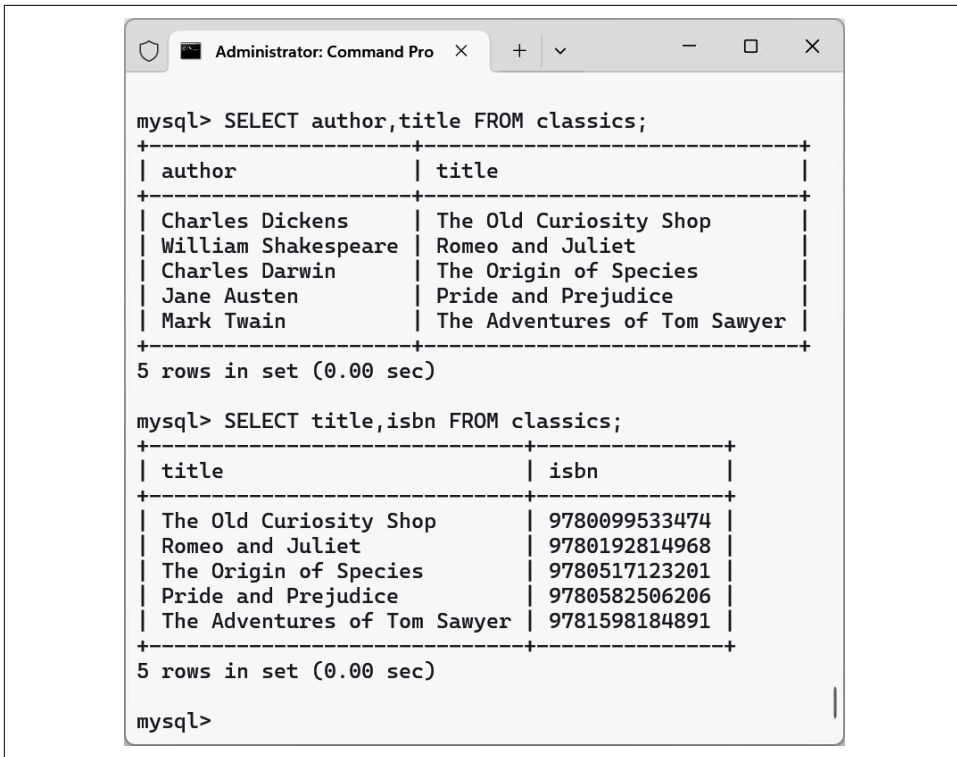
Let's now examine `SELECT` in more detail. The basic syntax is:

```
SELECT something FROM tablename;
```

The *something* can be an `*` (asterisk) as you saw before, which means *every column*, or you can choose to select only certain columns. For instance, [Example 8-16](#) shows how to select just the *author* and *title* and just the *title* and *isbn*. The result of typing these commands can be seen in [Figure 8-9](#).

Example 8-16. Two different SELECT statements

```
SELECT author,title FROM classics;  
SELECT title,isbn FROM classics;
```



```
mysql> SELECT author,title FROM classics;
+-----+-----+
| author          | title                               |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop             |
| William Shakespeare | Romeo and Juliet                 |
| Charles Darwin  | The Origin of Species             |
| Jane Austen     | Pride and Prejudice               |
| Mark Twain      | The Adventures of Tom Sawyer      |
+-----+-----+
5 rows in set (0.00 sec)

mysql> SELECT title,isbn FROM classics;
+-----+-----+
| title                | isbn                |
+-----+-----+
| The Old Curiosity Shop | 9780099533474      |
| Romeo and Juliet      | 9780192814968      |
| The Origin of Species | 9780517123201      |
| Pride and Prejudice   | 9780582506206      |
| The Adventures of Tom Sawyer | 9781598184891    |
+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Figure 8-9. The output from two different *SELECT* statements

SELECT COUNT

Another replacement for the *something* parameter is `COUNT`, which can be used in many ways. In [Example 8-17](#), it displays the number of rows in the table by passing `*` as a parameter, which means *all rows*. As you'd expect, the result returned is 5, as there are five publications in the table.

Example 8-17. Counting rows

```
SELECT COUNT(*) FROM classics;
```

SELECT DISTINCT

The `DISTINCT` qualifier (and its partner `DISTINCTROW`) allows you to weed out multiple entries when they contain the same data. For instance, suppose that you want a list of all authors in the table. If you select just the *author* column from a table containing multiple books by the same author, you'll normally see a long list with the same author names over and over. But by adding the `DISTINCT` keyword, you can

show each author just once. Let's test that out by adding another row that repeats one of our existing authors ([Example 8-18](#)).

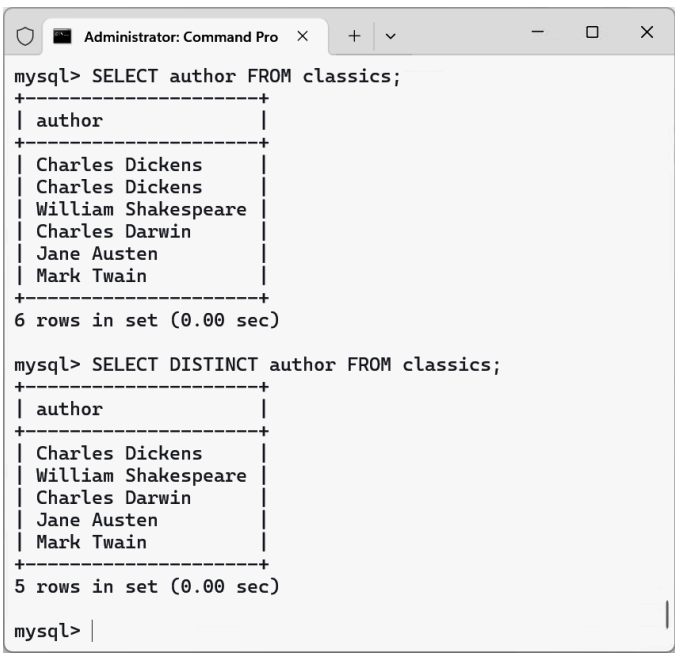
Example 8-18. Duplicating data

```
INSERT INTO classics(author, title, category, year, isbn)
VALUES('Charles Dickens','Little Dorrit','Fiction','1857','9780141439969');
```

Now that Charles Dickens appears twice in the table, we can compare the results of using SELECT with and without the DISTINCT qualifier. [Example 8-19](#) and [Figure 8-10](#) show that the simple SELECT lists Dickens twice, and the command with the DISTINCT qualifier shows him only once.

Example 8-19. With and without the DISTINCT qualifier

```
SELECT author FROM classics;
SELECT DISTINCT author FROM classics;
```



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Pro". Inside, a MySQL command prompt is active. The first query is `mysql> SELECT author FROM classics;`, which returns a table with 6 rows: Charles Dickens (listed twice), William Shakespeare, Charles Darwin, Jane Austen, and Mark Twain. The second query is `mysql> SELECT DISTINCT author FROM classics;`, which returns a table with 5 rows: Charles Dickens, William Shakespeare, Charles Darwin, Jane Austen, and Mark Twain. The prompt ends with `mysql> |`.

```
mysql> SELECT author FROM classics;
+-----+
| author          |
+-----+
| Charles Dickens |
| Charles Dickens |
| William Shakespeare |
| Charles Darwin  |
| Jane Austen     |
| Mark Twain      |
+-----+
6 rows in set (0.00 sec)

mysql> SELECT DISTINCT author FROM classics;
+-----+
| author          |
+-----+
| Charles Dickens |
| William Shakespeare |
| Charles Darwin  |
| Jane Austen     |
| Mark Twain      |
+-----+
5 rows in set (0.00 sec)

mysql> |
```

Figure 8-10. Selecting data with and without DISTINCT

DELETE

When you need to remove a row from a table, use the DELETE command. Its syntax is similar to the SELECT command and allows you to narrow down the exact row or rows to delete using qualifiers such as WHERE and LIMIT.

Now that you've seen the effects of the DISTINCT qualifier, if you typed [Example 8-18](#), you should remove *Little Dorrit* by entering the commands in [Example 8-20](#).

Example 8-20. Removing the new entry

```
DELETE FROM classics WHERE title='Little Dorrit';
```

This example issues a DELETE command for all rows whose *title* column contains the exact string Little Dorrit.

The WHERE keyword is very powerful and important to enter correctly; an error could lead a command to the wrong rows (or have no effect in cases where nothing matches the WHERE clause). So now we'll spend some time on that clause, which is the heart and soul of SQL.

WHERE

The WHERE keyword enables you to narrow queries by returning only those where a certain expression is true. [Example 8-20](#) returns only the rows where the column exactly matches the string Little Dorrit, using the equality operator =. [Example 8-21](#) shows a couple more examples of using WHERE with the = operator.

Example 8-21. Using the WHERE keyword

```
SELECT author,title FROM classics WHERE author="Mark Twain";  
SELECT author,title FROM classics WHERE isbn="9781598184891";
```

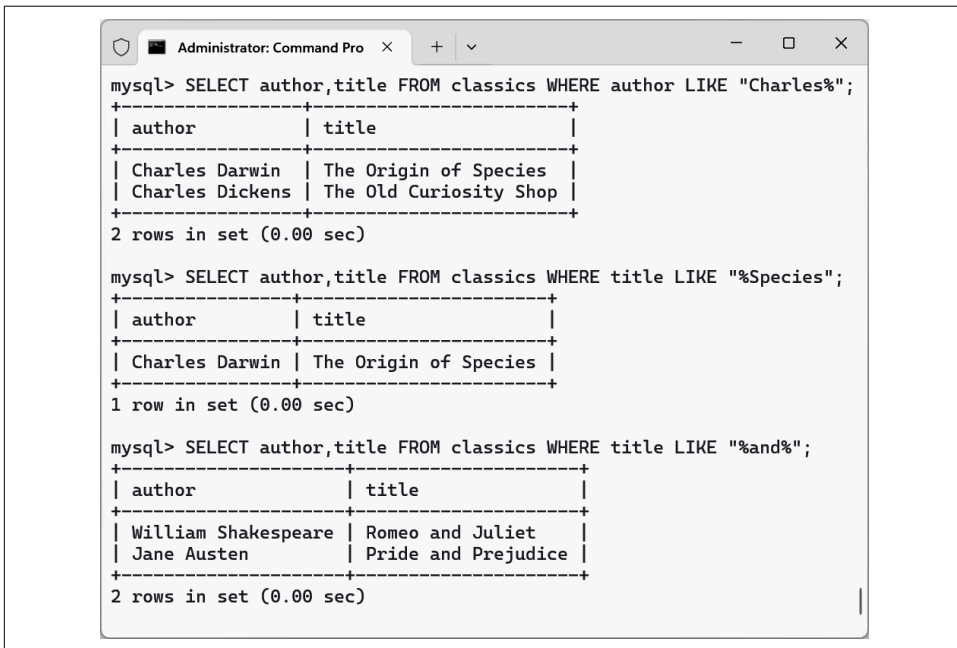
Given our current table, the two commands in [Example 8-21](#) display the same results. But we could easily add more books by Mark Twain, in which case the first line would display all the titles he wrote and the second line would continue (because we know the ISBN is unique) to display *The Adventures of Tom Sawyer*. In other words, searches using a unique key are more predictable, and you'll see further evidence later of the value of unique and primary keys.

You can also do pattern matching for your searches using the LIKE qualifier, which allows searches on parts of strings. This qualifier should be used with a % character before or after some text. When placed before a keyword, % means *anything before*. After a keyword, it means *anything after*. [Example 8-22](#) performs three different queries, one for the start of a string, one for the end, and one for anywhere in a string.

Example 8-22. Using the LIKE qualifier

```
SELECT author,title FROM classics WHERE author LIKE "Charles%";
SELECT author,title FROM classics WHERE title LIKE "%Species";
SELECT author,title FROM classics WHERE title LIKE "%and%";
```

You can see the results of these commands in [Figure 8-11](#). The first command outputs the publications by both Charles Darwin and Charles Dickens because the LIKE qualifier was set to return anything matching the string Charles followed by any other text. Then just *The Origin of Species* is returned, because it's the only row whose column ends with the string Species. Last, both *Pride and Prejudice* and *Romeo and Juliet* are returned, because they both matched the string and anywhere in the column. The % will also match if there is nothing in the position it occupies; in other words, it can match an empty string.



```
mysql> SELECT author,title FROM classics WHERE author LIKE "Charles%";
+-----+-----+
| author      | title                               |
+-----+-----+
| Charles Darwin | The Origin of Species              |
| Charles Dickens | The Old Curiosity Shop             |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT author,title FROM classics WHERE title LIKE "%Species";
+-----+-----+
| author      | title                               |
+-----+-----+
| Charles Darwin | The Origin of Species              |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT author,title FROM classics WHERE title LIKE "%and%";
+-----+-----+
| author      | title                               |
+-----+-----+
| William Shakespeare | Romeo and Juliet                  |
| Jane Austen   | Pride and Prejudice               |
+-----+-----+
2 rows in set (0.00 sec)
```

Figure 8-11. Using WHERE with the LIKE qualifier

LIMIT

The LIMIT qualifier enables you to choose how many rows to return in a query and where in the table to start returning them. When passed a single parameter, it tells MySQL to start at the beginning of the results and return just the number of rows given in that parameter. If you pass it two parameters, the first indicates the offset from the start of the results where MySQL should start the display, and the second

indicates how many to return. You can think of the first parameter as saying, “Skip this number of results at the start.”

Example 8-23 includes three commands. The first returns the first three rows from the table. The second returns two rows starting at position 1 (skipping one row). The last command returns a single row starting at position 3 (skipping the first three rows). **Figure 8-12** shows the results of issuing these three commands.

Example 8-23. Limiting the number of results returned

```
SELECT author,title FROM classics LIMIT 3;  
SELECT author,title FROM classics LIMIT 1,2;  
SELECT author,title FROM classics LIMIT 3,1;
```



Be careful with the LIMIT keyword, because offsets start at 0, but the number of rows to return starts at 1. So, LIMIT 1,3 means return *three* rows starting from the *second* row. You could look at the first argument as stating how many rows to skip, so that in English the instruction would be “Return 3 rows, skipping the first 1.”

```
Administrator: Command Pro x + - □ ×  
  
mysql> SELECT author,title FROM classics LIMIT 3;  
+-----+-----+  
| author          | title                |  
+-----+-----+  
| Charles Dickens | The Old Curiosity Shop |  
| William Shakespeare | Romeo and Juliet |  
| Charles Darwin  | The Origin of Species |  
+-----+-----+  
3 rows in set (0.00 sec)  
  
mysql> SELECT author,title FROM classics LIMIT 1,2;  
+-----+-----+  
| author          | title                |  
+-----+-----+  
| William Shakespeare | Romeo and Juliet |  
| Charles Darwin  | The Origin of Species |  
+-----+-----+  
2 rows in set (0.00 sec)  
  
mysql> SELECT author,title FROM classics LIMIT 3,1;  
+-----+-----+  
| author          | title                |  
+-----+-----+  
| Jane Austen    | Pride and Prejudice |  
+-----+-----+  
1 row in set (0.00 sec)
```

Figure 8-12. Restricting the rows returned with LIMIT

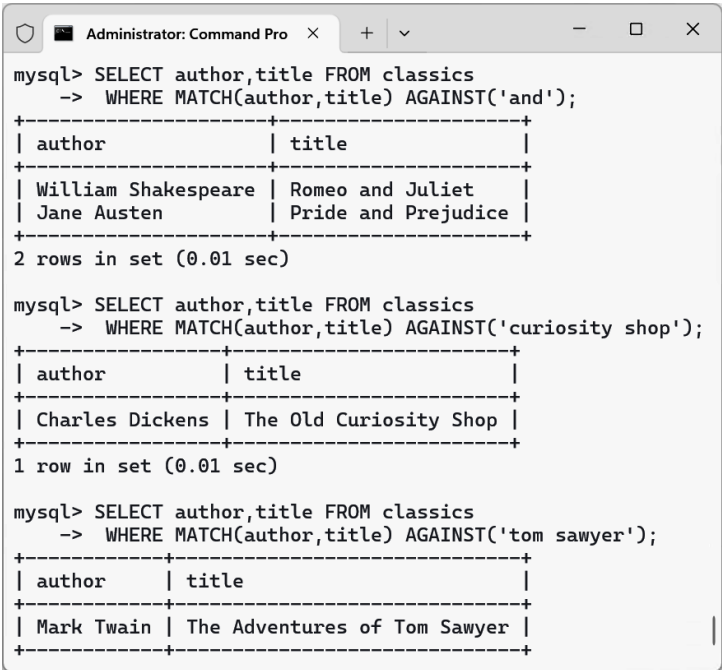
MATCH...AGAINST

The MATCH...AGAINST construct can be used on columns that have been given a FULLTEXT index (see “[Creating a FULLTEXT index](#)” on page 188). With it, you can make natural-language searches as you would in an internet search engine. Unlike the use of WHERE...= or WHERE...LIKE, MATCH...AGAINST lets you enter multiple words in a search query and checks them against all words in the FULLTEXT columns. FULLTEXT indexes are case-insensitive, so it makes no difference what case is used in your queries.

Assuming that you have added a FULLTEXT index to the *author* and *title* columns, enter the three queries shown in [Example 8-24](#). The first asks for any rows that contain the word *and* to be returned. If you are using the MyISAM storage engine, then because *and* is a stopword in that engine, MySQL will ignore it and the query will always produce an empty set—no matter what is stored in the column. Otherwise, if you are using InnoDB, *and* is an allowed word. The second query asks for any rows that contain both of the words *curiosity* and *shop* anywhere in them, in any order, to be returned. And the last query applies the same kind of search for the words *tom* and *sawyer*. [Figure 8-13](#) shows the results of these queries.

Example 8-24. Using MATCH...AGAINST on FULLTEXT indexes

```
SELECT author,title FROM classics
WHERE MATCH(author,title) AGAINST('and');
SELECT author,title FROM classics
WHERE MATCH(author,title) AGAINST('curiosity shop');
SELECT author,title FROM classics
WHERE MATCH(author,title) AGAINST('tom sawyer');
```



```
mysql> SELECT author,title FROM classics
-> WHERE MATCH(author,title) AGAINST('and');

+-----+-----+
| author          | title          |
+-----+-----+
| William Shakespeare | Romeo and Juliet |
| Jane Austen       | Pride and Prejudice |
+-----+-----+
2 rows in set (0.01 sec)

mysql> SELECT author,title FROM classics
-> WHERE MATCH(author,title) AGAINST('curiosity shop');

+-----+-----+
| author          | title          |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop |
+-----+-----+
1 row in set (0.01 sec)

mysql> SELECT author,title FROM classics
-> WHERE MATCH(author,title) AGAINST('tom sawyer');

+-----+-----+
| author          | title          |
+-----+-----+
| Mark Twain      | The Adventures of Tom Sawyer |
+-----+-----+
```

Figure 8-13. Using *MATCH...AGAINST* on *FULLTEXT* indexes

MATCH...AGAINST in Boolean mode

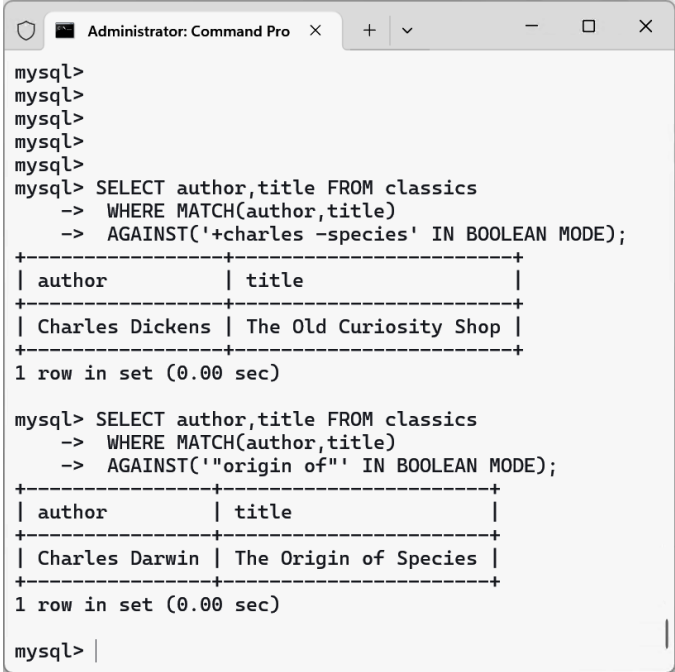
If you wish to give your *MATCH...AGAINST* queries even more power, use *Boolean mode*. This changes the effect of the standard *FULLTEXT* query so that it searches for any combination of search words, instead of requiring all search words to be in the text. The presence of a single word in a column causes the search to return the row.

Boolean mode also allows you to preface search words with a + or - sign to indicate whether they must be included or excluded. If normal Boolean mode says, “Any of these words will do,” a plus sign means, “This word must be present; otherwise, don’t return the row.” A minus sign means, “This word must not be present; its presence disqualifies the row from being returned.”

Example 8-25 illustrates Boolean mode through two queries. The first asks for all rows containing the word *charles* and not the word *species* to be returned. The second uses double quotes to request that all rows containing the exact phrase *origin of* be returned. **Figure 8-14** shows the results of these queries.

Example 8-25. Using *MATCH...AGAINST* in Boolean mode

```
SELECT author,title FROM classics
WHERE MATCH(author,title)
AGAINST('+charles -species' IN BOOLEAN MODE);
SELECT author,title FROM classics
WHERE MATCH(author,title)
AGAINST('"origin of"' IN BOOLEAN MODE);
```



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Pro". It contains two MySQL queries and their results. The first query filters for 'charles' and excludes 'species', returning one row: Charles Dickens, The Old Curiosity Shop. The second query filters for 'origin of', returning one row: Charles Darwin, The Origin of Species.

```
mysql>
mysql>
mysql>
mysql>
mysql> SELECT author,title FROM classics
        -> WHERE MATCH(author,title)
        -> AGAINST('+charles -species' IN BOOLEAN MODE);
+-----+-----+
| author          | title                      |
+-----+-----+
| Charles Dickens | The Old Curiosity Shop    |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT author,title FROM classics
        -> WHERE MATCH(author,title)
        -> AGAINST('"origin of"' IN BOOLEAN MODE);
+-----+-----+
| author          | title                      |
+-----+-----+
| Charles Darwin  | The Origin of Species     |
+-----+-----+
1 row in set (0.00 sec)

mysql> |
```

Figure 8-14. Using *MATCH...AGAINST* in Boolean mode

As you would expect, the first request returns only *The Old Curiosity Shop* by Charles Dickens; any rows containing the word *species* have been excluded, so Charles Darwin's publication is ignored.



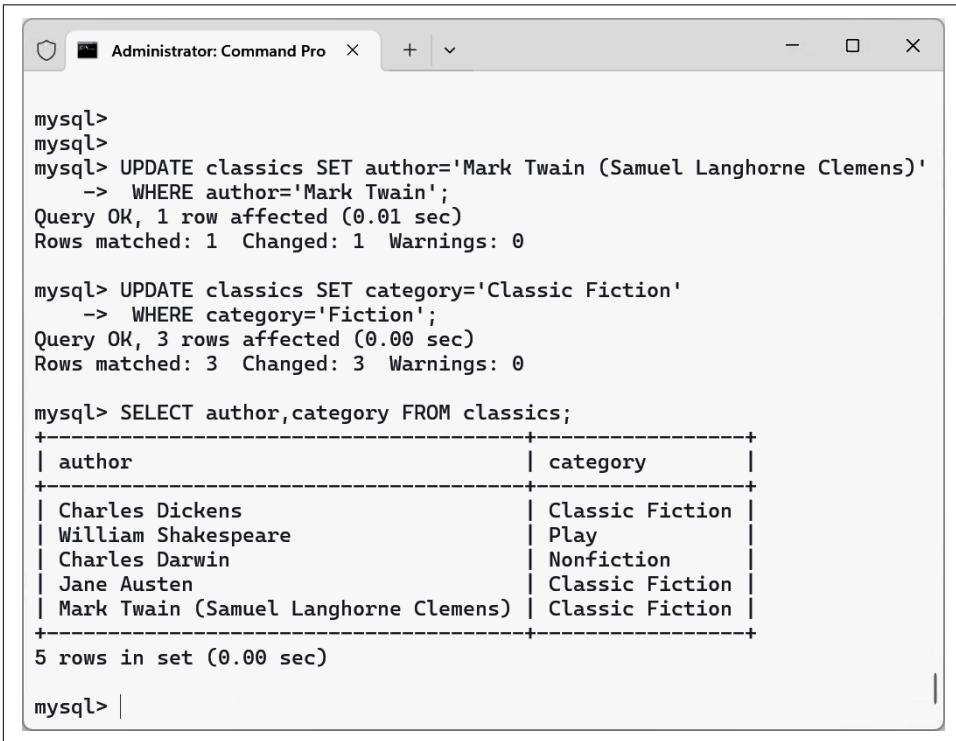
Something of interest to note in the second query: the stopword *of* is part of the search string, but it is still used by the search because the double quotation marks override stopwords.

UPDATE...SET

This construct allows you to update the contents of a field. If you wish to change the contents of one or more fields, you first need to focus on just the field or fields to be changed, in much the same way you use the SELECT command. [Example 8-26](#) shows the use of UPDATE...SET in two different ways. You can see the results in [Figure 8-15](#).

Example 8-26. Using UPDATE...SET

```
UPDATE classics SET author='Mark Twain (Samuel Langhorne Clemens)'  
WHERE author='Mark Twain';  
UPDATE classics SET category='Classic Fiction'  
WHERE category='Fiction';
```



```
mysql>  
mysql>  
mysql> UPDATE classics SET author='Mark Twain (Samuel Langhorne Clemens)'  
-> WHERE author='Mark Twain';  
Query OK, 1 row affected (0.01 sec)  
Rows matched: 1 Changed: 1 Warnings: 0  
  
mysql> UPDATE classics SET category='Classic Fiction'  
-> WHERE category='Fiction';  
Query OK, 3 rows affected (0.00 sec)  
Rows matched: 3 Changed: 3 Warnings: 0  
  
mysql> SELECT author,category FROM classics;  
+-----+-----+  
| author                                | category |  
+-----+-----+  
| Charles Dickens                      | Classic Fiction |  
| William Shakespeare                  | Play       |  
| Charles Darwin                       | Nonfiction  |  
| Jane Austen                          | Classic Fiction |  
| Mark Twain (Samuel Langhorne Clemens) | Classic Fiction |  
+-----+-----+  
5 rows in set (0.00 sec)  
  
mysql> |
```

Figure 8-15. Updating columns in the classics table

In the first query, Mark Twain's real name of Samuel Langhorne Clemens was appended to his pen name in parentheses, which affected only one row. The second query, however, affected three rows, because it changed all occurrences of *Fiction* in the *category* column to the term *Classic Fiction*.

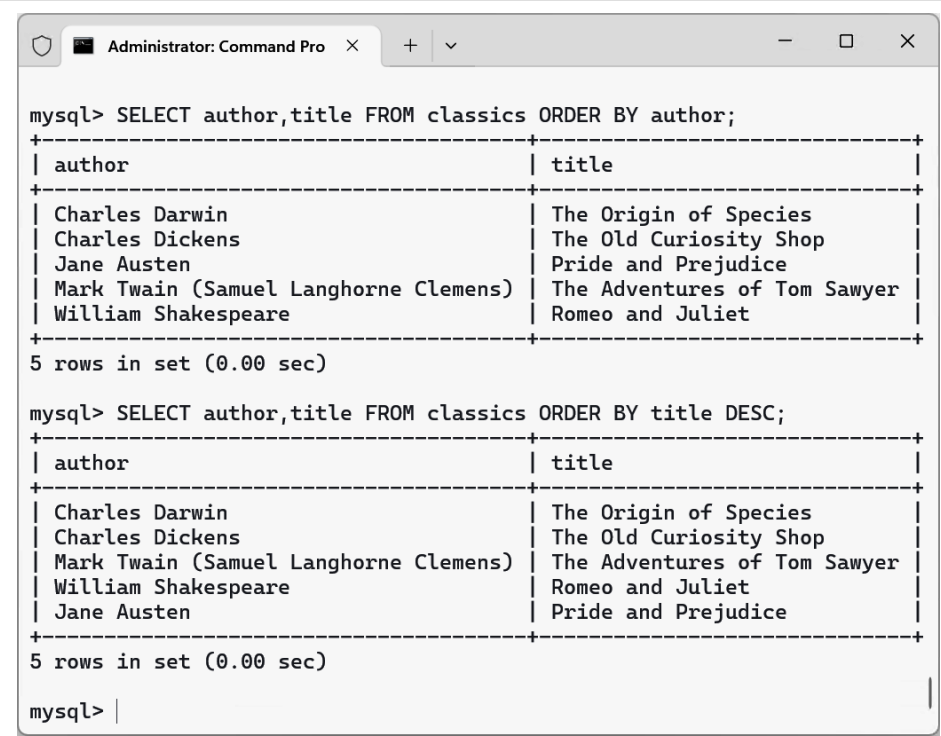
When performing an update, you can also use the qualifiers you have already seen, such as `LIMIT`, and the following `ORDER BY` and `GROUP BY` keywords.

ORDER BY

`ORDER BY` sorts returned results by one or more columns in ascending or descending order. [Example 8-27](#) shows two such queries, the results of which can be seen in [Figure 8-16](#).

Example 8-27. Using `ORDER BY`

```
SELECT author,title FROM classics ORDER BY author;
SELECT author,title FROM classics ORDER BY title DESC;
```



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Pro". It contains two MySQL queries and their results. The first query sorts by author in ascending order, and the second sorts by title in descending order. Both queries return 5 rows.

```
mysql> SELECT author,title FROM classics ORDER BY author;
+-----+-----+
| author                                | title                                |
+-----+-----+
| Charles Darwin                        | The Origin of Species              |
| Charles Dickens                       | The Old Curiosity Shop             |
| Jane Austen                           | Pride and Prejudice                |
| Mark Twain (Samuel Langhorne Clemens) | The Adventures of Tom Sawyer       |
| William Shakespeare                   | Romeo and Juliet                   |
+-----+-----+
5 rows in set (0.00 sec)

mysql> SELECT author,title FROM classics ORDER BY title DESC;
+-----+-----+
| author                                | title                                |
+-----+-----+
| Charles Darwin                        | The Origin of Species              |
| Charles Dickens                       | The Old Curiosity Shop             |
| Mark Twain (Samuel Langhorne Clemens) | The Adventures of Tom Sawyer       |
| William Shakespeare                   | Romeo and Juliet                   |
| Jane Austen                           | Pride and Prejudice                |
+-----+-----+
5 rows in set (0.00 sec)

mysql> |
```

Figure 8-16. Sorting the results of requests

As you can see, the first query returns the publications by *author* in ascending alphabetical order (the default), and the second returns them by *title* in descending order.

If you wanted to sort all the rows by *category* and then by descending *year* of publication (to view the most recent first), you would issue this query:

```
SELECT author,title,category,year FROM classics
ORDER BY category,year DESC;
```

This shows that each ascending and descending qualifier applies to a single column. The DESC keyword applies only to the preceding column, *year*. Because you allow *category* to use the default sort order, it is sorted in ascending order. You could also have explicitly specified ascending order for that column, with the same results:

```
SELECT author,title,category,year FROM classics
ORDER BY category ASC,year DESC;
```

GROUP BY

In a similar fashion to ORDER BY, you can group results returned from queries using GROUP BY, which is good for retrieving information about a group of data. For example, if you want to know how many publications of each category are in the *classics* table, you can issue the query:

```
SELECT category,COUNT(author) FROM classics GROUP BY category;
```

which returns this output:

```
+-----+-----+
| category | COUNT(author) |
+-----+-----+
| Classic Fiction | 3 |
| Nonfiction | 1 |
| Play | 1 |
+-----+-----+
3 rows in set (0.00 sec)
```

Joining Tables

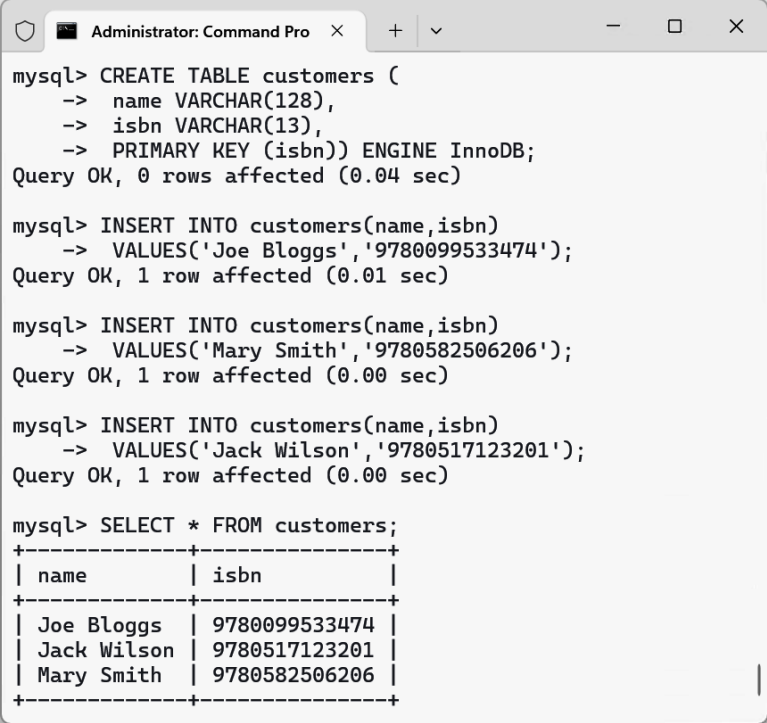
It is quite normal to maintain multiple tables within a database, each holding a different type of information. For example, consider the case of a *customers* table that needs to be able to be cross-referenced with publications purchased from the *classics* table. Enter the commands in [Example 8-28](#) to create this new table and populate it with three customers and their purchases. [Figure 8-17](#) shows the result.



Joining tables is a really big topic that we're going to cover very quickly here. In [Chapter 9](#), you'll also learn more about database design, which includes a process called *database normalization*.

Example 8-28. Creating and populating the customers table

```
CREATE TABLE customers (  
  name VARCHAR(128),  
  isbn VARCHAR(13),  
  PRIMARY KEY (isbn)) ENGINE InnoDB;  
INSERT INTO customers(name,isbn)  
  VALUES('Joe Bloggs','9780099533474');  
INSERT INTO customers(name,isbn)  
  VALUES('Mary Smith','9780582506206');  
INSERT INTO customers(name,isbn)  
  VALUES('Jack Wilson','9780517123201');  
SELECT * FROM customers;
```



```
Administrator: Command Pro  x  +  -  □  X  
mysql> CREATE TABLE customers (  
-> name VARCHAR(128),  
-> isbn VARCHAR(13),  
-> PRIMARY KEY (isbn)) ENGINE InnoDB;  
Query OK, 0 rows affected (0.04 sec)  
  
mysql> INSERT INTO customers(name,isbn)  
-> VALUES('Joe Bloggs','9780099533474');  
Query OK, 1 row affected (0.01 sec)  
  
mysql> INSERT INTO customers(name,isbn)  
-> VALUES('Mary Smith','9780582506206');  
Query OK, 1 row affected (0.00 sec)  
  
mysql> INSERT INTO customers(name,isbn)  
-> VALUES('Jack Wilson','9780517123201');  
Query OK, 1 row affected (0.00 sec)  
  
mysql> SELECT * FROM customers;  
+-----+-----+  
| name      | isbn      |  
+-----+-----+  
| Joe Bloggs | 9780099533474 |  
| Jack Wilson | 9780517123201 |  
| Mary Smith | 9780582506206 |  
+-----+-----+
```

Figure 8-17. Creating the customers table



There's also a shortcut for inserting multiple rows of data, as in [Example 8-28](#), in which you can replace the three separate INSERT INTO queries with a single one listing the data to be inserted, separated by commas, like this:

```
INSERT INTO customers(name,isbn) VALUES
('Joe Bloggs','9780099533474'),
('Mary Smith','9780582506206'),
('Jack Wilson','9780517123201');
```

Of course, in a proper table containing customers' details there also would be addresses, phone numbers, email addresses, and so on, but they aren't necessary for this explanation. While creating the new table, you should have noticed that it has something in common with the *classics* table: a column called *isbn*. Because it has the same meaning in both tables (an ISBN refers to a book, and always the same book), we can use this column to tie the two tables together into a single query, as in [Example 8-29](#).

Example 8-29. Joining two tables into a single SELECT

```
SELECT name,author,title FROM customers,classics
WHERE customers.isbn=classics.isbn;
```

The result of this operation is:

```
+-----+-----+-----+
| name      | author      | title      |
+-----+-----+-----+
| Joe Bloggs | Charles Dickens | The Old Curiosity Shop |
| Mary Smith | Jane Austen    | Pride and Prejudice    |
| Jack Wilson | Charles Darwin | The Origin of Species  |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

See how this query has neatly linked the tables to show the publications purchased from the *classics* table by the people in the *customers* table?

NATURAL JOIN

Using NATURAL JOIN, you can save yourself some typing and make queries a little clearer. This kind of join takes two tables and automatically joins columns that have the same name. So, to achieve the same results as from [Example 8-29](#), you would enter:

```
SELECT name,author,title FROM customers NATURAL JOIN classics;
```

JOIN...ON

If you wish to specify the column on which to join two tables, use the JOIN...ON construct, as follows, to achieve results identical to those of [Example 8-29](#):

```
SELECT name,author,title FROM customers
JOIN classics ON customers.isbn=classics.isbn;
```

Using AS

You can also save yourself some typing and improve query readability by creating aliases using the AS keyword. Simply follow a table name with AS and the alias to use. The following code, therefore, is also identical in action to [Example 8-29](#):

```
SELECT name,author,title FROM
customers AS cust, classics AS cclass WHERE cust.isbn=cclass.isbn;
```

The result of this operation is:

```
+-----+-----+-----+
| name      | author      | title      |
+-----+-----+-----+
| Joe Bloggs | Charles Dickens | The Old Curiosity Shop |
| Mary Smith | Jane Austen   | Pride and Prejudice    |
| Jack Wilson | Charles Darwin | The Origin of Species   |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

You can also use AS to assign an alias to a column for the current query (whether or not joining tables), like this:

```
SELECT name AS customer FROM customers ORDER BY customer;
```

which results in the output:

```
+-----+
| customer |
+-----+
| Jack Wilson |
| Joe Bloggs  |
| Mary Smith  |
+-----+
3 rows in set (0.00 sec)
```

Aliases can be particularly useful when you have long queries that reference the same table names many times.

Using Logical Operators

You can also use the logical operators AND, OR, and NOT in your MySQL WHERE queries to further narrow your selections. [Example 8-30](#) shows one instance of each, but you can mix and match them in any way you need.

Example 8-30. Using logical operators

```
SELECT author,title FROM classics WHERE
  author LIKE "Charles%" AND author LIKE "%Darwin";
SELECT author,title FROM classics WHERE
  author LIKE "%Mark Twain%" OR author LIKE "%Samuel Langhorne Clemens%";
SELECT author,title FROM classics WHERE
  author LIKE "Charles%" AND author NOT LIKE "%Darwin";
```

I've chosen the first query because Charles Darwin might be listed in some rows by his full name, Charles Robert Darwin. The query returns any publications for which the *author* column starts with *Charles* and ends with *Darwin*. The second query searches for publications written using either Mark Twain's pen name or his real name, Samuel Langhorne Clemens. The third query returns publications written by authors with the first name Charles but not the surname Darwin.

MySQL Functions

You might wonder why anyone would want to use MySQL functions when PHP comes with a whole bunch of powerful functions of its own. The answer is very simple: the MySQL functions work on the data right there in the database. If you were to use PHP, you would first have to extract raw data from MySQL, manipulate it, and then perform the database query you wanted.

Having functions built into MySQL substantially reduces the time needed for performing complex queries, as well as their complexity. You can learn more about all the available **string** and **date/time** functions from the documentation.

Accessing MySQL via phpMyAdmin

Although to use MySQL you have to learn these main commands and how they work, once you understand them, it can be much quicker and simpler to use a program such as *phpMyAdmin* to manage your databases and tables.

To do this, assuming you have installed AMPPS as described in **Chapter 2**, type the following to open the program (see **Figure 8-18**):

```
http://localhost/phpmyadmin
```

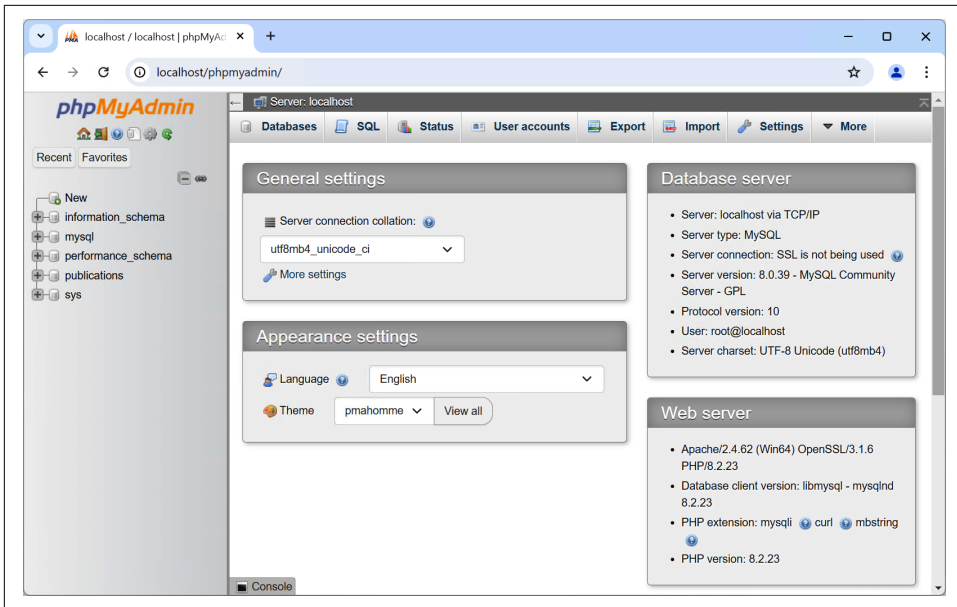


Figure 8-18. The phpMyAdmin main screen

If prompted (and you haven't changed them), the default login and password to enter are *root* and *mysql*.

In the lefthand pane of the main phpMyAdmin screen, you can click to select any tables you wish to work with (although none will be available until created). You can also click *New* to create a new database.

From here, you can perform all the main operations, such as creating new databases, adding tables, creating indexes, and much more. To find out more about phpMyAdmin, consult the [documentation](#).

If you worked with me through the examples in this chapter, congratulations—it has been quite a journey. You've come all the way from learning how to create a MySQL database, through issuing complex queries that combine multiple tables, to using Boolean operators and leveraging MySQL's various qualifiers.

In [Chapter 9](#), we'll start looking at how to approach efficient database design, advanced SQL techniques, and MySQL functions and transactions, but first test your knowledge of what you have learned in this chapter with the following questions.

Questions

1. What is the purpose of the semicolon in MySQL queries?
2. Which command would you use to view the available databases or tables?
3. How would you create a new MySQL user on the local host called *newuser* with a password of *newpass* and with access to everything in the database *newdatabase*?
4. How can you view the structure of a table?
5. What is the purpose of a MySQL index?
6. What benefit does a FULLTEXT index provide?
7. What is a stopword?
8. Both SELECT DISTINCT and GROUP BY cause the display to show only one output row for each value in a column, even if multiple rows contain that value. What are the main differences between SELECT DISTINCT and GROUP BY?
9. Using the SELECT...WHERE construct, how would you return only rows containing the word *Langhorne* somewhere in the *author* column of the *classics* table used in this chapter?
10. What needs to be defined in two tables to make it possible for you to join them together?

See “Chapter 8 Answers” on page 573 in the [Appendix](#) for the answers to these questions.

Mastering MySQL

Chapter 8 provided you with a good grounding in the practice of using relational databases with the Structured Query Language. You've learned about creating databases and the tables they comprise, as well as inserting, looking up, changing, and deleting data.

Next, we now need to look at how to design databases for maximum speed and efficiency. For example, how do you decide what data to place in which table? Over the years, a number of guidelines have been developed that—if you follow them—ensure that your databases will be efficient and capable of growing as you feed them more and more data.

Database Design

It's very important that you design a database correctly before you start to create it; otherwise, you are almost certainly going to have to go back and change it by splitting up some tables, merging others, and moving various columns to achieve sensible relationships that MySQL can easily use.

Sitting down with a sheet of paper and a pencil (or a digital equivalent) and writing a selection of the queries that you think you and your users are likely to ask is an excellent starting point. In the case of an online bookstore's database, some of your questions could be:

- How many authors, books, and customers are in the database?
- Which author wrote a certain book?
- Which books were written by a certain author?
- What is the most expensive book?

- What is the best-selling book?
- Which books have not sold this year?
- Which books did a certain customer buy?
- Which books have been purchased at the same time as other books?

Of course, you could make many more queries on such a database, but even this small sample will begin to give you insights into how to lay out your tables. For example, books and ISBNs can probably be combined into one table, because they are closely linked (we'll examine some of the subtleties later). In contrast, books and customers should be in separate tables, because their connection is very loose. A customer can buy any book, and even multiple copies of a book, yet a book can be bought by many customers and be ignored by still more potential customers.

When you plan to do a lot of searches on something, a search can often benefit from having its own table. And when couplings between things are loose, it's best to put them in separate tables.

Taking into account those simple rules, we can guess we'll need at least three tables to accommodate all these queries:

Authors

There will be lots of searches for authors, many of whom have collaborated on titles, and many of whom will be featured in collections. Listing all the information about each author together, linked to that author, will produce optimal results for searches—hence an *Authors* table.

Books

Many books appear in different editions. Sometimes they change publisher, and sometimes they have the same titles as other unrelated books. So, the links between books and authors are complicated enough to call for a separate table.

Customers

It's even more clear why customers should get their own table, as they are free to purchase any book by any author.

Primary Keys: The Keys to Relational Databases

Using the power of relational databases, we can define information for each author, book, and customer in just one place. Obviously, what interests us is the links between them—such as who wrote each book and who purchased it—but we can store that information just by making links between the three tables. I'll show you the basic principles, and then it just takes practice for it to feel natural.

The magic involves giving every author a unique identifier. We'll do the same for every book and for every customer. We saw the means of doing that in [Chapter 8](#): the *primary key*. For a book, it makes sense to use the ISBN, although you then have to deal with multiple editions that have different ISBNs. For authors and customers, you can just assign arbitrary keys, which the `AUTO_INCREMENT` feature that you saw in the last chapter makes easy.

In short, every table will be designed around some object that you're likely to search for a lot—an author, book, or customer, in this case—and that object will have a primary key. Don't choose a key that could possibly have the same value for different objects. The ISBN is a rare case for which an industry has provided a primary key that you can rely on to be unique for each product. Most of the time, you'll create an arbitrary key for this purpose, using `AUTO_INCREMENT`.



Globally Unique Identifiers

Some databases and tables may also use globally unique identifiers called UUIDs (Universally Unique Identifiers) or GUIDs (Globally Unique Identifiers), a 128-bit label usually formatted as `189aa781-5d03-11ef-ac40-00155d7f2216`. They take up more storage space and may negatively impact performance if not used correctly and thus are not common in MySQL databases.

Normalization

The process of separating your data into tables and creating primary keys is called *normalization*. Its main goal is to make sure each piece of information appears in the database only once. The presence of duplicates creates a strong risk that you'll update only one row of duplicated data, creating inconsistencies in a database and potentially causing serious errors.

For example, if you list the titles of books in the *Authors* table as well as the *Books* table, and you have to correct a typographic error in a title, you'll have to search through both tables and make sure you make the same change every place the title is listed. It's better to keep the title in one place and use the ISBN in other places.

But in the process of splitting a database into multiple tables, it's important not to go too far and create more tables than is necessary, which would also lead to inefficient design and slower access.

Luckily, E. F. Codd, the inventor of the relational model, analyzed the concept of normalization and came up with a series of so-called normal forms: *First*, *Second*, and *Third Normal Form*. If you modify a database to satisfy each of these forms in order, you will ensure that your database is optimally balanced for fast access and minimum memory and disk space usage.

To see how the normalization process works, let's start with the rather monstrous database in **Table 9-1**, which shows a single table containing all of the author names, book titles, and (fictional) customer details. You could consider it a first attempt at a table intended to keep track of which customers have ordered books. Obviously this is an inefficient design, because data is duplicated all over the place (duplications are highlighted in bold), but it represents a starting point. Also, to make this and the following tables less cluttered, ISBN-10 numbers are used in place of ISBN-13.

Table 9-1. A highly inefficient design for a database table

Author 1	Author 2	Title	ISBN	Price	Customer name	Customer address	Purchase date
David Sklar	Adam Trachtenberg	PHP Cookbook	0596101015	44.99	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Danny Goodman		Dynamic HTML	0596527403	59.99	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
Hugh E. Williams	David Lane	PHP and MySQL	0596005436	44.95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
David Sklar	Adam Trachtenberg	PHP Cookbook	0596101015	44.99	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
Rasmus Lerdorf	Kevin Tatroe & Peter MacIntyre	Programming PHP	0596006815	39.99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

In the following three sections, we will examine this database design, and you'll see how we can improve it by removing the various duplicate entries and splitting the single table into multiple tables, each containing one type of data.

First Normal Form

For a database to satisfy the *First Normal Form*, it must fulfill three requirements:

- There should be no repeating columns containing the same kind of data.
- All columns should contain a single value.
- There should be a primary key to uniquely identify each row.

Looking at these requirements in order, you should notice straightaway that both the *Author 1* and *Author 2* columns constitute repeating data types. So we already have a target column for pulling into a separate table, as the repeated *Author* columns violate Rule 1.

Second, there are three authors listed for the final book, *Programming PHP*. I've handled that by making Kevin Tatroe and Peter MacIntyre share the *Author 2* column, which violates Rule 2—yet another reason to transfer the *Author* details to a separate table.

However, Rule 3 is satisfied, because the primary key of ISBN has already been created.

Table 9-2 shows the result of removing the *Author* columns from **Table 9-1**. Already it looks a lot less cluttered, although duplications remain (highlighted in bold).

Table 9-2. The result of stripping the Author columns from Table 9-1

Title	ISBN	Price	Customer name	Customer address	Purchase date
PHP Cookbook	0596101015	44.99	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Dynamic HTML	0596527403	59.99	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
PHP and MySQL	0596005436	44.95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
PHP Cookbook	0596101015	44.99	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
Programming PHP	0596006815	39.99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

The new *Authors* table shown in **Table 9-3** is small and simple. It just lists the ISBN of a title along with an author. If a title has more than one author, additional authors get their own rows. At first, you may feel ill at ease with this table, because you can't tell which author wrote which book. But don't worry: MySQL can quickly tell you. All you have to do is tell it which book you want information for, and MySQL will use its ISBN to search the *Authors* table in a matter of milliseconds.

Table 9-3. The new Authors table

ISBN	Author
0596101015	David Sklar
0596101015	Adam Trachtenberg
0596527403	Danny Goodman
0596005436	Hugh E. Williams
0596005436	David Lane
0596006815	Rasmus Lerdorf
0596006815	Kevin Tatroe
0596006815	Peter MacIntyre

As I mentioned earlier, the ISBN will be the primary key for the *Books* table, when we get around to creating that table. I mention that here to emphasize that the ISBN is not, however, the primary key for the *Authors* table. In the real world, the *Authors* table would deserve a primary key, too, so that each author would have a key to uniquely identify them.

So, in the *Authors* table, *ISBN* is just a column that—for the purposes of speeding up searches—we’ll probably make a key, but not the primary key. In fact, it *cannot* be the primary key in this table because it’s not unique: the same *ISBN* appears multiple times whenever two or more authors have collaborated on a book.

Because we’ll use it to link authors to books in another table, this column is called a *foreign key*.



Keys (also called *indexes*) have several purposes in MySQL. The fundamental reason for defining a key is to make searches faster. You’ve seen examples in [Chapter 8](#) in which keys are used in `WHERE` clauses for searching. But a key can also be useful to uniquely identify an item. Thus, a unique key is often used as a primary key in one table and as a foreign key to link rows in that table to rows in another table.

Second Normal Form

The First Normal Form deals with duplicate data (or redundancy) across multiple columns. The *Second Normal Form* is all about redundancy across multiple rows. To achieve Second Normal Form, your tables must already be in First Normal Form. Once this has been done, you achieve Second Normal Form by identifying columns whose data repeats in different places and then removing them to their own tables.

So, let’s look again at [Table 9-2](#). Notice how Darren Ryder bought two books, and therefore his details are duplicated. This tells us that the *Customer* columns need to be pulled into their own table. [Table 9-4](#) shows the result of removing the *Customer* columns from [Table 9-2](#).

Table 9-4. The new *Titles* table

ISBN	Title	Price
0596101015	PHP Cookbook	44.99
0596527403	Dynamic HTML	59.99
0596005436	PHP and MySQL	44.95
0596006815	Programming PHP	39.99

As you can see, all that’s left in [Table 9-4](#) are the *ISBN*, *Title*, and *Price* columns for four unique books, so this now constitutes an efficient and self-contained table that satisfies the requirements of both the First and Second Normal Forms. Along the way, we’ve managed to reduce the information to data closely related to book titles. This table could also include years of publication, page counts, numbers of reprints, and so on, as these details are also closely related. The only rule is that we can’t put in any column that could have multiple values for a single book, because then we’d have

to list the same book in multiple rows and would thus violate Second Normal Form. Restoring an *Author* column, for instance, would violate this normalization.

However, looking at the extracted *Customer* columns, now in [Table 9-5](#), we can see that there's more normalization work to do, because Darren Ryder's details are still duplicated. And it could also be argued that First Normal Form Rule 2 (all columns should contain a single value) has not been properly complied with, because the addresses really need to be broken into separate columns for *Address*, *City*, *State*, and *Zip*.

Table 9-5. The customer details from [Table 9-2](#)

ISBN	Customer name	Customer address	Purchase date
0596101015	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
0596527403	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
0596005436	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
0596101015	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
0596006815	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

What we have to do is split this table further to ensure that each customer's details are entered only once. Because the ISBN is not and cannot be used as a primary key to identify customers (or authors), a new key must be created.

[Table 9-6](#) is the result of normalizing the *Customers* table into both First and Second Normal Forms. Each customer now has a unique customer number called *CustNo* that is the table's primary key and that will most likely have been created via `AUTO_INCREMENT`. All the parts of customer addresses have also been separated into distinct columns to make them easily searchable and updatable.

Table 9-6. The new *Customers* table

CustNo	Name	Address	City	State	Zip
1	Emma Brown	1565 Rainbow Road	Los Angeles	CA	90014
2	Darren Ryder	4758 Emily Drive	Richmond	VA	23219
3	Earl B. Thurston	862 Gregory Lane	Frankfort	KY	40601
4	David Miller	3647 Cedar Lane	Waltham	MA	02154

At the same time, to normalize [Table 9-6](#), we had to remove the information on customer purchases, because otherwise there would be multiple instances of customer details for each book purchased. Instead, the purchase data is now placed in a new table called *Purchases* (see [Table 9-7](#)).

Table 9-7. The new Purchases table

CustNo	ISBN	Date
1	0596101015	Mar 03 2009
2	0596527403	Dec 19 2008
2	0596101015	Dec 19 2008
3	0596005436	Jun 22 2009
4	0596006815	Jan 16 2009

Here the *CustNo* column from Table 9-6 is reused as a key to tie the *Customers* and *Purchases* tables together. Because the *ISBN* column is also repeated here, this table can be linked with the *Authors* and *Titles* tables, too.

The *CustNo* column may be a useful key in the *Purchases* table, but it's not a primary key. A single customer can buy multiple books (and even multiple copies of one book), so the *CustNo* column is not a primary key. In fact, the *Purchases* table has no primary key. That's all right, because we don't expect to need to keep track of unique purchases. If one customer buys two copies of the same book on the same day, we'll just allow two rows with the same information. For easy searching, we can define both *CustNo* and *ISBN* as keys—just not as primary keys.



There are now four tables, one more than the three we had initially assumed would be needed. We arrived at this decision through the normalization process, by methodically following the First and Second Normal Form rules, which made it plain that a fourth table called *Purchases* would also be required.

The tables we now have are *Authors* (Table 9-3), *Titles* (Table 9-4), *Customers* (Table 9-6), and *Purchases* (Table 9-7), and we can link each table to any other using either the *CustNo* or the *ISBN* key.

For example, to see which books Darren Ryder has purchased, you can look him up in Table 9-6, the *Customers* table, where you will see his *CustNo* is 2. Armed with this number, you can now go to Table 9-7, the *Purchases* table; looking at the *ISBN* column here, you will see that he purchased titles 0596527403 and 0596101015 on December 19, 2008. This looks like a lot of trouble for a human, but it's not so hard for MySQL.

To determine what these titles were, you can then refer to Table 9-4, the *Titles* table, and see that the books he bought were *Dynamic HTML* and *PHP Cookbook*. Should you wish to know the authors of these books, you could also use the ISBNs you just looked up on Table 9-3, the *Authors* table, and you would see that ISBN 0596527403, *Dynamic HTML*, was written by Danny Goodman, and that ISBN 0596101015, *PHP Cookbook*, was written by David Sklar and Adam Trachtenberg.

Third Normal Form

Once you have a database that complies with both the First and Second Normal Forms, it is in pretty good shape, and you might not have to modify it any further. However, if you wish to be very strict with your database, you can ensure that it adheres to the *Third Normal Form*, which requires moving data that is *not* directly dependent on the primary key but *is* dependent on another value in the table. Such data should be moved into separate tables, according to the dependence.

For example, in [Table 9-6](#), the *Customers* table, it could be argued that the *State*, *City*, and *Zip* keys are not directly related to each customer, because many other people will have the same details in their addresses, too. However, they are directly related to each other, in that the *Address* relies on the *City*, and the *City* relies on the *State*.

Therefore, to satisfy Third Normal Form for [Table 9-6](#), you would need to split it into [Tables 9-8 through 9-11](#).

Table 9-8. Third Normal Form Customers table

CustNo	Name	Address	Zip
1	Emma Brown	1565 Rainbow Road	90014
2	Darren Ryder	4758 Emily Drive	23219
3	Earl B. Thurston	862 Gregory Lane	40601
4	David Miller	3647 Cedar Lane	02154

Table 9-9. Third Normal Form Zip codes table

Zip	CityID
90014	1234
23219	5678
40601	4321
02154	8765

Table 9-10. Third Normal Form Cities table

CityID	Name	StateID
1234	Los Angeles	5
5678	Richmond	46
4321	Frankfort	17
8765	Waltham	21

Table 9-11. Third Normal Form States table

StateID	Name	Abbreviation
5	California	CA
46	Virginia	VA
17	Kentucky	KY
21	Massachusetts	MA

So, how would you use this set of four tables instead of the single Table 9-6? Well, you would look up the *Zip code* in Table 9-8 and then find the matching *CityID* in Table 9-9. Given this information, you could look up the city *Name* in Table 9-10 and then also find the *StateID*, which you could use in Table 9-11 to look up the state's *Name*.

Although using the Third Normal Form in this way may seem like overkill, it can have advantages. For example, look at Table 9-11, where it has been possible to include both a state's name and its two-letter abbreviation. It could also contain population details and other demographics, if you desired.



Table 9-10 could also contain even more localized demographics that could be useful to you and/or your customers. By splitting up these pieces of data, you can make it easier to maintain your database in the future, should it be necessary to add columns.

Deciding whether to use the Third Normal Form can be tricky. Your evaluation should rest on what data you may need to add at a later date. If you are absolutely certain that the name and address of a customer is all that you will ever require, you probably will want to leave out this final normalization stage.

On the other hand, suppose you are writing a database for a large organization such as the US Postal Service. What would you do if a city were to be renamed? With a table such as Table 9-6, you would need to perform a global search-and-replace on every instance of that city. But if you have your database set up according to the Third Normal Form, you would have to change only a single entry in Table 9-10 for the change to be reflected throughout the entire database.

Therefore, I suggest that you ask yourself two questions to help you decide whether to perform a Third Normal Form normalization on any table:

- Is it likely that many new columns will need to be added to this table?
- Could any of this table's fields require a global update at any point?

If either of the answers is yes, you should consider performing this final stage of normalization.

When Not to Use Normalization

Now that you know all about normalization, I'm going to tell you why you should not implement these rules on high-traffic sites. That's right—you should never fully normalize your tables on sites that will cause MySQL to thrash.

Normalization generally requires spreading data across multiple tables, and multi-table queries and joins are, by nature, less efficient than single-table queries, and that might produce challenges at scale. On a very popular site, if you have normalized tables, your database access will slow down considerably once you get above a few dozen concurrent users, because they will be creating hundreds of database accesses between them. In fact, I would go so far as to say you should *denormalize* any commonly looked-up data as much as you can.

You see, if you have data duplicated across your tables, you can substantially reduce the number of additional requests that need to be made, because most of the data you want is available in each table. This means that you can simply add an extra column to a query and that field will be available for all matching results.

Of course, you have to deal with the downsides previously mentioned, such as using large amounts of disk space and ensuring that you update every single duplicate copy of the data when it needs modifying.

Multiple updates can be computerized, though. MySQL provides a feature called *triggers* that make automatic changes to the database in response to changes you make. (Triggers are, however, beyond the scope of this book.) Another way to propagate redundant data is to set up a PHP program to run regularly and keep all copies in sync. The program reads changes from a “main” table and updates all the others. (You'll see how to access MySQL from PHP in [Chapter 10](#).)

However, until you are very experienced with MySQL, I recommend that you fully normalize all your tables (at least to First and Second Normal Form), as this will instill the habit and help you create effective structure. Only when you start to see MySQL logjams should you consider looking at denormalization.

Relationships

MySQL is called a *relational* database management system because its tables store not only data but also the *relationships* among the data. There are three categories of relationships.

One-to-One

A *one-to-one relationship* is like a monogamous marriage: each item has a relationship to only one item of the other type. This is surprisingly rare. For instance, an author can write multiple books, a book can have multiple authors, and even an address can be associated with multiple customers. The best example of a one-to-one relationship in this chapter thus far is the relationship between the name of a state and its two-character abbreviation.

However, for the sake of argument, let's assume there can always be only one customer at any address. In such a case, the Customers–Addresses relationship in **Figure 9-1** is a one-to-one relationship: only one customer lives at each address, and each address can have only one customer.

Table 9-8a (Customers)		Table 9-8b (Addresses)	
CustNo	Name	Address	Zip
1	Emma Brown	1565 Rainbow Road	90014
2	Darren Ryder	4758 Emily Drive	23219
3	Earl B. Thurston	862 Gregory Lane	40601
4	David Miller	3647 Cedar Lane	02154

Figure 9-1. The Customers table, **Table 9-8**, split into two tables

Usually, when two items have a one-to-one relationship, you just include them as columns in the same table. There are two reasons for splitting them into separate tables:

- You want to be prepared in case the relationship changes later and is no longer one-to-one.
- The table has a lot of columns, and you think that performance or maintenance would be improved by splitting it.

Of course, when you build your own databases in the real world, you will have to create one-to-many Customer–Address relationships (*one* address, *many* customers).

One-to-Many

One-to-many (or many-to-one) relationships occur when one row in one table is linked to many rows in another table. You have already seen how **Table 9-8** would take on a one-to-many relationship if multiple customers were allowed at the same address, which is why it would have to be split up if that were the case.

So, looking at Table 9-8a within Figure 9-1, you can see that it shares a one-to-many relationship with Table 9-7 because there is only one of each customer in Table 9-8a. However Table 9-7, the *Purchases* table, can (and does) contain more than one purchase from a given customer. Therefore, *one* customer has a relationship with *many* purchases.

You can see these two tables alongside each other in Figure 9-2, where the dashed lines joining rows in each table start from a single row in the lefthand table but can connect to more than one row in the righthand table. This one-to-many relationship is also the preferred scheme to use when describing a many-to-one relationship, in which case you would normally swap the left and right tables to view them as a one-to-many relationship.

Table 9-8a (Customers)			Table 9-7. (Purchases)		
CustNo	Name		CustNo	ISBN	Date
1	Emma Brown	 1	1	0596101015	Mar 03 2009
2	Darren Ryder	 2	2	0596527403	Dec 19 2008
	(etc...)	 2	2	0596101015	Dec 19 2008
3	Earl B. Thurston	 3	3	0596005436	Jun 22 2009
4	David Miller	 4	4	0596006815	Jan 16 2009

Figure 9-2. Illustrating the relationship between two tables

To represent a one-to-many relationship in a relational database, create a table for the “many” and a table for the “one.” The table for the “many” must contain a column that lists the primary key from the “one” table. Thus, the *Purchases* table will contain a column that lists the primary key of the customer.

Many-to-Many

In a *many-to-many relationship*, many rows in one table are linked to many rows in another table. To create this relationship, add a third table containing the same key column from each of the other tables. This third table contains nothing else, as its sole purpose is to link the other tables.

Table 9-12 is such a table. It was extracted from Table 9-7, the *Purchases* table but omits the purchase date information. It contains a copy of the ISBN of every title sold, along with the customer number of each purchaser.

Table 9-12. An intermediary table

CustNo	ISBN
1	0596101015
2	0596527403
2	0596101015
3	0596005436
4	0596006815

With this intermediary table in place, you can traverse all the information in the database through a series of relations. You can take an address as a starting point and find out the authors of any books purchased by the customer living at that address.

For example, let's suppose that you want to find out about purchases in the 23219 zip code. Look up that zip code in Table 9-8b, and you'll find that customer number 2 has bought at least one item from the database. At this point, you can use Table 9-8a to find out that customer's name or use the new intermediary Table 9-12 to see the book(s) purchased.

From here, you can see that two titles were purchased and follow them back to Table 9-4 to find the titles and prices of these books or to Table 9-3 to see the authors.

If you're thinking this is really combining multiple one-to-many relationships, you are absolutely correct. To illustrate, Figure 9-3 brings three tables together.

Columns from Table 9-8 (Customers)		Intermediary Table 9-12 (Customer/ISBN)		Columns from Table 9-4 (Titles)	
Zip	CustNo	CustNo	ISBN	ISBN	Title
90014	1	1	0596101015	0596101015	PHP Cookbook
23219	2	2	0596101015	(etc...)	
(etc...)		2	0596527403	0596527403	Dynamic HTML
40601	3	3	0596005436	0596005436	PHP and MySQL
02154	4	4	0596006815	0596006815	Programming PHP

Figure 9-3. Creating a many-to-many relationship via a third table

Follow any zip code in the lefthand table to the associated customer IDs. From there, you can link to the middle table, which joins the left and right tables by linking customer IDs and ISBNs. Now all you have to do is follow an ISBN over to the righthand table to see which book it relates to.

You can also use the intermediary table to work your way backward from book titles to zip codes. The *Titles* table can tell you the ISBN, which you can use in the middle

table to find the ID numbers of customers who bought the book, and finally you can use the *Customers* table to match the customer ID numbers to the customers' zip codes.

Databases and Privacy

An interesting aspect of using relations is that you can accumulate a lot of information about some item—such as a customer—without actually knowing who that customer is. Note that in the previous example we went from customers' zip codes to customers' purchases, and back again, without knowing the name of a customer. Databases can be used to track people, but they also can be used to help preserve people's privacy while still finding useful information, by returning information about a purchase without revealing other customer details, for example.

Transactions

In some applications, it is vitally important that a sequence of queries runs in the correct order and that every single query successfully completes. For example, suppose you are creating a sequence of queries to transfer funds from one bank account to another. You would not want either of the following events to occur:

- You add the funds to the second account, but when you try to subtract them from the first account, the update fails, and now both accounts have the funds.
- You subtract the funds from the first bank account, but the update request to add them to the second account fails, and the funds have disappeared into thin air.

As you can see, not only is it important how you order queries in this type of transaction but it is also vital that all parts of the transaction complete successfully. But how can you ensure this happens, because surely after a query has occurred, it cannot be undone? Do you have to keep track of all parts of a transaction and then undo them all one at a time if any one fails? The answer is absolutely not, because MySQL comes with powerful transaction-handling features to cover just these types of eventualities.

In addition, transactions allow concurrent access to a database by many users or programs at the same time. MySQL handles this seamlessly by ensuring that all transactions are queued and that users or programs take their turns and don't tread on each other's toes.

Transaction Storage Engines

To be able to use MySQL's transaction facility, you have to be using MySQL's InnoDB storage engine (which is the default from version 5.5 onward). If you are not sure

which version of MySQL your code will be running on, rather than assuming InnoDB is the default engine, you can force its use when creating a table, as follows.

Create a table of bank accounts by typing the commands in [Example 9-1](#). (Remember that to do this, you will need access to the MySQL command line or a graphical tool, and you must also have already selected a suitable database in which to create this table.)

Example 9-1. Creating a transaction-ready table

```
CREATE TABLE accounts (  
  number INT, balance INT, PRIMARY KEY(number)  
) ENGINE InnoDB;  
DESCRIBE accounts;
```

The final line of this example displays the contents of the new table so you can ensure that it was correctly created. The output from it should look like this:

```
+-----+-----+-----+-----+-----+-----+  
| Field | Type | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+-----+  
| number | int(11) | NO | PRI | NULL | |  
| balance | int(11) | YES | | NULL | |  
+-----+-----+-----+-----+-----+-----+  
2 rows in set (0.00 sec)
```

Now let's create two rows within the table so that you can practice using transactions. Type the commands in [Example 9-2](#).

Example 9-2. Populating the accounts table

```
INSERT INTO accounts(number, balance) VALUES(12345, 1025);  
INSERT INTO accounts(number, balance) VALUES(67890, 140);  
SELECT * FROM accounts;
```

The third line displays the contents of the table to confirm that the rows were correctly inserted. The output should look like this:

```
+-----+-----+  
| number | balance |  
+-----+-----+  
| 12345 | 1025 |  
| 67890 | 140 |  
+-----+-----+  
2 rows in set (0.00 sec)
```

With this table created and prepopulated, you are ready to start using transactions.

Using START TRANSACTION

Transactions in MySQL start with either a `START TRANSACTION` or a `BEGIN` statement, where the former is a standard SQL syntax. Type the commands in [Example 9-3](#) to send a transaction to MySQL.

Example 9-3. A MySQL transaction

```
START TRANSACTION;
UPDATE accounts SET balance=balance+25 WHERE number=12345;
COMMIT;
SELECT * FROM accounts;
```

The result of this transaction is displayed by the final line and should look like this:

```
+-----+-----+
| number | balance |
+-----+-----+
|  12345 |    1050 |
|  67890 |     140 |
+-----+-----+
2 rows in set (0.00 sec)
```

As you can see, the balance of account number 12345 was increased by 25 and is now 1050. You also may have noticed the `COMMIT` command in [Example 9-3](#), which is explained next.

Using COMMIT

When you are satisfied that a series of queries in a transaction has successfully completed, issue a `COMMIT` command to commit all the changes to the database. Until it receives a `COMMIT`, MySQL considers all the changes you make to be temporary. This feature gives you the opportunity to cancel a transaction by not sending a `COMMIT` but issuing a `ROLLBACK` command instead.

Using ROLLBACK

Using the `ROLLBACK` command, you can tell MySQL to forget all the queries made since the start of a transaction and to cancel the transaction. See this in action by entering the funds transfer transaction in [Example 9-4](#).

Example 9-4. A funds transfer transaction

```
START TRANSACTION;
UPDATE accounts SET balance=balance-250 WHERE number=12345;
UPDATE accounts SET balance=balance+250 WHERE number=67890;
SELECT * FROM accounts;
```

Once you have entered these lines, you should see this result:

```
+-----+-----+
| number | balance |
+-----+-----+
|  12345 |      800 |
|  67890 |      390 |
+-----+-----+
2 rows in set (0.00 sec)
```

The first bank account now has a value that is 250 less than before, and the second has been incremented by 250; you have transferred a value of 250 between them. But let's assume that something went wrong and you wish to undo this transaction. All you have to do is issue the commands in [Example 9-5](#).

Example 9-5. Canceling a transaction using ROLLBACK

```
ROLLBACK;
SELECT * FROM accounts;
```

You should now see the following output, showing that the two accounts have had their previous balances restored, due to the entire transaction being canceled via the ROLLBACK command:

```
+-----+-----+
| number | balance |
+-----+-----+
|  12345 |     1050 |
|  67890 |       140 |
+-----+-----+
2 rows in set (0.00 sec)
```

Using EXPLAIN

MySQL comes with a powerful tool for investigating how the queries you issue to it are interpreted. Using EXPLAIN, you can get a snapshot of any query to find out whether you could issue it in a better or more efficient way. [Example 9-6](#) shows how to use this command with the *accounts* table you created earlier.

Example 9-6. Using the EXPLAIN command

```
EXPLAIN SELECT * FROM accounts WHERE number='12345';
```

The results of this EXPLAIN command should look like:

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select|table  |part- |type |possible|key   |key |ref |rows|fil- |Extra|
| |_type |        |itions|     |_keys  |      |_len|    |    |tered|     |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1 |SIMPLE|accounts|NULL  |const|PRIMARY |PRIMARY|4   |const|1   |100.00|NULL |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Here is an explanation of the information that MySQL is giving you:

select_type

The selection type is SIMPLE. If you were joining tables together, this would show the join type.

table

The current table being queried is accounts.

type

The query type is const. From the least efficient to the most efficient type, the possible values can be ALL, index, range, ref, eq_ref, const, system, and NULL.

possible_keys

There is a possible PRIMARY key, which means that accessing should be fast.

key

The key actually used is PRIMARY. This is good.

key_len

The key length is 4. This is the number of bytes of the index that MySQL will use.

ref

The ref column displays which columns or constants are used with the key. In this case, a constant key is being used.

rows

The number of rows that need to be searched by this query is 1. This is good.



Partitions

Partitioning allows you to store parts of individual tables in separate files called partitions, used to store more data in one table than one disk can handle.

Whenever you have a query that seems to be taking longer to execute than you think it should, try using `EXPLAIN` to see where you can optimize it. You will discover which keys (if any) are being used, their lengths, and so on, and you will be able to adjust your query or the design of your table(s) accordingly.



When you have finished experimenting with the temporary *accounts* table, you can remove it by entering the following command:

```
DROP TABLE accounts;
```

Backing Up and Restoring

Whatever kind of data you are storing in your database, it has some value, even if it's only the cost of the time for reentering it should the hard disk fail. Therefore, it's important that you keep backups to protect your investment. In addition, there will be times when you have to migrate your database to a new server; the best way to do this is to back it up first. It is important that you test your backups from time to time to ensure they are valid and will work if they need to be used.

Thankfully, backing up and restoring MySQL data is easy with the `mysqldump` command.

Using `mysqldump`

With `mysqldump`, you can dump a database or collection of databases into one or more files containing all the instructions necessary to re-create all your tables and repopulate them with your data. This command can also generate files in CSV (comma-separated values) and other delimited text formats, or even in XML. Its main drawback is that you must make sure that no one writes to a table while you're backing it up. There are various ways to do this, but the easiest is to shut down the MySQL server before running `mysqldump` and start the server again after `mysqldump` finishes.

Alternatively, you can lock the tables you are backing up before running `mysqldump`. To lock tables for reading (as we want to read the data), from the MySQL command line issue this command:

```
LOCK TABLES tablename1 READ, tablename2 READ ...
```

Then, to release the lock(s), enter:

```
UNLOCK TABLES;
```

By default, the output from `mysqldump` is simply printed out, but you can capture it in a file through the `>` redirect symbol.

The basic format of the `mysqldump` command is:

```
mysqldump -u user -ppassword database
```

However, before you can dump the contents of a database, you must make sure that `mysqldump` is in your path or else specify its location as part of your command. **Table 9-13** shows the likely locations of the program for the different installations and operating systems covered in **Chapter 2**. If you have a different installation, it may be in a slightly different location.

Table 9-13. Likely locations of `mysqldump` for different installations

Operating system and program	Likely folder location
Windows AMPPS	C:\Program Files\Ampps\mysql\bin
macOS AMPPS	/Applications/ampps/mysql/bin
Linux	/usr/bin

So, to dump the contents of the *publications* database that you created in **Chapter 8** to the screen, first exit MySQL and then enter the command in **Example 9-7** (specifying the full path to `mysqldump` if necessary).

*Example 9-7. Dumping the *publications* database to screen*

```
mysqldump -u user -ppassword publications
```

Make sure that you replace *user* and *password* with the correct details for your installation of MySQL. If no password is set for the user, you can omit that part of the command, but the `-u user` part is mandatory unless you have root access without a password and are executing as root (not recommended). The result of issuing this command will look something like **Figure 9-4**.

```
Administrator: Command Prompt
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `customers`
--

LOCK TABLES `customers` WRITE;
/*!40000 ALTER TABLE `customers` DISABLE KEYS */;
INSERT INTO `customers` VALUES ('Joe Bloggs','9780099533474'),('Jack Wilson','97
80517123201'),('Mary Smith','9780582506206');
/*!40000 ALTER TABLE `customers` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2024-09-14 3:33:42

C:\Program Files\Amps\mysql\bin>
```

Figure 9-4. Dumping the publications database to the screen

Creating a Backup File

Now that you have `mysqldump` working and have verified it outputs correctly to the screen, you can send the backup data directly to a file using the `>` redirect symbol. Assuming that you wish to call the backup file *publications.sql*, type the command in **Example 9-8** (remembering to replace *user* and *password* with the correct details).



The command in **Example 9-8** stores the backup file into the current directory. If you need it to be saved elsewhere, you should insert a filepath before the filename. You must also ensure that the directory you are backing up to has the right permissions set to allow the file to be written but not to be accessed by any unprivileged user!

Example 9-8. Dumping the publications database to a file

```
mysqldump -u user -ppassword publications > publications.sql
```



Sometimes you may get errors accessing MySQL using Windows PowerShell, which you will not see in a standard Command Prompt window.

If you echo the backup file to screen or load it into a text editor, you will see that it comprises sequences of SQL commands such as:

```
DROP TABLE IF EXISTS 'classics';
CREATE TABLE 'classics' (
  'author' varchar(128) default NULL,
  'title' varchar(128) default NULL,
  'category' varchar(16) default NULL,
  'year' smallint(6) default NULL,
  'isbn' char(13) NOT NULL default '',
  PRIMARY KEY ('isbn'),
  KEY 'author' ('author'(20)),
  KEY 'title' ('title'(20)),
  KEY 'category' ('category'(4)),
  KEY 'year' ('year'),
  FULLTEXT KEY 'author_2' ('author','title')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

This is smart code that can be used to restore a database from a backup, even if it currently exists; it will first drop any tables that need to be re-created, thus avoiding potential MySQL errors.

Backing up a single table

To back up only a single table from a database (such as the *classics* table from the *publications* database), you should first lock the table from within the MySQL command line, by issuing a command such as:

```
LOCK TABLES publications.classics READ;
```

This ensures that MySQL remains running for read purposes but that writes cannot be made. Then, while keeping the MySQL command line open, use another terminal window to issue the following command from the operating system command line:

```
mysqldump -u user -ppassword publications classics > classics.sql
```

You must now release the table lock by entering the following command from the MySQL command line in the first terminal window, which unlocks all tables that have been locked during the current session:

```
UNLOCK TABLES;
```

Backing up all databases

If you want to back up all your MySQL databases at once (including the system databases such as *mysql*), you can use a command such as the one in [Example 9-9](#), which would enable you to restore an entire MySQL database installation. Remember to use locking where required.

Example 9-9. Dumping all the MySQL databases to file

```
mysqldump -u user -ppassword --all-databases > all_databases.sql
```



Of course, there's a lot more than just a few lines of SQL code in backed-up database files. I recommend that you take a few minutes to examine a couple to familiarize yourself with the types of commands that appear in backup files and how they work.

Restoring from a Backup File

To perform a restore from a file, call the *mysql* executable, passing it the file to restore from using the < symbol. So, to recover all databases that you dumped using the `--all-databases` option, use a command such as that in [Example 9-10](#).

Example 9-10. Restoring an entire set of databases

```
mysql -u user -ppassword < all_databases.sql
```

To restore a single database, use the `-D` option followed by the name of the database, as in [Example 9-11](#), where the *publications* database is being restored from the backup made in [Example 9-8](#).

Example 9-11. Restoring the publications database

```
mysql -u user -ppassword -D publications < publications.sql
```

To restore a single table to a database, use a command such as that in [Example 9-12](#), where just the *classics* table is being restored to the *publications* database.

Example 9-12. Restoring the classics table to the publications database

```
mysql -u user -ppassword -D publications < classics.sql
```


Dumping Data in CSV Format

As previously mentioned, the `mysqldump` program is very flexible and supports various types of output, such as the CSV format, which contains just the data and no commands. You might use it to import data into a spreadsheet or an analytical tool, among other purposes. **Example 9-13** shows how you can dump the data from the *classics* and *customers* tables in the *publications* database to the files *classics.txt* and *customers.txt* in the folder *c:/temp*. On macOS or Linux systems, you should modify the destination path to an existing folder.

Example 9-13. Dumping data to CSV-format files

```
mysqldump -u user -ppassword --no-create-info --tab=c:/temp  
--fields-terminated-by=',' publications
```

This command is quite long and is shown here wrapped over two lines, but you must type it all as a single line. The result is:

```
Mark Twain (Samuel Langhorne Clemens)', 'The Adventures of Tom Sawyer',  
  'Classic Fiction', '1876', '9781598184891  
Jane Austen', 'Pride and Prejudice', 'Classic Fiction', '1811', '9780582506206  
Charles Darwin', 'The Origin of Species', 'Nonfiction', '1856', '9780517123201  
Charles Dickens', 'The Old Curiosity Shop', 'Classic Fiction', '1841', '9780099533474  
William Shakespeare', 'Romeo and Juliet', 'Play', '1594', '9780192814968  
  
Mary Smith', '9780582506206  
Jack Wilson', '9780517123201
```

Planning Your Backups

The more valuable your data, the more often you should back it up, and the more copies you should make. If your database gets updated at least once a day, you should back it up daily. If, on the other hand, it is not updated very often, you can get by with less frequent backups.



Consider making multiple backups and storing them in different locations. If you have several servers, it is a simple matter to copy your backups between them. You also should make physical backups on removable hard disks, thumb drives, and so on, and keep these in separate locations—preferably somewhere like a fireproof safe.

It's important to test restoring a database once in a while, too, to make sure your backups are done correctly. You also want to be familiar with restoring a database because you may have to do so when you are stressed and in a hurry, such as after a power failure that takes down the website. You can restore a database to a private server and run a few SQL commands to make sure the data is as it should be.

Once you've digested the contents of this chapter, you will be proficient in using both PHP and MySQL. You can check your understanding using the following questions. [Chapter 10](#) introduces you to accessing MySQL using PHP.

Questions

1. What does the word *relationship* mean in reference to a relational database?
2. What is the term for the process of removing duplicate data and optimizing tables?
3. What are the three rules of the First Normal Form?
4. How can you make a table satisfy the Second Normal Form?
5. What do you put in a column to tie together two tables that contain items having a one-to-many relationship?
6. How can you create a database with a many-to-many relationship?
7. What commands initiate and end a MySQL transaction?
8. What feature does MySQL provide to enable you to examine how a query will work in detail?
9. What command would you use to back up the database *publications* to a file called *publications.sql*?

See “[Chapter 9 Answers](#)” on page 574 in the [Appendix](#) for the answers to these questions.

Accessing MySQL Using PHP

If you worked through the previous chapters, you're now proficient in using both MySQL and PHP. In this chapter, you will learn how to integrate the two by using PHP's built-in functions to access MySQL.

Querying a MySQL Database with PHP

The reason for using PHP as an interface to MySQL is to format the results of SQL queries in a form visible in a web page. As long as you can log in to your MySQL installation using your username and password, you can also do so from PHP.

However, instead of using MySQL's command line to enter instructions and view output, you will create query strings that are passed to MySQL. When MySQL returns its response, it will come as a data structure that PHP can recognize instead of the formatted output you see when you work on the command line. Further PHP commands can retrieve the data and format it for the web page.

The Process

The process of using MySQL with PHP is:

1. Connect to MySQL and select the database to use.
2. Build a query string.
3. Execute the query.
4. Fetch the results and output them to a web page.
5. Repeat steps 2 to 4 until all desired data has been retrieved.
6. Disconnect from MySQL.

We'll work through these steps in turn, but first it's important to set up your login details securely so people snooping around on your system won't have access to your database.

Creating a Login File

Most websites developed with PHP contain multiple program files that will require access to MySQL and will thus need the login and password details. Therefore, it's sensible to create a single file to store these and then include that file wherever it's needed. [Example 10-1](#) shows such a file, which I've called *login.php*.

Example 10-1. The login.php file

```
<?php // login.php
$host = 'localhost'; // Change as necessary
$db   = 'publications'; // Change as necessary
$user = 'root'; // Change as necessary
$pass = 'mysql'; // Change as necessary
$chrs = 'utf8mb4';
$attr = "mysql:host=$host;dbname=$db;charset=$chrs";
$opts =
[
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES  => false,
];
?>
```

Either type the example or edit the file (from the [GitHub repo](#)), replacing the username *root* and password of *mysql* with the values you use for your MySQL database (and the host and database name too, if necessary), and save it to the document root directory you set up in [Chapter 2](#). We'll be using the file shortly.

PDO, as seen in the `$opts` array, stands for PHP Data Objects and is a built-in data access library that uses the same functions regardless of which database you're using.

The hostname *localhost* should work as long as you're using a MySQL database on your local system, and the database *publications* should work if you're using the same examples I've used so far.

The enclosing `<?php` and `?>` tags are especially important for the *login.php* file in [Example 10-1](#), because they mean that the lines between can be interpreted *only* as PHP code. If you were to leave them out and someone were to call up the file directly from your website, it would display as text and reveal your secrets. But, with the tags in place, all that person will see is a blank page. The file will correctly include your other PHP files.



Direct Inclusion of Login Details

This chapter includes placing a username and password in the *login.php* file to make the following short examples work as-is from the repo. At this point you are simply learning about MySQL itself and how it works on a very simple set of non-valuable data in a nonproduction environment. Please be aware that real-world projects must never place secure information anywhere that it might be discovered and compromise your web server or data. Hacking prevention and security are covered toward the end of this chapter, and better ways of handling, storing, and processing sensitive information are provided in [Chapter 12](#), once you have learned the information and techniques needed to understand how to do this.

The `$host` variable will tell PHP which computer to use when connecting to a database. This is required because you can access MySQL databases on any computer connected to your PHP installation, and that potentially includes any host anywhere on the web. However, the examples in this chapter will be working on the local server. So, in place of specifying a domain such as `mysql.myserver.com`, you can use the word `localhost` (or the IP address `127.0.0.1`).

The database we'll be using, in the string variable `$db`, is the one called *publications* that we created in [Chapter 8](#) (if you're using a different database—one provided by your server administrator—you'll have to modify *login.php* accordingly).

`$chars` stands for character set, and in this case we are using `utf8mb4`, a Unicode character set that includes Latin letters, digits, punctuation signs, European and Middle East script letters as well as Korean, Chinese, and Japanese ideographs, and emoji. Characters are encoded using UTF-8 and can take up to 4 bytes per character. `utf8mb4` is the recommended UTF-8 character set in MySQL.



In earlier versions of the book we used direct access to MySQL, which was not at all secure, and later switched to using *mysqli*, which was more secure. But, as they say, time marches on, and now the most secure and easiest way yet to access a MySQL database from PHP is PDO, which we now use by default in this edition of the book as a lightweight, consistent interface for accessing databases in PHP. Each database driver that implements the PDO interface can expose database-specific features as regular extension functions.

Finally, `$attr` and `$opts` contain additional options needed to access the database.



Never store usernames and passwords in the clear on any production project. Refer to [Chapter 12](#) for more on encryption, salting, and security.

Connecting to a MySQL Database

Now that you have saved the *login.php* file, you can include it in any PHP files that will need to access the database by using the `require_once` statement. This is preferable to an `include` statement, as it will generate a fatal error if the file is not found—and believe me, not finding the file containing the login details to your database *is* a fatal error.

Also, using `require_once` instead of `require` means that the file will be read in only when it has not previously been included, which prevents wasteful duplicate disk accesses. [Example 10-2](#) shows the code to use.

Example 10-2. Connecting to a MySQL server using PDO

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}
?>
```



Resist the temptation to output the contents of any error message received from MySQL in your production project. Rather than helping your users, you could give away sensitive information to hackers, such as login details. Instead, just guide the user with information on how to overcome their difficulty based on what the error message reports to your code. The examples use `$e->getMessage()`, which you should replace with a custom message. The exception message as returned by the `getMessage` method can be stored in a logfile, for example. Also, by throwing a new exception we ensure that the stack trace will be different and won't contain login details present in the arguments of the original exception stack trace.

This example creates a new PDO object called `$pdo`, passing all the values retrieved from the `login.php` file to the constructor. We achieve error checking by using the `try...catch` pair of commands.

The PDO object is used in the following examples to access the MySQL database.

Building and Executing a Query

Sending a query to MySQL from PHP is as simple as including the relevant SQL in the `query` method of a connection object. [Example 10-3](#) shows you how to do this.

Example 10-3. Querying a database with PDO

```
<?php
$query = "SELECT * FROM classics";
$result = $pdo->query($query);
?>
```

As you can see, the MySQL query looks just like what you would type directly at the command line, except there is no trailing semicolon inside the string, as none is needed when you are accessing MySQL from PHP. Here the variable `$query` is assigned a string containing the query to be made and then passed to the `query` method of the `$pdo` object, which returns a result that we place in the object `$result`. All the data returned by MySQL is now stored in an easily interrogable format in the `$result` object.

Selecting all columns with `SELECT * FROM ...` instead of listing only the columns you'll need is used here for brevity, but in general it should be avoided especially when querying tables with many large columns, to avoid exhausting server memory.

Fetching a Result

Once you have an object returned in `$result`, you can use it to extract the data you want, one item at a time, using the `fetch` method of the object. [Example 10-4](#) combines and extends the previous examples into a program that you can run yourself to retrieve the results (as depicted in [Figure 10-1](#)). Type this script and save it using the filename `query.php`, or download it from the [example repository](#).

Example 10-4. Fetching results one row at a time

```
<?php // query.php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
```

```

}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "SELECT * FROM classics";
$result = $pdo->query($query);

while ($row = $result->fetch())
{
    echo 'Author:   '.htmlspecialchars($row['author']) . "<br>";
    echo 'Title:   '.htmlspecialchars($row['title']) . "<br>";
    echo 'Category: '.htmlspecialchars($row['category']) . "<br>";
    echo 'Year:    '.htmlspecialchars($row['year']) . "<br>";
    echo 'ISBN:    '.htmlspecialchars($row['isbn']) . "<br><br>";
}
?>

```

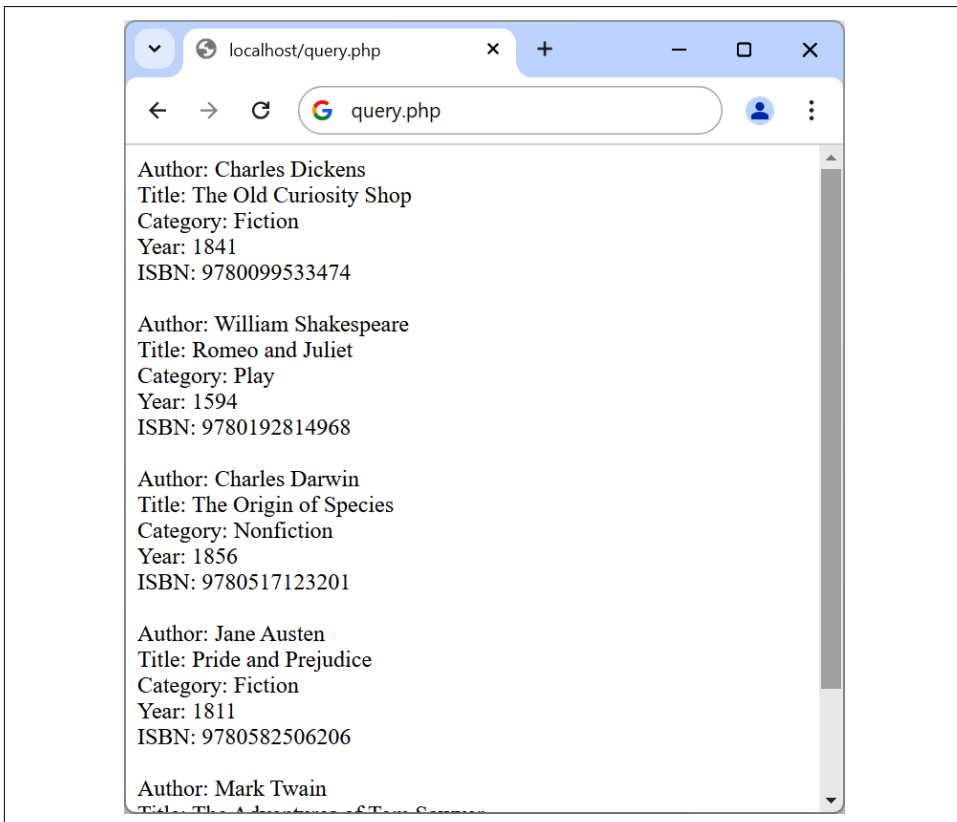


Figure 10-1. The output from *query.php*

Here, each time around the loop, we call the `fetch` method of the `$pdo` object to retrieve the value stored in each row and output the result using `echo` statements. Don't worry if you see the results in a different order. This is because we have not used an `ORDER BY` command to specify the order in which they should be returned, so the order will be unspecified.

When displaying data in a browser whose source was (or may have been) user input, there's always a risk of sneaky HTML characters being embedded within it—even if you believe it to have been previously sanitized—which could potentially be used for a cross-site scripting (XSS) attack. The simple way to prevent this possibility is to embed all such output within a call to the function `htmlspecialchars`, which replaces all such characters with harmless HTML entities in which, for example, the `<` character is replaced with the entity `<`, and so forth. This technique was implemented in the preceding example and will be used in many of the following examples.

In [Chapter 9](#), I talked about First, Second, and Third Normal Form. You may have noticed that the *classics* table doesn't satisfy these, because both author and book details are included within the same table. That's because we created this table before encountering normalization. However, for the purposes of illustrating access to MySQL from PHP, reusing this table prevents the hassle of typing in a new set of test data, so we'll stick with it for the time being.

Fetching a Row While Specifying the Style

The `fetch` method can return data in various styles, including the following (where anonymous means unnamed):

`PDO::FETCH_ASSOC`

Returns the next row as an array indexed by column name

`PDO::FETCH_BOTH` (*default*)

Returns the next row as an array indexed by both column name and number

`PDO::FETCH_LAZY`

Returns the next row as an anonymous object with names as properties

`PDO::FETCH_OBJ`

Returns the next row as an anonymous object with column names as properties

`PDO::FETCH_NUM`

Returns an array indexed by column number

For the full list of PDO fetch styles, please refer to the [online reference](#).

Therefore, the following (slightly changed) example (shown in [Example 10-5](#)) shows more clearly the intention of the `fetch` method in this case. You may wish to save this revised file using the name *fetchrow.php*.

Example 10-5. Fetching results one row at a time

```
<?php //fetchrow.php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "SELECT * FROM classics";
$result = $pdo->query($query);

while ($row = $result->fetch(PDO::FETCH_ASSOC)) // Style of fetch
{
    echo 'Author:   '.htmlspecialchars($row['author']) . "<br>";
    echo 'Title:    '.htmlspecialchars($row['title'])  . "<br>";
    echo 'Category: '.htmlspecialchars($row['category']) . "<br>";
    echo 'Year:     '.htmlspecialchars($row['year'])   . "<br>";
    echo 'ISBN:     '.htmlspecialchars($row['isbn'])    . "<br><br>";
}
?>
```

The `fetch` method in this example returns only an associative array, leaving out the numeric indexes that would be returned when the fetch style wouldn't be specified, or if `PDO::FETCH_BOTH` would be used. Associative arrays can be more useful than numeric ones because you can refer to each column by name, such as `$row['author']`, instead of trying to remember where it is located in the column order. The numeric indexes are often unused so the `fetch` method does not need to return them.

Closing a Connection

PHP will eventually return the memory it has allocated for objects after you have finished with the script, so in small scripts, you don't usually need to worry about releasing memory yourself. However, should you wish to close a PDO connection manually, you simply set it to `null` like this:

```
$pdo = null;
```

A Practical Example

It's time to write our first example of inserting data in and deleting it from a MySQL table using PHP. I recommend that you type [Example 10-6](#) and save it to your web development directory using the filename *sqltest.php*. You can see an example of the program's output in [Figure 10-2](#).



[Example 10-6](#) creates a standard HTML form. [Chapter 11](#) explains forms in detail, but in this chapter I take form handling for granted and just deal with database interactions.

*Example 10-6. Inserting and deleting using *sqltest.php**

```
<?php // sqltest.php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

if (isset($_POST['delete']) && isset($_POST['isbn']))
{
    $isbn = sanitize_post_value($pdo, 'isbn');
    $query = "DELETE FROM classics WHERE isbn=$isbn";
    $result = $pdo->query($query);
}

if (isset($_POST['author']) &&
    isset($_POST['title']) &&
    isset($_POST['category']) &&
    isset($_POST['year']) &&
    isset($_POST['isbn']))
{
    $author = sanitize_post_value($pdo, 'author');
    $title = sanitize_post_value($pdo, 'title');
    $category = sanitize_post_value($pdo, 'category');
    $year = sanitize_post_value($pdo, 'year');
    $isbn = sanitize_post_value($pdo, 'isbn');

    $query = "INSERT INTO classics VALUES " .
        "($author, $title, $category, $year, $isbn)";
    $result = $pdo->query($query);
}
```

```

echo <<<_END
<form action="sqltest.php" method="post"><pre>
    Author <input type="text" name="author">
    Title <input type="text" name="title">
    Category <input type="text" name="category">
    Year <input type="text" name="year">
    ISBN <input type="text" name="isbn">
        <input type="submit" value="ADD RECORD">
</pre></form>
_END;

$query = "SELECT * FROM classics";
$result = $pdo->query($query);

while ($row = $result->fetch())
{
    $r0 = htmlspecialchars($row['author']);
    $r1 = htmlspecialchars($row['title']);
    $r2 = htmlspecialchars($row['category']);
    $r3 = htmlspecialchars($row['year']);
    $r4 = htmlspecialchars($row['isbn']);

    echo <<<_END
    <pre>
        Author $r0
        Title $r1
    Category $r2
        Year $r3
        ISBN $r4
    </pre>
    <form action='sqltest.php' method='post'>
    <input type='hidden' name='delete' value='yes'>
    <input type='hidden' name='isbn' value='$r4'>
    <input type='submit' value='DELETE RECORD'></form>
_END;
}

function sanitize_post_value($pdo, $var)
{
    return $pdo->quote($_POST[$var]);
}
?>

```

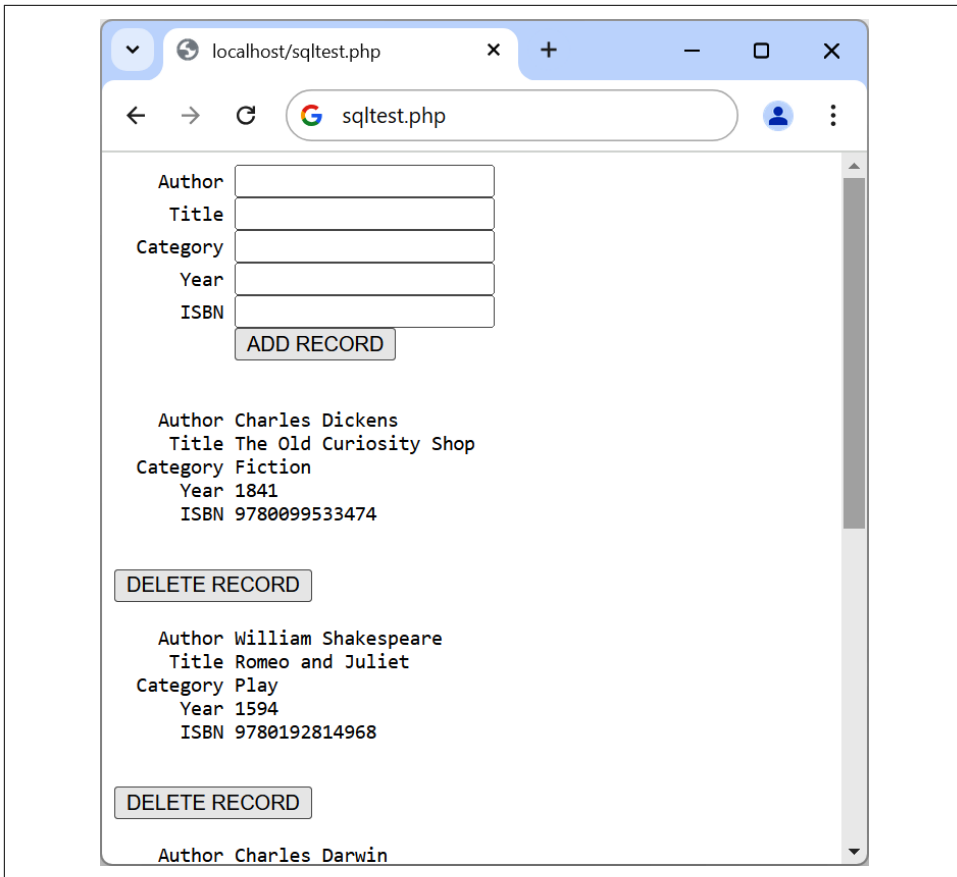


Figure 10-2. The output from *Example 10-6*, *sqltest.php*

At almost 80 lines of code, this program may appear daunting, but don't worry—you've already covered many of those lines in *Example 10-4*, and what the code does is actually quite simple.

It first checks for any inputs that may have been made and then either inserts new data into the table *classics* of the *publications* database or deletes a row from it, according to the input supplied. Regardless of whether there was input, the program then outputs all rows in the table to the browser. Let's see how it works.

The first section of new code starts by using the `isset` function to check whether values for all the fields have been posted to the program. Upon confirmation, each line within the `if` statement calls the function `sanitize_post_value`, which appears at the end of the program. This function has one small but critical job: fetching input from the browser.



For clarity and brevity, and to explain things as simply as possible, many of the following examples omit sensible security precautions that would have made them longer and could detract from clearly explaining their function. Don't skip past **“Preventing Hacking Attempts”** on page 259 on preventing your database from being hacked, where you will learn about additional actions you can take to secure your code.

The \$_POST Array

I mentioned in an earlier chapter that a browser sends user input through either a GET request or a POST request. Form data is almost always sent using the POST method as putting all the form data in GET request URLs would be unsightly as well as a potential security and privacy risk. Once a POST method form has been submitted, the web server bundles up all of the user input and puts it into an array named \$_POST.

Whether a form has been set to use either the GET or POST method, the \$_GET associative array will always be populated with URL query parameters, if present, from the form's action attribute. Additionally, the \$_GET array will also contain form field values if the GET method has been used. If the form was submitted using the POST method, the form field data will be returned in the \$_POST array.

Each field has an element in the array named after that field. So, if a form contains a field named isbn, the \$_POST array contains an element keyed by the word isbn. The PHP program can read that field by referring to either \$_POST['isbn'] or \$_POST["isbn"] (single and double quotes have the same effect in this case).

If the \$_POST syntax still seems complex to you, remember you can just use the convention shown in **Example 10-6**: copy the user's input to other variables and forget about \$_POST after that. This is normal in PHP programs: they retrieve all the fields from \$_POST at the beginning of the program and then ignore it.



There is no reason to write to an element in the \$_POST array. Its only purpose is to communicate information from the browser to the program, and you're better off copying data to your own variables before altering it.

The `sanitize_post_value` function from **Example 10-6** passes each item it retrieves through the quote method of the PDO object to escape any quotes that a hacker may have inserted to break into or alter your database, like this, and it adds quotes around each string for you:

```
function sanitize_post_value($pdo, $var)
{
```

```
    return $pdo->quote($_POST[$var]);  
}
```

Deleting a Record

Prior to checking whether new data has been posted, the program checks whether the variable `$_POST['delete']` has a value. If so, the user has clicked the DELETE RECORD button to erase a record. In this case, the value of `$isbn` will also have been posted.

As you will recall, the ISBN uniquely identifies each record. The script receives the identifier as the value of the hidden HTML form field named `isbn` in `$_POST['isbn']`. The `sanitize_post_value` function is then used to escape any dangerous characters and add quotes. The returned value is stored in `$isbn`, which is then used in the `DELETE FROM` query created in the variable `$query`, which is then passed to the `query` method of the `pdo` object to issue it to MySQL.

If `$_POST['delete']` is not set (and there is no record to be deleted), `$_POST['author']` and other posted values are checked. If they have all been given values, `$query` is set to an `INSERT INTO` command, followed by the five values to be inserted. The string is then passed to the `query` method.



Helpful Error Messages

If any query fails, PHP will throw an error. On a production website, you will not want these very programmer-oriented error messages to show, so you will need to add more `try...catch` commands and replace the existing `catch` statement, which handles connection errors only, with one in which you handle the error yourself neatly and decide what sort of error message (if any) to give to your users.

Displaying the Form

Before displaying the little form (as shown in [Figure 10-2](#)), the program sanitizes copies of the elements we will be outputting from the `$row` array into the variables `$r0` through `$r4` by passing them to the `htmlspecialchars` function, to replace any potentially dangerous HTML characters with harmless HTML entities.

Then the part of code that displays the output follows, using a heredoc `echo <<<_END..._END` structure as seen in previous chapters, which outputs everything between the `_END` tags.

The HTML form section simply sets the form's action to `sqltest.php`. This means that when the form is submitted, the contents of the form fields will be sent to the file

sqltest.php, which is the program itself. The form is also set up to send the fields as a POST rather than a GET request. This is because GET requests are appended to the URL being submitted and can look messy in your browser. They also allow users to easily modify submissions and try to hack your server (although that also can be achieved with in-browser developer tools). Additionally, avoiding GET requests prevents too much information appearing in server logfiles. Therefore, whenever possible, you should use POST submissions, which also have the benefit of revealing less posted data.

Having output the form fields, the HTML displays a submit button with the name ADD RECORD and closes the form. Note the `<pre>` and `</pre>` tags here, which have been used to force a monospaced font that lines up all the inputs neatly. The carriage returns at the end of each line are also output when inside `<pre>` tags.



Instead of using the `echo` command, the program could drop out of PHP using `?>`, issue the HTML, and then reenter PHP processing with `<?php`. Which style is used is a matter of programmer preference.

Querying the Database

Next, the code returns to the familiar territory of [Example 10-4](#), where a query is sent to MySQL asking to see all the records in the *classics* table, like this:

```
$query = "SELECT * FROM classics";  
$result = $pdo->query($query);
```

A `while` loop is then entered to display the contents of each row. Then the program populates the array `$row` with a row of results by calling the `fetch` method of `$result`.

With the data in `$row`, it's now a simple matter to display it within the heredoc `echo` statement that follows, in which I have chosen to use a `<pre>` tag to line up the display of each record in a pleasing manner. After the display of each record, a second form also posts to *sqltest.php* (the program itself) but this time contains two hidden fields: `delete` and `isbn`. The `delete` field is set to `yes` and `isbn` to the value held in `$row[isbn]`, which contains the ISBN for the record.

Then a submit button with the name DELETE RECORD is displayed, and the form is closed. A curly brace then completes the `while` loop, which will continue until all records have been displayed.

Finally, you see the definition for the function `sanitize_post_value`, which we've already looked at. And that's it—our first PHP program to manipulate a MySQL database. So, let's check out what it can do.

Once you've typed the program (and corrected any errors), enter this data into the various input fields to add a new record for the book *Moby Dick* to the database:

```
Herman Melville  
Moby Dick  
Fiction  
1851  
9780199535729
```

Running the Program

When you have submitted this data using the ADD RECORD button, scroll down the web page to see the new addition. It should look something like [Figure 10-3](#), although since we have not ordered the results using `ORDER BY`, the position in which it appears is undetermined.

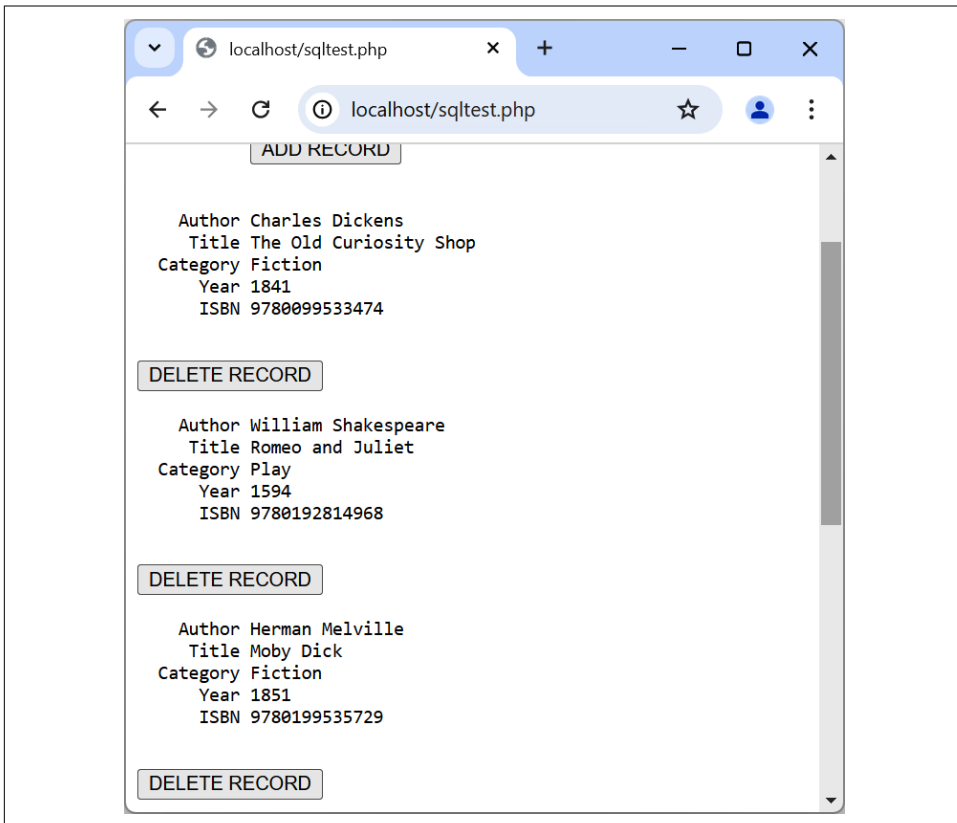


Figure 10-3. The result of adding *Moby Dick* to the database

Now let's look at how deleting a record works by creating a dummy record. Try entering just the number 1 in each of the five fields and clicking the ADD RECORD

button. If you scroll down, you'll see a new record consisting just of 1s. Obviously, this record isn't useful in this table, so now click the DELETE RECORD button and scroll down again to confirm that the record has been deleted.



Assuming that everything worked, you now can add and delete records at will. Try doing this a few times, but leave the main records in place (including the new one for *Moby Dick*), as we'll be using them later. You also can try adding the record with all 1s again a couple of times and if you haven't already deleted it, note the error message that you receive the second time, indicating that there is already an ISBN with the number 1.

Practical MySQL

You are now ready for some practical techniques you can use in PHP to access the MySQL database, including tasks such as creating and dropping tables; inserting, updating, and deleting data; and protecting your database and website from malicious users. The following examples assume that you've already created the *login.php* program discussed earlier in this chapter.

Creating a Table

Let's assume that you are working for a wildlife park and need to create a database to hold details about all the types of cats it houses. You know there are nine *families* of cats—Lion, Tiger, Jaguar, Leopard, Cougar, Cheetah, Lynx, Caracal, and Domestic—so you'll need a column for that. Then each cat has been given a *name*, so that's another column, and you also want to keep track of their *ages*, which is another. Of course, you will probably need more columns later, perhaps to hold dietary requirements, inoculations, and other details, but for now that's enough to get going. A unique identifier is also needed for each animal, so you also decide to create a column for that called *id*.

Example 10-7 shows the code you might use to create a MySQL table to hold this data, with the main query assignment in bold text.

Example 10-7. Creating a table called cats

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
```

```

{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "CREATE TABLE cats (
    id SMALLINT NOT NULL AUTO_INCREMENT,
    family VARCHAR(32) NOT NULL,
    name VARCHAR(32) NOT NULL,
    age TINYINT NOT NULL,
    PRIMARY KEY (id)
)";

$result = $pdo->query($query);
?>

```

As you can see, the MySQL query looks just like what you would type directly at the command line, except without the trailing semicolon.

Describing a Table

When you aren't logged in to the MySQL command line, here's a handy piece of code that you can use to verify that a table has been correctly created from inside a browser. It simply issues the query `DESCRIBE cats` and then outputs an HTML table with four headings—*Column*, *Type*, *Null*, and *Key*—underneath which all columns within the table are shown. To use it with other tables, simply replace the name `cats` in the query with that of the new table (see [Example 10-8](#)).

Example 10-8. Describing the cats table

```

<?php
    require_once 'login.php';

    try
    {
        $pdo = new PDO($attr, $user, $pass, $opts);
    }
    catch (PDOException $e)
    {
        throw new PDOException($e->getMessage(), (int)$e->getCode());
    }

    $query = "DESCRIBE cats";
    $result = $pdo->query($query);

    echo "<table><tr><th>Column</th><th>Type</th>";
    echo "<th>Null</th><th>Key</th></tr>";

    while ($row = $result->fetch(PDO::FETCH_NUM))
    {

```

```

        echo "<tr>";
        for ($k = 0 ; $k < 4 ; ++$k)
            echo "<td>" . htmlspecialchars($row[$k]) . "</td>";
        echo "</tr>";
    }

    echo "</table>";
?>

```

See how the PDO fetch style of `FETCH_NUM` is used to return a numeric array so that it is easy to display the contents of the returned data without using names. The output from the program should look like this:

Column	Type	Null	Key
id	smallint(6)	NO	PRI
family	varchar(32)	NO	
name	varchar(32)	NO	
age	tinyint(4)	NO	

Dropping a Table

Dropping a table is very easy to do and therefore very dangerous, so be careful. It is not something you would usually do in a PHP project, but [Example 10-9](#) shows the code that you'd use if needed. However, I don't recommend that you try it until you have been through the other examples (up to [“Performing Additional Queries” on page 257](#)), as it will drop the table *cats* and you'll have to re-create it using [Example 10-7](#).

Example 10-9. Dropping the cats table

```

<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "DROP TABLE cats";
$result = $pdo->query($query);
?>

```

Adding Data

Let's add some data to the table, using the code in [Example 10-10](#).

Example 10-10. Adding data to the cats table

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "INSERT INTO cats VALUES(NULL, 'Lion', 'Leo', 4)";
$result = $pdo->query($query);
?>
```

You may wish to add a couple more items of data by modifying `$query` as follows and calling up the program in your browser again:

```
$query = "INSERT INTO cats VALUES(NULL, 'Cougar', 'Growler', 2)";
$query = "INSERT INTO cats VALUES(NULL, 'Cheetah', 'Charly', 3)";
```

By the way, did you notice the `NULL` value passed as the first parameter? This is because the *id* column is of type `AUTO_INCREMENT`, and MySQL will decide what value to assign according to the next available number in sequence. So, we simply pass a `NULL` value, which will be ignored.

Of course, the most efficient way to populate MySQL with data is to create an array and insert the data with a single query using multiple lists of column values specified within parentheses and separated by commas:

```
INSERT INTO cats VALUES
(NULL, 'Cougar', 'Growler', 2), (NULL, 'Cheetah', 'Charly', 3)
```



At this point, I am concentrating on showing you how to directly insert data into MySQL (and providing some security precautions to keep the process safe). However, later in the book we'll move on to a better method you can employ that involves placeholders (see [“Using Placeholders” on page 261](#)), which make it virtually impossible for users to inject malicious hacks into your database. So, as you read this section, understand that these are the basics of how MySQL insertion works and remember that we will improve on it later.

Retrieving Data

Now that some data has been entered into the *cats* table, [Example 10-11](#) shows how you can check that it was correctly inserted.

Example 10-11. Retrieving rows from the cats table

```
<?php
    require_once 'login.php';

    try
    {
        $pdo = new PDO($attr, $user, $pass, $opts);
    }
    catch (PDOException $e)
    {
        throw new PDOException($e->getMessage(), (int)$e->getCode());
    }

    $query = "SELECT * FROM cats";
    $result = $pdo->query($query);

    echo "<table><tr> <th>Id</th> <th>Family</th>";
    echo "<th>Name</th><th>Age</th></tr>";

    while ($row = $result->fetch(PDO::FETCH_NUM))
    {
        echo "<tr>";
        for ($k = 0 ; $k < 4 ; ++$k)
            echo "<td>" . htmlspecialchars($row[$k]) . "</td>";
        echo "</tr>";
    }

    echo "</table>";
?>
```

This code simply issues the MySQL query `SELECT * FROM cats` and then displays all the rows returned by requiring them in the form of numerically accessed arrays with the style of `PDO::FETCH_NUM`. Its output is:

	Id	Family	Name	Age
1	Lion	Leo	4	
2	Cougar	Growler	2	
3	Cheetah	Charly	3	

Here you can see that the *id* column has correctly auto-incremented.

Updating Data

Changing data that you have already inserted is also quite simple. Did you notice the spelling of *Charly* for the cheetah's name? Let's correct that to *Charlie*, as in [Example 10-12](#).

Example 10-12. Changing the name Charly the cheetah to Charlie

```
<?php
    require_once 'login.php';

    try
    {
        $pdo = new PDO($attr, $user, $pass, $opts);
    }
    catch (PDOException $e)
    {
        throw new PDOException($e->getMessage(), (int)$e->getCode());
    }

    $query = "UPDATE cats SET name='Charlie' WHERE name='Charly'";
    $result = $pdo->query($query);
?>
```

If you run [Example 10-11](#) again, you'll see that it now outputs:

Id	Family	Name	Age
1	Lion	Leo	4
2	Cougar	Growler	2
3	Cheetah	Charlie	3

Deleting Data

Growler the cougar has been transferred to another zoo, so it's time to remove him from the database; see [Example 10-13](#).

Example 10-13. Removing Growler the cougar from the cats table

```
<?php
    require_once 'login.php';

    try
    {
        $pdo = new PDO($attr, $user, $pass, $opts);
    }
    catch (PDOException $e)
    {
        throw new PDOException($e->getMessage(), (int)$e->getCode());
    }
```

```
$query = "DELETE FROM cats WHERE name='Growler'";
$result = $pdo->query($query);
?>
```

This uses a standard DELETE FROM query, and when you run [Example 10-11](#), you can see that the row has been removed:

	Id	Family	Name	Age
1	Lion	Leo	4	
3	Cheetah	Charlie	3	

Using AUTO_INCREMENT

When using AUTO_INCREMENT, you cannot know what value has been given to a column before a row is inserted. Instead, if you need to know it, you must ask MySQL afterward by calling `$pdo->lastInsertId()`. This need is common: for instance, when you process a purchase, you might insert a new customer into a *Customers* table and then refer to the newly created *CustId* when inserting a purchase into the *Purchases* table.



Using AUTO_INCREMENT is recommended instead of selecting the highest ID in the *id* column and incrementing it by one, because concurrent queries could change the values in that column after the highest value has been fetched and before the calculated value is stored.

[Example 10-10](#) can be rewritten as [Example 10-14](#) to display this value after each insert.

Example 10-14. Adding data to the cats table and reporting the insert ID

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$query = "INSERT INTO cats VALUES(NULL, 'Lynx', 'Stumpy', 5)";
$result = $pdo->query($query);

echo "The Insert ID was: " . $pdo->lastInsertId();
?>
```


The contents of the table should now look like the following (note how the previous *id* value of 2 is *not* reused, as this could cause complications in some instances):

Id	Family	Name	Age
1	Lion	Leo	4
3	Cheetah	Charlie	3
4	Lynx	Stumpy	5

Using insert IDs

It's very common to insert data in multiple tables: a book followed by its author, a customer followed by their purchase, and so on. When doing this with an auto-increment column, you will need to retain the insert ID returned for storing in the related table.

For example, let's assume that these cats can be “adopted” by the public as a means of raising funds, and that when a new cat is stored in the *cats* table, we also want to create a key to tie it to the animal's adoptive owner. The code to do this is similar to that in [Example 10-14](#), except that the returned insert ID is stored in the variable `$insertID` and is then used as part of the subsequent query:

```
$query = "INSERT INTO cats VALUES(NULL, 'Lynx', 'Stumpy', 5)";
$result = $pdo->query($query);
$insertID = $pdo->lastInsertId();

$query = "INSERT INTO owners VALUES($insertID, 'Ann', 'Smith')";
$result = $pdo->query($query);
```

Now the cat is connected to its “owner” through the cat's unique ID, which was created automatically by `AUTO_INCREMENT`. This example, and especially the last two lines, is theoretical code showing how to use an insert ID as a key if we had created a table called *owners*.

Performing Additional Queries

Okay, that's enough feline fun. To explore some slightly more complex queries, we need to revert to using the *customers* and *classics* tables that you created in [Chapter 8](#). There will be three customers in the *customers* table, while the *classics* table holds the details of a few books. They also share a common column of ISBNs, called *isbn*, that you can use to perform additional queries.

For example, to display all of the customers along with the titles and authors of the books they have bought, you can use the code in [Example 10-15](#).

Example 10-15. Performing a secondary query

```
<?php
require_once 'login.php';
```


Preventing Hacking Attempts

You might at first find it difficult to understand just how dangerous it is to pass user input unchecked to MySQL. For example, suppose you have a simple piece of code to verify a user, and it looks like this:

```
$user = $_POST['user'];  
$pass = $_POST['pass'];  
$query = "SELECT * FROM users WHERE user='$user' AND pass='$pass'";
```

At first glance, you might think this code is perfectly fine. If the user enters values of fredsmith and mypass for \$user and \$pass, respectively, then the query string, as passed to MySQL, will be:

```
SELECT * FROM users WHERE user='fredsmith' AND pass='mypass'
```

This is all well and good, but what if someone enters the following for \$user (and doesn't even enter anything for \$pass)?

```
admin' #
```

Here's the string that would be sent to MySQL:

```
SELECT * FROM users WHERE user='admin' #' AND pass=''
```

Do you see the problem? An *SQL injection* attack has occurred. In MySQL, the # symbol represents the start of a comment. Therefore, the user will be logged in as *admin* (assuming there is a user *admin*), without having to enter a password. In the following, the part of the query that will be executed is shown in bold; the rest will be ignored:

```
SELECT * FROM users WHERE user='admin' #' AND pass=''
```

Count yourself very lucky if that's all a malicious user does to you. You might still be able to go into your application and undo any changes the user makes as *admin*. But what if your application code removes a user from the database? The code might look something like this:

```
$user = $_POST['user'];  
$pass = $_POST['pass'];  
$query = "DELETE FROM users WHERE user='$user' AND pass='$pass'";
```

Again, this looks quite normal at first glance, but what if someone entered the following for \$user?

```
anything' OR 1=1 #
```

This would be interpreted by MySQL as:

```
DELETE FROM users WHERE user='anything' OR 1=1 #' AND pass=''
```

Ouch—because any statement followed by OR 1=1 is always TRUE, that SQL query will always be TRUE, and therefore, since the rest of the statement is ignored due to the #

character, you've now lost your whole *users* database! So what can you do about this kind of attack?

Steps You Can Take

First, don't rely on PHP's built-in *magic quotes*, used to automatically escape any characters such as single and double quotes by prefacing them with a backslash (\). The feature was removed in PHP 5.4.0.

Instead, as we showed earlier, you could use the `quote` method of the PDO object to escape all characters and surround strings with quotation marks. [Example 10-16](#) is a function you can use that will properly sanitize a user-inputted string for you.

Example 10-16. How to properly sanitize user input for MySQL

```
<?php
function mysql_fix_string($pdo, $string)
{
    return $pdo->quote($string);
}
?>
```

[Example 10-17](#) illustrates how you would incorporate `mysql_fix_string` within your own code.

You could also call `$pdo->quote($string)` directly instead of wrapping it in a function like `mysql_fix_string`.

Example 10-17. How to safely access MySQL with user input

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$user = mysql_fix_string($pdo, $_POST['user']);
$pass = mysql_fix_string($pdo, $_POST['pass']);
$query = "SELECT * FROM users WHERE user=$user AND pass=$pass";

// Etc...

function mysql_fix_string($pdo, $string)
```

```
{
    return $pdo->quote($string);
}
?>
```



Remember: because the `quote` method automatically adds quotes around strings, you should *not* use them in any query that uses these sanitized strings. So, in place of using this:

```
$query = "SELECT * FROM users WHERE user='$user' AND pass='$pass'";
```

you should enter:

```
$query = "SELECT * FROM users WHERE user=$user AND pass=$pass";
```

These precautions are becoming less important, however, because there's a much easier and safer way to access MySQL, which obviates the need for these types of functions—the use of placeholders, which is explained next.

Using Placeholders

All the methods shown thus far work with MySQL but have security implications, with strings constantly requiring escaping to prevent security risks. So, now that you know the basics, let me introduce the best and recommended way to interact with MySQL that is pretty much bulletproof in terms of security. Once you have read this section, you should no longer use direct inserting of data into MySQL but instead always use placeholders. It was still important to show you how to do it without placeholders because a lot of existing or older code doesn't use them.

So what are placeholders? They are positions within prepared statements in which data is transferred directly to the database, without the possibility of user-submitted (or other) data being interpreted as MySQL statements (and the potential for hacking that could result).

The technology requires that you first prepare the statement you wish to be executed in MySQL but leave all the parts of the statement that refer to data as simple question marks.

In plain MySQL, prepared statements look like [Example 10-18](#).

Example 10-18. MySQL placeholders

```
PREPARE statement FROM "INSERT INTO classics VALUES(?,?,?,?)";
```

```
SET @author = "Emily Brontë",
    @title = "Wuthering Heights",
    @category = "Classic Fiction",
    @year = "1847",
    @isbn = "9780553212587";
```

```
EXECUTE statement USING @author,@title,@category,@year,@isbn;
DEALLOCATE PREPARE statement;
```

This can be cumbersome to submit to MySQL, so the PDO extension makes handling placeholders easier with a ready-made method called `prepare`, which you call like this:

```
$stmt = $pdo->prepare('INSERT INTO classics VALUES(?,?,?,?,?)');
```

The object `$stmt` (shorthand for *statement*) returned by this method is then used for sending the data to the server in place of the question marks. Its first use is to bind some PHP variables to each of the question marks (the placeholder parameters) in turn, like this:

```
$stmt->bindParam(1, $author, PDO::PARAM_STR, 128);
$stmt->bindParam(2, $title, PDO::PARAM_STR, 128);
$stmt->bindParam(3, $category, PDO::PARAM_STR, 16 );
$stmt->bindParam(4, $year, PDO::PARAM_INT );
$stmt->bindParam(5, $isbn, PDO::PARAM_STR, 13 );
```

The first argument to `bindParam` is a number representing the position in the query string of the value to insert (in other words, which question mark placeholder is being referred to). This is followed by the variable that will supply the data for that placeholder, and then the type of data the variable must be and, if a string, another value follows stating its maximum length.

With the variables bound to the prepared statement, it is now necessary to populate them with the data to be passed to MySQL, like this:

```
$author = 'Emily Brontë';
$title = 'Wuthering Heights';
$category = 'Classic Fiction';
$year = '1847';
$isbn = '9780553212587';
```

At this point, PHP has everything it needs to execute the prepared statement, so you can issue the following command, which calls the `execute` method of the `$stmt` object created earlier:

```
$stmt->execute();
```

Before going any further, it makes sense to verify whether the command was executed successfully. You can do that by calling the `rowCount` method of `$stmt`:

```
printf("%d Row inserted.\n", $stmt->rowCount());
```

In this case, the output should indicate that one row was inserted.

When using the `bindParam` method, you need to correctly specify the position, the type, and need to use a variable. Luckily, there's an easier and clearer way to use

placeholders. Instead of specifying positions and using question marks, you can use named values (for example `:name`), and instead of binding variables with `bindParam`, you can pass values directly to the `execute` method:

```
$stmt = $pdo->prepare('INSERT INTO classics
VALUES(:author,:title,:category,:year,:isbn)');
$stmt->execute([
    'author'    => 'Emily Brontë',
    'title'     => 'Wuthering Heights',
    'category'  => 'Classic Fiction',
    'year'      => 1847,
    'isbn'      => '9780553212587'
]);
```

Note how the keys in the array passed to `execute` have the same names as the named parameters in the query passed to the `prepare` method. The order of the array items is irrelevant; the key is what binds the value to the named parameter.

The colon prefix (`:`) is optional in the `execute` call (the array keys can be named either `author` or `:author`) but required in the `prepare` call. PHP will guess the data types from the array values, which can be type-casted if needed; you don't need to specify them.

When you put all this together, the result is [Example 10-19](#).

Example 10-19. Issuing prepared statements

```
<?php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}

$stmt = $pdo->prepare('INSERT INTO classics
VALUES(:author,:title,:category,:year,:isbn)');
$stmt->execute([
    'author'    => 'Emily Brontë',
    'title'     => 'Wuthering Heights',
    'category'  => 'Classic Fiction',
    'year'      => 1847,
    'isbn'      => '9780553212587'
]);
printf("%d Row inserted.\n", $stmt->rowCount());
?>
```

Every time you use prepared statements in place of nonprepared ones, you will be closing a potential security hole, so it's worth spending some time getting to know how to use them.

Preventing JavaScript Injection into HTML

There's another type of injection you need to be concerned about—not for the safety of your own websites but for your users' privacy and protection. That's *cross-site scripting*, also referred to as an *XSS attack*.

This occurs when you allow HTML or, more often, JavaScript code to be input by a user and then displayed by your website. One place this is common is in a comment form. What happens most often is that a malicious user will try to write code that steals cookies from your site's users, which even allows them to discover username and password pairs if those are poorly handled or other information that could enable session hijacking (in which a user's login is taken over by a hacker, who could then take over that person's account!). Or the malicious user might launch a phishing attack to steal login credentials from a fake login form.

Preventing this is as simple as calling the `htmlspecialchars` function, which strips out all HTML markup and replaces it with a form that displays the characters but does not allow a browser to act on them. For example, consider this HTML:

```
<script src='http://example.com/hack.js'></script>
<script>hack();</script>
```

This code loads in a JavaScript program and then executes malicious functions. But if it is first passed through `htmlspecialchars`, it will be turned into the following totally harmless string:

```
&lt;script src=&#039;http://example.com/hack.js&#039;&gt; &lt;/script&gt;
&lt;script&gt;hack();&lt;/script&gt;
```

Therefore, if you are ever going to display anything that your users enter, either immediately or after storing it in a database, you first need to sanitize it using the `htmlspecialchars` function. To do this, I recommend that you create a new function, like the first one in [Example 10-20](#), but you can also use `htmlspecialchars` directly.

Example 10-20. Functions for preventing both SQL and XSS injection attacks

```
<?php
function entities_fix_string($string)
{
    return htmlspecialchars($string);
}

function mysql_fix_string($pdo, $string)
{

```



```

        return $pdo->quote($string);
    }
?>

```

The `entities_fix_string` function passes the string through `htmlentities` before returning the fully sanitized string. To use the `mysql_fix_string` function, you must already have an active connection object open to a MySQL database.

Example 10-21 shows the new “higher protection” version of **Example 10-17**. This is just example code, and you need to add the code to access the results returned where you see the `//Etc...` comment line.

Example 10-21. How to safely access MySQL and prevent XSS attacks

```

<?php
    require_once 'login.php';

    try
    {
        $pdo = new PDO($attr, $user, $pass, $opts);
    }
    catch (PDOException $e)
    {
        throw new PDOException($e->getMessage(), (int)$e->getCode());
    }

    $user = mysql_fix_string($pdo, $_POST['user']);
    $pass = mysql_fix_string($pdo, $_POST['pass']);
    $query = "SELECT * FROM users WHERE user='$user' AND pass='$pass'";

    echo 'Search result: ' . entities_fix_string($_GET['search']);

    //Etc...

    function entities_fix_string($string)
    {
        return htmlentities($string);
    }

    function mysql_fix_string($pdo, $string)
    {
        return $pdo->quote($string);
    }
?>

```

In **Chapter 11**, we’ll expand on ways to access MySQL from PHP by looking at form handling. Before moving on, you can test your knowledge of what you’ve learned in this chapter with the following questions.

Questions

1. How do you connect to a MySQL database using PDO?
2. How do you submit a query to MySQL using PDO?
3. What style of the `fetch` method can be used to return a row as an array indexed by column number?
4. How can you manually close a PDO connection?
5. When adding a row to a table with an `AUTO_INCREMENT` column, what value should be passed to that column?
6. Which PDO method can be used to properly escape user input to prevent code injection?
7. What is the best way to ensure database security when accessing it?

See “[Chapter 10 Answers](#)” on page 575 in the [Appendix](#) for the answers to these questions.

Form Handling

One of the main ways that website users interact with PHP and MySQL is through HTML forms. These were introduced very early on in the development of the World Wide Web, in 1993—even before the advent of ecommerce—and have remained a mainstay ever since, due to their simplicity and ease of use, although formatting them can be a nightmare.

Of course, enhancements have been made over the years to add extra functionality to HTML form handling, so this chapter will bring you up to speed on the state of the art and show you the best ways to implement forms for good usability and security. Plus, the latest HTML specification has further improved the use of forms.

Building Forms

Handling forms is a multipart process. First is the creation of a form into which a user can enter the required details. This data is then sent to the web server, where it is interpreted, often with some error checking. If the PHP code identifies one or more fields that require reentering, the form may be redisplayed with an error message. When the code is satisfied with the validity of the input, it takes some action that may often involve a database, such as entering details about a purchase.

A useful form consists of the following elements:

- An opening `<form>` and closing `</form>` tag
- A submission type specifying either a GET or POST method using the `method` attribute; defaults to GET if omitted

- One or more input fields
- The destination URL in the action attribute, to which the form data is to be submitted; defaults to the same page if not specified

Example 11-1 shows a very simple form created with HTML, which you should type and save as *formtest.php*, or download it from the [examples repo](#).

Example 11-1. formtest.php—a simple HTML form

```
<html>
  <head>
    <title>Form Test</title>
  </head>
  <body>
    <form method="post" action="formtest.php">
      What is your name?
      <input type="text" name="name">
      <input type="submit">
    </form>
  </body>
</html>
```

Inside this multiline output is standard code for commencing an HTML document, displaying its title, and starting the body of the document. This is followed by the form, which is set to send its data using the POST method to the PHP program *formtest.php*, which is the name of the program itself.

The rest of the file closes all the items it opened: the form and the body of the HTML document. The result of opening this program in a web browser is shown in [Figure 11-1](#).

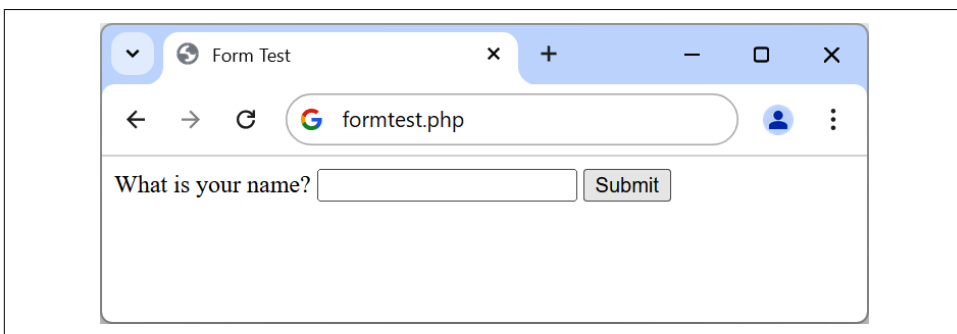


Figure 11-1. The result of opening formtest.php in a web browser

Retrieving Submitted Data

Example 11-1 is one part of the multi-step form-handling process. If you enter a name and click the submit button, it will appear that nothing will happen other than the form being redisplayed (and the entered data lost). So now it's time to add some PHP code to process the data submitted by the form.

Example 11-2 expands on the previous program to include data processing. Type it or modify *formtest.php* by adding in the new lines, save it as *formtest2.php*, and try the program for yourself. The result of entering a name and clicking Submit is shown in **Figure 11-2**.

Example 11-2. Updated version of formtest.php

```
<?php // formtest2.php
if (!empty($_POST['name'])) $name = htmlentities($_POST['name']);
else $name = "(Not Entered)";

echo <<<_END
    <html>
    <head>
        <title>Form Test</title>
    </head>
    <body>
        Your name is: $name<br>
        <form method="post" action="formtest2.php">
            What is your name?
            <input type="text" name="name">
            <input type="submit">
        </form>
    </body>
</html>
_END;
?>
```

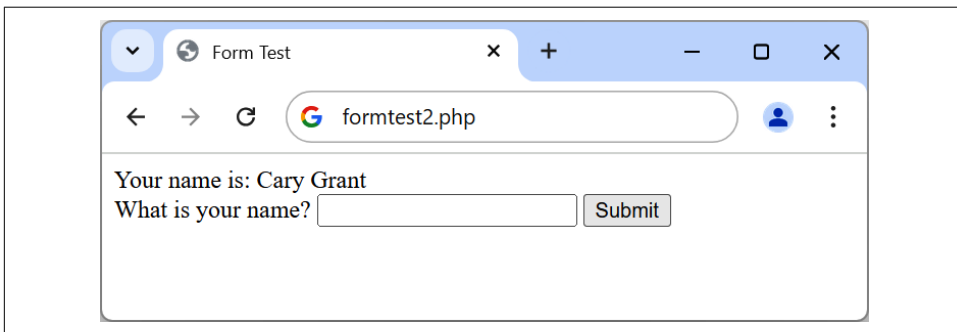


Figure 11-2. formtest2.php with data handling

The first thing to notice about this example is that, as you have seen in earlier chapters, rather than dropping in and out of PHP code, the echo <<<_END..._END heredoc construct is used whenever multiline HTML must be output.

The only other changes are a couple of lines at the start that check the name field of the \$_POST associative array and echo it back to the user. [Chapter 10](#) introduced the \$_POST associative array, which contains an element for each field in an HTML form. In [Example 11-2](#), the input name used was name and the form method was POST, so the element name of the \$_POST array contains the value in \$_POST['name'].

The PHP isset function is used to test whether \$_POST['name'] has been assigned a value. If nothing was posted, the program assigns the value (Not entered); otherwise, it stores the value that was entered. Then a single line has been added after the <body> statement to display that value, which is stored in \$name.

Default Values

Sometimes it's convenient to offer your site visitors a default value in a web form. For example, suppose you put up a loan repayment calculator widget on a real estate website. It could make sense to enter default values of, say, 15 years and 3% interest so that the user can simply type either the principal sum to borrow or the amount that they can afford to pay each month.

In this case, the HTML for those two values would be something like [Example 11-3](#).

Example 11-3. Setting default values

```
<form method="post" action="calc.php"><pre>
    Loan Amount <input type="text" name="principal">
Monthly Repayment <input type="text" name="monthly">
    Number of Years <input type="text" name="years" value="25">
    Interest Rate <input type="text" name="interest" value="6">
    <input type="submit">
</pre></form>
```

Take a look at the third and fourth inputs. By populating the value attribute, you display a default value in the field, which the users can then change if they wish. With sensible default values, you can make your web forms more user-friendly by minimizing unnecessary typing. The result of the previous code looks like [Figure 11-3](#). Of course, this was created to illustrate default values, and, because the program *calc.php* has not been written, the form will return a 404 error message if you submit it.

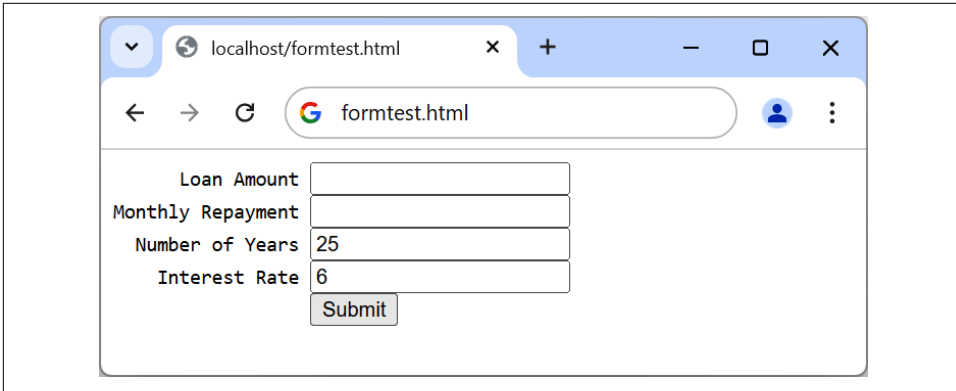


Figure 11-3. Using default values for selected form fields

Input Types

HTML forms are very versatile and allow you to submit a wide range of input types, from text boxes and text areas to checkboxes, radio buttons, and more.

Text boxes

The input type used most often is the text box. It accepts a wide range of alphanumeric text and other characters in a single-line box. The general format of a text box input is:

```
<input type="text" name="name" size="size" maxlength="length" value="value">
```

We've already covered the name and value attributes, but two more are introduced here: size and maxlength. The size attribute specifies the width of the box (in characters of the current font) as it should appear on the screen, and maxlength specifies the maximum number of characters that a user is allowed to enter into the field.

The type attribute, which tells the web browser what type of input to expect, can be omitted since text is the default, but it's recommended to add it even if not required. The name attribute gives the input a name that will be used to process the field upon receipt of the submitted form.

Text areas

When you need to accept input of more than a short line of text, use a text area. This is similar to a text box but, because it allows multiple lines, it has some different attributes. Its general format looks like this:

```
<textarea name="name" cols="width" rows="height" wrap="type">
</textarea>
```

The first thing to notice is that `<textarea>` has its own tag and is not a subtype of the `<input>` tag. It therefore requires a closing `</textarea>` to end input.

Instead of a default attribute, if you have default text to display, you must put it before the closing `</textarea>`, and it will then be displayed and be editable by the user:

```
<textarea name="name" cols="width" rows="height" wrap="type">
  This is some default text.
</textarea>
```

To control the width and height, use the `cols` and `rows` attributes (or CSS). Both use the character spacing of the current font to determine the size of the area. If you omit these values, a default input box will be created that will vary in dimensions depending on the browser used, so you should always define them to be certain about how your form will appear.

Last, you can control how the text entered into the box will wrap (and how any such wrapping will be sent to the server) using the `wrap` attribute. Table 11-1 shows the wrap types available. If you leave out the `wrap` attribute, soft wrapping is used.

Table 11-1. The wrap types available in a `<textarea>` input

Type	Action
off	Text does not wrap, and lines appear exactly as the user types them.
soft	Text wraps but is sent to the server as one long string without carriage returns and line feeds.
hard	Text wraps and is sent to the server in wrapped format with soft or hard returns and line feeds.

Checkboxes

When you want to offer a number of different options to a user, from which they can select one or more items, checkboxes are the way to go. Here is the format to use:

```
<input type="checkbox" name="name" value="value" checked>
```

By default, checkboxes are square. If you include the `checked` attribute, the box is already checked when the page is loaded. The string you assign to the attribute should either be surrounded with double or single quotes or the value `"checked"`, or no value should be assigned (just `checked`). If you don't include the attribute, the box is shown unchecked. Here is an example of creating an unchecked box; we'll talk about the `<label>` tag in a moment:

```
<label for="agree">I Agree</label>
<input type="checkbox" id="agree" name="agree">
```

If the user doesn't check the box, no value will be submitted. But if they do, a value of `"on"` will be submitted for the field named `agree`. If you prefer to have your own value submitted instead of the word *on* (such as the number 1), you could use the following syntax:


```
<label for="agree">I Agree</label>  
<input type="checkbox" id="agree" name="agree" value="1">
```

On the other hand, if you'd like to offer your users a default option to deliver a package to their billing address, for example, you might want to have the checkbox already checked as the default value:

```
<label for="same">Deliver to the same address?</label>  
<input type="checkbox" id="same" name="sameaddress" checked>
```

If you want to allow groups of items to be selected at one time, assign them all the same name. However, only the last item checked will be submitted, unless you pass an array as the name. For example, [Example 11-4](#) allows the user to select their favorite ice creams (see [Figure 11-4](#) for how it displays in a browser, also note that I have left out the label tags for brevity).

Example 11-4. Offering multiple checkbox choices

```
Vanilla <input type="checkbox" name="ice" value="Vanilla">  
Chocolate <input type="checkbox" name="ice" value="Chocolate">  
Strawberry <input type="checkbox" name="ice" value="Strawberry">
```

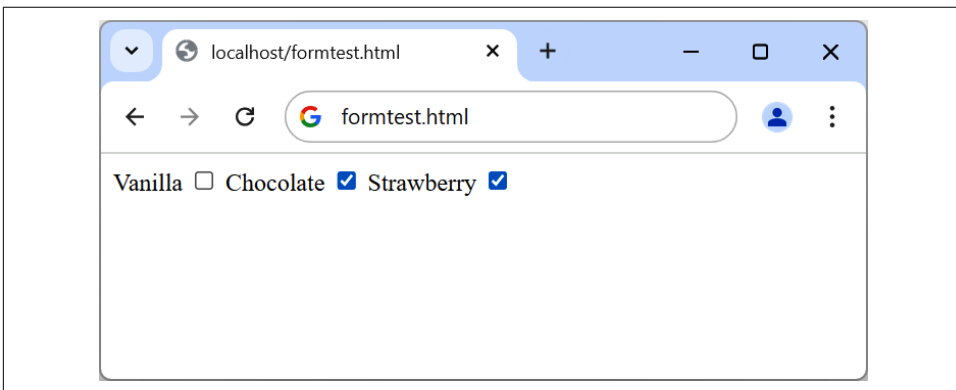


Figure 11-4. Using checkboxes to make quick selections

If only one of the checkboxes is selected, such as the second one, only that item will be submitted (the field named `ice` would be assigned the value "Chocolate"). But if two or more are selected, only the last value will be submitted, with prior values being ignored.

If you *want* exclusive behavior—so that only one item can be submitted—then you should use radio buttons instead (see [“Radio buttons” on page 275](#)). Otherwise, to allow multiple submissions, you have to slightly alter the HTML, as in [Example 11-5](#) (note the addition of the square brackets, `[]`, following the values of `ice`, and again label tags left out for brevity).

Example 11-5. Submitting multiple values with an array

```
Vanilla <input type="checkbox" name="ice[]" value="Vanilla">
Chocolate <input type="checkbox" name="ice[]" value="Chocolate">
Strawberry <input type="checkbox" name="ice[]" value="Strawberry">
```

Now when the form is submitted, if any of these items have been checked, an array called `ice` will be submitted that contains all the selected values. You can extract either the single submitted value or the array of values to a variable like this:

```
$ice = $_POST['ice'];
```

If the field `ice` has been posted as a single value, `$ice` will be a single string, such as "Strawberry". But if `ice` was defined in the form as an array (like in [Example 11-5](#)), `$ice` will be an array, and its number of elements will be the number of values submitted. [Table 11-2](#) shows the seven possible sets of values that could be submitted by this HTML for one, two, or all three selections. In each case, an array of one, two, or three items is created.

Table 11-2. The seven possible sets of values for the array `$ice`

One value submitted	Two values submitted	Three values submitted
<code>\$ice[0] => Vanilla</code>	<code>\$ice[0] => Vanilla</code> <code>\$ice[1] => Chocolate</code>	<code>\$ice[0] => Vanilla</code> <code>\$ice[1] => Chocolate</code> <code>\$ice[2] => Strawberry</code>
<code>\$ice[0] => Chocolate</code>	<code>\$ice[0] => Vanilla</code> <code>\$ice[1] => Strawberry</code>	
<code>\$ice[0] => Strawberry</code>	<code>\$ice[0] => Chocolate</code> <code>\$ice[1] => Strawberry</code>	

If `$ice` is an array, the PHP code to display its contents is quite simple and might look like this:

```
foreach($ice as $item) echo "$item<br>";
```

This uses the standard PHP `foreach` construct to iterate through the array `$ice` and pass each element's value into the variable `$item`, which is then displayed via the `echo` command. The `<br>` is just an HTML formatting device to force a new line after each flavor in the display.

Labels

You can provide an even better user experience by utilizing the `<label>` tag. Going back to the delivery address example, it uses a `label` tag, which is explicitly associated with an `input` (a checkbox in this case) by using the `for` and `id` attributes. This allows the user to click the checkbox itself *and* the associated text:

```
<label for="same">Deliver to the same address?</label>
<input type="checkbox" id="same" name="sameaddress" checked>
```

A label tag can also surround a form element (no need for the for and id attributes), making it selectable by clicking any visible part contained between the opening and closing <label> tags:

```
<label>
  Deliver to the same address?
  <input type="checkbox" name="sameaddress" checked>
</label>
```

The text will not be underlined like a hyperlink when you add a label, but as the mouse pointer passes over it, it will change to an arrow instead of a text cursor, indicating that the whole item is clickable.

Labels can be added to all form fields, not just checkboxes, and we'll be using them extensively in the following examples.

Radio buttons

Radio buttons are named after the push-in preset buttons found on many older radios, where any previously depressed button pops back up when another is pressed. They are used when you want only a single value to be returned from a selection of two or more options. All the buttons in a group must use the same name, and, because only a single value is returned, you do not have to pass an array.

For example, if your website offers a choice of delivery times for items purchased from your store, you might use HTML like that in [Example 11-6](#) (see [Figure 11-5](#) to see how it displays). By default, radio buttons are round.

Example 11-6. Using radio buttons

```
<label for="morning">8am-Noon</label>
  <input type="radio" id="morning" name="time" value="1">
<label for="afternoon">Noon-4pm</label>
  <input type="radio" id="afternoon" name="time" value="2" checked>
<label for="evening">4pm-8pm</label>
  <input type="radio" id="evening" name="time" value="3">
```

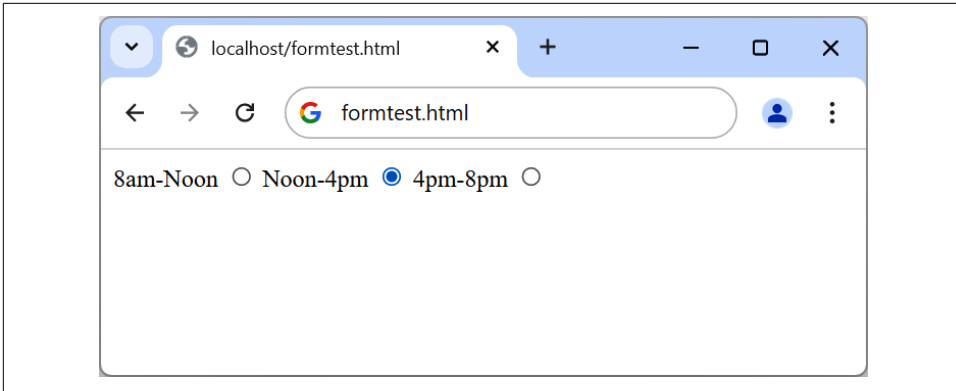


Figure 11-5. Selecting a single value with radio buttons

Here, the second option of Noon–4pm has been selected by default. This default choice ensures that at least one delivery time will be chosen by the user, which they can change to one of the other two options if they prefer. Had one of the items not been already checked, the user might forget to select an option, and no value would be submitted for the delivery time.

Unlike checkboxes, once selected, radio buttons cannot be deselected, so if you would like to provide an option like “no preference” you should make it an explicit radio button.

Hidden fields

Sometimes it is convenient to have hidden form fields so that you can keep track of the state of form entry. For example, you might wish to know whether a form has already been submitted. You can achieve this by adding some HTML in your PHP code, such as:

```
<input type="hidden" name="submitted" value="yes">
```

Let’s assume the form was created outside the program, without the hidden field, and displayed to the user. The first time the PHP program receives the input, the hidden field is missing, so there will be no field named `submitted`. The PHP program re-creates the form, adding the hidden input field. So when the visitor resubmits the form, the PHP program receives it with the `submitted` field set to “yes”. The code can simply check whether the field is present:

```
if (isset($_POST['submitted']))  
{...
```

Hidden fields can also be useful for storing other details, such as an ID string that you might create to identify a user, and so on.



Never treat hidden fields as secure—because they are not. Someone could easily view the HTML containing them by using a browser's View Source feature. A malicious attacker could also craft a post that removes, adds, or changes a hidden field.

<select>

The <select> tag lets you create a drop-down list of options, offering either single or multiple selections. It conforms to the following syntax:

```
<select name="name" size="size" multiple>
```

The attribute `size` is the number of lines to display before the dropdown is expanded; the default is 1. Clicking on the display causes a list to drop down, showing all the options. If you use the optional `multiple` attribute, a user can select multiple options from the list by pressing the Ctrl key when clicking. So, to ask a user for their favorite vegetable from a choice of five, you might use HTML like that in [Example 11-7](#), which offers a single selection.

Example 11-7. Using <select>

Vegetables

```
<select name="veg">
  <option value="Peas">Peas</option>
  <option value="Beans">Beans</option>
  <option value="Carrots">Carrots</option>
  <option value="Cabbage">Cabbage</option>
  <option value="Broccoli">Broccoli</option>
</select>
```

This HTML offers five choices, with the first one, *Peas*, preselected (due to it being the first item). [Figure 11-6](#) shows the output where the list has been clicked to drop it down, and the option *Carrots* has been highlighted. If you want to have a different default option offered first (such as *Beans*), use the `selected` attribute, like this:

```
<option selected value="Beans">Beans</option>
```

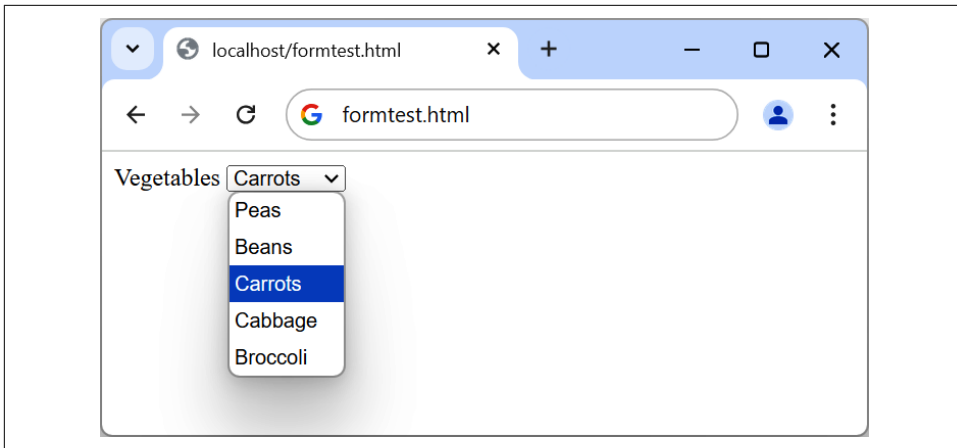


Figure 11-6. Creating a drop-down list with `<select>`

You can also allow users to select more than one item, as in [Example 11-8](#).

Example 11-8. Using `<select>` with the `multiple` attribute

```
Vegetables
<select name="veg" size="5" multiple>
  <option value="Peas">Peas</option>
  <option value="Beans">Beans</option>
  <option value="Carrots">Carrots</option>
  <option value="Cabbage">Cabbage</option>
  <option value="Broccoli">Broccoli</option>
</select>
```

This HTML is not very different; the size has been changed to "5", and the attribute `multiple` has been added. But, as you can see from [Figure 11-7](#), it is now possible for the user to select more than one option by using the Ctrl key when clicking. You can leave out the `size` attribute if you wish, and the output will be the same; however, with a larger list, the drop-down box may display more items, so I recommend that you pick a suitable number of rows and stick with it. I also recommend not using multiple select boxes smaller than two rows in height—some browsers may not correctly display the scroll bars needed to access them.

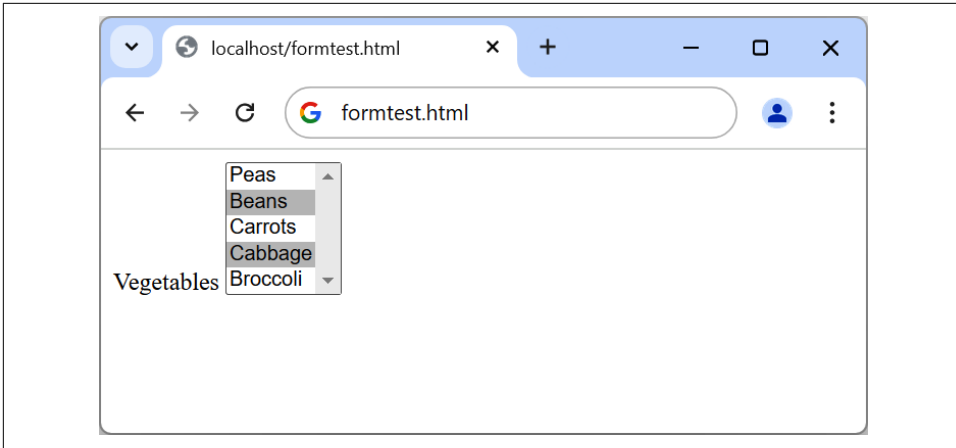


Figure 11-7. Using a `<select>` with the `multiple` attribute

You can also use the `selected` attribute within a multiple select and can, in fact, have more than one option preselected if you wish.

The submit button

To match the type of form being submitted, you can change the text of the submit button to anything you like by using the `value` attribute, like this:

```
<input type="submit" value="Search">
```

You can also replace the standard text button with a graphic image of your choice, using HTML:

```
<input type="image" name="submit" src="image.gif" alt="Submit">
```

Instead of an `<input>` tag, you can also use a `<button>` tag to create a submit button; the advantage is that you can use other HTML tags to style the text or even include an image:

```
<button><em>Search</em></button>  
<button></button>
```

The autocomplete attribute

You can apply the `autocomplete` attribute to the `<form>` element, or to any of the `color`, `date`, `email`, `password`, `range`, `search`, `tel`, `text`, or `url` types of the `<input>` element.

With autocomplete enabled, previous user inputs are recalled and automatically entered into fields as suggestions. You can also disable this feature by turning autocomplete off. Here's how to turn autocomplete on for an entire form but disable it for specific fields (highlighted in bold):

```
<form action='myform.php' method='post' autocomplete='on'>
  <input type='text' name='type'>
  <input type='text' name='amount' autocomplete='off'>
</form>
```

There are many possible values for the autocomplete attribute, for example email, username, current-password, and new-password to help password managers prefill the fields with respective values. For the full list please visit the [MDN page on the autocomplete attribute](#).

The autofocus attribute

The autofocus attribute gives immediate focus to an element when a page loads. It can be applied to any <input>, <textarea>, or <button> element, like this:

```
<input type='text' name='query' autofocus='autofocus'>
```



Browsers that use touch interfaces (such as Android or iOS) usually ignore the autofocus attribute, leaving it to the user to tap on a field to give it focus; otherwise, the zooming, focusing, and pop-up keyboards this attribute would generate could quickly become annoying.

Because this feature will cause the focus to move into an input element, the Backspace key will no longer take the user back a web page (although Alt-Left arrow and Alt-Right arrow will still move backward and forward within the browsing history).

The placeholder attribute

The placeholder attribute lets you place into any blank input field a helpful hint to explain to users what they should enter. You use it like this:

```
<input type='text' name='name' size='50' placeholder='First & Last name'>
```

The input field will display the placeholder text as a prompt until the user starts typing, at which point the placeholder will disappear.

The required attribute

The required attribute ensures that a field has been completed before a form is submitted:

```
<input type='text' name='creditcard' required>
```


When the browser detects an attempted form submission where there's an uncompleted required input, a message is displayed, prompting the user to complete the field.

Override attributes

With override attributes, you can override form settings on an element-by-element basis. So, for example, using the `formaction` attribute, you can specify that a submit button should submit a form to a different URL from the one specified in the form itself, like the following (in which the default and overridden action URLs are bold):

```
<form action='url1.php' method='post'>
  <input type='text' name='field'>
  <input type='submit' formaction='url2.php'>
</form>
```

HTML also brings support for the `formenctype`, `formmethod`, `formnovalidate`, and `formtarget` override attributes, which you can use in exactly the same manner as `formaction` to override one of these settings.

The width and height attributes

Using these new attributes, you can alter the displayed dimensions of an input image, like this:

```
<input type='image' src='picture.png' width='120' height='80'>
```

The step attribute

Often used with `min` and `max`, the `step` attribute supports stepping through number or date values, like this:

```
<input type='time' name='meeting' value='12:00'
  min='09:00' max='16:00' step='3600'>
```

When you are stepping through date or time values, each unit represents 1 second.

The form attribute

You no longer have to place `<input>` elements within `<form>` elements, because you can specify the form to which an input applies by supplying a `form` attribute. The following code shows a form being created, but with its input outside of the `<form>` and `</form>` tags:

```
<form action='myscript.php' method='post' id='form1'>
</form>

<input type='text' name='username' form='form1'>
```

To do this, you must give the form an ID using the `id` attribute and refer to this ID in the `form` attribute of the input element. This is most useful for adding hidden input

fields when you can't control how or if the field is placed inside the `<form>` tag in the HTML code, or for using JavaScript to modify forms and inputs on the fly.

The list attribute

Attaching lists to inputs enables users to easily select from a predefined list, which you can use like this:

```
Select destination:
<input type='url' name='site' list='links'>

<datalist id='links'>
  <option label='Google' value='http://google.com'>
  <option label='Yahoo!' value='http://yahoo.com'>
  <option label='Bing' value='http://bing.com'>
  <option label='Ask' value='http://ask.com'>
</datalist>
```

The color input type

The color input type calls up a color picker so that you can simply click the color of your choice. After submitting the form, the server receives the color in the hex format. You use the input like this:

```
Choose a color <input type='color' name='color'>
```

The min and max attributes

With the `min` and `max` attributes, you can specify minimum and maximum values for inputs. The browser will then either offer up and down selectors for the range of values allowed or simply disallow values outside of that range or mark such values as invalid. See the following types for example usage.

The number and range input types

The number and range input types restrict input to a number and optionally also specify an allowed range, like this:

```
<input type='number' name='age'>
<input type='range' name='num' min='0' max='100' value='50' step='1'>
```

Date and time pickers

When you choose an input type of date, month, week, time, datetime, or datetime-local, a picker will pop up on supported browsers from which the user can make a selection, like this one, which inputs the time:

```
<input type='time' name='time' value='12:34' min='09:00' max='17:00'>
```

Chapter 12 will show you how to use cookies and authentication to store users' preferences and keep them logged in, and how to maintain a complete user session.

Sanitizing Input

Now we return to PHP programming. It can't be emphasized enough that handling user data is a security minefield, and it is essential to learn to treat all such data with the utmost caution from the start. It's not that difficult to sanitize user input from potential hacking attempts, and it must be done.

Remember that the safest way to secure MySQL from hacking attempts is to use placeholders and prepared statements, as described in Chapter 10. If you do so for all accesses to MySQL, it is not necessary to manually escape data being transferred into or out of the database. You will, however, still need to sanitize input when including it within HTML.

The first thing to remember is that regardless of any constraints you have placed in an HTML form to limit the types and sizes of inputs, it is a trivial matter for a hacker to use their browser's View Source feature to extract the form and modify it to provide malicious input to your website.

To prevent such attacks, you must never trust any variable that you fetch from either the `$_GET` or `$_POST` arrays until you have sanitized it. If you don't, users may try to inject JavaScript into the data to interfere with your site's operation, or even attempt to add MySQL commands to compromise your database.

Preventing SQL injection is easy. Use prepared statements and placeholders as described in Chapter 10. Script injection and XSS attacks can be stopped by using the `htmlspecialchars` function:

```
$variable = htmlspecialchars($variable);
```

For example, this would change a string of interpretable HTML code like `<b>hi</b>` into `<b>hi</b>`, which then displays as text and won't be interpreted as HTML tags.

The `htmlspecialchars` function is identical to `htmlspecialchars`, and both can be used to stop the attacks, but where the former converts all characters which have HTML entity equivalents, the latter converts only the special ones:

```
htmlspecialchars("caffè d'orzo"); // the result is caffè&grave; d&#039;orzo
htmlspecialchars("caffè d'orzo") // the result is caffè d&#039;orzo
```



This book uses `htmlspecialchars` in the following examples as it's slightly shorter to write, but you could as well use `htmlspecialchars` chars.

If you use `htmlentities` before storing data in your database, and then when the data is retrieved from the database to be rendered to an HTML page, and the component or code that renders it calls `htmlentities` a second time, you probably won't get what you want. It'll double-encode and mangle legitimate quotation marks, ampersands, and angle brackets. You want to call the function just once.

If you legitimately want to allow user-provided HTML to be rendered as HTML (which is common, for example, with popular WYSIWYG editors), look to a robust tool such as the [DOMPurify library available on GitHub](#).

Since the danger in user-provided content is at the time of *use* (as opposed to the time that it is *submitted*; or at least PHP handles most of those dangers for you), you may want to consider deferring sanitization until you know the requirements for the output data. You may even use the same piece of data in different contexts: for example, a user-provided field in a database could be rendered to a web page, a mobile app, a text email, an HTML email, and SMS. Each may have different sanitization needs and concerns.

Having solid documentation in your system about the nature of the content in your database is more valuable than rote sanitization. Even such a simple expedient as suffixing fields that may contain HTML with `_html` can be helpful. You could expand this by having suffixes like `_safe_html` (after having been run through something like DOMPurify), or `_html_entities` (for text that was run through `htmlentities`).

If you'd like to dive deeper into XSS prevention, you can check out the article by OWASP (Open Worldwide Application Security Project) published in its [Cheat Sheet Series](#).



Stripping HTML Is Not Enough

The function used to strip HTML from an input, `strip_tags`, won't reliably prevent XSS attacks and, depending on the input, can produce mangled HTML.

An Example Program

Let's look at how a real-life PHP program integrates with an HTML form by creating the program *convert.php* listed in [Example 11-9](#). Type it as shown and try it for yourself.

Example 11-9. A program to convert values between Fahrenheit and Celsius

```
<?php // convert.php
$f = $c = $f_html_entities = $c_html_entities = '';

if (isset($_POST['f'])) {
    $f = $_POST['f'];
    $f_html_entities = htmlentities($f);
}
if (isset($_POST['c'])) {
    $c = $_POST['c'];
    $c_html_entities = htmlentities($c);
}

if (is_numeric($f)) {
    $c = intval((5 / 9) * ($f - 32));
    $out = "$f &deg;F equals $c &deg;C";
} elseif (is_numeric($c)) {
    $f = intval((9 / 5) * $c + 32);
    $out = "$c &deg;C equals $f &deg;F";
} else
    $out = "";

?>
<html>
  <head>
    <title>Temperature Converter</title>
  </head>
  <body>
    <pre>
      Enter either Fahrenheit or Celsius and click on Convert

      <b><?php echo $out; ?></b>
      <form method="post" action="">
        <label>Fahrenheit <input type="text" name="f"
          value="<?php echo $f_html_entities; ?>" size="7"></label>
        <label>Celsius <input type="text" name="c"
          value="<?php echo $c_html_entities; ?>" size="7"></label>
        <input type="submit" value="Convert">
      </form>
    </pre>
  </body>
</html>
```

When you call up *convert.php* in a browser, the result should look something like Figure 11-8.

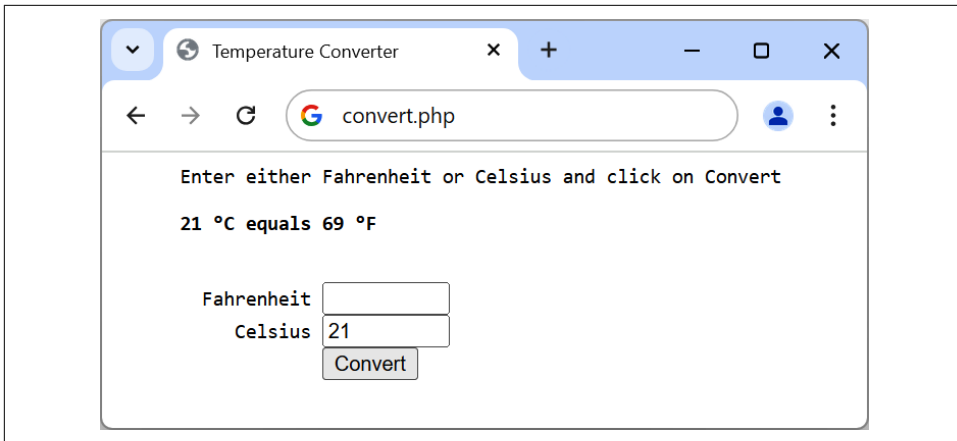


Figure 11-8. The temperature conversion program in action

Let's break down this code. The first line initializes the variables `$c`, `$f`, `$f_html_entities`, and `$c_html_entities` in case the respective form fields do not get posted to the program. The next two `if` blocks fetch the values of either the field named `f` or the one named `c`, for an input Fahrenheit or Celsius value, and create sanitized values by calling `htmlentities`. If the user inputs both, the Celsius is simply ignored and the Fahrenheit value is converted. The values will be echoed back to the form later, and the sanitization prevents the XSS attack.

So, having either submitted values or empty strings in both `$f` and `$c`, the next portion of code constitutes an `if...elseif...else` structure that first tests whether `$f` has a numeric value. If not, it checks `$c`; if `$c` does not have a numeric value, the variable `$out` is set to the empty string (more on that in a moment).

If `$f` has a numeric value, the variable `$c` is assigned a simple mathematical expression that converts the value of `$f` from Fahrenheit to Celsius. The formula used is $\text{Celsius} = (5 / 9) \times (\text{Fahrenheit} - 32)$. The variable `$out` is then set to a message string explaining the conversion.

On the other hand, if `$c` has a numeric value, a complementary operation is performed to convert the value of `$c` from Celsius to Fahrenheit and assign the result to `$f`. The formula used is $\text{Fahrenheit} = (9 / 5) \times \text{Celsius} + 32$. Then again, the string `$out` is set to contain a message about the conversion.

In both conversions, the PHP `intval` function is called to convert the result of the conversion to an integer value. It's not necessary, but it looks better.

With all the arithmetic done, the program now outputs the HTML, which starts with the basic head and title and then contains some introductory text before displaying the value of `$out`. If no temperature conversion was made, `$out` will have a value of

NULL and nothing will be displayed, which is exactly what we want when the form hasn't yet been submitted. But if a conversion was made, `$out` contains the result, which is displayed.

After this, we come to the form, which is set to submit using the POST method to the program itself (represented by a pair of double quotation marks so that the file can be saved with any name). Within the form, there are two inputs for either a Fahrenheit or a Celsius value to be entered. The original entered value is printed in the `value` attribute of the respective field and is sanitized, because it's a user input. A submit button with the text Convert is then displayed, and the form is closed.

Try playing with the example by inputting different values into the fields; for a bit of fun, can you find a value for which Fahrenheit and Celsius are the same? You may also try entering HTML (for example `>XSS here`) to see why calling `htmlspecialchars` is important. Then remove the `htmlspecialchars` call and try inputting the same HTML again; you should see a broken input field with the HTML you have injected. Don't forget to put the `htmlspecialchars` call back after you're done playing.



All the examples in this chapter have used the POST method to send form data. I recommend this, as it's the neatest and most secure method. However, the forms can easily be changed to use the GET method, as long as values are fetched from the `$_GET` array instead of the `$_POST` array. Reasons to do this might include making the result of a search bookmarkable or directly linkable from another page.

At this point, you should be familiar with various form fields and able to process them in PHP. This will be useful in [Chapter 12](#), where you'll learn about logins and sessions. But before that, let's refresh what you've learned by answering these questions.

Questions

1. You can submit form data using either the POST or the GET method. Which associative arrays are used to pass this data to PHP?
2. What is the difference between a text box and a text area?
3. If a form needs to offer three choices to a user, each of which is mutually exclusive so that only one of the three can be selected, which input type would you use, given a choice between checkboxes and radio buttons?
4. How can you submit a group of selections from a web form using a single field name?
5. How can you submit a form field without displaying it in the browser?

6. Which HTML tag is used to encapsulate a form element and supporting text or graphics, making the entire unit selectable with a mouse-click?
7. Which PHP function converts HTML into a format that can be displayed but will not be interpreted as HTML by a browser, preventing attacks like XSS?
8. What form attribute can be used to help users complete input fields?
9. How can you ensure that an input is completed before a form gets submitted?

See “[Chapter 11 Answers](#)” on [page 575](#) in the [Appendix](#) for the answers to these questions.

Cookies, Sessions, and Authentication

As your web projects grow larger and more complicated, you will find an increasing need to keep track of your users. Even if you aren't offering logins and passwords, you still will often need to store details about a user's current session and possibly also recognize them when they return to your site.

Several technologies support this kind of interaction, ranging from simple browser cookies to session handling and HTTP authentication. Between them, they offer the opportunity for you to configure your site to your users' preferences and ensure a smooth and enjoyable transition through it.

Using Cookies in PHP

A *cookie* is an item of data that a web server saves to your computer's hard disk via a web browser. It can contain almost any alphanumeric information (as long as it's under 4 KB) and can be retrieved from your computer and returned to the server. Common uses include session tracking and identifiers, maintaining data across multiple visits, holding shopping cart contents, storing non-secure login details (not passwords), and more.

Because of their privacy implications, cookies can be read only from the issuing domain. In other words, if a cookie is issued by, for example, *oreilly.com*, it can be retrieved only by a web server using that domain. This prevents other websites from gaining access to details for which they are not authorized.

Because of the way the internet works, multiple elements on a web page can be embedded from multiple domains, each of which can issue its own cookies. When this happens, they are referred to as *third-party cookies*. Most commonly, these are created by advertising companies to track users across multiple websites or for analytic purposes.

Because of this, most browsers allow users to turn cookies off either for the current server's domain, third-party servers, or both. Fortunately, most people who disable cookies do so only for third-party websites.

Cookies are exchanged during the transfer of headers, before the actual HTML of a web page is sent in the response body, and it is impossible to send a cookie once any HTML has been transferred. Therefore, careful planning of cookie usage is important. **Figure 12-1** illustrates a typical request and response dialog between a web browser and web server passing cookies.

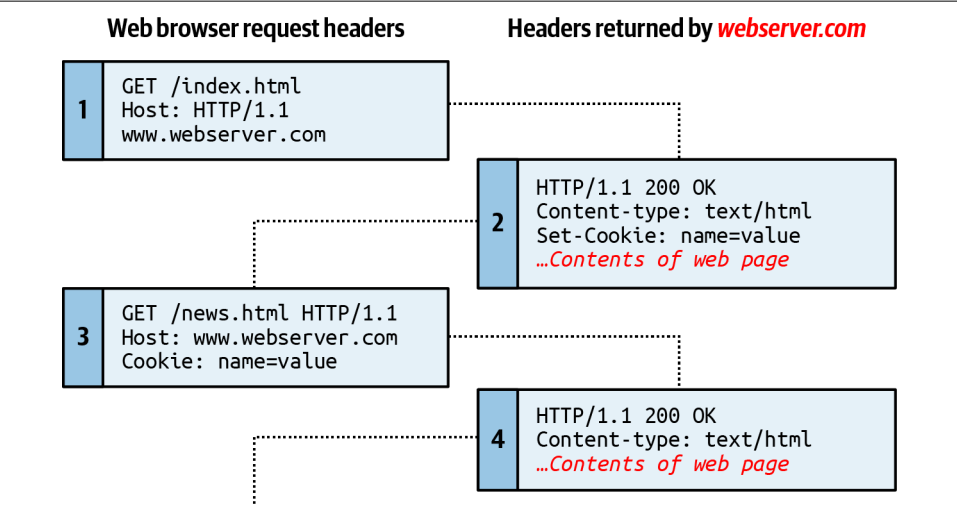


Figure 12-1. A browser/server request/response dialog with cookies

This exchange shows a browser receiving two pages:

1. The browser issues a request to retrieve the main page, *index.html*, at the website *http://www.webserver.com*. The first line specifies the file, and the second header specifies the server.
2. When the web server at *webserver.com* receives this pair of headers, it returns some of its own. The second header defines the type of content to be sent (*text/html*), and the third one sends a cookie of the name *name* and with the value *value*. Only then are the contents of the web page transferred.
3. Once the browser has received the cookie, it will then return it with every future request made to the issuing server until the cookie expires or is deleted. So, when the browser requests the new page */news.html*, it also returns the cookie *name* with the value *value*.

4. Because the cookie has already been set, when the server receives the request to send `/news.html`, it does not have to resend the cookie but just returns the requested page.



It is relatively straightforward to edit cookies directly from within the browser by using built-in developer tools or extensions. Therefore, because users can change cookie values, you should not put key information such as usernames in a cookie and trust it blindly without verifying it, or you face the possibility of having your website manipulated in unexpected ways. Cookies are best used for storing data such as language or currency settings.

Setting a Cookie

Setting a cookie in PHP is simple. As long as no HTML has yet been transferred, you can call the `setcookie` function (see [Table 12-1](#)), which has the following syntax:

```
setcookie(name, value, expire, path, domain, secure, httponly);
```

Table 12-1. The `setcookie` parameters

Parameter	Description	Example
name	The name of the cookie. This is the name that your server will use to access the cookie on subsequent browser requests.	location
value	The value of the cookie or the cookie's contents. This can contain up to 4 KB of alphanumeric text.	USA
expire	(Optional.) The Unix timestamp of the expiration date. Generally, you will probably use <code>time()</code> plus a number of seconds. If not set, the cookie becomes a session cookie that can be deleted when the browser is closed, but you can't rely on that behavior: many browsers will restore session cookies when the browser restarts.	<code>time() + 2592000</code>
path	(Optional.) The path of the cookie on the server. If this is a <code>/</code> (forward slash), the cookie is available over the entire domain, such as <code>www.webserver.com</code> . If it is a subdirectory, the cookie is available only within that subdirectory. The default is the current directory that the cookie is being set in, and this is the setting you will normally use.	<code>/</code>
domain	(Optional.) The internet domain of the cookie. If this is <code>webserver.com</code> , the cookie is available to all of <code>webserver.com</code> and its subdomains, such as <code>www.webserver.com</code> and <code>images.webserver.com</code> . If it is <code>images.webserver.com</code> , the cookie is available only to <code>images.webserver.com</code> and its subdomains, such as <code>sub.images.webserver.com</code> but not, say, to <code>www.webserver.com</code> .	<code>webserver.com</code>
secure	(Optional.) Whether the cookie must use a secure connection (<code>https://</code>). If this value is <code>TRUE</code> , the cookie can be transferred only across a secure connection. The default is <code>FALSE</code> .	<code>FALSE</code>
httponly	(Optional; implemented since PHP version 5.2.0.) Whether the cookie must use the HTTP protocol. If this value is <code>TRUE</code> , scripting languages such as JavaScript cannot access the cookie. The default is <code>FALSE</code> .	<code>FALSE</code>

So, to create a cookie with the name `location` and the value `USA` that is accessible across the entire web server on the current domain and that will be removed from the browser's cache in seven days, use:

```
setcookie('location', 'USA', time() + 60 * 60 * 24 * 7, '/');
```

Accessing a Cookie

Reading the value of a cookie is as simple as accessing the `$_COOKIE` system array. For example, if you wish to see whether the browser that issued the request already stores the cookie called `location` and, if so, to read its value, use:

```
if (isset($_COOKIE['location'])) $location = $_COOKIE['location'];
```

Note that you can read a cookie back only after it has been sent to a web browser. This means when you issue a cookie, you cannot read it in again until the browser reloads the page (or another with access to the cookie) from your website and passes the cookie back to the server in the process.

Destroying a Cookie

To delete a cookie, you must issue it again and set a date in the past. It is important for all parameters in your new `setcookie` call except the timestamp to be identical to the parameters when the cookie was first issued; otherwise, the deletion will fail. Therefore, to delete the cookie created earlier, you would use:

```
setcookie('location', 'USA', time() - 2592000, '/');
```

As long as the time given is in the past, the cookie should be deleted. However, I have used a time of 2,592,000 seconds (one month) in the past in case the client computer's date and time are not correctly set. You can also provide an empty string for the cookie value (or a value of `FALSE`), and PHP will automatically set its time in the past.

HTTP Authentication

HTTP authentication uses the web server to manage users and passwords for the application. It's adequate for simple applications that ask users to log in, although most applications will have specialized needs or more stringent security requirements that call for other techniques.

To use HTTP authentication, PHP sends a header request asking to start an authentication dialog with the browser. The server must have this feature turned on for it to work, but because it's so common, your server is very likely to offer the feature.



Although it is usually installed with Apache, the HTTP authentication module may not necessarily be installed on the server you use. So, attempting to run these examples could generate an error telling you that the feature is not enabled, in which case you must either install the module and change the configuration file to load it or ask your system administrator to make these changes.

After entering your URL into the browser or visiting the page via a link, the user will see an “Authentication Required” prompt pop-up, requesting two fields: Username and Password (Figure 12-2 shows how this looks in Firefox).

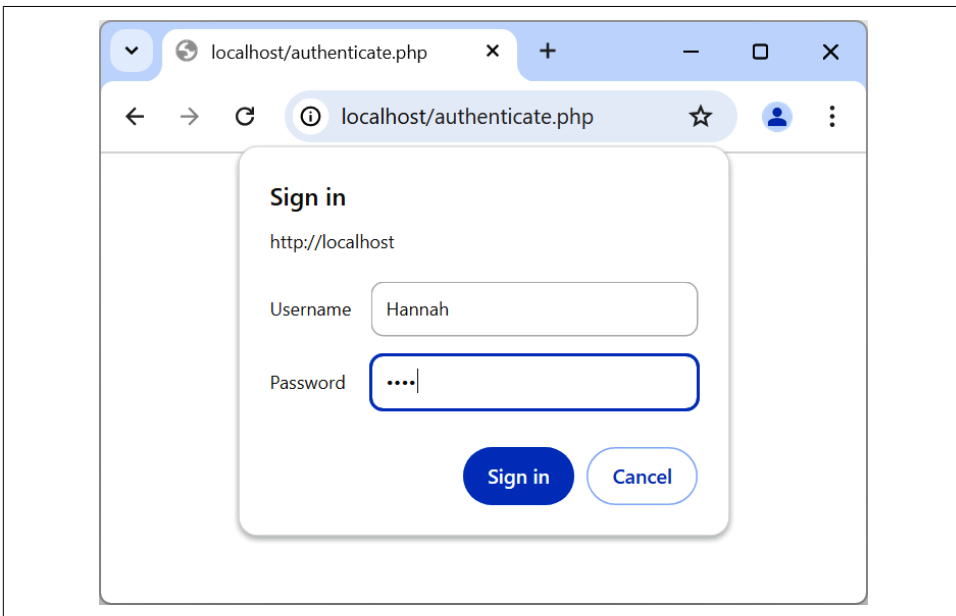


Figure 12-2. An HTTP authentication login prompt

Example 12-1 shows the code to make this happen.

Example 12-1. PHP authentication

```
<?php
if (isset($_SERVER['PHP_AUTH_USER']) &&
    isset($_SERVER['PHP_AUTH_PW']))
{
    echo "Welcome User: " . htmlspecialchars($_SERVER['PHP_AUTH_USER']) .
        " Password: " . htmlspecialchars($_SERVER['PHP_AUTH_PW']);
}
else
{
    header('WWW-Authenticate: Basic realm="Restricted Area"');
```

```

header('HTTP/1.1 401 Unauthorized');
die("Please enter your username and password");
}
?>

```

The first thing the program does is look for two particular array values: `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`. If they both exist, they represent the username and password entered by a user into an authentication prompt.



Notice that when being displayed to the screen, the values that have been returned in the `$_SERVER` array are first processed through the `htmlspecialchars` function. This is because these values have been entered by the user and therefore cannot be trusted, as a hacker could make a cross-site scripting attempt by adding HTML characters and other symbols to the input. `htmlspecialchars` translates any such input into harmless HTML entities.

If either value does not exist, the user has not yet been authenticated, and you display the prompt in [Figure 12-2](#) by issuing the following header, where `Basic realm` is the name of the section that is protected and appears as part of the pop-up prompt:

```
WWW-Authenticate: Basic realm="Restricted Area"
```

If the user fills out the fields, the PHP program runs again from the top. But if the user clicks the Cancel button, the program proceeds to the following two lines, which send the following header and an error message:

```
HTTP/1.1 401 Unauthorized
```

The `die` statement causes the text “Please enter your username and password” to be displayed (see [Figure 12-3](#)).

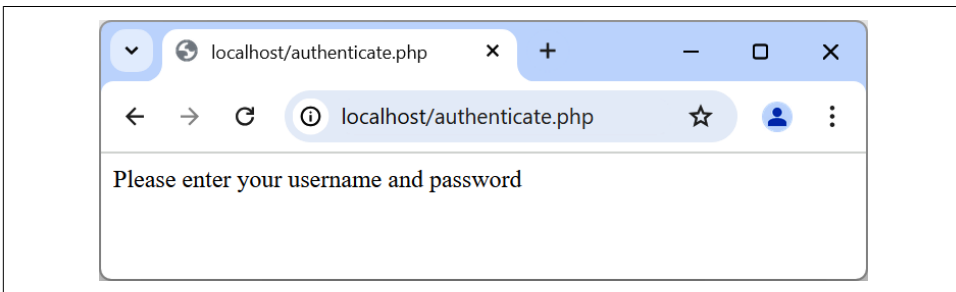


Figure 12-3. The result of clicking the Cancel button



Once a user has been authenticated, you will not be able to get the authentication dialog to pop up again unless the user closes and reopens all browser windows, because the web browser will keep returning the same username and password to PHP. You may need to close and reopen your browser a few times as you work through this section and try different things. The easiest way to do this is to open up a new private or anonymous window to run these examples, so you won't need to close the entire browser.

Now let's check for a valid username and password. The code in [Example 12-1](#) doesn't require you to change much to add this check, other than modifying the previous welcome message code to test for a correct username and password and then issuing a welcome message. A failed authentication causes an error message to be sent (see [Example 12-2](#)).

Example 12-2. PHP authentication with input checking

```
<?php
$username = 'admin';
$password = 'letmein';

if (isset($_SERVER['PHP_AUTH_USER']) &&
    isset($_SERVER['PHP_AUTH_PW']))
{
    if ($_SERVER['PHP_AUTH_USER'] === $username &&
        $_SERVER['PHP_AUTH_PW'] === $password)
        echo "You are now logged in";
    else die("Invalid username/password combination");
}
else
{
    header('WWW-Authenticate: Basic realm="Restricted Area"');
    header('HTTP/1.0 401 Unauthorized');
    die ("Please enter your username and password");
}
?>
```

When comparing usernames and passwords the `===` (identity) operator is used, rather than the `==` (equals) operator. This is because we are checking whether the two values match *exactly*. The equality operator (`==`) is not suitable for comparing login information, because for example, `'0e123' == '0e456'` returns true, and this is not a suitable match for either username or password purposes.

In the previous instance, PHP automatically converted the strings to numbers, where `0e123` is 0 times 10 raised to the 123rd power, which results in zero, and `0e456` is 0 times 10 raised to the 456th power, which also evaluates to zero. Therefore, using the `==` operator, they will match due to their values both evaluating to zero, and so the

result of the comparison will be `true`, but the `===` operator says that the two parts must be identical in every way, and as these two strings are different, the test will return `false`.

A mechanism is now in place to authenticate users, but only for a single username and password. Also, the password appears in clear text within the PHP file, and if someone managed to hack into your server, they would instantly know it. So, let's look at a better way to handle usernames and passwords.

Storing Usernames and Passwords

MySQL is a natural way to store usernames and passwords. But again, we don't want to store the passwords as clear text, because our website could be compromised if the database were accessed by a hacker. Instead, we'll use a neat trick called a *one-way function*.

This simple function converts a string of text into a seemingly random string. Because they are one-way, such functions are impossible to reverse, so their output can be safely stored in a database—and anyone who steals it will be none the wiser as to the passwords used.

In previous editions of this book, I recommended using the *MD5* hashing algorithm for your data security. Time marches on, however, and now MD5 is considered easily hackable and therefore unsafe. Indeed, even its previously recommended replacement of *SHA-1* can be hacked.

So, now that PHP 5.5 is pretty much the minimum standard everywhere, I have moved on to using its built-in hashing function, which is vastly more secure and neatly handles everything for you.

Previously, to store a password securely, you would have needed to *salt* the password, which is a term for adding extra characters to a password that the user did not enter (to further obscure it). You then needed to run that new string through a one-way function to turn it into a seemingly random set of characters, which used to be hard to crack.

For example, code such as the following (which is now *very insecure*, because modern graphics processing units have such speed and power):

```
echo hash('ripemd128', 'saltstringmypassword');
```

would display this value:

```
9eb8eb0584f82e5d505489e6928741e7
```

Remember this method is *never* recommended. Treat this as an example of what *not* to do, as it is very insecure. Instead, please read on.

Using password_hash

From version 5.5 of PHP, there's a far better way to create salted password hashes: the `password_hash` function. Supply `PASSWORD_DEFAULT` as its second (required) argument to ask the function to select the most secure hashing function currently available. `password_hash` will also choose a random salt for every password. (Don't be tempted to add any more salting of your own, as this could compromise the algorithm's security.) So, the following code:

```
echo password_hash("mypassword", PASSWORD_DEFAULT);
```

will return a string such as the following, which includes the salt and all information required for the password to be verified:

```
$2y$10$K0Yljbc2dmmCq8WKGf8oteBGiXlM9Zx0ss4PEtb5kz22EoIkXBtbG
```



If you are letting PHP choose the hashing algorithm for you, you should allow for the returned hash to expand in size over time as better security is implemented. The developers of PHP recommend that you store hashes in a database field that can expand to at least 255 characters (even though 60–72 seems to be around the current length at the time of writing). Should you wish, you can manually select the BCrypt algorithm to guarantee a hash string of only 60 characters by supplying the constant `PASSWORD_BCRYPT` as the second argument to the function. However, I don't recommend this unless you have a very good reason.

You can supply options (in the form of an optional third argument) to further tailor how hashes are calculated, such as the cost or amount of processor time to allocate to the hashing (more time means more security but a slower server). In PHP 8.3 and older, the cost has a default value of 10, which is the minimum you should use with BCrypt. The default cost value has been increased to 12 in PHP 8.4.

However, I don't want to confuse you with more information than you need to be able to store password hashes securely with minimal fuss, so please refer to the [documentation](#) if you'd like more details on the available options.

Using password_verify

To verify that a password matches a hash, use the `password_verify` function, passing it the password string a user has just entered and the stored hash value for that user's password (generally retrieved from your database).

So, assuming your user had previously entered the (very insecure) password of *mypassword*, and you now have their password's hash string (from when the user created their password) stored in the variable `$hash`, you could verify that they match, like this:

```
if (password_verify("mypassword", $hash))  
    echo "Valid";
```

If the correct password for the hash has been supplied, `password_verify` returns the value `TRUE`, so this `if` statement will display the word “Valid.” If it doesn’t match, then `FALSE` is returned, and you can ask the user to try again.

An Example Program

Let’s see how these functions work together when combined with MySQL. First you need to create a new table to store password hashes, so type the program in [Example 12-3](#) and save it as `setupusers.php` (or download it from [GitHub](#)), and then open it in your browser.

Example 12-3. Creating a users table and adding two accounts

```
<?php //setupusers.php  
require_once 'login.php';  
  
try  
{  
    $pdo = new PDO($attr, $user, $pass, $opts);  
}  
catch (\PDOException $e)  
{  
    throw new \PDOException($e->getMessage(), (int)$e->getCode());  
}  
  
$query = "CREATE TABLE users (  
    forename VARCHAR(32) NOT NULL,  
    surname  VARCHAR(32) NOT NULL,  
    username VARCHAR(32) NOT NULL UNIQUE,  
    password VARCHAR(255) NOT NULL  
)";  
  
$result = $pdo->query($query);  
  
$forename = 'Bill';  
$surname  = 'Smith';  
$username = 'bsmith';  
$password = 'mysecret';  
$hash     = password_hash($password, PASSWORD_DEFAULT);  
  
add_user($pdo, $forename, $surname, $username, $hash);  
  
$forename = 'Pauline';  
$surname  = 'Jones';  
$username = 'pjones';  
$password = 'acrobat';  
$hash     = password_hash($password, PASSWORD_DEFAULT);
```

```

add_user($pdo, $forename, $surname, $username, $hash);

function add_user($pdo, $fn, $sn, $un, $pw)
{
    $stmt = $pdo->prepare('INSERT INTO users VALUES(:fn,:sn,:un,:pw)');
    $stmt->execute([
        ':fn' => $fn,
        ':sn' => $sn,
        ':un' => $un,
        ':pw' => $pw
    ]);
}
?>

```

This program will create the table *users* within your *publications* database (or whichever database you set up for the *login.php* file in [Chapter 10](#)). In this table, it will create two users: Bill Smith and Pauline Jones. They have the usernames and passwords of *bsmith/mysecret* and *pjones/acrobat*, respectively.

Using the data in this table, we can now modify [Example 12-2](#) to properly authenticate users, and [Example 12-4](#) shows the code needed to do this. Type it in or download it from the [companion website](#), make sure it is saved as *authenticate.php*, and then call it up in your browser.

Example 12-4. PHP authentication using MySQL

```

<?php // authenticate.php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (\PDOException $e)
{
    throw new \PDOException($e->getMessage(), (int)$e->getCode());
}

if (isset($_SERVER['PHP_AUTH_USER']) &&
    isset($_SERVER['PHP_AUTH_PW']))
{
    $stmt = $pdo->prepare('SELECT * FROM users WHERE username=:un');
    $stmt->execute([':un' => $_SERVER['PHP_AUTH_USER']]);

    if (!$stmt->rowCount()) die("User not found");

    $row = $stmt->fetch();
    $fn = $row['forename'];
    $sn = $row['surname'];
}

```

```

$un = $row['username'];
$pw = $row['password'];

if (password_verify($_SERVER['PHP_AUTH_PW'], $pw))
    echo htmlspecialchars("$fn $sn : Hi $fn,
        you are now logged in as '$un'");
else die("Invalid username/password combination");
}
else
{
    header('WWW-Authenticate: Basic realm="Restricted Area"');
    header('HTTP/1.1 401 Unauthorized');
    die ("Please enter your username and password");
}
?>

```



Depending on hardware, using HTTP authentication will impose approximately an 80 ms penalty on every request when using `password_verify` with passwords hashed with BCrypt, with the default cost of 10. This slowdown serves as a barrier for attackers, preventing them from trying to crack the passwords at maximum speed. Therefore, HTTP authentication is not a good solution on very busy sites, where you will likely prefer to use sessions (see “Using Sessions” on page 301).

As you might expect at this point in the book, some of these examples are starting to get quite a bit longer. But don’t be put off. The only lines to really concern yourself with at this point are the ones highlighted in bold. They start by issuing a query (using placeholders and prepared statements) to MySQL to look up the user and, if a result is returned, to assign the first row to `$row`. Because usernames are unique, there will be only one row.

Now all that’s necessary is to check the hash value stored in the database, which is in `$row['password']` and is the previous hash value calculated with `password_hash` when the user created their password.

If the hash and the password just supplied by the user verify, `password_verify` will return TRUE and a friendly welcome string will be output, calling the user by their first name (see [Figure 12-4](#)). Otherwise, an error message is displayed.

You can try this by calling up the program in your browser and entering a username of `bsmith` and password of `mysecret` (or `pjones` and `acrobat`), the values that were saved in the database by [Example 12-3](#).

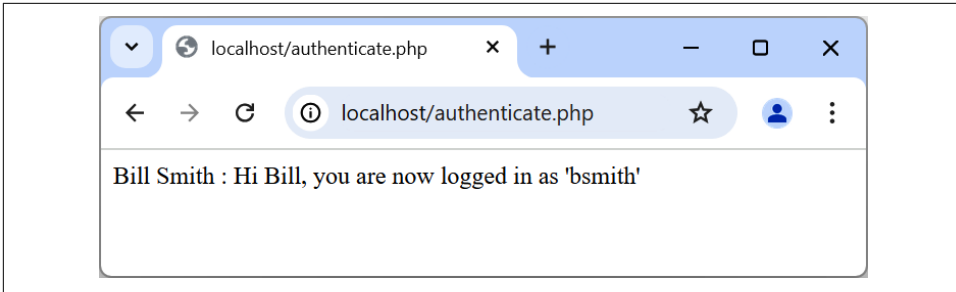


Figure 12-4. Bill Smith has now been authenticated



By replacing dangerous HTML characters with harmless HTML entities in your data (by calling `htmlspecialchars`) when sent to the browser you will block any malicious HTML or JavaScript, and by using placeholders and prepared statements when querying MySQL you will block SQL injection attacks.

Using Sessions

Because your program can't tell what variables were set in other programs—or even what values the same program set the previous time it ran—you'll sometimes want to track what your users are doing from one web page to another. You can do this by setting hidden fields in a form, as seen in [Chapter 10](#), and checking the values of the fields after the form is submitted, but PHP provides a much more powerful, more secure, and simpler solution in the form of *sessions*. These are groups of variables that are stored on the server but relate only to the current user. To ensure that the right variables are applied to the right users, PHP saves a cookie in the users' web browsers to uniquely identify them.



Google is now phasing out third-party cookies in its browser with a project called Privacy Sandbox. No doubt other browsers will follow suit, particularly Opera and Microsoft Edge, which both rely on the open source Google Chromium codebase. Google is starting to lump users into groups of 1,000 or so who have similar browser usage and product interests, so that nobody can be uniquely identified or traced. In Google's own words, "The Privacy Sandbox for the Web will phase out third-party cookies and limit covert tracking. By creating new web standards it will provide publishers with safer alternatives to existing technology, so they can continue building digital businesses while your data stays private."

This cookie has meaning only to the web server and cannot be used to ascertain any information about a user; the cookie contains just a random, arbitrary ID. You

might ask about those users who turned off cookies. Well, today, anyone with cookies disabled should not expect to have the best browsing experience, and if you find them disabled you should probably inform such a user that they require cookies enabled if they wish to fully benefit from your site, rather than trying to find ways around the use of cookies, which could create security issues.

Starting a Session

Starting a session requires calling the PHP function `session_start` before any HTML has been output, similarly to how cookies are sent during header exchanges. Then, to begin saving session variables, you just assign them as part of the `$_SESSION` array, like this:

```
$_SESSION['variable'] = $value;
```

They can then be read back just as easily in later program runs, like this:

```
$variable = $_SESSION['variable'];
```

Now assume that you have an application that always needs access to the first name and last name of each user, as stored in the table *users*, which you created a little earlier. Let's further modify *authenticate.php* from [Example 12-4](#) to set up a session once a user has been authenticated.

[Example 12-5](#) shows the changes needed. The only difference is the content of the `if (password_verify...` section, which we now start by opening a session and saving these variables into it. Type this program (or modify [Example 12-4](#)) and save it as *authenticate2.php*. But don't run it in your browser yet, as you will also need to create a second program in a moment.

Example 12-5. Setting a session after successful authentication

```
<?php // authenticate2.php
require_once 'login.php';

try
{
    $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (\PDOException $e)
{
    throw new \PDOException($e->getMessage(), (int)$e->getCode());
}

if (isset($_SERVER['PHP_AUTH_USER']) &&
    isset($_SERVER['PHP_AUTH_PW']))
{

    $stmt = $pdo->prepare('SELECT * FROM users WHERE username=:un');
```

```

$stmt->execute([':un' => $_SERVER['PHP_AUTH_USER']]);

if (!$stmt->rowCount()) die("User not found");

$row = $stmt->fetch();
$fn = $row['forename'];
$sn = $row['surname'];
$un = $row['username'];
$pw = $row['password'];

if (password_verify($_SERVER['PHP_AUTH_PW'], $pw))
{
    session_start();

    $_SESSION['forename'] = $fn;
    $_SESSION['surname'] = $sn;

    echo htmlspecialchars("$fn $sn : Hi $fn,
        you are now logged in as '$un'");
    die("<p><a href='continue.php'>Click here to continue</a></p>");
}
else die("Invalid username/password combination");
}
else
{
    header('WWW-Authenticate: Basic realm="Restricted Area"');
    header('HTTP/1.0 401 Unauthorized');
    die("Please enter your username and password");
}
?>

```

One other addition to the program is the “Click here to continue” link with a destination URL of *continue.php*. This will be used to illustrate how the session will transfer to another program or PHP web page. So, create *continue.php* by typing the program in **Example 12-6** and saving it.

Example 12-6. Retrieving session variables

```

<?php // continue.php
session_start();

if (isset($_SESSION['forename']))
{
    $forename = htmlspecialchars($_SESSION['forename']);
    $surname = htmlspecialchars($_SESSION['surname']);

    echo "Welcome back $forename.<br>
        Your full name is $forename $surname.<br>";
}
else echo "Please <a href='authenticate2.php'>click here</a> to log in.";
?>

```

Now you are ready to call up *authenticate2.php* into your browser. Enter a username of bsmith and password of mysecret (or pjones and acrobat) when prompted and click the link to load *continue.php*. When your browser calls it up, the result should be something like [Figure 12-5](#).

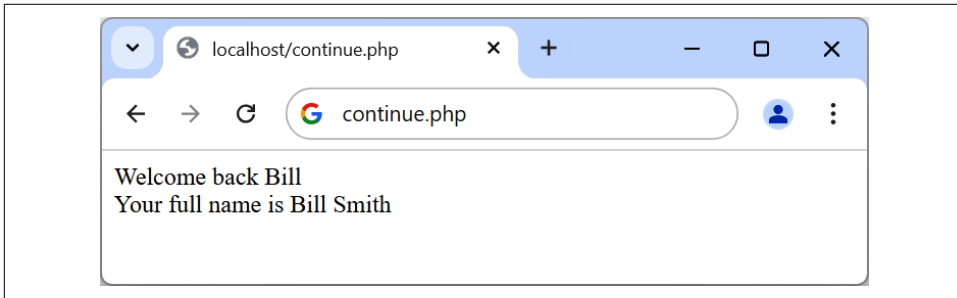


Figure 12-5. Maintaining user data with sessions

Sessions neatly confine to a single program the extensive code required to authenticate and log in a user. Once a user has been authenticated, and you have created a session, your program code becomes very simple indeed. You need only call up `session_start` and look in `$_SESSION` for any variables you need to access.

In [Example 12-6](#), a quick test of whether `$_SESSION['forename']` has a value is enough to let you know that the current user is authenticated, because session variables are stored on the server (unlike cookies, which are stored in the web browser) and can therefore be trusted.

If `$_SESSION['forename']` has not been assigned a value, no session is active, so the last line of code in [Example 12-6](#) directs users to the login page at *authenticate2.php*.

Ending a Session

When the time comes to end a session, usually when a user requests to log out from your site, you can use the `session_destroy` function, as in [Example 12-7](#). This example provides a useful function for totally destroying a session, logging a user out, and unsetting all session variables. The `session_get_cookie_params` function returns the current session cookie information, and we'll use it to get the path used to set the cookie.

Example 12-7. A handy function to destroy a session and its data

```
<?php
function destroy_session_and_data()
{
    $_SESSION = array();
    $params = session_get_cookie_params();
```



```

        setcookie(session_name(), '', time() - 2592000, $params['path']);
        session_destroy();
    }
?>

```

To see this in action, you could modify *continue.php* as in [Example 12-8](#).

Example 12-8. Retrieving session variables and then destroying the session

```

<?php
    session_start();

    if (isset($_SESSION['forename']))
    {
        $forename = $_SESSION['forename'];
        $surname  = $_SESSION['surname'];

        destroy_session_and_data();

        echo htmlspecialchars("Welcome back $forename");
        echo "<br>";
        echo htmlspecialchars("Your full name is $forename $surname.");
    }
    else echo "Please <a href='authenticate.php'>click here</a> to log in.";

    function destroy_session_and_data()
    {
        $_SESSION = array();
        $params = session_get_cookie_params();
        setcookie(session_name(), '', time() - 2592000, $params['path']);
        session_destroy();
    }
?>

```

The first time you navigate from *authenticate2.php* to *continue.php*, it will display all the session variables. But, because of the call to `destroy_session_and_data`, if you then click your browser's Reload button, the session will have been destroyed and you'll be prompted to return to the login page.

Setting a Timeout

At other times you might wish to close a user's session yourself, such as when the user has forgotten or neglected to log out, and you want the program to do so for them for their own security. You do this by setting the timeout after which a logout will automatically occur if there has been no activity. It may also be a good idea to warn the user through a message or a dialog that their session will end soon and

allow them to choose to continue the session, but that goes beyond the scope of the following example.

To set a timeout, use the `ini_set` function to set the timeout to exactly one day (the letters `gc` standing for garbage collection):

```
ini_set('session.gc_maxlifetime', 60 * 60 * 24);
```

If you wish to know the current timeout period, you can display it using:

```
echo ini_get('session.gc_maxlifetime');
```

Session Security

Although I mentioned that once you had authenticated a user and set up a session you could safely assume that the session variables were trustworthy, this isn't exactly the case. The reason is that it's possible to use *packet sniffing* (sampling of data being transferred across an internet connection) to discover session IDs passing across a network.

The only truly secure way of preventing these from being discovered is to implement *Transport Layer Security* (TLS, the more secure successor to the *Secure Sockets Layer*, or SSL) and run HTTPS instead of HTTP web pages. That's beyond the scope of this book, although you can refer to the [Apache documentation](#) for details on setting up a secure web server.

Preventing session hijacking

You can further authenticate users by storing their IP addresses along with their other details by adding a line such as the following when you store their sessions:

```
$_SESSION['ip'] = $_SERVER['REMOTE_ADDR'];
```

Then, as an extra check, whenever any page loads and a session is available, perform the following check. It calls the function `different_user` if the stored IP address doesn't match the current one:

```
if ($_SESSION['ip'] !== $_SERVER['REMOTE_ADDR']) different_user();
```

The code you place in your `different_user` function is up to you. I recommend that you either delete the current session and ask the user to log in again due to a technical error or, if you have their email address, email them a link to confirm their identity, which will enable them to retain all the data in the session.

Of course, you need to be aware that users on the same proxy server, or sharing the same IP address on a home or business network, will have the same IP address. And on the other hand, many networks will change the assigned IP address at random and mobile devices can use multiple IP addresses in a given browsing session, so this approach could eventually result in usability problems. Storing IP addresses

can also present a privacy challenge because some countries view them as personal information.

You can also store a copy of the browser *user-agent string* (a string that developers put in their browsers to identify them by type and version), which might also distinguish users due to the wide variety of browser types, versions, and computer platforms in use (although this is not a perfect solution, and the string will change if the browser auto-updates). Use the following to store the user agent:

```
$_SESSION['ua'] = $_SERVER['HTTP_USER_AGENT'];
```

And use this to compare the current user-agent string with the saved one:

```
if ($_SESSION['ua'] !== $_SERVER['HTTP_USER_AGENT']) different_user();
```

Or, better still, combine the two checks like this and save the combination as a hash hexadecimal string:

```
$hash_algo = 'ripemd128'; // See https://www.php.net/function.hash-algos
$_SESSION['check'] = hash($hash_algo, $_SERVER['REMOTE_ADDR'] .
    $_SERVER['HTTP_USER_AGENT']);
```

And use the following code, which uses the `hash_equals` function to safely compare two hashes, the current and stored strings:

```
$hash_algo = 'ripemd128'; // See https://www.php.net/function.hash-algos
$check = hash($hash_algo, $_SERVER['REMOTE_ADDR'] . $_SERVER['HTTP_USER_AGENT']);
if (!hash_equals($_SESSION['check'], $check)) different_user();
```

Forcing cookie-only sessions

You should require your users to enable cookies for your website. It solves a lot of security problems, at least partially, like the session fixation attack mentioned in “[Preventing session fixation](#)” on page 308. There are two configuration options, `session.use_cookies` and `session.use_only_cookies`, both enabled by the default PHP configuration, but it’s recommended to set those explicitly using the `ini_set` function, because the PHP could be configured differently on your server:

```
ini_set('session.use_cookies', 1);
ini_set('session.use_only_cookies', 1);
```

With the former one (`session.use_cookies`) enabled, PHP will use cookies, *if available*, to store the session ID and otherwise will store it in the URL. With the latter one (`session.use_only_cookies`) enabled, PHP will use *only* cookies, never the URL, to store the session ID.

If you use this security measure, I also recommend that you inform your users that your site requires cookies (but only if the user has cookies disabled, and especially if the user is in a part of the world that requires cookie notifications), so they know what’s wrong if they don’t get the results they want.

Preventing session fixation

Session fixation happens when a malicious third party obtains a valid session ID (which could be server-generated) and makes the user authenticate themselves with that session ID, instead of authenticating with their own. It can happen when an attacker takes advantage of the ability to pass a session ID in the GET part of a URL, like this:

```
http://yourserver.com/authenticate.php?PHPSESSID=123456789
```

In this example, the made-up session ID of 123456789 is being passed to the server. This case can be prevented by using *only* cookies to store session IDs, as already explained, and by using a strict session ID mode.

The following code will enable the strict mode and, once enabled, PHP will reject uninitialized, completely made-up session IDs (like in the previous example), even if passed in a cookie:

```
ini_set('session.use_strict_mode', 1);
```

Using the strict session ID mode is recommended; however, the attacker can still visit the application, let it generate a session ID, which is not made-up anymore, and then try to *fixate* that ID with, for example, an XSS attack even when cookies are used to store the ID. But with the strict mode, they at least have to do more work for their attack to succeed.

To prevent the session fixation attack, change the session ID using `session_regenerate_id` as soon as the user authentication status changes, for example after verifying the password during login. This function keeps all current session variable values but replaces the session ID with a new one that an attacker cannot know.

Example 12-9 shows how to add `session_regenerate_id` to the relevant part of the *authenticate2.php* file created in **Example 12-5**.

Example 12-9. Session regeneration

```
if (password_verify($_SERVER['PHP_AUTH_PW'], $pw))
{
    session_start();
    session_regenerate_id();

    $_SESSION['forename'] = $fn;
    $_SESSION['surname'] = $sn;

    echo htmlspecialchars("$fn $sn : Hi $fn,
        you are now logged in as '$sn'");
    die("<p><a href='continue.php'>Click here to continue</a></p>");
}
```

This way, an attacker can come back to your site using any of the session IDs that they generated, but none of them will call up another user's session, as they will all have been replaced with regenerated IDs, which you can verify by watching the cookie value change in developer tools after you load the script in your browser.

Using a shared server

On a server shared with other accounts, you do not want to save all your session data into the same directory as theirs. Instead, you should choose a directory that only your account has access to (and that is not web-visible) to store your sessions, by placing an `ini_set` call near the start of your program, like this:

```
ini_set('session.save_path', '/home/user/myaccount/sessions');
```

The configuration option will keep this new value only during the program's execution, and the original configuration will be restored at the program's ending.

This *sessions* folder can fill up quickly; you might want to periodically clear out older sessions according to how busy your server gets. The more it's used, the less time you will want to keep a session stored.



Remember that your websites can and will be subject to hacking attempts. There are automated bots running riot around the internet, trying to find sites vulnerable to exploits. So, whatever you do, whenever you are handling data that is not 100% generated within your own program, you should always treat it with the utmost caution.

You should now have a very good grasp of both PHP and MySQL, which you can confirm by answering the following questions. Then **Chapter 13** introduces the third major technology covered by this book: JavaScript.

Questions

1. Why must a cookie be transferred at the start of a program?
2. Which PHP function stores a cookie in a web browser?
3. How can you destroy a cookie?
4. Where are the username and password stored in a PHP program when you are using HTTP authentication?
5. Why is the `password_hash` function a powerful security measure?
6. What is meant by *salting* a string?

7. What is a PHP session?
8. How do you initiate a PHP session?
9. What is session hijacking?
10. What is session fixation?

See “[Chapter 12 Answers](#)” on page 576 in the [Appendix](#) for the answers to these questions.

Exploring JavaScript

JavaScript brings dynamic functionality to your websites. Every time you see something pop up when you mouse over an item in the browser, or see new text, colors, or images appear on the page in front of your eyes, or grab an object on the page and drag it to a new location—these are done through JavaScript (or CSS). JavaScript offers effects that are not otherwise possible, because it runs inside the browser and has direct access to all the elements in a web document.

JavaScript first appeared in the Netscape Navigator browser in 1995, coinciding with the addition of support for Java technology in the browser. Because of the initial incorrect impression that JavaScript was a spin-off of Java, there has been some long-term confusion over their relationship. However, the naming was just a marketing ploy to help the new scripting language benefit from the popularity of the Java programming language.

JavaScript gained new power when the HTML elements of the web page got a more formal, structured definition in what is called the *Document Object Model* (DOM). The DOM makes it relatively easy to add a new paragraph or focus on a piece of text and change it.

Because both JavaScript and PHP support much of the structured programming syntax used by the C programming language, they look very similar to each other. They are both fairly high-level languages, too. Also, they are weakly typed, so it's easy to change a variable to a new type just by using it in a new context.

Now that you have learned PHP, you should find JavaScript even easier. And you'll be glad you did, because it's at the heart of the asynchronous communication technology that provides the fluid web frontends that savvy web users expect these days.

Outputting the Results

When teaching programming, it's necessary to have a quick and easy way to display the results of expressions. In PHP (for example) there are the `echo` and `print` statements, which simply send text to the browser or the terminal if the script is executed from the command line, so that's easy. In JavaScript, though, there are the following alternatives.

Using `console.log`

The `console.log` function will output the result of any value or expression passed to it in the console of the current browser. This is a special mode with a frame or window separate from the browser window, and in which errors and other messages can be made to display. You can find the console in the browser developer tools, so you may want to open it when trying out the following examples as the console will be used to display the output.

Using `alert`

The `alert` function displays values or expressions passed to it in a pop-up window, which requires you to click a button to close. Clearly this can become quite irritating, and it has the downside of displaying only the current message—previous ones are erased.

Writing into Elements

It is possible to write directly into the text of an HTML element, which is a fairly elegant solution (and the best one for production websites)—except that for this book every example would require such an element to be created, and some lines of code to access it. This gets in the way of teaching the core of an example and would make the code overly cumbersome and confusing.

Using `document.write`

The `document.write` function writes a value or expression at the current browser location and at first glance seems the perfect choice for quickly displaying results. It keeps all the examples short and sweet by placing the output right there in the browser next to the web content and code.

You may, however, have heard that some developers regard this function as unsafe, because when you call it after a web page is fully loaded, it will overwrite the current document.

I never use `document.write` in production code (except in the very rarest circumstances where it is necessary). Instead, I almost always use the preceding option of writing directly into a specially prepared element, per the more complex examples in [Chapter 17](#) onward (which access the `innerHTML` property of elements for program output).

JavaScript and HTML Text

Executing JavaScript code requires a JavaScript *engine*. All modern web browsers have a JavaScript engine, allowing the browser itself to execute JavaScript. Node.js, which we will explore later, is a JavaScript engine that doesn't require a browser and is suitable for desktop or server-side use.

To add a JavaScript code to your web page, you place it between opening `<script>` and closing `</script>` HTML tags. Note that a typical “Hello World” document using JavaScript might look like [Example 13-1](#).

Example 13-1. “Hello World” displayed using JavaScript

```
<html>
  <head><title>Hello World</title></head>
  <body>
    <script>
      console.log("Hello World")
    </script>
    <noscript>
      Your browser doesn't support or has disabled JavaScript
    </noscript>
  </body>
</html>
```

Within the `<script>` tags is a single line of JavaScript code that uses its equivalent of the PHP `echo` or `print` commands, `console.log`. As you'd expect and as already explained, it simply outputs the supplied string to the browser console, where it is displayed.

You also may have noticed that, unlike with PHP, there is no trailing semicolon (;). This is because a newline often, but not always, serves the same purpose as a semicolon in JavaScript. However, if you wish to have more than one statement on a single line, you do need to place a semicolon after each command except the last one. Of course, if you wish, you can add a semicolon to the end of every statement, and your JavaScript will work fine. My personal preference is to leave out the semicolon because it's often superfluous. At the end of the day, though, the choice may come down to the team you work on. So, if in doubt, just add the semicolons.

The other thing to note in this example is the `<noscript>` and `</noscript>` pair of tags. These are used when you wish to offer alternative HTML to users whose browsers do not support or have disabled JavaScript. Using these tags is up to you, as they are not required, but you ought to use them at least to tell users that JavaScript is required, because in complex apps, providing static HTML alternatives to the operations you provide using JavaScript may be difficult. However, the remaining examples in this book will omit `<noscript>` tags, because we're focusing on what you can do *with* JavaScript, not what you can do *without* it.

When [Example 13-1](#) is loaded, a web browser with JavaScript enabled will output the following (as shown in [Figure 13-1](#)):

Hello World

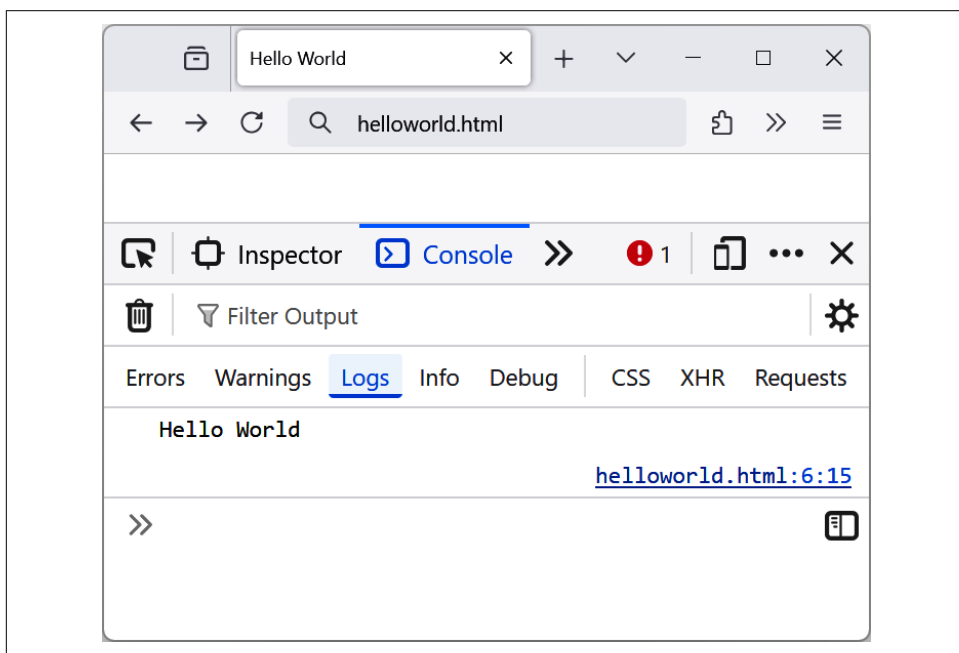


Figure 13-1. JavaScript, enabled and working

A browser with JavaScript disabled will display the following message (as shown in [Figure 13-2](#)):

Your browser doesn't support or has disabled JavaScript

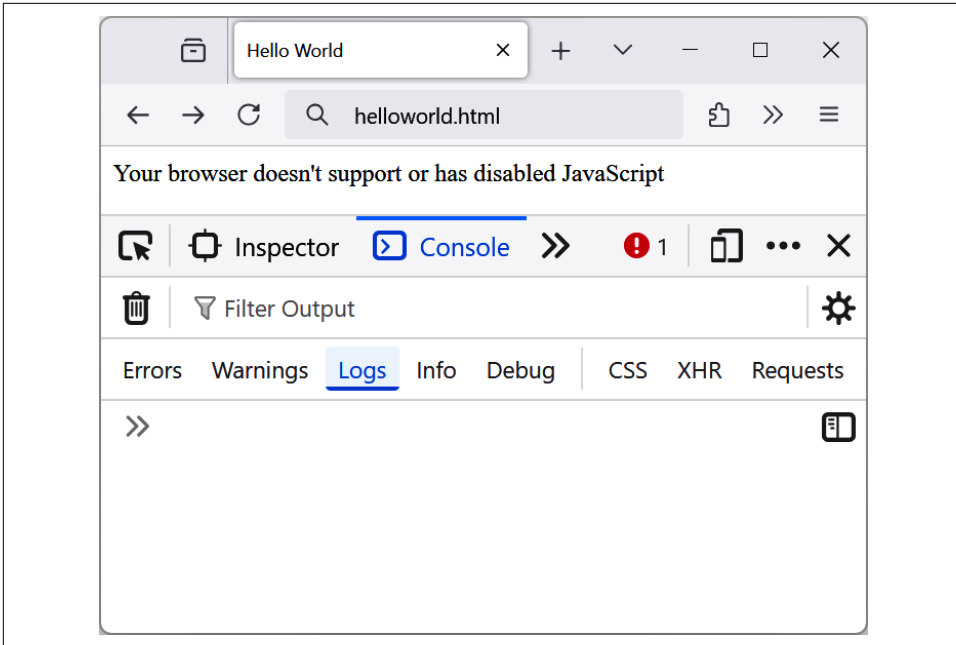


Figure 13-2. JavaScript, disabled

Using Scripts Within a Document Head

In addition to placing a script within the body of a document, you can put it in the `<head>` section, which is the ideal place if you wish to execute a script when a page loads.

A generally accepted best practice is to place framework files (like jQuery or React or the like) in the `<head>` element but actual functionality at the bottom of the page just before the closing `</body>` tag to wait for the entire page to load and the Document Object Model (we'll discuss it a bit later) to be available.

Including JavaScript Files

In addition to writing JavaScript code directly in HTML documents, you can include files of JavaScript code either from your website or from anywhere on the internet. The syntax for this is:

```
<script src="script.js"></script>
```

Or, to pull in a file from the internet, use:

```
<script src="http://someserver.com/script.js"></script>
```

As for the script files themselves, they must *not* include any `<script>` or `</script>` tags; putting them in the JavaScript files will cause an error.

Including script files is the preferred way to use third-party JavaScript files on your website.

Debugging JavaScript Errors

When you're learning JavaScript, it's important to be able to track typing or other coding errors. Unlike PHP, which displays error messages in the browser, JavaScript displays the errors in the browser console in the developer tools. You can open the console by pressing F12 and selecting the Console tab, or by pressing Ctrl-Shift-J on a PC or Cmd-Shift-J on a Mac.



Developer Tools in Safari

To view the JavaScript console in Safari, you first need to enable the Develop menu by selecting “Show features for developers” in Safari → Preferences → Advanced. Then press Cmd-Opt-C, or select the Show JavaScript Console item in the Develop menu in Safari’s menu bar.

Please refer to the browser developers’ documentation on their websites for full details on using them.

Using Comments

Because of their shared inheritance from the C programming language, PHP and JavaScript have many similarities, one of which is commenting. First, there’s the single-line comment, like this:

```
// This is a comment
```

This style uses a pair of forward slash characters (`//`) to inform JavaScript that everything that follows on the current line is to be ignored. You also have multiline comments, like this:

```
/* This is a section  
   of multiline comments  
   that will not be  
   interpreted */
```

You start a multiline comment with the sequence `/*` and end it with `*/`. Just remember that you cannot nest multiline comments, so make sure that you don’t comment out large sections of code that already contain multiline comments.

A common variant of a multiline comment that is used to document functions and other code is called JSDoc:

```
/**
 * Show a page notification.
 * @param {string} message
 */
```

Semicolons

Unlike PHP, JavaScript generally does not require semicolons if you have only one statement on a line. Therefore, the following is valid:

```
x += 10
```

However, when you wish to place more than one statement on a line, you must separate them with semicolons, like this:

```
x += 10; y -= 5; z = 0
```

You can leave the final semicolon off, because the newline terminates the final statement.



There are exceptions to the semicolon rule. If you write JavaScript bookmarklets or end a statement with a variable or function reference, *and* the first character of the line below is a left parenthesis or bracket, you *must* remember to append a semicolon or the JavaScript will fail. When in doubt, use a semicolon.

Variables

No particular character identifies a variable in JavaScript like the dollar sign does in PHP. Instead, variables use these rules:

- A variable may include only the letters a-z, A-Z, 0-9, the \$ symbol, and the underscore (_).
- No other characters, such as spaces or punctuation, are allowed in a variable name.
- The first character of a variable name can be only a-z, A-Z, \$, or _ (no numbers).
- Names are case-sensitive. Count, count, and COUNT are all different variables.
- There is no set limit on variable name lengths.

And yes, you're right: a \$ is in that list of allowed characters. It *is* allowed by JavaScript and *may* be the first character of a variable or function name. This lets

you port a lot of PHP code more quickly to JavaScript. That said, I don't recommend keeping the `$` character because it is frequently employed by jQuery as an alias.

String Variables

JavaScript string variables should be enclosed in either single or double quotation marks, like this:

```
greeting = "Hello there"  
warning  = 'Be careful'
```

You may include a single quote within a double-quoted string or a double quote within a single-quoted string. But you must escape a quote of the same type by using the backslash character, like this:

```
greeting = "\"Hello there\" is a greeting"  
warning  = '\Be careful\' is a warning'
```

To use or copy a string variable, you can assign it to another one, like this:

```
newstring = oldstring
```

or you can use it in a function, like this:

```
status = "All systems are working"  
console.log(status)
```

Numeric Variables

Creating a numeric variable is as simple as assigning a value, as in these examples:

```
count      = 42  
temperature = 98.4
```

Like strings, numeric variables can be read from and used in expressions and functions.

Arrays

JavaScript arrays are also very similar to those in PHP, in that an array can contain string or numeric data, as well as other arrays. To assign values to an array, use the following syntax (which in this case creates an array of strings):

```
toys = ['bat', 'ball', 'whistle', 'puzzle', 'doll']
```

To create a multidimensional array (or, more accurately, an array of arrays), nest smaller arrays within a larger one. So, to create a two-dimensional array containing the colors of a single face of a scrambled Rubik's Cube (where the colors red, green, orange, yellow, blue, and white are represented by their capitalized initial letters), you could use this code:

```
face =
[
  ['R', 'G', 'Y'],
  ['W', 'R', 'O'],
  ['Y', 'W', 'G']
]
```

The preceding example has been formatted to make it obvious what is going on, but it could also be written like this:

```
face = [['R', 'G', 'Y'], ['W', 'R', 'O'], ['Y', 'W', 'G']]
```

or even like this:

```
top = ['R', 'G', 'Y']
mid = ['W', 'R', 'O']
bot = ['Y', 'W', 'G']

face = [top, mid, bot]
```

To access the element two down and three along in this matrix, you would use the following (because array elements start at position 0):

```
console.log(face[1][2])
```

This statement will output the letter O for *orange*.



JavaScript arrays are powerful storage structures, and [Chapter 15](#) discusses them in much greater depth.

Operators

Operators in JavaScript, as in PHP, can involve mathematics, changes to strings, and comparison and logical operations (and, or, etc.). JavaScript mathematical operators look a lot like plain arithmetic—for instance, the following statement outputs 15:

```
console.log(13 + 2)
```

The following sections teach you about the various operators.

Arithmetic Operators

Arithmetic operators are used to perform mathematics. You can use them for the main four operations (addition, subtraction, multiplication, and division) as well as to find the modulus (more precisely, the remainder after a division) and to increment or decrement a value (see [Table 13-1](#)).

Table 13-1. Arithmetic operators

Operator	Description	Example
+	Addition	j + 12
-	Subtraction	j - 22
*	Multiplication	j * 7
/	Division	j / 3.13
%	Modulus (division remainder)	j % 6
++	Increment	++j
--	Decrement	--j

Assignment Operators

The *assignment operators* are used to assign values to variables. They start with the very simple = and move on to +=, -=, and so on. The operator += adds the value on the right side to the variable on the left, instead of totally replacing the value on the left. Thus, if count starts with the value 6, the statement:

```
count += 1
```

sets count to 7, just like the more familiar assignment statement:

```
count = count + 1
```

Table 13-2 lists the assignment operators available.

Table 13-2. Assignment operators

Operator	Example	Equivalent to
=	j = 99	j = 99
+=	j += 2	j = j + 2
+=	j += 'string'	j = j + 'string'
-=	j -= 12	j = j - 12
*=	j *= 2	j = j * 2
/=	j /= 6	j = j / 6
%=	j %= 7	j = j % 7

Comparison Operators

Comparison operators are used inside a construct such as an if statement, where you need to compare two items. For example, you may wish to know whether a variable you have been incrementing has reached a specific value, or whether another variable is less than a set value, and so on (see Table 13-3).

Table 13-3. Comparison operators

Operator	Description	Example
==	Is <i>equal</i> to	j == 42
!=	Is <i>not equal</i> to	j != 17
>	Is <i>greater than</i>	j > 0
<	Is <i>less than</i>	j < 100
>=	Is <i>greater than or equal</i> to	j >= 23
<=	Is <i>less than or equal</i> to	j <= 13
===	Is <i>equal</i> to (and of the same type)	j === 56
!==	Is <i>not equal</i> to (and of the same type)	j !== '1'

Logical Operators

Unlike PHP, JavaScript's *logical operators* do not include and and or equivalents to && and ||, and there is no xor operator (see [Table 13-4](#)).

Table 13-4. Logical operators

Operator	Description	Example
&&	And	j == 1 && k == 2
	Or	j < 100 j > 0
!	Not	! (j == k)

Incrementing, Decrementing, and Shorthand Assignment

The following forms of post- and pre-incrementing and decrementing that you learned to use in PHP are also supported by JavaScript, as are shorthand assignment operators:

```
++x
--y
x += 22
y -= 3
```

String Concatenation

JavaScript handles string concatenation slightly differently from PHP. Instead of the . (period) operator, it uses the plus sign (+), like this:

```
console.log("You have " + messages + " messages.")
```

Assuming that the variable `messages` is set to the value 3, the output from this line of code will be:

```
You have 3 messages.
```

Just as you can add a value to a numeric variable with the += operator, you can also append one string to another the same way:

```
name = "James"
name += " Dean"
```

Escape Characters

Escape characters, which you’ve seen used to insert quotation marks in strings, can also be used to insert various special characters such as tabs, newlines, and carriage returns. Here is an example using tabs to lay out a heading—it is included here merely to illustrate escapes, because in web pages, there are better ways to do layout:

```
heading = "Name\tAge\tLocation"
```

Table 13-5 details the escape characters available.

Table 13-5. JavaScript’s escape characters

Character	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\'	Single quote (or apostrophe)
\"	Double quote
\\	Backslash
\XXX	An octal number between 000 and 377 that represents the Latin-1 character equivalent (such as \251 for the © symbol)
\xXX	A hexadecimal number between 00 and FF that represents the Latin-1 character equivalent (such as \xA9 for the © symbol)
\uXXXX	A hexadecimal number between 0000 and FFFF that represents the Unicode character equivalent (such as \u00A9 for the © symbol)

Variable Typing

Like PHP, JavaScript is a very loosely typed language; the *type* of a variable is determined only when a value is assigned, and it can change as the variable appears in different contexts. Usually, you don’t have to worry about the type; JavaScript figures out what you want and just does it.



JavaScript but with Types

If you'd like to use JavaScript but also love types, you can use TypeScript, a programming language released in 2012 that adds types to JavaScript.

Take a look at [Example 13-2](#), in which:

1. The variable `n` is assigned the string value `'838102050'`. The next line prints out its value, and the `typeof` operator is used to look up the type.
2. `n` is given the value returned when the numbers 12345 and 67890 are multiplied together. This value is also 838102050, but it is a number, not a string. The type of the variable is then looked up and displayed.
3. Some text is appended to the number `n` and the result is displayed.

Example 13-2. Setting a variable's type by assignment

```
<script>
  n = '838102050'           // Set 'n' to a string
  console.log('n = ' + n + ', and is a ' + typeof n)

  n = 12345 * 67890;         // Set 'n' to a number
  console.log('n = ' + n + ', and is a ' + typeof n)

  n += ' plus some text'    // Change 'n' from a number to a string
  console.log('n = ' + n + ', and is a ' + typeof n)
</script>
```

The output from this script looks like this:

```
n = 838102050, and is a string
n = 838102050, and is a number
n = 838102050 plus some text, and is a string
```

If there is ever any doubt about the type of a variable, or you need to ensure that a variable has a particular type, you can force it to that type by using statements such as the following (which, respectively, turn a string into a number and a number into a string):

```
n = "123"
n *= 1    // Convert 'n' into a number

n = 123
n += ""   // Convert 'n' into a string
```

Or you can use these functions in the same way:

```
n = "123"
n = parseInt(n) // Convert 'n' into an integer number
n = parseFloat(n) // Convert 'n' into a floating point number

n = 123
n = n.toString() // Convert 'n' into a string
```

You can read more about type conversion in JavaScript at javascript.info/type-conversions. And you can always look up a variable's type by using the `typeof` operator.



Using `typeof` on values like `null` or `[]` may give unexpected results as both `typeof null` and `typeof []` return `"object"`.

Functions

As with PHP, JavaScript functions are used to separate out sections of code that perform a particular task. To create a function, declare it as shown in [Example 13-3](#).

Example 13-3. A simple function declaration

```
<script>
  function product(a, b) {
    return a * b
  }
</script>
```

This function takes the two parameters passed, multiplies them together, and returns the product.

Global Variables

Global variables are ones defined outside of any functions (or defined within functions but without the `var` keyword). They can be defined as:

```
a = 123 // Global scope
var b = 456 // Global scope
if (a == 123) var c = 789 // Global scope
```

Regardless of whether you are using the `var` keyword, as long as a variable is defined outside of a function, it is global in scope and every part of a script can have access to it.

Local Variables

Parameters passed to a function automatically have *local* scope, that is, they can be referenced only from within that function. However, there is one exception. Arrays are passed to a function by reference, so if you modify any elements in an array parameter, the elements of the original array will be modified.

To define a local variable that has scope only within the current function, and has not been passed as a parameter, use the `var` keyword. [Example 13-4](#) shows a function that creates one variable with global scope (not recommended to create variables like this) and two with local scope.

Example 13-4. A function creating variables with global and local scope

```
<script>
  function test() {
    a = 123           // Global scope, discouraged
    var b = 456       // Local scope
    if (a == 123) var c = 789 // Local scope
  }
</script>
```

To test whether scope setting has worked in PHP, we can use the `isset` function. But in JavaScript there is no such function, so [Example 13-5](#) uses the `typeof` operator, which returns the string `undefined` when a variable is not defined.

Example 13-5. Checking the scope of the variables defined in the function test

```
<script>
  test()

  if (typeof a !== 'undefined') console.log('a = ' + a + '')
  if (typeof b !== 'undefined') console.log('b = ' + b + '')
  if (typeof c !== 'undefined') console.log('c = ' + c + '')

  function test() {
    a = 123
    var b = 456

    if (a == 123) var c = 789
  }
</script>
```

The output from this script is the following single line:

```
a = "123"
```

This shows that only the variable `a` was given global scope, which is exactly what we would expect, since the variables `b` and `c` were given local scope by being prefaced with the `var` keyword.

If your browser issues a warning about `b` being undefined, the warning is correct but it can be ignored.

Using `let`

JavaScript now offers two new keywords: `let` and `const`, and you should be using them instead of the rather legacy `var`. The `let` keyword is pretty much a swap-in for `var`, but it has the advantage that you cannot redeclare a variable in the same scope once you have done so with `let`, although you can with `var`.

You see, the fact that you could redeclare variables using `var` was leading to obscure bugs, such as:

```
var hello = "Hello there"
var counter = 1

if (counter > 0)
{
    var hello = "How are you?"
}

console.log(hello)
```

Can you see the problem? Because `counter` is greater than 0 (since we initialized it to 1), the string `hello` is redefined as “How are you?” which is then displayed in the console.

If you replace the `var` with `let` (as follows), the second declaration is seemingly ignored as the string “How are you?” is visible only in the `if` block, where it is unused. The original string “Hello there” will be displayed instead:

```
let hello = "Hello there"
let counter = 1

if (counter > 0)
{
    let hello = "How are you?"
}

console.log(hello)
```

The `var` keyword is either globally scoped (if outside of any blocks or functions) or *function* scoped, and variables declared with it are initialized with `undefined`, so they can be referenced before the `var` declaration, but the `let` keyword is either

globally or *block* scoped, and variables are not initialized, meaning variables cannot be referenced before the `let` declaration.

Any variable assigned using `let` has scope either within the entire document if declared outside of any block, or, if declared within a block bounded by `{}` (which includes functions), its scope is limited to that block (and any nested sub-blocks). If you declare a variable within a block but try to access it from outside that block, an error will be returned, as with the following, which will fail at the `console.log` because `hello` will have no value:

```
let counter = 1

if (counter > 0)
{
  let hello = "How are you?"
}

console.log(hello)
```

Although the practice is discouraged, you can use `let` to declare variables of the same name as previously declared ones, as long as it is within a new scope, in which case any previous value assigned to a variable of the same name in the previous scope will become inaccessible to the new scope, because the new variable of the same name is treated as totally different from the previous one. It has scope only within the current block, or any sub-blocks (unless another `let` is used to declare yet another variable of the same name in a sub-block).

It is good practice to avoid the reuse of meaningful variable names, or you risk causing confusion. However, loop or index variables such as `i` (or other short and simple names) can be reused in new scopes without causing confusion.

Using `const`

You can further increase your control over scope by declaring a variable to have a constant value, that is, one that cannot be changed. This is beneficial when you have created a variable that you are treating as a constant but had declared it only using `var` or `let`, because you might have instances in your code where you try to change that value, which would be allowed but would be a bug.

However, if you use the `const` keyword to declare the variable and assign its value, any attempt to change the value later will be disallowed, and your code will halt with an error message in the console similar to:

```
Uncaught TypeError: Assignment to constant variable
```

The following code will cause that error:

```
const hello = "Hello there"
let counter = 1

if (counter > 0)
{
  hello = "How are you?"
}

console.log(hello)
```

Unlike strings, arrays and objects can be modified, for example added to, as you don't modify the variable itself but only change its internals:

```
const array = [1, 2]
array.push(3) // Works, will add 3 to the array
array = [4, 5, 6] // Will throw an error
```

Just like `let`, `const` declarations are also block scoped (within `{}` sections and any sub-blocks), meaning that you can have constant variables of the same name but have different values in different scopes of a piece of code. However, I strongly recommend you try to avoid duplication of names and keep any constant name for one single value throughout each program, using a new constant name wherever you need a new constant.

In summary: `var` has global or function scope, and `let` and `const` have global or block scope. Both `var` and `let` can be declared without being initialized, while `const` must be initialized during declaration. The `var` keyword can be reused to redeclare a `var` variable, but `let` and `const` cannot. Finally, `const` can be neither redeclared nor reassigned.

The Document Object Model

JavaScript's design is very smart. Rather than creating yet another scripting language (which would have still been a pretty good improvement at the time), there was a vision to build it around the already-existing HTML Document Object Model. This breaks down the parts of an HTML document into discrete *objects*, each with its own *properties* and *methods* and each subject to JavaScript's control.

JavaScript separates objects, properties, and methods by using a period (one good reason why `+` is the string concatenation operator in JavaScript, rather than the period). For example, let's consider a business card as an object we'll call `card`. This object contains properties such as a name, address, phone number, and so on. In the syntax of JavaScript, these properties would look like this:


```
card.name  
card.phone  
card.address
```

Its methods are functions that retrieve, change, and otherwise act on the properties. For instance, to invoke a method that displays the properties of the object `card`, you might use syntax such as this:

```
card.display()
```

Have a look at some of the earlier examples in this chapter and notice where the statement `console.log` is used. Now that you understand how JavaScript is based around objects, you will see that `log` is actually a method of the `console` object.

Within JavaScript, there is a hierarchy of parent and child objects, known as the Document Object Model or DOM (see [Figure 13-3](#)).

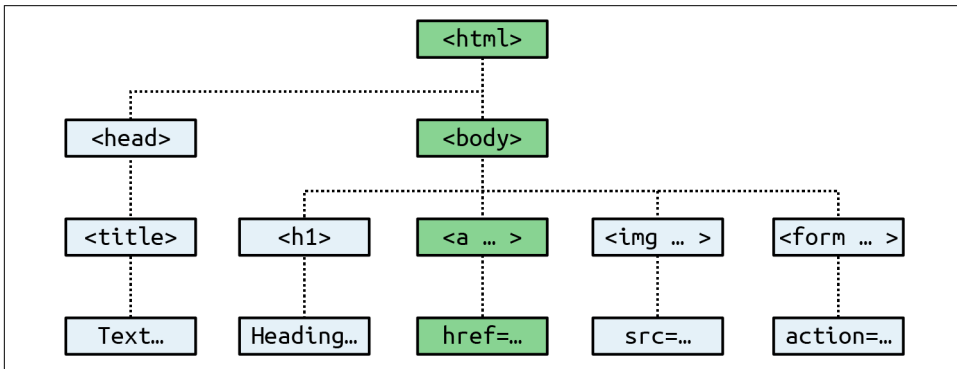


Figure 13-3. Example of DOM object hierarchy

The figure uses HTML tags you are already familiar with to illustrate the parent/child relationship between the various objects in a document. The last row shows object content "Text..." and "Heading...", and object properties `href`, `src`, and `action`. For example, a URL within a link is part of the body of an HTML document. In JavaScript, it is referenced like this:

```
url = document.links.linkname.href
```

Notice how this follows the central column down. The first part, `document`, refers to the `<html>` and `<body>` tags in the figure; `links.linkname` refers to the `<a>` tag, and `href` to the `href` attribute.

Let's turn this into some HTML and a script to read a link's properties. Type [Example 13-6](#) and save it as `linktest.html`; then call it up in your browser.

Example 13-6. Reading a link URL with JavaScript

```
<html>
  <head>
    <title>Link Test</title>
  </head>
  <body>
    <a id="mylink" href="http://mysite.com">Click me</a><br>
    <script>
      url = document.links.mylink.href
      console.log('The URL is ' + url)
    </script>
  </body>
</html>
```

If you wish, just for the purposes of testing this (and other examples), you could also omit everything outside of the `<script>` and `</script>` tags. The output from this example in the browser console is:

```
The URL is http://mysite.com/
```

Notice how the code follows the document tree down from document to links to mylink (the id given to the link) to href (the URL destination value).

A short form that works equally well starts with the value in the id attribute: mylink.href. So, you can replace this:

```
url = document.links.mylink.href
```

with this:

```
url = mylink.href
```

Another Use for the \$ Symbol

As mentioned earlier, the \$ symbol is allowed in JavaScript variable and function names. Because of this, you may sometimes encounter strange-looking code like this:

```
url = $('mylink').href
```

Some enterprising programmers have decided that the getElementById function is so prevalent in JavaScript that they have written a function to replace it called \$, like in jQuery (although jQuery uses the \$ for much more than just getting an element by its ID).

Using the DOM

The links object is actually an array of URLs, so the mylink URL in [Example 13-6](#) can also be safely referred to in all browsers in the following way (because it's the first, and only, link):

```
url = document.links[0].href
```

If you want to know how many links are in an entire document, you can query the `length` property of the `links` object, like this:

```
numlinks = document.links.length
```

You can extract and display all links in a document, like this:

```
for (j=0 ; j < document.links.length ; ++j)
  console.log(document.links[j].href)
```

The `length` of something is a property of every array, and many objects as well. For example, the number of items in your browser's web history can be queried like this:

```
console.log(history.length)
```

To stop websites from snooping on your browsing history, the `history` object stores only the number of sites in the array: you cannot read the full history; you can modify only the current entry with `history.replaceState` or add a new entry with `history.pushState`.

You can also replace the current page with one from the history, if you know what position it has within the history. This can be very useful for cases in which you know that certain pages in the history came from your site, or you simply wish to send the browser back one or more pages, which you do with the `go` method of the `history` object. For example, to send the browser back three pages, issue the command:

```
history.go(-3)
```

You can also use the following methods to move back or forward a page at a time:

```
history.back()
history.forward()
```

Similarly, you can replace the currently loaded URL with one of your choosing, like this:

```
document.location.href = 'http://google.com'
```

Of course, there's a whole lot more to the DOM than reading and modifying links. As you progress through the following chapters on JavaScript, you'll become quite familiar with the DOM and how to access it.

In [Chapter 14](#) we'll continue our exploration by looking at how to control program flow and write expressions, but first let's repeat what you've learned by answering the following questions.

Questions

1. Which tags do you use to enclose JavaScript code?
2. How can you include JavaScript code from another source in your documents?
3. Which JavaScript function is the equivalent of `echo` or `print` used in PHP for quick output of values or expressions?
4. How can you create a comment in JavaScript?
5. What is the JavaScript string concatenation operator?
6. Which keyword can you use within a JavaScript function to define a variable that has local scope?
7. Give two cross-browser methods to display the URL assigned to the link with an `id` of `thisLink`.
8. Which two JavaScript commands will make the browser load the previous page in its history array?
9. What JavaScript command would you use to replace the current document with the main page at the *oreilly.com* website?

See “Chapter 13 Answers” on page 577 in the [Appendix](#) for the answers to these questions.

Expressions and Control Flow in JavaScript

In [Chapter 13](#), I introduced the basics of JavaScript and the DOM. Now it's time to look at how to construct complex expressions in JavaScript and how to control the program flow of your scripts by using conditional statements.

Expressions

JavaScript expressions are very similar to those in PHP. As you learned in [Chapter 4](#), an expression is a combination of values, variables, operators, and functions that results in a value.

[Example 14-1](#) shows some simple expressions. For each line, it prints out a letter between a and d, followed by a colon and the result of the expressions.

Example 14-1. Four simple Boolean expressions

```
<script>
  console.log("a: " + (42 > 3))
  console.log("b: " + (91 < 4))
  console.log("c: " + (8 === 2))
  console.log("d: " + (4 < 17))
</script>
```

The output from this code is:

```
a: true
b: false
c: false
d: true
```

Notice that both expressions `a:` and `d:` evaluate to `true`, but `b:` and `c:` evaluate to `false`. Unlike PHP (which would print the number 1 and nothing, respectively), the actual strings `true` and `false` are displayed.

In JavaScript, when you are checking whether a value is `true` or `false`, all values evaluate to `true` except the following, which evaluate to `false`:

- The string `false` itself
- `0`
- `-0`
- The empty string
- `null`
- `undefined`
- `NaN` (Not a Number, a computer engineering concept for the result of an illegal floating-point operation such as division by zero)

Note that I am referring to `true` and `false` in lowercase. This is because, unlike in PHP, in JavaScript these values *must* be lowercase. Therefore, only the first of the two following `if` statements will display, printing the lowercase word `true`, because the second will cause a `TRUE is not defined` error:

```
const foo = true
if (foo === true) console.log('foo is true') // True
if (foo === TRUE) console.log('foo is TRUE') // Will cause an error
```



Remember that any code snippets you wish to type and try for yourself in an HTML file need to be enclosed within `<script>` and `</script>` tags.

Literals and Variables

The simplest form of an expression is a *literal*, which means something that evaluates to itself, such as the number 22 or the string "Press Enter". An expression could also be a variable, which evaluates to the value assigned to it. They are both types of expressions, because they return a value.

Example 14-2 shows three different literals and two variables, all of which return values, albeit of different types.

Example 14-2. Five types of literals

```
<script>
  const myname = "Peter"
  const myage  = 24
  console.log("a: " + 42    ) // Numeric literal
  console.log("b: " + "Hi"  ) // String literal
  console.log("c: " + true  ) // Boolean literal
  console.log("d: " + myname) // String variable
  console.log("e: " + myage ) // Numeric variable
</script>
```

And, as you'd expect, you see a return value from all of these in the following output:

```
a: 42
b: Hi
c: true
d: Peter
e: 24
```

Operators let you create more complex expressions that evaluate to useful results. In contrast with an expression, a *statement* is code that does *not* evaluate to a value. Most control flow constructs in JavaScript are statements.

Example 14-3 shows one of each. The first assigns the result of the expression `366 - day_number` to the variable `days_to_new_year`, and the second outputs a friendly message only if the expression `days_to_new_year < 30` evaluates to true.

Example 14-3. Two simple JavaScript statements

```
<script>
  const day_number      = 127 // For example
  const days_to_new_year = 366 - day_number
  if (days_to_new_year < 30) console.log("It's nearly New Year")
  else                      console.log("A long time to go")
</script>
```

Operators

JavaScript offers a lot of powerful operators, ranging from arithmetic, string, and logical operators to assignment, comparison, and more (see [Table 14-1](#)).

Table 14-1. JavaScript operator types

Operator	Description	Example
Arithmetic	Basic mathematics	<code>a + b</code>
Assignment	Assign values	<code>a = b + 23</code>
Bitwise	Manipulate bits within bytes	<code>12 ^ 9</code>
Comparison	Compare two values	<code>a < b</code>

Operator	Description	Example
Increment/decrement	Add or subtract one	a++ / b--
Logical	Boolean	a && b
String	Concatenation	a + 'string'

Each operator takes a different number of operands:

- *Unary* operators, such as incrementing (a++) or negation (-a), take a single operand.
- *Binary* operators, which represent the bulk of JavaScript operators—including addition, subtraction, multiplication, and division—take two operands.
- The one *ternary* operator, which takes the form ? x : y, requires three operands. It's a terse single-line if statement that chooses between two expressions depending on a third one.

Operator Precedence

Like PHP, JavaScript utilizes operator precedence, in which some operators in an expression are processed before others and are therefore evaluated first. [Table 14-2](#) lists JavaScript's operators and their precedences. Check the [MDN page on operator precedence](#) for a detailed description.

Table 14-2. Precedence of JavaScript operators (high to low)

Operator(s)	Type(s)
() [] .	Parentheses, call, and member
++ --	Increment/decrement
+ - ~ !	Unary, bitwise, and logical
* / %	Arithmetic
+ -	Arithmetic and string
<< >> >>>	Bitwise
< > <= >=	Comparison
== != === !==	Comparison
& ^	Bitwise
&&	Logical
	Logical
? :	Ternary
= += -= *= /= %=	Assignment
<<= >>= >>>= &= ^= =	Assignment
,	Separator

Associativity

Most JavaScript operators are processed in order from left to right in an equation. But some operators require processing from right to left instead. The direction of processing is called the operator's *associativity*.

This associativity becomes important where you do not explicitly force precedence (which you should always do, by the way, because it makes code more readable and less error prone). For example, look at the following assignment operators, by which three variables are all set to the value 0:

```
level = score = time = 0
```

This multiple assignment is possible only because the rightmost part of the expression is evaluated first and then processing continues in a right-to-left direction. **Table 14-3** lists the JavaScript operators and their associativity.

Table 14-3. Operators and associativity

Operator	Description	Associativity
++ --	Increment and decrement	None
new	Create a new object	Right
+ - ~ !	Unary and bitwise	Right
?:	Ternary	Right
= *= /= %= += -=	Assignment	Right
<<= >>= >>>= &= ^= =	Assignment	Right
,	Separator	Left
+ - * / %	Arithmetic	Left
<< >> >>>	Bitwise	Left
< <= > >= == != === !==	Arithmetic	Left

Relational Operators

Relational operators test two operands and return a Boolean result of either true or false. There are three types of relational operators: *equality*, *comparison*, and *logical*.

Equality operators

The *equality* operator is == and the *strict equality* operator (sometimes called *identity* operator) is === (neither should be confused with the = assignment operator). Similar to PHP, you're encouraged to always use the *strict equality* operators === and !== as they are more safe and only very rarely use == and !=. In **Example 14-4**, the first statement assigns a value, and the second tests it for equality. As it stands, nothing will be printed out, because month is assigned the string value July, and therefore the check for it having a value of October will fail.

Example 14-4. Assigning a value and testing for equality

```
<script>
  const month = "July"
  if (month === "October") console.log("It's the Fall")
</script>
```

If the non-strict equality operator `==` is used and if the two operands of the expression are of different types, JavaScript will convert them to whatever type makes best sense to it, and this can result in an unexpected behavior. For example, any strings composed entirely of numbers will be converted to numbers whenever compared with a number. In [Example 14-5](#), `a` and `b` are two different values (one is a number, and the other is a string, although an empty one), and we would therefore normally expect neither of the `if` statements to output a result.

Example 14-5. The equality and identity operators

```
<script>
  const a = 0
  const b = ""
  if (a == b) console.log("1")
  if (a === b) console.log("2")
</script>
```

However, if you run the example, you will see that it outputs the number 1, which means that the first `if` statement evaluated to `true`. This is because the string value of `b` was temporarily converted to a number, and therefore both halves of the equation had a numerical value of 0.

In contrast, the second `if` statement uses the *identity* operator, three equals signs in a row, which prevents JavaScript from automatically converting types. `a` and `b` are therefore found to be different, so nothing is output.

To avoid unexpected behavior, you should always use the *strict equality* (*identity*) operator.

Comparison operators

Using comparison operators, you can test for more than just equality and inequality. JavaScript also gives you `>` (is greater than), `<` (is less than), `>=` (is greater than or equal to), and `<=` (is less than or equal to) to play with. [Example 14-6](#) shows these operators in use.

Example 14-6. The four comparison operators

```
<script>
  const a = 7
  const b = 11
  if (a > b) console.log("a is greater than b")
  if (a < b) console.log("a is less than b")
  if (a >= b) console.log("a is greater than or equal to b")
  if (a <= b) console.log("a is less than or equal to b")
</script>
```

In this example, where *a* is 7 and *b* is 11, the following is output (because 7 is less than 11 and also less than or equal to 11):

```
a is less than b
a is less than or equal to b
```

Truthy and falsy values

A *truthy* value is a value that evaluates to true after casting it to Boolean, and vice versa, a *falsy* value is a value that evaluates to false after casting it to Boolean.

Some truthy values, for example:

- true
- Any number except 0, for example 303
- A nonempty string like "hello"
- An array like [1, 2, 3]
- And even an empty array like []

And some falsy values, for example:

- false
- null
- undefined
- Number 0
- Empty string ""

The following code will take the truthy values from the preceding list and typecast them to Boolean, except the first one which is already a Boolean, and print the result:

```

console.log(
  true,           // true
  Boolean(303),   // true
  Boolean("hello"), // true
  Boolean([1, 2, 3]), // true
  Boolean([]),    // true
  Boolean('hello') // true
)

```

And the same for the falsy values:

```

console.log(
  false,           // false
  Boolean(null),   // false
  Boolean(undefined), // false
  Boolean(0),      // false
  Boolean("")      // false
)

```

Logical operators

Logical operators produce *truthy* or *falsy* results and are also known as *Boolean* operators. JavaScript has three of them (see [Table 14-4](#)).

Table 14-4. JavaScript's logical operators

Logical operator	Description
<code>x && y</code> (<i>and</i>)	<code>y</code> if <code>x</code> is truthy, otherwise <code>x</code>
<code>x y</code> (<i>or</i>)	<code>x</code> if <code>x</code> is truthy, otherwise <code>y</code>
<code>!x</code> (<i>not</i>)	true if <code>x</code> is falsy, otherwise false

You can see how these can be used in [Example 14-7](#), which outputs 0, 1, and true.

Example 14-7. The logical operators in use

```

<script>
  const a = 1
  const b = 0
  console.log(a && b)
  console.log(a || b)
  console.log(!b)
</script>

```

The `&&` statement requires both operands to be true to return a value of true, the `||` statement will be true if either value is true, and the third statement performs a NOT on the value of `b`, turning it from 0 into a value of true.

Both the `||` and `&&` operators can cause unexpected problems, because the second operand will not be evaluated if the first is evaluated as true in case of `||`, or as false in case of `&&`. This is a feature called *short-circuit evaluation*. In [Example 14-8](#), the `getNext` function will never be called if `finished` has a value of 1 (these are just examples, and the action of `getNext` is irrelevant to this explanation—just think of it as a function that does *something* when called).

Example 14-8. A statement using the `||` operator

```
<script>
  if (finished === 1 || getNext() === 1) done = 1
</script>
```

If you *need* `getNext` to be called at each `if` statement, you should rewrite the code as shown in [Example 14-9](#).

Example 14-9. The `if...or` statement modified to ensure calling of `getNext`

```
<script>
  gn = getNext()
  if (finished === 1 || gn === 1) done = 1;
</script>
```

In this case, the code in the function `getNext` will be executed and its return value stored in `gn` before the `if` statement.

[Table 14-5](#) shows all the possible variations of using the logical operators. You should also note that `!true` equals false and `!false` equals true.

Table 14-5. All possible logical expressions

Inputs		Operators and results	
a	b	&&	
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Using onerror

Using either the `onerror` event or a combination of the `try` and `catch` keywords, you can catch JavaScript errors and deal with them yourself.

Events are actions that can be detected by JavaScript. Every element on a web page has certain events that can trigger JavaScript functions. For example, the `click` event (sometimes rather incorrectly called the `onclick` event, as the `on` prefix is used only for event handler property names) of a button element can be set to call a function and make it run whenever a user clicks the button.

Example 14-10 illustrates how to use the `onerror` event.

Example 14-10. A script employing the `onerror` event

```
<script>
  onerror = errorHandler
  console.log("Welcome to this website") // Deliberate error

  function errorHandler(message, url, line)
  {
    out = "Sorry, an error was encountered.\n\n";
    out += "Error: " + message + "\n";
    out += "URL: "   + url   + "\n";
    out += "Line: "  + line  + "\n\n";
    out += "Click OK to continue.\n\n";
    alert(out);
    return true;
  }
</script>
```

The first line of this script tells the error event to use the new `errorHandler` function from now on. This function takes three parameters—a `message`, a `url`, and a `line` number—so it's a simple matter to display all these in an alert pop-up.

Then, to test the new function, we deliberately place a syntax error in the code with a call to `console.log` instead of `console.log` (the final `g` is replaced with `l`).

Figure 14-1 shows the result of running this script in a browser. Using `onerror` this way can also be quite useful during debugging.

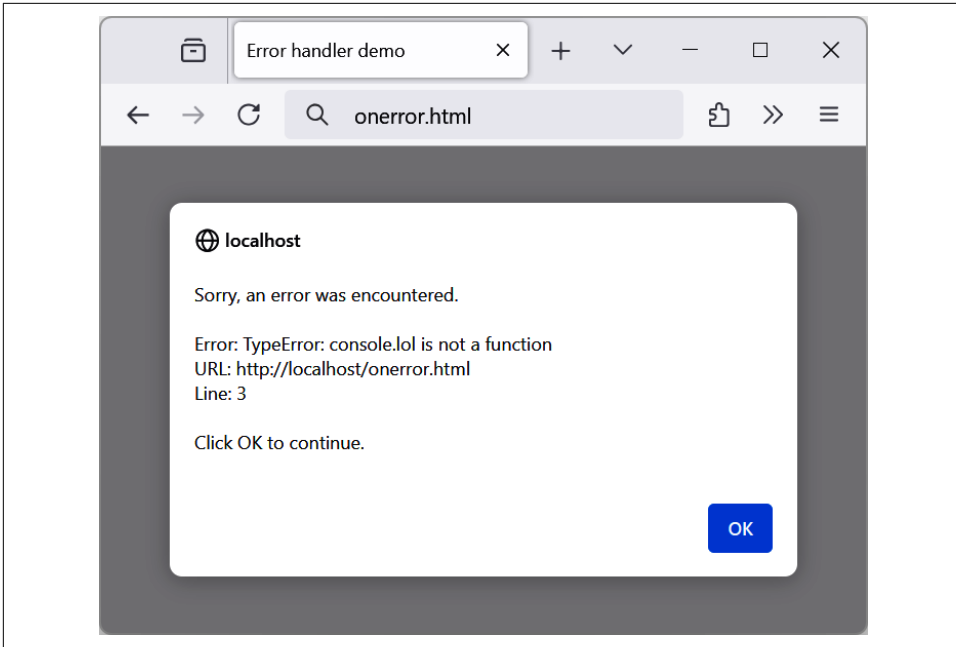


Figure 14-1. Using the `onerror` event with an alert method pop-up

Using `try...catch`

The `try` and `catch` keywords are more standard and more flexible than the `onerror` technique shown in “Using `onerror`” on page 342. These keywords let you trap errors for a selected section of code, rather than all scripts in a document. However, they do not catch syntax errors, for which you need `onerror`.

The `try...catch` construct is supported by all major browsers and is handy when you want to catch a certain condition that you are aware could occur in a specific part of your code.

When working with elements, you can use `try` and `catch` to do something else if the element is not available. Example 14-11 shows how.

Example 14-11. Trapping an error with `try` and `catch`

```
<script>
  try {
    document.getElementById('el').innerHTML = '...';
  }
  catch(err) {
    alert("Oh no! There's no element with ID 'el!'")
  }
</script>
```

```
}  
</script>
```

Another keyword associated with `try` and `catch` called `finally` is always executed, regardless of whether an error occurs in the `try` clause. To use it, for example to clean up some resources, just add something like the following statements after a `catch` statement:

```
finally {  
    alert("The 'try' clause was encountered")  
}
```

Conditionals

Conditionals alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways. There are three types of nonlooping conditionals: the `if` statement, the `switch` statement, and the `?` operator.

The if Statement

Several examples in this chapter have already used `if` statements. The code within such a statement is executed only if the given expression evaluates to `true`. Multiline `if` statements require curly braces around them, but as in PHP, you can omit the braces for single statements, although it's often a good idea to use them anyway, especially when writing code in which the number of actions within an `if` statement might change as development proceeds. Therefore, the following statements are valid:

```
if (a > 100) {  
    b = 2  
    console.log("a is greater than 100")  
}  
  
if (b === 10) console.log("b is equal to 10")
```

The else Statement

When a condition has not been met, you can execute an alternative by using an `else` statement, like this:

```
if (a > 100) {  
    console.log("a is greater than 100")  
}  
else {  
    console.log("a is less than or equal to 100")  
}
```


Unlike PHP, JavaScript has no `elseif` statement, but that's not a problem because you can use an `else` followed by another `if` to form the equivalent of an `elseif` statement, like this:

```
if (a > 100) {  
    console.log("a is greater than 100")  
}  
else if (a < 100) {  
    console.log("a is less than 100")  
}  
else {  
    console.log("a is equal to 100")  
}
```

As you can see, you can use another `else` after the new `if`, which could equally be followed by another `if` statement, and so on. Although I have shown braces on the statements, because each is a single line, the previous example could be written:

```
if (a > 100) console.log("a is greater than 100")  
else if (a < 100) console.log("a is less than 100")  
else console.log("a is equal to 100")
```

The switch Statement

The `switch` statement is useful when one variable or the result of an expression can have multiple values and you want to perform a different function for each value.

For example, the following code takes the PHP menu system we put together in [Chapter 4](#) and converts it to JavaScript. It works by passing a single string to the main menu code according to what the user requests. Let's say the options are Home, About, News, Login, and Links, and we set the variable `page` to one of these according to the user's input.

The code for this written using `if...else if...` will look like [Example 14-12](#).

Example 14-12. A multiline `if...else if...` statement

```
<script>  
if (page === "Home") console.log("You selected Home")  
else if (page === "About") console.log("You selected About")  
else if (page === "News") console.log("You selected News")  
else if (page === "Login") console.log("You selected Login")  
else if (page === "Links") console.log("You selected Links")  
</script>
```

But using a `switch` construct, the code could look like [Example 14-13](#).

Example 14-13. A *switch* construct

```
<script>
  switch (page) {
    case "Home":
      console.log("You selected Home")
      break
    case "About":
      console.log("You selected About")
      break
    case "News":
      console.log("You selected News")
      break
    case "Login":
      console.log("You selected Login")
      break
    case "Links":
      console.log("You selected Links")
      break
  }
</script>
```

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



You can also supply multiple cases for a single action; this is called *fall-through cases*. For example:

```
switch (heroName) {
  case "Superman":
  case "Batman":
  case "Wonder Woman":
    console.log("Justice League")
    break
  case "Iron Man":
  case "Captain America":
  case "Spiderman":
    console.log("The Avengers")
    break
}
```

Breaking out

As you can see in [Example 14-13](#), just as with PHP, the `break` command allows your code to break out of the `switch` statement once a condition has been satisfied. Remember to include the `break` unless you want to continue executing the statements under the next case.

Default action

When no condition is satisfied, you can specify a default action for a switch statement by using the `default` keyword. [Example 14-14](#) shows a code snippet that could be inserted into [Example 14-13](#).

Example 14-14. A default statement to add to [Example 14-13](#)

```
default:
  console.log("Unrecognized selection")
  break
```

The ? Operator

The ternary operator, which looks like "*condition ? ifTrue : ifFalse*" provides a shorthand alternative to `if...else`. With it you can write an expression to evaluate and then follow it with a `?` symbol and the code to execute (*ifTrue*) if the expression is true. After that, place a `:` and the code to execute (*ifFalse*) if the expression evaluates to false.

[Example 14-15](#) shows the ternary operator being used to print out whether the variable `a` is less than or equal to 5 and prints something either way.

Example 14-15. Using the ternary operator

```
<script>
  console.log(
    a <= 5
    ? "a is less than or equal to 5"
    : "a is greater than 5"
  )
</script>
```

In this example, the statement has been broken into several lines for clarity. However, if the operands are short, it's common to write ternaries on one line, such as:

```
size = a <= 5 ? "short" : "long"
```

Looping

Again, you will find many close similarities between JavaScript and PHP when it comes to looping. Both languages support `while`, `do...while`, and `for` loops.

while Loops

A JavaScript `while` loop first checks the value of an expression and starts executing the statements within the loop only if that expression is `true`. If it is `false`, the loop terminates.

Upon completing an iteration of the loop, the expression is again tested to see if it is `true`, and the process continues until the expression evaluates to `false` or until execution is otherwise halted. [Example 14-16](#) shows such a loop.

Example 14-16. A `while` loop

```
<script>
  let counter = 0

  while (counter < 5)
  {
    console.log("Counter: " + counter)
    ++counter
  }
</script>
```

This script outputs this:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
```



If the variable `counter` were not incremented within the loop, it is quite possible that some browsers could become unresponsive due to a never-ending loop, and the page might not even be easy to terminate with Escape or the Stop loading page button. So, be careful with your JavaScript loops.

do...while Loops

When you require a loop to iterate at least once before any tests are made, use a `do...while` loop, which is similar to a `while` loop, except that the test expression is checked only after each iteration of the loop. So, to output the first seven results in the 7 times table, you could use code like in [Example 14-17](#).

Example 14-17. A `do...while` loop

```
<script>
  let count = 1
```

```

do {
    console.log(count + " times 7 is " + count * 7)
} while (++count <= 7)
</script>

```

As you might expect, this loop outputs:

```

1 times 7 is 7
2 times 7 is 14
3 times 7 is 21
4 times 7 is 28
5 times 7 is 35
6 times 7 is 42
7 times 7 is 49

```

for Loops

A for loop gives you extensive control over the loop conditions by providing three parameters:

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

These are separated by semicolons, like this: `for (expr1 ; expr2 ; expr3)`. The initialization expression is executed at the start of the first iteration of the loop. In the case of the code for the multiplication table for 7, `count` would be initialized to the value 1. Then, each time around the loop, the condition expression (in this case, `count <= 7`) is tested, and the loop is entered only if the condition is true. Finally, at the end of each iteration, the modification expression is executed. In the case of the multiplication table for 7, the variable `count` is incremented. [Example 14-18](#) shows what the code would look like.

Example 14-18. Using a for loop

```

<script>
    for (count = 1 ; count <= 7 ; ++count) {
        console.log(count + "times 7 is " + count * 7);
    }
</script>

```

As in PHP, you can assign multiple variables in the first parameter of a for loop by separating them with a comma, like this:

```

for (i = 1, j = 1 ; i < 10 ; i++)

```

Likewise, you can perform multiple modifications in the last parameter, like this:

```
for (i = 1 ; i < 10 ; i++, --j)
```

Or you can do both at the same time:

```
for (i = 1, j = 1 ; i < 10 ; i++, --j)
```

A variant worth mentioning is the `for...of` loop. You can use it to loop over values coming from iterable objects, for example, arrays:

```
const array = [1, 2, 3]
for (let value of array) {
  console.log(value)
}
```

Breaking Out of a Loop

The `break` command, which you'll recall is important inside a `switch` statement, is also available within `for` loops. You might need to use this, for example, when searching for a match of some kind. Once the match is found, you know that continuing to search will only waste time and make your visitor wait. [Example 14-19](#) shows how to use the `break` command.

Example 14-19. Using the `break` command in a `for` loop

```
<script>
const haystack = new Array()
haystack[17] = "Needle"

for (let j = 0 ; j < 20 ; ++j)
{
  if (haystack[j] === "Needle")
  {
    console.log("- Found at location " + j)
    break
  }
  else console.log(j + ", ")
}
</script>
```

This script outputs:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
- Found at location 17
```

The continue Statement

Sometimes you don't want to entirely exit from a loop but instead wish to skip the remaining statements just for this iteration of the loop. In such cases, you can use the `continue` command. [Example 14-20](#) shows this in use.

Example 14-20. Using the `continue` command in a `for` loop

```
<script>
  const haystack = new Array()
  haystack[4]    = "Needle"
  haystack[11]   = "Needle"
  haystack[17]   = "Needle"

  for (let j = 0 ; j < 20 ; ++j)
  {
    if (haystack[j] === "Needle")
    {
      console.log("- Found at location " + j)
      continue
    }

    console.log(j + ", ")
  }
</script>
```

Notice how the second `console.log` call does not have to be enclosed in an `else` statement (as it did before), because the `continue` command will skip it if a match has been found. The output from this script is:

```
0, 1, 2, 3,
- Found at location 4
5, 6, 7, 8, 9, 10,
- Found at location 11
12, 13, 14, 15, 16,
- Found at location 17
18, 19,
```

Explicit Casting

Unlike PHP, JavaScript has no explicit casting of types such as `(int)` or `(float)`. Instead, when you need a value to be of a certain type, use one of JavaScript's built-in functions, shown in [Table 14-6](#).

Table 14-6. JavaScript's type-changing functions

Change to type	Function to use
Int, Integer	<code>parseInt()</code>
Bool, Boolean	<code>Boolean()</code>
Float, Double, Real	<code>parseFloat()</code>
String	<code>String()</code>
Array	<code>split()</code>

So, for example, to change a floating-point number to an integer, you could use the following code (which displays the value 3):

```
n = 3.1415927
i = parseInt(n, 10) // 10 is an optional radix, recommended to always use it
console.log(i)
```

That's it for control flow and expressions; you can use the following questions to confirm your understanding. [Chapter 15](#) focuses on the use of functions, objects, and arrays in JavaScript.

Questions

1. How are Boolean values handled differently by PHP and JavaScript?
2. What characters are used to define a JavaScript variable name?
3. What is the difference between unary, binary, and ternary operators?
4. What is the best way to force your own operator precedence?
5. When would you use the `===` (identity) operator?
6. What are the simplest two forms of expressions?
7. Name the three conditional statement types.
8. How do `if` and `while` statements interpret conditional expressions of different data types?
9. When might you prefer a `for` loop over a `while` loop, and vice versa?
10. How can you cast one type to another in JavaScript?

See “[Chapter 14 Answers](#)” on page 578 in the [Appendix](#) for the answers to these questions.

JavaScript Functions, Objects, and Arrays

Just like PHP, JavaScript offers access to functions and objects. In JavaScript objects are the primary means for accessing the Document Object Model (DOM), because—as you’ve seen—every element of an HTML document is available to be manipulated as an object.

The usage and syntax are also quite similar to those of PHP, so you should feel right at home as I take you through using functions and objects in JavaScript, as well as through an in-depth exploration of array handling.

JavaScript Functions

In addition to having access to dozens of built-in functions (or methods), such as `log`, which you have already seen being used in `console.log`, you can easily create your own functions. A relatively complex piece of code that is likely to be reused is a candidate for a function.

Defining a Function

The general syntax for a function is:

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

The first line of the syntax indicates:

- A definition starts with the word `function`.
- A name follows that must start with a letter or underscore, followed by any number of letters, digits, dollar signs, or underscores, the same rules as for any other identifier.

- The parentheses are required.
- One or more parameters, separated by commas, are optional (indicated by the square brackets, which are not part of the function syntax).

Like all identifiers in JavaScript, function names are case-sensitive, so all of the following strings refer to different functions: `getInput`, `GETINPUT`, and `getinput`.

JavaScript has a general naming convention for functions: the first letter of each word in a name is capitalized, except for the very first letter, which is lowercase. Therefore, of the previous examples, `getInput` would be the preferred name used by most programmers. This convention is referred to as *camelCase*, because the capitalized letters resemble a camel's humps (or sometimes called *bumpyCaps* or *bumpyCase*).

The opening curly brace starts the statements that will execute when you call the function; a matching curly brace must close it. These statements may include one or more `return` statements, which force the function to cease execution and return to the calling code. If a value is attached to the `return` statement, the calling code can retrieve it. If a function does not have a `return` statement it implicitly returns `undefined`.

The rest parameter

With the *rest parameter* syntax `...params`, a function can accept a virtually infinite number of parameters. Take the example of a function called `displayItems`. **Example 15-1** shows one way of writing it, using exactly 5 parameters.

Example 15-1. Defining a function

```
<script>
  displayItems("Dog", "Cat", "Pony", "Hamster", "Tortoise")

  function displayItems(v1, v2, v3, v4, v5)
  {
    console.log(v1)
    console.log(v2)
    console.log(v3)
    console.log(v4)
    console.log(v5)
  }
</script>
```

When you call up this script in your browser, it will display the following in the browser console:

Dog
Cat
Pony
Hamster
Tortoise

All of this is fine, but what if you wanted to pass more than five items to the function? Also, reusing the `console.log` call multiple times instead of employing a loop is wasteful programming. Luckily, the rest parameter syntax `...params`, gives you the flexibility to handle a variable number of arguments. You can use any other name, for example `...args`; the leading dots `...` are the important part of the syntax, not the parameter name.

Inside the function, the rest parameter is available as an array of values passed to the function as parameters when the function is called. [Example 15-2](#) shows how you can use it to rewrite the previous example much more efficiently.

Example 15-2. Modifying the function to use the rest parameter syntax

```
<script>
  let c = "Car"

  displayItems("Bananas", 32.3, c)

  function displayItems(...params)
  {
    for (j = 0 ; j < params.length ; ++j)
      console.log(params[j])
  }
</script>
```

Note the use of the `length` property, which you encountered in [Chapter 14](#). Also note that I reference the `params` array using the variable `j` as an offset into it. And I chose to keep the function short and sweet by not surrounding the contents of the `for` loop in curly braces, as it contains only a single statement. Remember that because the `<` operator is used, the loop must stop when `j` is one less than `length`, not equal to `length`.

Using this technique, you now have a function that can take as many (or as few) arguments as you like and act on each argument as you desire.

The arguments array

The `arguments` array is automatically available inside every function and offers a similar way of working with function parameters. Using the array, the `displayItems` function from [Example 15-2](#) can be written like this:

```
function displayItems()
{
  for (j = 0 ; j < arguments.length ; ++j)
    console.log(arguments[j])
}
```

However, the *rest parameter* syntax is strongly preferred over the `arguments` array because the *rest parameter* is visible when you look at the function parameters, so it's clear that the function accepts *some* parameters, unlike when you use the `arguments` array.

Returning a Value

Functions are not used just to display things. In fact, they are used mostly to perform calculations or data manipulations and then return a result. The function `fixNames` in [Example 15-3](#) uses the *rest parameter* syntax (discussed in “[The rest parameter](#)” on [page 354](#)) to take a series of strings passed to it and return them as a single string. The “fix” it performs is to convert every character in the arguments to lowercase except for the first character of each argument, which is set to a capital letter.

Example 15-3. Cleaning up a full name

```
<script>
  console.log(fixNames("the", "DALLAS", "CowBoys"))

  function fixNames(...names)
  {
    let s = ""

    for (j = 0 ; j < names.length ; ++j)
      s += names[j].charAt(0).toUpperCase() +
          names[j].substring(1).toLowerCase() + " "

    return s.substring(0, s.length-1)
  }
</script>
```

When called with the parameters `the`, `DALLAS`, and `CowBoys`, for example, the function returns the string `The Dallas Cowboys`. Let's walk through the function.

It first initializes the temporary (and local) variable `s` to the empty string. Then a `for` loop iterates through each of the passed parameters, isolating the parameter's first character using the `charAt` method and converting it to uppercase with the `toUpperCase` method. The various methods shown in this example are all built into JavaScript and available by default.

Then the `substring` method is used to fetch the rest of each string, which is converted to lowercase via the `toLowerCase` method. A fuller version of the `substring`

method here would specify how many characters are part of the substring as a second argument:

```
substring(1, (names[j].length) - 1)
```

In other words, this `substring` method says, “Start with the character at position 1 (the second character) and return the rest of the string (the length minus one).” As a nice touch, though, the `substring` method assumes that you want the rest of the string if you omit the second argument.

After the whole argument is converted to the desired case, a space character is added to the end, and the result is appended to the temporary variable `s`.

Finally, the `substring` method is used again to return the contents of the variable `s`, except for the final space—which is unwanted. We remove this by using `substring` to return the string up to, but not including, the final character.

This example is particularly interesting in that it illustrates the use of multiple properties and methods in a single expression, for example:

```
names[j].substring(1).toLowerCase()
```

You have to interpret the statement by mentally dividing it into parts at the periods. JavaScript evaluates these elements of the statement from left to right:

1. Start with the *rest parameter* `names` representing an array of `fixNames` arguments.
2. Extract element `j` from the array.
3. Invoke `substring` with a parameter of 1 to the extracted element. This passes all but the first character to the next section of the expression.
4. Apply the method `toLowerCase` to the string that has been passed thus far.

This practice is often referred to as *method chaining*. So, for example, if the string `mixedCASE` is passed to the example expression, it will go through the following transformations:

```
mixedCASE  
ixedCASE  
ixedcase
```

In other words, `names[j]` produces “mixedCASE,” then `substring(1)` takes “mixedCASE” and produces “ixedCASE,” and finally `toLowerCase()` takes “ixedCASE” and produces “ixedcase.”

One final reminder: the `s` variable created inside the function is local and therefore cannot be accessed outside the function. By returning `s` in the `return` statement, we made its value available to the caller, which could store or use it any way it wanted. But `s` itself disappears at the end of the function. Although we could make

a function operate on global variables (and sometimes that's necessary in legacy code for example), it's much better to just return the values you want to preserve and let JavaScript clean up all the other variables used by the function.

Returning an Array

In [Example 15-3](#), the function returned only one parameter—but what if you need to return multiple parameters? You can do this by returning an array, as shown in [Example 15-4](#).

Example 15-4. Returning an array of values

```
<script>
  words = fixNames("the", "DALLAS", "CowBoys")

  for (j = 0 ; j < words.length ; ++j)
    console.log(words[j])

  function fixNames(...names)
  {
    let s = []

    for (j = 0 ; j < names.length ; ++j)
      s[j] = names[j].charAt(0).toUpperCase() +
            names[j].substr(1).toLowerCase()

    return s
  }
</script>
```

Here the variable `words` is automatically defined as an array and populated with the returned result of a call to the function `fixNames`. Then a `for` loop iterates through the array and displays each member. You can also use `console.log(words)` here instead of the `for` loop to see the full array.

As for the `fixNames` function, it's almost identical to [Example 15-3](#), except that the variable `s` is now an array; after each word has been processed, it is stored as an element of this array, which is returned by the `return` statement.

This function enables the extraction of individual parameters from its returned values, like the following (the output is simply `The Cowboys`):

```
words = fixNames("the", "DALLAS", "CowBoys")
console.log(words[0] + " " + words[2])
```

JavaScript Objects

A JavaScript object is more complex than a variable, which can contain only one value at a time. In contrast, objects can contain multiple values and even functions. An object groups data together with the functions needed to manipulate it.

The simplest object you can have in JavaScript is an empty object with no values and no functions. This is how you can create it:

```
const user = {}  
console.log(typeof user) // displays "object"
```

Objects created this way are often used to emulate PHP's associative arrays that JavaScript doesn't support natively, something we'll explore later in this chapter.

Declaring a Class

When writing a code that uses the *object-oriented programming* approach, you need to design a composite of data and code called a *class*. Classes should ideally be modeled after real-world items, so you should create different classes for items such as a user, an order, or a cart. Each new object based on such class is called an *instance* (or *occurrence*) of that class. As you've already seen, the data associated with an object is called its *properties*, while the functions it uses are called *methods*.

Let's look at how to declare the class for an object called `User` that will contain details about the current user. The class will have three properties, a special method called a *constructor* (I'll show later how it's invoked), and a method to display the data, `showUser`. **Example 15-5** shows the code declaring the class and creating an instance with the `new` keyword.

Example 15-5. Declaring the `User` class, its properties and methods, creating an instance

```
<script>  
class User {  
  constructor(forename, username, timezone) {  
    this.forename = forename  
    this.username = username  
    this.timezone = timezone  
  }  
  
  showUser() {  
    console.log("Forename: " + this.forename)  
    console.log("Username: " + this.username)  
    console.log("Timezone: " + this.timezone)  
  }  
}  
</script>
```

Creating an Instance

When you want to work with an instance (or object) of a class, you first need to create it. Here is the line with the new keyword:

```
const user = new User("Fred", "fred303", "UTC")
```

Immediately after creating the instance, the constructor method will be automatically called and the values Fred, fred303, and UTC will be passed to it as the forename, username, and timezone parameters. The constructor is then used to initialize the object properties referencing an object named `this`, which refers to the instance being created:

```
this.forename = forename
this.username = username
this.timezone = timezone
```

The created instance is then assigned to the `user` variable. The class also contains the `showUser` method, which will show the values of the object properties in your browser console when called.

You can create multiple instances from the same class by using the `new` keyword again, possibly passing different values to the constructor, like this:

```
const details = new User("Waldo", "waldo2600", "UTC+2")
```

The naming convention I have used is to start the name of the class itself with a capital letter (`User`), unlike the instance name (`user`), which can be different than the class name (for example, `details`), and keep all properties in lowercase and to use at least one uppercase character in method names, following the camelCase convention mentioned earlier in the chapter.

Accessing Objects

To access an object, you can refer to its properties, as in the following two unrelated example statements:

```
const name = user.forename
if (user.username === "admin") loginAsAdmin()
```

So, to access the `showUser` method of an object of class `User`, you would use the following syntax, in which the object `user` has already been created and populated with data:

```
user.showUser()
```

Assuming the data supplied earlier, this code would display:

```
Forename: Fred
Username: fred303
Timezone: UTC
```


Static Methods and Properties

When reading about PHP objects, you learned that classes can have static properties and methods as well as properties and methods associated with a particular instance of a class. JavaScript also supports static properties and methods; these are prefixed by the `static` keyword in the class declaration as you can see in [Example 15-6](#).

Example 15-6. Adding a static property and a static method to the User class

```
<script>
  class User {
    constructor(forename, username, timezone) {
      this.forename = forename
      this.username = username
      this.timezone = timezone
    }

    static greeting = "Hello";

    showUser() {
      console.log("Forename: " + this.forename)
      console.log("Username: " + this.username)
      console.log("Timezone: " + this.timezone)
    }

    static greet(name) {
      console.log(this.greeting + " " + name)
    }
  }
</script>
```

These can be accessed without creating an instance, by referencing the class:

```
console.log(User.greeting)
User.greet("Jack")
```

The Legacy Objects Simulated with Functions

A lot of existing code uses functions to simulate objects and classes seen in the previous sections. I show you the following forms mainly because you are certain to encounter them when perusing other programmers' code. When writing a new code, it's recommended to use the class syntax seen in [Example 15-5](#).

Example 15-7. Declaring the User class using functions

```
<script>
  function User(forename, username, timezone)
  {
    this.forename = forename
```

```

    this.username = username
    this.timezone = timezone

    this.showUser = function()
    {
        console.log("Forename: " + this.forename)
        console.log("Username: " + this.username)
        console.log("Timezone: " + this.timezone)
    }
}
const user = new User("Wolfgang", "w.a.mozart", "UTC+2")
user.showUser()
</script>

```

Note that there are no `class` and `constructor` keywords, the function name serves as the class name, and the `User` function itself acts similarly to the constructor:

```
function User(forename, username, timezone)
```

The `User` function also emulates methods by storing functions in properties:

```
    this.showUser = function()
```

Some other existing code also refers to functions defined outside the constructor when setting the method-properties, as in [Example 15-8](#). This approach shouldn't be used for any new code.

Example 15-8. Separately defining a class and method

```

<script>
    function showUser()
    {
        console.log("Forename: " + this.forename)
        console.log("Username: " + this.username)
        console.log("Timezone: " + this.timezone)
    }

    function User(forename, username, timezone)
    {
        this.forename = forename
        this.username = username
        this.timezone = timezone
        this.showUser = showUser
    }
</script>

```

JavaScript Arrays

Array handling in JavaScript is very similar to PHP, although the syntax is a little different. Nevertheless, given all you have already learned about arrays, this section should be relatively straightforward.

Arrays

To create a new array, use the following bracket syntax:

```
arrayname = []
```

Or you can use the longer form:

```
arrayname = new Array()
```

Assigning element values

In PHP, you could add a new element to an array by simply assigning it without specifying the element offset, like this:

```
$arrayname[] = "Element 1";  
$arrayname[] = "Element 2";
```

But in JavaScript you use the push method to achieve the same thing:

```
arrayname.push("Element 1")  
arrayname.push("Element 2")
```

This allows you to keep adding items to an array without having to keep track of the number of items. When you need to know how many elements are in an array, you can use the length property:

```
console.log(arrayname.length)
```

Alternatively, if you wish to keep track of the element locations yourself and place them in specific locations, you can use this syntax:

```
arrayname[0] = "Element 1"  
arrayname[1] = "Element 2"
```

Example 15-9 shows a simple script that creates an array, loads it with some values, and then displays them.

Example 15-9. Creating, building, and printing an array

```
<script>  
  numbers = []  
  numbers.push("One")  
  numbers.push("Two")  
  numbers.push("Three")
```

```
for (j = 0 ; j < numbers.length ; ++j)
  console.log("Element " + j + " = " + numbers[j])
</script>
```

The output from this script is:

```
Element 0 = One
Element 1 = Two
Element 2 = Three
```

Assignment using the array keyword

You can also create an array together with some initial elements, like this:

```
numbers = ["One", "Two", "Three"]
```

Nothing is stopping you from adding more elements afterward as well.

You’ve now seen a couple of ways you can add items to an array, and one way of referencing them. JavaScript offers many more, which I’ll get to shortly—but first, we’ll look at another type of array.

Associative Arrays

An *associative array* is one in which the elements are referenced by name rather than by an integer offset. However, JavaScript doesn’t support such things. Instead, we can achieve a similar result by creating an object with properties that will act the same way.

So, to create an “associative array,” define a block of elements within curly braces. For each element, place the key on the left and the contents on the right of a colon (:). **Example 15-10** shows how you might create an “associative array” to hold the contents of the “balls” section of an online sports equipment retailer.

Example 15-10. Creating and displaying an associative array

```
<script>
  balls = {golf:    "Golf balls, 6",
          tennis:  "Tennis balls, 3",
          soccer:  "Soccer ball, 1",
          ping:    "Ping Pong balls, 1 doz"}

  for (const ball in balls)
    console.log(ball + " = " + balls[ball])
</script>
```

I’ve used “associative array” in quotes because we’re actually creating an object, which you can verify with `console.log(typeof balls)`.

To verify that the object has been correctly created and populated, I have used another kind of for loop using the `in` keyword. This creates a new variable to use only within the loop (`ball`, in this example) and iterates through all elements of the object to the right of the `in` keyword (`balls`, in this example). The loop acts on each element of `balls`, placing the key value into `ball`.

Using this property name stored in `ball`, you can also get the value of the current element of `balls`. The result of calling up the example script in a browser is:

```
golf = Golf balls, 6
tennis = Tennis balls, 3
soccer = Soccer ball, 1
ping = Ping Pong balls, 1 doz
```

To get a specific element of an object, you can specify a key explicitly (in this case, outputting the value `Soccer ball, 1`):

```
console.log(balls.soccer)
```

Or you can use the “array” syntax to access the property, like this:

```
console.log(balls['soccer'])
```

Multidimensional Arrays

To create a multidimensional (or, more accurately, a nested) array in JavaScript, just place arrays inside other arrays. For example, to create an array to hold the details of a two-dimensional checkerboard (8×8 squares), you could use the code in [Example 15-11](#).

Example 15-11. Creating a multidimensional numeric array

```
<script>
checkerboard = [
  [' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'],
  ['o', ' ', 'o', ' ', 'o', ' ', 'o', ' '],
  [' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'],
  [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
  [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
  ['o', ' ', 'o', ' ', 'o', ' ', 'o', ' '],
  [' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'],
  ['o', ' ', 'o', ' ', 'o', ' ', 'o', ' ']]

let s = '';
for (let j = 0 ; j < 8 ; ++j) {
  for (let k = 0 ; k < 8 ; ++k)
    s += checkerboard[j][k] + " "

  s += '\n'
```

```

    }
    console.log(s)
</script>

```

In this example, the lowercase letters represent black pieces, and the uppercase white. A pair of nested for loops walks through the array and builds a string that is then displayed in the console.

The outer loop contains two statements, so curly braces enclose them. The inner loop then processes each square in a row, outputting the character at location `[j][k]`, followed by a space (to square up the printout). This loop contains a single statement, so curly braces are not required to enclose it. The result looks like this:

```

    o   o   o   o
o  o   o   o
    o   o   o   o

```

```

0   0   0   0
0   0   0   0
0   0   0   0

```

You can also directly access any element within this array by using square brackets:

```

console.log(checkerboard[7][2])

```

This statement outputs the uppercase letter O, the eighth element down and the third along—remember that array indexes start at 0, not 1.

Using Array Methods

Given the power of arrays, JavaScript comes ready-made with a number of methods for manipulating them and their data. These are almost exclusively used with arrow functions, which will be explained later in this chapter, but we'll use the regular named functions in the following examples to help you better understand the array methods. Here is a selection of the most useful ones.

some and every

When you need to know whether at least one array element matches a certain criterion, you can use the `some` function, which will test all the elements and automatically stop and return the required value as soon as one matches. When you need to know whether all elements match, you can use `every`. This saves you from having to write your own code to perform such searches, like this:

```

function isBiggerThan10(element)
{
    return element > 10
}

```

```
result = [2, 5, 8, 1, 4].some(isBiggerThan10); // result will be false
result = [12, 5, 8, 1, 4].some(isBiggerThan10); // result will be true
result = [12, 5, 8, 1, 4].every(isBiggerThan10); // result will be false
result = [12, 42, 2600].every(isBiggerThan10); // result will be true
```

includes

If you want to know whether an array contains a value, use the `includes` method:

```
result = [2, 5, 8, 1, 4].includes(7); // result will be false
result = [2, 5, 8, 1, 4].includes(8); // result will be true
result = ['Hello', 'Hi'].includes('Hi'); // result will be true
result = ['Hello', 'Hi'].includes('Bye'); // result will be false
```

map

Sometimes, you would like to apply the result of calling a function to all array values. JavaScript offers a `map` method you can use to return the new array, like this:

```
function add10(element)
{
    return element + 10
}

result = [2, 5, 8, 1, 4].map(add10);
console.log(result) // result is [12, 15, 18, 11, 14]
```

filter

You can use the `filter` method to return a new array that will contain only the elements which, when passed to the provided function, returned `true`:

```
function isBiggerThan10(element)
{
    return element > 10
}

result = [12, 5, 18, 11, 4].filter(isBiggerThan10);
console.log(result) // result is [12, 18, 11]
```

indexOf

To find out where an element can be found in an array, you can call the `indexOf` function on the array, which will return the offset of the located element (starting from 0), or -1 if it is not found. For example, the following gives offset the value 2:

```
animals = ['cat', 'dog', 'cow', 'horse', 'elephant']
offset = animals.indexOf('cow')
```

concat

The `concat` method concatenates two arrays or a series of values within an array. For example, the following code outputs `Banana, Grape, Carrot, Cabbage`:

```
fruit = ["Banana", "Grape"]
veg   = ["Carrot", "Cabbage"]

console.log(fruit.concat(veg))
```

The same can be accomplished with the *spread syntax* using `...` (not to be confused with the *rest parameter syntax*, which is used only for function parameters), which, when followed by an array name, will be replaced by all array values, as if they were directly written there. The following code outputs the same array as the previous example:

```
fruit = ["Banana", "Grape"]
veg   = ["Carrot", "Cabbage"]

console.log([...fruit, ...veg])
```

When using `concat`, you also can specify multiple arrays as arguments, in which case `concat` adds all their elements in the order that the arrays are specified.

Here's another way to use `concat`. This time, plain values are concatenated with the array `pets`, which outputs `Cat, Dog, Fish, Rabbit, Hamster`:

```
pets      = ["Cat", "Dog", "Fish"]
more_pets = pets.concat("Rabbit", "Hamster")

console.log(more_pets)
```

forEach

The `forEach` method in JavaScript is another way of achieving functionality similar to the PHP `foreach` keyword. To use it, you pass it the name of a function, which will be called for each element within the array. [Example 15-12](#) shows how.

Example 15-12. Using the `forEach` method

```
<script>
  pets = ["Cat", "Dog", "Rabbit", "Hamster"]
  pets.forEach(output)

  function output(element, index)
  {
    console.log("Element at index " + index + " has the value " + element)
  }
</script>
```


In this case, the function passed to `forEach` is called `output`. It takes two parameters: the `element` and its `index`. The function may also take a third parameter, `array`, which contains the array `forEach` was called upon, but I've omitted it as it is unused in this example. All these parameters can be used as required by your function. This example uses and displays just the `element` and `index` values using the function `console.log`.

Once an array has been populated, the method is called, like this:

```
pets.forEach(output)
```

This is the output:

```
Element at index 0 has the value Cat
Element at index 1 has the value Dog
Element at index 2 has the value Rabbit
Element at index 3 has the value Hamster
```

join

With the `join` method, you can convert all the values in an array to strings and then join them together into one large string, placing an optional separator between them.

[Example 15-13](#) shows three ways of using this method.

Example 15-13. Using the `join` method

```
<script>
  pets = ["Cat", "Dog", "Rabbit", "Hamster"]

  console.log(pets.join())
  console.log(pets.join(' '))
  console.log(pets.join(' : '))
</script>
```

Without a parameter, `join` uses a comma to separate the elements; otherwise, the string passed to `join` is inserted between each element. The output of [Example 15-13](#) looks like this:

```
Cat,Dog,Rabbit,Hamster
Cat Dog Rabbit Hamster
Cat : Dog : Rabbit : Hamster
```

push and pop

You already saw how the `push` method can be used to insert a value into an array. The inverse method is `pop`. It removes the last element from an array and returns it.

[Example 15-14](#) shows an example of its use.

Example 15-14. Using the push and pop methods

```
<script>
  sports = ["Football", "Tennis", "Baseball"]
  console.log("Start = " + sports)

  sports.push("Hockey")
  console.log("After Push = " + sports)

  removed = sports.pop()
  console.log("After Pop = " + sports)
  console.log("Removed = " + removed)

  removed = sports.pop()
  console.log("After Pop = " + sports)
  console.log("Removed = " + removed)
</script>
```

The four main statements of this script are shown in bold type. First, the script creates an array called `sports` with three elements and then pushes a fourth element into the array. After that, it pops that element back off, and then it pops once more. In the process, the various current values are displayed via `console.log`. The script outputs the following:

```
Start = Football,Tennis,Baseball
After Push = Football,Tennis,Baseball,Hockey
After Pop = Football,Tennis,Baseball
Removed = Hockey
After Pop = Football,Tennis
Removed = Baseball
```

The push and pop functions are useful in situations where you need to divert from some activity to do another and then return. For example, let's suppose you want to put off some activities until later, while you get on with something more important now. This often happens in real life when we're going through "to-do" lists, so let's emulate that in code, with tasks number 2 and 5 in a list of six items being granted priority status, as in [Example 15-15](#).

Example 15-15. Using push and pop inside and outside of a loop

```
<script>
  const numbers = []

  for (j = 1 ; j < 6 ; ++j) {
    if (j === 2 || j === 5) {
      console.log("Processing 'todo' #" + j)
    }
    else {
      console.log("Putting off 'todo' #" + j + " until later")
    }
  }
</script>
```

```

        numbers.push(j)
    }
}

console.log("Finished processing the priority tasks.")
console.log("Commencing stored tasks, most recent first.")

console.log("Now processing 'todo' #" + numbers.pop())
console.log("Now processing 'todo' #" + numbers.pop())
console.log("Now processing 'todo' #" + numbers.pop())
</script>

```

Of course, nothing is actually getting processed here, just text being output to the browser, but you get the idea. The output from this example is:

```

Putting off 'todo' #1 until later
Processing 'todo' #2
Putting off 'todo' #3 until later
Putting off 'todo' #4 until later
Processing 'todo' #5
Finished processing the priority tasks.
Commencing stored tasks, most recent first.
Now processing 'todo' #4
Now processing 'todo' #3
Now processing 'todo' #1

```

Using reverse

The reverse method simply reverses the order of all elements in an array. [Example 15-16](#) shows this in action.

Example 15-16. Using the reverse method

```

<script>
    sports = ["Football", "Tennis", "Baseball", "Hockey"]
    sports.reverse()
    console.log(sports)
</script>

```

The original array is modified, and the output from this script is:

```

Hockey, Baseball, Tennis, Football

```

sort

With the sort method, you can place all the elements of an array in alphabetical order, depending on the parameters used. [Example 15-17](#) shows four types of sort.

Example 15-17. Using the sort method

```
<script>
  // Alphabetical sort
  sports = ["Football", "Tennis", "Baseball", "Hockey"]
  sports.sort()
  console.log(sports)

  // Reverse alphabetical sort
  sports = ["Football", "Tennis", "Baseball", "Hockey"]
  sports.sort().reverse()
  console.log(sports)
</script>
```

The first of the two example sections uses the default `sort` method to perform an *alphabetical sort*, while the second uses the default `sort` and then applies the `reverse` method to get a *reverse alphabetical sort*.

Anonymous Functions

Some functions do not need a name, because they're used only once, for example, and coming up with a name would be a waste of time. A function without a name, called an *anonymous function*, can be used, for example, as a comparison function for sorting arrays with the `sort` method you've already seen in [Example 15-17](#). Let's take that example but sort the array using a numeric sort this time, which requires an anonymous comparison function. The code is in [Example 15-18](#).

Example 15-18. Using the sort method with an anonymous function

```
<script>
  // Ascending numeric sort
  numbers = [7, 23, 6, 74]
  numbers.sort(function(a,b){return a - b})
  console.log(numbers)

  // Descending numeric sort
  numbers = [7, 23, 6, 74]
  numbers.sort(function(a,b){return b - a})
  console.log(numbers)
</script>
```

In both cases, the `numbers.sort` call here uses a function to compare the relationships between `a` and `b`. The function doesn't have a name, because it's used only in the `sort`. You have already seen the function named `function` used to create an anonymous function; we used it to define a method in a class (the `showUser` method).

Here, `function` creates an anonymous function meeting the needs of the `sort` method. If the function returns a value greater than zero, the sort assumes that `b` comes before `a`. If the function returns a value less than zero, the sort assumes that `a` comes before `b`. The sort runs this function across all the values in the array to determine their order. (Of course, if `a` and `b` have the same value, the function returns zero, and it doesn't matter which value is first.)

By manipulating the value returned (`a - b` in contrast to `b - a`), the two sections of [Example 15-18](#) choose between an *ascending numerical sort* and a *descending numerical sort*.

Arrow Functions

The *arrow function* syntax is a simplified version of the general anonymous function syntax you have just seen. The syntax is very common because it's short and concise. In the following example, we'll convert the general anonymous function to use the arrow function syntax. The line with the anonymous function in [Example 15-18](#) looks like this:

```
numbers.sort(function(a,b){return a - b})
```

You can remove the `function` keyword as well as the `return` keyword and remove the body braces, like this:

```
numbers.sort((a,b) => a - b)
```

If the arrow function would have exactly one parameter, you could also omit the parameter parentheses, but that's not the case here; the function has two parameters, `a` and `b`.

And, believe it or not, this represents the end of your introduction to JavaScript. You should now have a core knowledge of the three main technologies covered in this book. [Chapter 16](#) will look at some advanced techniques used across these technologies, such as pattern matching and input validation. But before we continue, let's try to answer the following questions to repeat the main things you've learned in this chapter.

Questions

1. Are JavaScript functions and variable names case-sensitive or case-insensitive?
2. How can you write a function that accepts and processes an arbitrary number of parameters?
3. Describe a way to return multiple values from a function.
4. When you're defining a class, what keyword do you use to refer to the current object?

5. Do all the methods of a class have to be defined within the class definition?
6. What keyword is used to create an object instance from a class?
7. How can you create a multidimensional array?
8. What syntax is used to create an “associative array”?
9. Write a statement to sort an array of numbers in descending numerical order.

See “[Chapter 15 Answers](#)” on page 579 in the [Appendix](#) for the answers to these questions.

JavaScript and PHP Validation and Error Handling

With your solid foundation in both PHP and JavaScript, it's time to bring these technologies together to create web forms that are as user-friendly as possible.

We'll be using PHP to create the forms and JavaScript to perform client-side validation to ensure that the data is as complete and correct as it can be before it is submitted. Final validation of the input will then be done by PHP, which will, if necessary, present the form again to the user for further modification.

In the process, this chapter will cover validation and regular expressions in both JavaScript and PHP.

Validating User Input with JavaScript

JavaScript validation should be considered assistance to your users more than to your websites because, as I have stressed many times, you cannot trust any data submitted to your server, even if it has supposedly been validated with JavaScript. Hackers can quite easily simulate your web forms and submit any data of their choosing.

Another reason you cannot rely on JavaScript to perform all your input validation is that some users disable JavaScript or use browsers that don't support it.

So, the best types of validation to do in JavaScript are checking that fields have content if they are not to be left empty, ensuring that email addresses conform to the proper format, and ensuring that values entered are within expected bounds.

The validate.html Document (Part 1)

To keep the code listing easier to follow, this example is split into two parts: the main HTML and associated JavaScript, and the JavaScript functions that get called by the main part. Let's begin with a general signup form, common on most sites that offer user registration. The inputs requested will be *forename*, *surname*, *username*, *password*, *age*, and *email address*. [Example 16-1](#) provides a good template for such a form, which you can retrieve from the [repo of examples on GitHub](#). Ensure it is saved as *validate.html*.

Example 16-1. A form with JavaScript validation (part 1)

```
<!DOCTYPE html>
<html>
  <head>
    <title>An Example Form</title>
    <style>
      .signup {
        border: 1px solid #999999;
        font: normal 14px helvetica;
        color: #444444;
        background-color: #eeeeee;
        border-spacing: 5px;
      }
      .signup th, .signup td {
        padding: 2px;
      }
    </style>
  </head>
  <body>
    <form method="post" action="" id="form">
      <table class="signup">
        <th colspan="2" align="center">Signup Form</th>
        <tr><td>Forename</td>
          <td><input type="text" maxlength="32" name="forename" required></td></tr>
        <tr><td>Surname</td>
          <td><input type="text" maxlength="32" name="surname" required></td></tr>
        <tr><td>Username</td>
          <td><input type="text" maxlength="16" name="username" required></td></tr>
        <tr><td>Password</td>
          <td><input type="password" name="password" required></td></tr>
        <tr><td>Age</td>
          <td><input type="number" max="110" name="age" required></td></tr>
        <tr><td>Email</td>
          <td><input type="email" maxlength="64" name="email" required></td></tr>
        <tr><td colspan="2" align="center"><input type="submit"
          value="Signup"></td></tr>
      </table>
    </form>
  </script>
```



```

function validateFields(form)
{
    const errors = []
    const elements = {}
    let error = ''

    for (let element of form.elements)
        elements[element.name] = element.value.trim()

    error = validateForename(elements.forename)
    if (error) errors.push({field: 'forename', message: error})
    error = validateSurname(elements.surname)
    if (error) errors.push({field: 'surname', message: error})
    error = validateUsername(elements.username)
    if (error) errors.push({field: 'username', message: error})
    error = validatePassword(elements.password)
    if (error) errors.push({field: 'password', message: error})
    error = validateAge(elements.age)
    if (error) errors.push({field: 'age', message: error})
    error = validateEmail(elements.email)
    if (error) errors.push({field: 'email', message: error})
    return errors
}

const validate = function(event)
{
    const errors = validateFields(event.target)
    if (errors.length) {
        const alerts = []
        for (error of errors) {
            alerts.push(error.field + ": " + error.message)
        }
        alert(alerts.join("\n"))
        event.preventDefault()
    }
}

document.getElementById('form').addEventListener('submit', validate)
</script>
</body>
</html>

```

As it stands, this form will display correctly but will not yet have any JavaScript-based validation, because the main validation functions have not been added. The HTML attributes like `required`, `type="number"` and `type="email"` will provide at least some built-in validation. Even so, save it as *validate.html*, and when you call it up in your browser, it will look like [Figure 16-1](#).

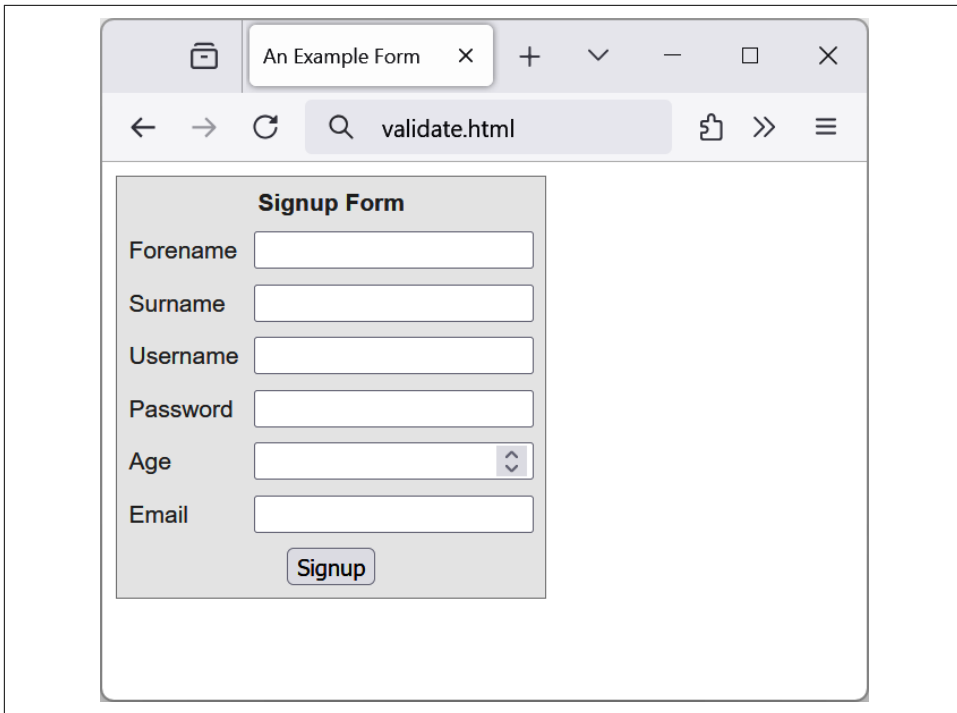


Figure 16-1. The output from *Example 16-1*

Let's look at how this document is made up. The first few lines set up the document and use a little CSS to make the form look a little less plain. The parts of the document related to JavaScript come next and are shown in bold.

The first part of this example features the HTML for the form, with each field and its name placed within its own row of a table. This is pretty straightforward HTML: it uses correct input types for the age and email fields and adds the `required` attribute to provide some built-in validation.

Between the `<script>` and `</script>` tags lies a function called `validateFields` that first removes all leading and trailing spaces from all the submitted values by calling the `trim` method, and it stores the result in the `elements` "associative array" (in quotes because it's actually an object). Then it calls up six other functions to validate each of the form's input fields. We'll get to these functions shortly. For now I'll just explain that they return either an empty string if a field validates or an error message if it fails. If an error message is returned, it is added to the `errors` array as an object together with the input field name. The function then returns the `errors` array.

The `validateFields` function is then called in an anonymous function that, if any errors were returned, formats them as "field name: error message", stores them

in the `alerts` array, and eventually uses the `alert` function to show the error messages to the user, one per line. You could further enhance the anonymous function to, for example, display the error messages below the respective form fields; the `validateFields` function returns all the information you'd need.

The following line then attaches the anonymous function stored in `validate` as an event handler (or *listener*) that's called when the form is submitted:

```
document.getElementById('form').addEventListener('submit', validate)
```

The anonymous function receives a parameter I've called `event`, which references the form being submitted in `event.target`. If the `validateFields` function returned some errors in the array, besides alerting the messages, we'll prevent the form values from being submitted to the server by calling the following method on the event object:

```
event.preventDefault()
```

If this were omitted, the browser would display the error messages, but then upon closing, the alert pop-up would still submit the form, so the user would have no chance to correct the form data.

As you can see, there's no JavaScript used within the form's HTML. Browsers with JavaScript disabled or not available will display the form just fine.

The `validate.html` Document (Part 2)

Now we come to [Example 16-2](#), a set of six functions that do the actual form-field validation. I suggest that you type all of this second part and save it in the `<script>...</script>` section of [Example 16-1](#), which you have saved as `validate.html`.

Example 16-2. A form with JavaScript validation (part 2)

```
function validateForename(field)
{
    return (field === "") ? "No Forename was entered." : ""
}

function validateSurname(field)
{
    return (field === "") ? "No Surname was entered." : ""
}

function validateUsername(field)
{
    if (field == "")
        return "No Username was entered."
    else if (field.length < 5)
```

```

        return "Usernames must be at least 5 characters."
    else if (/^[a-zA-Z0-9_-]/.test(field))
        return "Only a-z, A-Z, 0-9, - and _ allowed in Usernames."
    return ""
}

function validatePassword(field)
{
    if (field == "")
        return "No Password was entered."
    else if (field.length < 6)
        return "Passwords must be at least 6 characters."
    else if (!/[a-z]/.test(field) || !/[A-Z]/.test(field) ||
        !/[0-9]/.test(field))
        return "Passwords require one each of a-z, A-Z and 0-9."
    return ""
}

function validateAge(field)
{
    if (field == "" || isNaN(field))
        return "No Age was entered."
    else if (field < 18 || field > 110)
        return "Age must be between 18 and 110."
    return ""
}

function validateEmail(field)
{
    return (field === "") ? "No Email was entered." : ""
}

```

We'll go through each of these functions in turn, starting with `validateForename`, so you can see how validation works.

Validating the forename

`validateForename` is a short function that accepts the parameter `field`, which is the value of the forename passed to it by the `validate` function.

If this value is the empty string, an error message is returned; otherwise, an empty string is returned to signify that no error was encountered.

Remember that spaces are already trimmed in `validateFields` so even if the user tried to submit a string with leading or trailing spaces, they would be removed before calling `validateForename`, and the value would be passed to it as an empty string.

Basic check for an empty string is already done by the browser as instructed by the required HTML attribute.

Validating the surname

The `validateSurname` function is almost identical to `validateForename` in that an error is returned only if the surname supplied was an empty string. I chose not to limit the characters allowed in either of the name fields to allow for possibilities such as non-English and accented characters.

Validating the username

The `validateUsername` function is a little more interesting, because it has a more complicated job. It can allow through only the characters a-z, A-Z, 0-9, `_`, and `-`, and it ensures that usernames are at least five characters long.

The `if...else` statements commence by returning an error if `field` has not been filled in. If it's not the empty string but is fewer than five characters in length, another error message is returned. This also can be done by adding `minlength="6"` attribute to the input's HTML instead.

Then the JavaScript test method is called, passing a regular expression (which matches any character that is *not* one of those allowed) to be matched against `field` (see “[Regular Expressions](#)” on page 383). If even one character that isn't one of the acceptable characters is encountered, the `test` function returns `true`, and so `validateUser` returns an error string. A `pattern` HTML attribute can be added to the input, which can do the same check in pure HTML, but its error message may not be clear to some users.

Validating the password

Similar techniques are used in the `validatePassword` function. First the function checks whether `field` is empty, and if it is, it returns an error. Next, an error message is returned if the password is shorter than six characters.

One of the requirements we're imposing on passwords is that they must have at least one each of a lowercase, uppercase, and numerical character, so the `test` function is called three times, once for each of these cases. If any one of these calls returns `false`, one of the requirements was not met, and so an error message is returned. Otherwise, the empty string is returned to signify that the password was OK.

The same HTML attributes (`minlength`, `pattern`) can also be used in this case.

Validating the age

`validateAge` returns an error message if `field` is not a number (determined by a call to the `isNaN` function) or if the age entered is lower than 18 or greater than 110. Your applications may well have different or no age requirements. Again, upon successful validation, the empty string is returned.

Entering numbers and the range is also enforced by the browser itself using the `type="number"` and `max="110"` HTML attributes. You could also add `min="18"` but you'd need to explain the allowed range directly in the form, which I have not done here for brevity; otherwise, users would be wondering why they cannot enter numbers less than *min*. If adding HTML attributes, make sure the range is the same as the range checked in JavaScript, similar to other attributes like `minlength` for example.

Validating the email

In the last example, the value is validated with `validateEmail`. Validating email addresses can be a difficult task: did you, for example, know that the plus sign (+) is a valid character of the username part? Addresses like `foo+bar@example.com` (and more) are valid email addresses, and it's not recommended you try to validate them with your own code. We'll check only for empty strings but leave the address format check on the browser's built-in validation of the `<input type="email">` field.

Figure 16-2 shows the result of the user clicking the Signup button without having completed any fields.

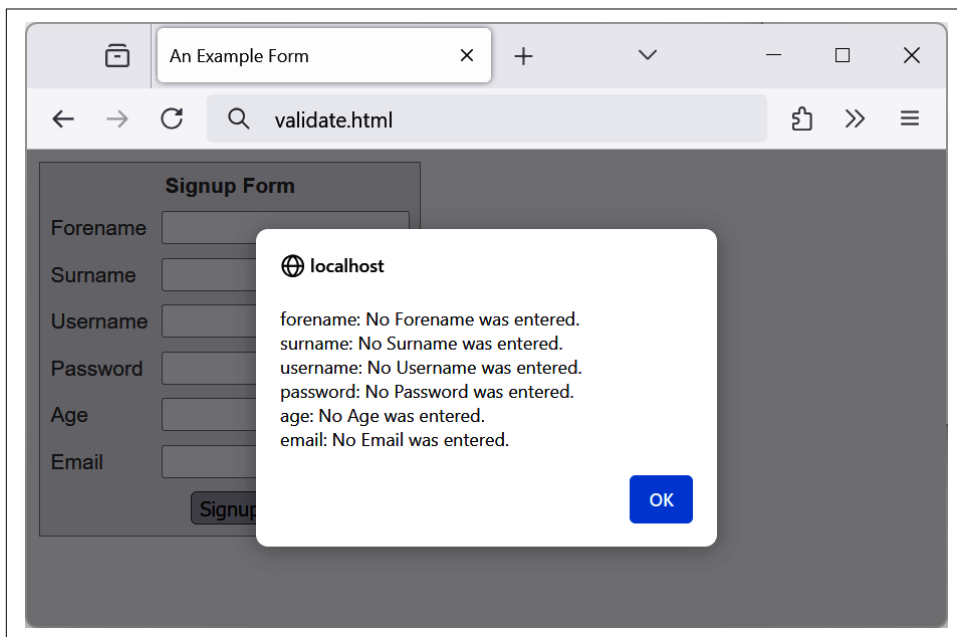


Figure 16-2. JavaScript form validation in action

Using a separate JavaScript file

Of course, because they are generic in construction and could apply to many types of validations you might require, these six functions make ideal candidates for moving out into a separate JavaScript file. You could name the file something like *validate_functions.js* and include it right after the initial script section in [Example 16-1](#), using the following statement:

```
<script src="validate_functions.js"></script>
```

Regular Expressions

Let's look a little more closely at the pattern matching we have been doing. We've achieved it using *regular expressions*, which are supported by both JavaScript and PHP. They make it possible to construct powerful pattern-matching algorithms within a single expression.

Matching Through Metacharacters

Regular expressions normally must be enclosed in slashes. Within these slashes, certain characters have special meanings; they are called *metacharacters*. For instance, an asterisk (*) has a meaning similar to what you have seen if you've used a shell or Windows command prompt (but not quite the same). An asterisk means "The text you're trying to match may have any number of the preceding characters—or none at all."

For instance, let's say you're looking for the name *Le Guin* and know that someone might spell it with or without a space. Because the text is laid out strangely (for instance, someone may have inserted extra spaces to right-justify lines), you could have to search for a line such as this:

```
The    difficulty    of    classifying    Le        Guin's    works
```

So you need to match *LeGuin*, as well as *Le* and *Guin* separated by any number of spaces. The solution is to follow a space with an asterisk:

```
/Le *Guin/
```

There's a lot more than the name *Le Guin* in the line, but that's OK. As long as the regular expression matches some part of the line, the test function returns a true value. What if it's important to make sure the line contains nothing but *Le Guin*? I'll show you how to ensure that later.

Suppose that you know there is always at least one space. In that case, you could use the plus sign (+), because it requires at least one of the preceding expressions to be present:

```
/Le +Guin/
```

Wildcard Matching

The dot (.) is particularly useful, because it can match anything except a newline. Suppose you are looking for HTML tags, something that generally shouldn't be done with regular expressions unless you want to do a quick and naive "pseudo-parsing" of HTML. Or unless, of course, you want to learn regular expressions, as HTML provides a lot of opportunities to showcase the usage.



Do Not Parse HTML with Regular Expressions

You can almost always construct valid HTML that will defeat your regular expression because regular expressions in general are not sufficient to completely parse HTML. This answer to a [question posted on StackOverflow](#) explains the details.

To parse HTML, you should use a full-featured parser like the one available in the `loadHTML` method of PHP's `DOMDocument` class.

HTML tags start with `<` and end with `>`. A simple way to find them is shown here:

```
/<.*>/
```

The dot matches any character, and the `*` expands it to match zero or more characters, so this is saying, "Match anything that lies between `<` and `>`, even if there's nothing." You will match `<>`, `<em>`, `<br>`, and so on. But if you don't want to match the empty case, `<>`, you should use `+` instead of `*`, like this:

```
/<.+>/
```

The plus sign expands the dot to match one or more characters, saying, "Match anything that lies between `<` and `>` as long as there's at least one character between them." You will match `<em>` and `</em>`, `<h1>` and `</h1>`, and tags with attributes, such as:

```
<a href="www.mozilla.org">
```

Unfortunately, the plus sign keeps on matching the last `>` on the line, so you might end up with this:

```
<h1><b>Introduction</b></h1>
```

That's a lot more than one tag! I'll show a better solution later in this section.



If you use the dot on its own between the angle brackets, without following it with either a `+` or `*`, then it matches a single character; you will match `<b>` and `<i>` but *not* `<em>` or `<textarea>`.

If you want to match the dot character itself (.), you have to escape it by placing a backslash (\) before it, because otherwise it's a metacharacter and matches anything. For example, suppose you want to match the floating-point number 5.0. The regular expression is:

```
/5\.0/
```

The backslash can escape any metacharacter, including another backslash (in case you're trying to match a backslash in text). However, you'll see later how backslashes sometimes give the following character a special meaning, which can be a bit confusing.

We just matched a floating-point number. But perhaps you want to match 5. as well as 5.0, because both mean the same thing as a floating-point number. You also want to match 5.00, 5.000, and so forth—any number of zeros is allowed. You can do this by adding an asterisk, as you've seen:

```
/5\.0*/
```

Grouping Through Parentheses

Suppose you want to match powers of increments of units, such as kilo, mega, giga, and tera. In other words, you want all the following to match:

```
1,000
1,000,000
1,000,000,000
1,000,000,000,000
...
```

The plus sign works here, too, but you need to group the string ,000 so the plus sign matches the whole thing. The regular expression is:

```
/1(,000)+ /
```

The parentheses mean “Treat this as a group when you apply something such as a plus sign.” Strings like 1,00,000 and 1,000,00 won't match because the text must have a 1 followed by one or more complete groups of a comma followed by three zeros.

The space after the + character indicates that the match must end when a space is encountered. Without it, 1,000,00 would incorrectly match because only the first 1,000 would be taken into account, and the remaining ,00 would be ignored. Requiring a space afterward ensures that matching will continue right through to the end of a number. If the number can also be followed by a full stop (.), not just by a space, like for example at the end of the sentence, you'd need to allow the full stop to also end the search using character classes.

Character Classes

Sometimes you want to match something fuzzy but not so broadly that you want to use a dot. Fuzziness is the great strength of regular expressions: they allow you to be as precise or vague as you want.

One of the key features supporting fuzzy matching is the pair of square brackets, `[]`. It matches a single character, like a dot, but inside the brackets you put a list of things that can match. If any of those characters appears, the text matches. For instance, if you wanted to match both the American spelling *gray* and the British spelling *grey*, you could specify:

```
/gr[ae]y/
```

After the `gr` in the text you're matching, there can be either an `a` or an `e`. But there must be only one of them: whatever you put inside the brackets matches exactly one character. The group of characters inside the brackets is called a *character class*.

Indicating a Range

Inside the brackets, you can use a hyphen (`-`) to indicate a range. One very common task is matching a single digit, which you can do with a range, as follows:

```
/[0-9]/
```

Digits are such a common item in regular expressions that a single character is provided to represent them: `\d`. You can use it in place of the bracketed regular expression to match a digit:

```
/\d/
```

Negation

One other important feature of the square brackets is *negation* of a character class. You can turn the whole character class on its head by placing a caret (`^`) after the opening bracket. Here it means “Match any characters *except* the following.” So let's say you want to find instances of *Yahoo* that lack the following exclamation point. (The name of the company officially contains an exclamation point!) You could do it like this:

```
/Yahoo[^!]/
```

The character class consists of a single character—an exclamation point—but it is inverted by the preceding `^`. This is actually not a great solution to the problem—for instance, it fails if *Yahoo* is at the end of the line, because then it's not followed by *anything*, whereas the brackets must match a character. A better solution involves negative *lookahead* (matching something that is not followed by anything else), but

that’s beyond the scope of this book, so please refer to the [Regular Expressions website](#), which shows how to apply negative lookahead for a regex in any language.

Some More Complicated Examples

With an understanding of character classes and negation, you’re ready to see a better solution to the problem of matching an HTML tag. This solution avoids going past the end of a single tag but still matches tags such as `<em>` and `</em>` as well as tags with attributes such as:

```
<a href="www.mozilla.org">
```

Here is one solution:

```
/<[^>]+>/
```

That regular expression may look like a cat just sauntered across the keyboard, but it is perfectly valid and very useful. Let’s break it apart. [Figure 16-3](#) shows the various elements, which I’ll describe one by one.

/	<	[^>]	+	>	/
Opening slash: Indicates a regular expression	Opening bracket of HTML tag: Matched exactly	Character class: Match anything except a closing angle bracket	Metacharacter: Any # of characters can match the <code>[^>]</code>	Closing bracket of HTML tag: Matched exactly	Closing slash: Indicates end of regular expression

Figure 16-3. Breakdown of a typical regular expression

The elements are:

/

Opening slash that indicates this is a regular expression.

<

Opening bracket of an HTML tag. This is matched exactly; it’s not a metacharacter.

[^>]

Character class. The embedded `^>` means “Match anything except a closing angle bracket.”

+

Allows any number of characters to match the previous `[^>]`, as long as there is at least one of them.

>

Closing bracket of an HTML tag. This is matched exactly.

/

Closing slash that indicates the end of the regular expression.



Another solution to the problem of matching HTML tags is to use a nongreedy operation. By default, pattern matching is greedy, returning the longest match possible. Nongreedy (or lazy) matching finds the shortest possible match. Its use is beyond the scope of this book, but there are more details on the [JavaScript.info website](https://www.javascript.info).

We'll look now at one of the expressions from [Example 16-1](#), where the `validateUsername` function is used:

```
/[^a-zA-Z0-9_-]/
```

[Figure 16-4](#) shows the various elements.

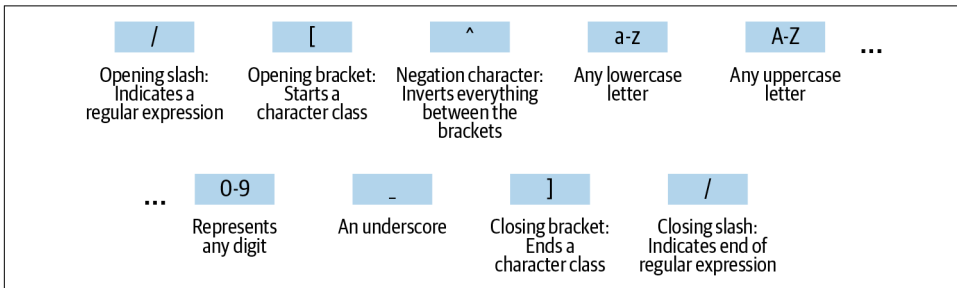


Figure 16-4. Breakdown of the `validateUsername` regular expression

Let's look at these elements in detail:

/

Opening slash that indicates this is a regular expression.

[

Opening bracket that starts a character class.

^

Negation character: inverts everything else between the brackets.

a-z

Represents any lowercase letter.

A-Z

Represents any uppercase letter.

0-9

Represents any digit.

- An underscore.
- A dash.
-]
- Closing bracket that ends a character class.
- /
- Closing slash that indicates the end of the regular expression.

Two other important metacharacters “anchor” a regular expression by requiring that it appear in a particular place. If a caret (^) appears at the beginning of the regular expression, the expression has to appear at the beginning of a line of text; otherwise, it doesn’t match. Similarly, if a dollar sign (\$) appears at the end of the regular expression, the expression has to appear at the end of a line of text.



It may be somewhat confusing that ^ can mean “negate the character class” inside square brackets and “match the beginning of the line” if it’s at the beginning of the regular expression. Unfortunately, the same character is used for two different purposes, so take care when using it.

We’ll finish our exploration of regular expression basics by answering a question raised earlier: suppose you want to make sure there is nothing extra on a line besides the regular expression? What if you want a line that has “Le Guin” and nothing else? We can do that by amending the earlier regular expression to anchor the two ends:

```
/^Le *Guin$/
```

Summary of Metacharacters

Table 16-1 shows the metacharacters available in regular expressions.

Table 16-1. Regular expression metacharacters

Metacharacters	Description
/	Begins and ends the regular expression
.	Matches any single character except the newline
<i>element</i> *	Matches <i>element</i> zero or more times
<i>element</i> +	Matches <i>element</i> one or more times
<i>element</i> ?	Matches <i>element</i> zero or one times
[<i>characters</i>]	Matches a character out of those contained within the brackets
[^ <i>characters</i>]	Matches a single character that is not contained within the brackets

Metacharacters	Description
<code>(regex)</code>	Treats the <i>regex</i> as a group for counting or a following <code>*</code> , <code>+</code> , or <code>?</code>
<code>left right</code>	Matches either <i>left</i> or <i>right</i>
<code>[l-r]</code>	Matches a range of characters between <i>l</i> and <i>r</i>
<code>^</code>	Requires match to be at the string's start
<code>\$</code>	Requires match to be at the string's end
<code>\b</code>	Matches a word boundary
<code>\B</code>	Matches where there is not a word boundary
<code>\d</code>	Matches a single digit
<code>\D</code>	Matches a single nondigit
<code>\n</code>	Matches a newline character
<code>\s</code>	Matches a whitespace character
<code>\S</code>	Matches a nonwhitespace character
<code>\t</code>	Matches a tab character
<code>\w</code>	Matches a word character (a-z, A-Z, 0-9, and <code>_</code>)
<code>\W</code>	Matches a nonword character (anything but a-z, A-Z, 0-9, and <code>_</code>)
<code>\x</code>	Matches <i>x</i> (useful if <i>x</i> is a metacharacter, but you really want <i>x</i>)
<code>{n}</code>	Matches exactly <i>n</i> times
<code>{n,}</code>	Matches <i>n</i> times or more
<code>{min,max}</code>	Matches at least <i>min</i> and at most <i>max</i> times

Provided with this table, and looking again at the expression `/[^a-zA-Z0-9_]/`, you can see that it could easily be shortened to `/[^\\w]/` because the single metacharacter `\\w` (with a lowercase *w*) specifies the characters a-z, A-Z, 0-9, and `_`.

In fact, we can be even more clever than that, because the metacharacter `\\W` (with an uppercase *W*) specifies all characters *except* for a-z, A-Z, 0-9, and `_`. Therefore, we could also drop the `^` metacharacter and simply use `/[\\W]/` for the expression, or go a step further and drop the square brackets, as in `/\\W/`, because it's a single character.

To give you more ideas of how this all works, [Table 16-2](#) shows a range of expressions and the patterns they match.

Table 16-2. Some example regular expressions

Example	Matches
<code>r</code>	The first <i>r</i> in <i>The quick brown</i>
<code>rec[ei][ei]ve</code>	Either of <i>receive</i> or <i>recieve</i> (but also <i>receeve</i> or <i>reciive</i>)
<code>rec[ei]{2}ve</code>	Either of <i>receive</i> or <i>recieve</i> (but also <i>receeve</i> or <i>reciive</i>)
<code>rec(ei ie)ve</code>	Either of <i>receive</i> or <i>recieve</i> (but not <i>receeve</i> or <i>reciive</i>)
<code>cat</code>	The word <i>cat</i> in <i>I like cats and dogs</i>
<code>cat dog</code>	The word <i>cat</i> in <i>I like cats and dogs</i> (matches either <i>cat</i> or <i>dog</i> , whichever is encountered first)
<code>\\.</code>	<code>.</code> (the <code>\\</code> is necessary because <code>.</code> is a metacharacter)

Example	Matches
<code>5\\.0*</code>	<i>5., 5.0, 5.00, 5.000, etc.</i>
<code>[a-f]</code>	Any of the characters <i>a, b, c, d, e, or f</i>
<code>cats\$</code>	Only the final <i>cats</i> in <i>My cats are friendly cats</i>
<code>^my</code>	Only the first <i>my</i> in <i>my cats are my pets</i>
<code>\\d{2,3}</code>	Any two- or three-digit number (<i>00 through 999</i>)
<code>7(,000)+</code>	<i>7,000; 7,000,000; 7,000,000,000; 7,000,000,000,000; etc.</i>
<code>[\\w]+</code>	Any word of one or more characters
<code>[\\w]{5}</code>	Any five-letter word

General Modifiers

Some additional modifiers are available for regular expressions:

- `/g` enables *global* matching. When using a replace function, specify this modifier to replace all matches, rather than only the first one.
- `/i` makes the regular expression match case-insensitive. Thus, instead of `/[a-zA-Z]/`, you could specify `/[a-z]/i` or `/[A-Z]/i`.
- `/m` enables multiline mode, in which the caret (^) and dollar sign (\$) match before and after any newlines in the subject string. Normally, in a multiline string, ^ matches only at the start of the string, and \$ matches only at the end of the string.

For example, the expression `/cats/g` will match both occurrences of the word *cats* in the sentence “I like cats, and cats like me.” Similarly, `/dogs/gi` will match both occurrences of the word *dogs* (*Dogs* and *dogs*) in the sentence “Dogs like other dogs,” because you can use these specifiers together.

Using Regular Expressions in JavaScript

In JavaScript, you will use regular expressions mostly in three methods: `test` (which you have already seen), `match`, and `replace`. Whereas `test` just tells you whether its argument matches the regular expression, the `match` method returns the result of matching a string against a regular expression passed as an argument, or `null` if no match is found.

The following line will try to find whether the sky was cloudy using both possible spellings of the color:

```
console.log("The sky was gray".match(/gr[ae]y/))
```

When using `match`, you can also pass the regular expression as a string enclosed in quotes, not in slashes:

```
console.log("The sky was grey".match("gr[ae]y"))
```

The `replace` method takes a second parameter: the string to replace the text that matches. Like most functions, `replace` generates a new string as a return value; it does not change the input.

To compare the `test` and `replace` methods, the following statement just returns `true` to let us know that the word *cats* appears at least once somewhere within the string:

```
console.log(/cats/i.test("Cats are funny. I like cats."))
```

But the following statement replaces both occurrences of the word *cats* with the word *dogs*, printing the result. The search has to be global (`/g`) to find all occurrences and case-insensitive (`/i`) to find the capitalized *Cats*:

```
console.log("Cats are friendly. I like cats.".replace(/cats/gi,"dogs"))
```

If you try out the statement, you'll see a limitation of `replace`: because it replaces text with exactly the string you tell it to use, the first word *Cats* is replaced by *dogs* instead of *Dogs*.

Using Regular Expressions in PHP

The most common regular expression functions that you are likely to use in PHP are `preg_match`, `preg_match_all`, and `preg_replace`.

To test whether the word *cats* appears anywhere within a string, in any combination of upper- and lowercase, you could use `preg_match` like this:

```
$n = preg_match("/cats/i", "Cats are crazy. I like cats.");
```

The function returns `1` if the match was found, `0` if it wasn't, and `FALSE` on failure. Because the word *cats* is in the tested string, the preceding statement sets `$n` to `1`. The first argument is the regular expression, and the second is the text to match. But `preg_match` is actually a good deal more powerful and complicated, because it takes a third argument that shows what text matched:

```
$n = preg_match("/cats/i", "Cats are curious. I like cats.", $match);  
echo "$n Matches: $match[0]";
```

The third argument is an array (here, given the name `$match`). The function puts the matching text into the first element, so if the match is successful, you can find the text that matched in `$match[0]`. In this example, the output lets us know that the matched text was capitalized:

```
1 Matches: Cats
```


If you wish to locate all matches, you use the `preg_match_all` function, like this:

```
$n = preg_match_all("/cats/i", "Cats are strange. I like cats.", $match);  
echo "$n Matches: ";  
for ($j=0 ; $j < $n ; ++$j) echo $match[0][$j]. " ";
```

As before, `$match` is passed to the function, and the element `$match[0]` is assigned the matches made but this time as a subarray. To display the subarray, this example iterates through it with a `for` loop.

When you want to replace part of a string, you can use `preg_replace`, as shown here. This example replaces all occurrences of the word *cats* with the word *dogs*, regardless of case:

```
echo preg_replace("/cats/i", "dogs", "Cats are furry. I like cats.");
```



The subject of regular expressions is a large one, and entire books have been written about it. If you would like more information, I suggest the [Wikipedia entry](#) or [Regular-Expressions.info](#), and I would also recommend the [MDN documentation](#). Also remember that while regular expressions are a useful tool, they should not be used as a general solution to all string-related problems.

Redisplaying a Form After PHP Validation

OK, back to form validation. So far we've created the HTML document *validate.html*, which will post through to the PHP program *adduser.php*, but only if JavaScript validates the fields or if JavaScript is disabled or unavailable.

So now it's time to add PHP code to do its own validation and then present the form again to the visitor if the validation fails. There's no need to create the form HTML again; you're supposed to rename (or copy) the *validate.html* file to *adduser.php* and continue in that file.

Example 16-3 contains the code that you should type and save (or download from the [GitHub repository](#)); the part with the bold typeface is the PHP we're adding. The form toward the end of the file is almost the same—only the error message output and the `value` attributes have been added.



The reason for first creating *validate.html* and then renaming it to *adduser.php* is to have two different filenames for each of the stages in case you downloaded the files from the [GitHub repository](#).

Example 16-3. The `adduser.php` program

```
<?php // adduser.php

// The PHP functions

function validate_forename($field)
{
    return ($field == '') ? 'No Forename was entered': '';
}

function validate_surname($field)
{
    return($field == '') ? 'No Surname was entered' : '';
}

function validate_username($field)
{
    if ($field == '')
        return 'No Username was entered';
    else if (strlen($field) < 5)
        return 'Usernames must be at least 5 characters';
    else if (preg_match('/^[a-zA-Z0-9_-]/', $field))
        return 'Only letters, numbers, - and _ in usernames';
    return '';
}

function validate_password($field)
{
    if ($field == '')
        return 'No Password was entered';
    else if (strlen($field) < 6)
        return 'Passwords must be at least 6 characters';
    else if (!preg_match('/[a-z]/', $field)
        || !preg_match('/[A-Z]/', $field)
        || !preg_match('/[0-9]/', $field))
        return 'Passwords require one each of a-z, A-Z and 0-9';
    return '';
}

function validate_age($field)
{
    if ($field == '')
        return 'No Age was entered';
    else if ($field < 18 || $field > 110)
        return 'Age must be between 18 and 110';
    return '';
}

function validate_email($field)
{
    if ($field == '')
```

```

        return 'No Email was entered';
    else if (!filter_var($field, FILTER_VALIDATE_EMAIL))
        return 'The Email address is invalid';
    return '';
}

```

// The PHP code

```

$forename_html_entities = '';
$surname_html_entities = '';
$username_html_entities = '';
$password_html_entities = '';
$age_html_entities = '';
$email_html_entities = '';
$errors = $values = [];

if ($_POST) {
    foreach ($_POST as $name => $value)
        $values[$name] = trim($value);

    $error = validate_forename($values['forename']);
    if ($error) $errors['forename'] = $error;
    $error = validate_forename($values['forename']);
    if ($error) $errors['forename'] = $error;
    $error = validate_surname($values['surname']);
    if ($error) $errors['surname'] = $error;
    $error = validate_username($values['username']);
    if ($error) $errors['username'] = $error;
    $error = validate_password($values['password']);
    if ($error) $errors['password'] = $error;
    $error = validate_age($values['age']);
    if ($error) $errors['age'] = $error;
    $error = validate_email($values['email']);
    if ($error) $errors['email'] = $error;

    if (!$errors) {
        /*
        This is where you would enter the posted fields into a database,
        reading the $values array, using password_hash for the password,
        then redirecting to a success page.

        For example:
        $stmt = $pdo->prepare('INSERT INTO users VALUES(:fn,:sn,:un,:pw)');
        $stmt->execute([
            ':fn' => $values['forename'],
            ':sn' => $values['surname'],
            ':un' => $values['username'],
            ':pw' => password_hash($values['forename'], PASSWORD_DEFAULT)
        ]);
        header('Location: success.php');
        exit;

```

```

    We'll simplify it and just output the data:
    */
    echo "<html><body>Form data successfully validated<pre>";
    echo htmlentities(print_r($values, true));
    echo "</pre></body></html>";
    exit;
}

// To echo the values back to the form when validation fails
$forename_html_entities = htmlentities($_POST['forename']);
$surname_html_entities = htmlentities($_POST['surname']);
$username_html_entities = htmlentities($_POST['username']);
$password_html_entities = htmlentities($_POST['password']);
$age_html_entities = htmlentities($_POST['age']);
$email_html_entities = htmlentities($_POST['email']);
}

// The HTML/JavaScript section
?>
<!DOCTYPE html>
<html>
  <head>
    <title>An Example Form</title>
    <style>
      .signup {
        border: 1px solid #999999;
        font: normal 14px helvetica;
        color: #444444;
        background-color: #eeeeee;
        border-spacing: 5px;
      }
      .signup th, .signup td {
        padding: 2px;
      }
      .error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form method="post" action="" id="form">
      <table class="signup">
        <th colspan="2" align="center">Signup Form</th>

        <?php if ($errors) { ?>
          <tr><td colspan="2">Sorry, the following errors were found<br>in your form:
            <p><i class="error">
              <?php foreach ($errors as $error) echo htmlentities($error) . '<br>'; ?>
            </i></p>
          </td></tr>
        <?php } ?>

```

```

<tr><td>Forename</td>
    <td><input type="text" maxlength="32" name="forename" required
        value="<?php echo $forename_html_entities; ?>"></td></tr>
<tr><td>Surname</td>
    <td><input type="text" maxlength="32" name="surname" required
        value="<?php echo $surname_html_entities; ?>"></td></tr>
<tr><td>Username</td>
    <td><input type="text" maxlength="16" name="username" required
        value="<?php echo $username_html_entities; ?>"></td></tr>
<tr><td>Password</td>
    <td><input type="password" name="password" required
        value="<?php echo $password_html_entities; ?>"></td></tr>
<tr><td>Age</td>
    <td><input type="number" max="110" name="age" required
        value="<?php echo $age_html_entities; ?>"></td></tr>
<tr><td>Email</td>
    <td><input type="email" maxlength="64" name="email" required
        value="<?php echo $email_html_entities; ?>"></td></tr>
<tr><td colspan="2" align="center"><input type="submit"
    value="Signup"></td></tr>
</table>
</form>
<script>
    function validateForename(field)
    {
        return (field === "") ? "No Forename was entered." : ""
    }

    function validateSurname(field)
    {
        return (field === "") ? "No Surname was entered." : ""
    }

    function validateUsername(field)
    {
        if (field == "")
            return "No Username was entered."
        else if (field.length < 5)
            return "Usernames must be at least 5 characters."
        else if (!/^[a-zA-Z0-9_-]/.test(field))
            return "Only a-z, A-Z, 0-9, - and _ allowed in Usernames."
        return ""
    }

    function validatePassword(field)
    {
        if (field == "")
            return "No Password was entered."
        else if (field.length < 6)
            return "Passwords must be at least 6 characters."
        else if (!/[a-z]/.test(field) || !/[A-Z]/.test(field) ||
            !/[0-9]/.test(field))

```

```

        return "Passwords require one each of a-z, A-Z and 0-9."
    }
    return ""
}

function validateAge(field)
{
    if (field == "" || isNaN(field))
        return "No Age was entered."
    else if (field < 18 || field > 110)
        return "Age must be between 18 and 110."
    return ""
}

function validateEmail(field)
{
    return (field === "") ? "No Email was entered." : ""
}

function validateFields(form)
{
    const errors = []
    const elements = {}
    let error = ''

    for (let element of form.elements)
        elements[element.name] = element.value.trim()

    error = validateForename(elements.forename)
    if (error) errors.push({field: 'forename', message: error})
    error = validateSurname(elements.surname)
    if (error) errors.push({field: 'surname', message: error})
    error = validateUsername(elements.username)
    if (error) errors.push({field: 'username', message: error})
    error = validatePassword(elements.password)
    if (error) errors.push({field: 'password', message: error})
    error = validateAge(elements.age)
    if (error) errors.push({field: 'age', message: error})
    error = validateEmail(elements.email)
    if (error) errors.push({field: 'email', message: error})
    return errors
}

const validate = function(event)
{
    const errors = validateFields(event.target)
    if (errors.length) {
        const alerts = []
        for (error of errors) {
            alerts.push(error.field + ": " + error.message)
        }
        alert(alerts.join("\n"))
        event.preventDefault()
    }
}

```

```

    }
  }
  document.getElementById('form').addEventListener('submit', validate)
</script>
</body>
</html>

```



In this example, the input is only trimmed before inserting it into the database; one exception is the password, which is hashed, as there's no need to sanitize the data against SQL injection attacks when placeholders and prepared statements are used in database queries.

When the form is redisplayed after it has been submitted with errors, and the submitted values are echoed back to the respective input fields, the sanitized values (suffixed with `_html_entities`) are used to prevent XSS attacks, but these sanitized values are not used for anything else.

For the email address, we've used the built-in validation provided by `type="email"` input when validating the email address in the browser. PHP also has a built-in email address validation, as used in the example:

```
filter_var($field, FILTER_VALIDATE_EMAIL)
```

When called with the `FILTER_VALIDATE_EMAIL` second parameter, the `filter_var` function returns the address (passed here as `$field`) if it's valid, or `FALSE` when `$field` is an invalid email address. Again, this is much easier and safer than writing the email address check yourself.

The result of submitting the form with JavaScript disabled (and two fields incorrectly completed) is shown in [Figure 16-5](#).

I have highlighted the PHP section of this code (and changes to the HTML section) in bold so that you can more clearly see the difference between this and Examples [16-1](#) and [16-2](#).

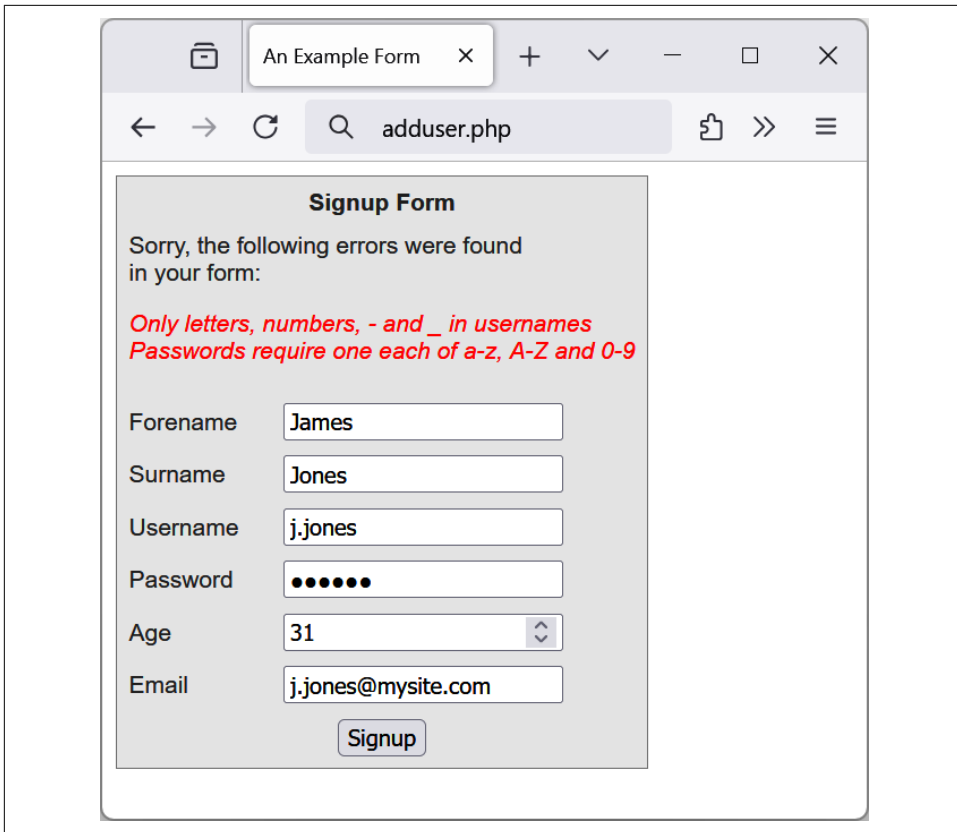


Figure 16-5. The form as represented after PHP validation fails

Now that you've seen how to bring PHP, HTML, and JavaScript together, **Chapter 17** will introduce *Ajax* (Asynchronous JavaScript and XML), which uses JavaScript calls to the server in the background to seamlessly update portions of a web page, without having to resubmit the entire page to the web server. But first, to better remember what you've learned, let's try to answer all the following questions.

Questions

1. What DOM event can you use to send a form for validation prior to submitting it?
2. What JavaScript method is used to test a string against a regular expression?
3. Write a regular expression to match any characters that are *not* in a word, as defined by regular expression syntax.
4. Write a regular expression to match either of the words *fox* or *fix*.

5. Write a regular expression to match any single word followed by any nonword character.
6. Using regular expressions, write a JavaScript function to test whether the word *fox* exists in the string `The quick brown fox`.
7. Using regular expressions, write a PHP function to replace all occurrences of the word *the* in `The cow jumps over the moon` with the word *my*.
8. What HTML attribute is used to precomplete form fields with a value?

See “Chapter 16 Answers” on page 579 in the [Appendix](#) for the answers to these questions.

Using Asynchronous Communication

The term *Ajax* was first coined in 2005. It stands for *Asynchronous JavaScript and XML*, which, in simple terms, means using a set of methods built into JavaScript to transfer data between the browser and a server in the background. This term has now been mostly abandoned in favor of simply talking about asynchronous communication, and one of the reasons is that these days, programmers are more likely to use **JavaScript Object Notation (JSON)** as their preferred data-interchange format, as it's a simple subset of JavaScript.

An excellent example of this technology is Google Maps (although there are numerous others), in which new sections of a map are downloaded from the server when needed, without requiring a page refresh.

Using asynchronous communication not only substantially reduces the amount of data that must be sent back and forth but also makes web pages seamlessly dynamic—allowing them to behave more like self-contained applications. The results are a much improved user interface and better responsiveness.

The Fetch API

In the past, making Ajax calls was a real pain in the neck because there were so many different implementations across various browsers. Luckily things vastly improved around 2015, when the simple `fetch` function was introduced to modern browsers. It is defined by the **Fetch standard**, which unifies how requests and responses work across the whole browser.

So, for example, to make a GET request, you use code such as this:

```
fetch("https://example.com")
```

Or, for a POST request, you can specify the method and add some fields to the body of the request using the second parameter:

```
const data = new FormData()
data.set("field", "value")
const options = {
  method: "POST",
  body: data
}
fetch("https://example.com", options)
```

Your First Asynchronous Program

Type and save the code in **Example 17-1** as *urlpost.html*, but don't load it into your browser yet.

Example 17-1. urlpost.html

```
<!DOCTYPE html>
<html> <!-- urlpost.html -->
  <head>
    <title>Asynchronous Communication Example</title>
    <style>
      body { text-align:center; }
    </style>
  </head>
  <body>
    <h1>Loading a web page into a DIV</h1>
    <div id="info">This sentence will be replaced</div>

    <script>
      const data = new FormData()
      data.set("url", "cnet.com")
      const options = {
        method: "POST",
        body: data,
      }
      fetch("http://localhost/18/urlpost.php", options)
        .then(response => response.text())
        .then(text => document.getElementById("info").innerHTML = text)
    </script>
  </body>
</html>
```



Use `innerText` or `textContent` for User Input

This example uses the `innerHTML` property because we want the data to be rendered as HTML. But assigning user input to this property will create new DOM elements, including the ones that could be used for a successful XSS attack. So if you're going to display user-entered data or data from untrusted sources, you should instead assign them to the `innerText` property or the `textContent` property; either will display the data as plain text only.

Let's go through this document and look at what it does, starting with the first eleven lines, which simply set up an HTML document and display a heading. The next line creates a `<div>` with the ID `info`, containing the text `This sentence will be replaced by default`. Later on, the text returned from the asynchronous call will be inserted here.

After this, a new `FormData` object is created called `data`, which is used to set the request field called `url` and then stored in the `options` object as the request body together with the method used to send the request. Calling the `fetch` function returns an object which we'll use to call the `then` method on and pass an anonymous arrow function as the parameter. The arrow function will receive the `response` object and call the `text` method on it and pass the result for further processing in the second `then` call. The second arrow function takes the `text` parameter, which contains the HTML from the URL that `fetch` was called with as returned by `response.text()` in the first arrow function, and displays it in the `<div>`.

Circling back to the `fetch` call, the object it returns is a `Promise`. It represents an asynchronous operation in JavaScript (not necessarily a network operation as you'll see soon) and, if the operation completed successfully, we say the promise was *fulfilled*. And when the promise becomes fulfilled, you can execute your own code, which is the first arrow function passed in the `then` call:

```
response => response.text()
```

The `text` method also returns a promise: that's why the second `then` call is needed. Then again, when the second promise is fulfilled, you can run your own code, which now sets the `innerHTML` property and finally displays the main page of *cnet.com*. The effect is that only the `<div>` element of the web page changes, while everything else remains the same.



If you set up a development server using AMPPS (or a similar WAMP, LAMP, or MAMP) as shown in [Chapter 2](#), downloaded the [example files from GitHub](#) and saved them in the document root of the web server (as described in that chapter), the [Chapter 18](#) folder will be in the right place for this code to work correctly. If any part of your setup is different, or you run this code on a development server using a domain of your choice, you will have to change those values in this code accordingly.

The Server Half of the Asynchronous Process

Now we get to the PHP half of the equation, which you can see in [Example 17-2](#). Type this code and save it as *urlpost.php*.

Example 17-2. urlpost.php

```
<?php // urlpost.php
if (isset($_POST['url']))
{
    echo file_get_contents('http://' . $_POST['url']);
}
?>
```

This program uses the `file_get_contents` PHP function to load in the web page at the URL supplied to it in the variable `$_POST['url']`. The `file_get_contents` function is versatile in that it loads in the entire contents of a file or web page from either a local or a remote server; it even takes into account moved pages and other redirects.

Once you have typed the program, you are ready to call up *urlpost.html* in your web browser, and after a few seconds you should see the contents of the *cnet.com* front page loaded into the `<div>` that we created for that purpose.

To test the program, enter the following into your browser:

`http://localhost/18/urlpost.html`

It won't be as fast as directly loading the web page, because it is transferred twice—once to the server and again from the server to your browser—but the result should look somewhat similar to [Figure 17-1](#).

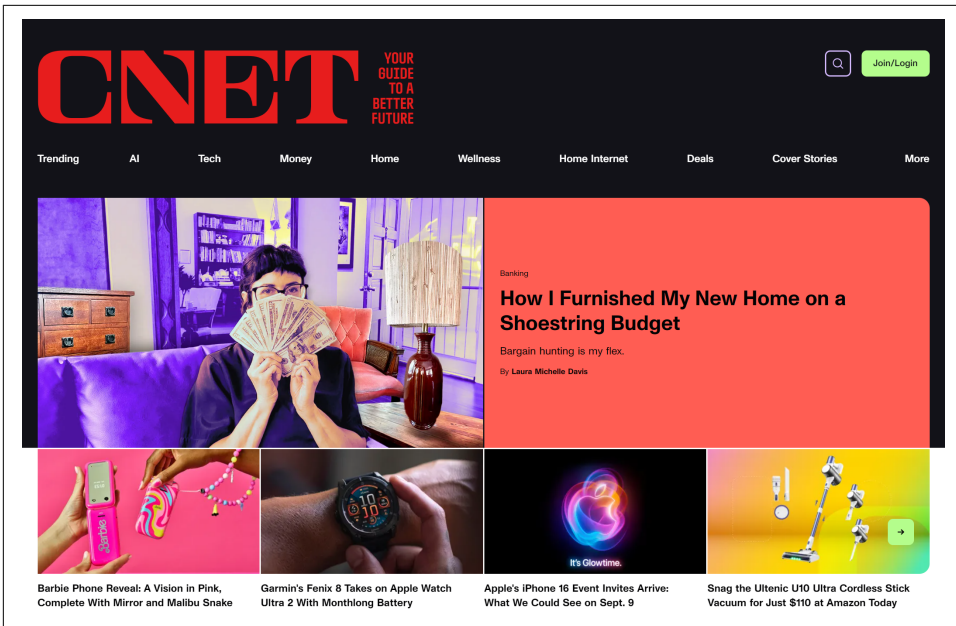


Figure 17-1. The *cnet.com* front page

Not only have we succeeded in making an asynchronous call and having a response returned to JavaScript, but we've also harnessed the power of PHP to merge in a totally unrelated external web page. Incidentally, if we had tried to find a way to asynchronously fetch this web page directly (without recourse to the PHP server-side module), we wouldn't have succeeded, because there are security blocks preventing cross-origin (sometimes called cross-domain) asynchronous communication. So, this example also illustrates a handy solution to a practical problem.

Cross-Origin Resource Sharing (CORS)

Cross-origin security makes using Ajax a little harder than it used to be when it was first introduced because your JavaScript code may not be allowed to read the response the server sent you. The mechanism that allows JavaScript code to access the response in such cases is called cross-origin resource sharing (CORS). First let's explain some of the terms.

Origin

An *origin* means the part of a URL that is a combination of *schema* (sometimes called *protocol*), *domain*, and *port*, if specified. Given a URL like this:

```
https://example.com/office/map.html?nonstop=true
```

the origin part of the URL is `https://example.com`.

Same-origin and cross-origin requests

Two URLs have the same origin when their origin parts are the same, like this:

```
https://example.com/office/map.html?nonstop=true
https://example.com/contact
```

When you load a page in your browser and create a request, for example with `fetch`, which loads a response from a different URL but with the same origin, we say it's a *same-origin request*.

A *cross-origin request* is when you'd load a page in your browser, let's say `https://example.com/contact`, and that page created a request to, for example, `https://maps.com/location?id=1337` or any other domain that does not have the same origin, either by using `fetch`, loading an image from that URL, or similar. All the following URLs have different origins. Can you spot why?

```
https://example.com/office
https://www.example.com/contact
https://www.example.net/contact
http://www.example.net/contact
```

By default, `fetch` can access the response only when a same-origin request was sent. This is why you will get an error if you load `http://example.com` in your browser, open the browser console, and try to run the following code:

```
fetch("https://www.cnet.com")
```

The browser console will show an error message similar to:

Access to fetch at 'https://www.cnet.com/' from origin 'http://localhost' has been blocked by CORS policy

To allow your JavaScript to access the response (shared from a different origin, to tie it back to the CORS mechanism), the `www.cnet.com` server would need to send a response HTTP header like this:

```
Access-Control-Allow-Origin: http://localhost
```

Which roughly translates to “I, `www.cnet.com`, allow a JavaScript running on pages starting with `http://localhost` to access the response I'm sending,” and I'm quite sure the `www.cnet.com` server will never send such header.

However, the server also can allow any origin to access the response, which sometimes happens, but at least at the time of writing, `www.cnet.com` doesn't allow that. The header to allow any origin to access the response looks like this:

```
Access-Control-Allow-Origin: *
```


CORS blocking the request is why we're using the `urlpost.php` script to load the home page of *cnet.com*. Because then the page `http://localhost/18/urlpost.html` can use `fetch` to send a request to `http://localhost/18/urlpost.php` because they have the same origin.

Using GET Instead of POST

As when you submit any data from a form, you have the option of submitting your data in the form of GET requests, and you will save a few lines of code if you do so. However, there is a possible downside: some browsers may cache GET requests, whereas POST requests will never be cached. You don't want to cache a request, because the browser will just redisplay what it got the last time instead of going to the server for fresh input. The solution is to use a workaround that adds a random parameter to each request, ensuring that each URL requested is unique. This technique is called *cachebusting*.

Example 17-3 shows how you would achieve the same result as with **Example 17-1** but using a GET request instead of a POST.

Example 17-3. urlget.html

```
<!DOCTYPE html>
<html> <!-- urlget.html -->
  <head>
    <title>Asynchronous Communication Example</title>
    <style>
      body { text-align:center; }
    </style>
  </head>
  <body>
    <h1>Loading a web page into a DIV</h1>
    <div id="info">This sentence will be replaced</div>

    <script>
      const params = new URLSearchParams({
        url: "cnet.com",
        nocache: Math.random() * 1000000
      })
      fetch("http://localhost/18/urlget.php?" + params)
        .then(response => response.text())
        .then(text => document.getElementById("info").innerHTML = text)
    </script>
  </body>
</html>
```

The differences to note between the two documents are highlighted in bold and described as follows:

- We create the query string by using the `URLSearchParams` object with the desired parameters passed to the constructor. The advantage of using this as compared to building the string manually is that the values will be properly sanitized automatically if needed.
- The first parameter is the URL, *cnet.com*, and the second parameter `nocache` is a random value between 0 and 1 million. This ensures that each URL requested is different and therefore that no requests will be cached.
- We call the `fetch` function with only one parameter (GET request is the default, no need to specify in the options parameter), supplying a URL that contains a `?` symbol followed by the `params` object that will be converted to parameter/value pairs.

To accompany this new document, the PHP program must be modified to respond to a GET request, as in [Example 17-4](#), *urlget.php*.

Example 17-4. urlget.php

```
<?php // urlget.php
if (isset($_GET['url']))
{
    echo file_get_contents('http://' . $_GET['url']);
}
?>
```

The only difference between this and [Example 17-2](#) is that the references to `$_POST` have been replaced with `$_GET`. The result of calling up *urlget.html* in your browser is identical to loading *urlpost.html*.

To test this revised version of the program, enter the following into your browser. You should see the same result as before, just loaded via a GET rather than a POST request:

`http://localhost/18/urlget.html`

Sending JSON Requests

Very often, your JavaScript needs to work with more data than just the HTML seen in the previous `fetch` examples. This section will show you how you can use JSON to transfer structured data from the server to the browser. For example, besides the HTML of *cnet.com* home page, we want to receive a color that will be used for our page header indicating success. The color could change based on the time the *cnet.com* home page was requested, but we'll simply set it to blue.

Let's modify the previous example document and PHP program to fetch some JSON data. To do this, first look at the PHP program, *jsonget.php*, shown in [Example 17-5](#).

Example 17-5. jsonget.php

```
<?php // jsonget.php
if (isset($_GET['url']))
{
    header('Content-Type: application/json');
    $data = [
        'html' => file_get_contents('http://' . $_GET['url']),
        'color' => 'blue',
    ];
    echo json_encode($data);
}
?>
```

This program outputs the correct JSON Content-Type header before printing a JSON-encoded associative array with two items: the first is the HTML of the requested page; the second is the color.

On to the HTML document, *jsonget.html*, shown in [Example 17-6](#).

Example 17-6. jsonget.html

```
<!DOCTYPE html>
<html> <!-- jsonget.html -->
<head>
    <title>Asynchronous Communication Example</title>
    <style>
        body { text-align:center; }
    </style>
</head>
<body>
    <h1 id="header">Loading a web page into a DIV</h1>
    <div id="info">This sentence will be replaced</div>

    <script>
        const params = new URLSearchParams({
            url: "cnet.com",
            nocache: Math.random() * 1000000
        })
        fetch("http://localhost/18/jsonget.php?" + params)
            .then(response => response.json())
            .then(data => {
                document.getElementById("header").style.backgroundColor = data.color
                document.getElementById("info").innerHTML = data.html
            })
    </script>
```

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

The differences between this and the previous example are highlighted in bold. As you can see, this code is substantially similar to the previous versions, except that `fetch` now requests `jsonget.php` and the first arrow function uses `response.json()` to parse the JSON in the server response.

The second then call uses an arrow function that receives the parsed JSON in the `data` parameter, and it uses the `data.color` property to set the background color of the `<h1>` header, now with the `id="header"` attribute, and the `data.html` property to show the home page HTML, similar to the previous examples.

You may have noticed that the property names `color` and `html` match the keys of the array encoded to JSON in the PHP code. You can easily add or use other data as needed.

Using XMLHttpRequest

Although more rare today, some existing or older code might still employ a different approach to Ajax using the `XMLHttpRequest` object, sometimes referred to as *XHR*. Using it for any new development is not recommend as it is a less flexible and less powerful alternative to `fetch`. It is a legacy way of sending asynchronous requests, so we'll show it only very shortly. If you'd like to know more, you can always read the [MDN web docs](#).

The code in [Example 17-7](#) performs the same as the code using `fetch` in [Example 17-1](#) except it uses `XMLHttpRequest`.

Example 17-7. `urlpostxhr.html`

```
<!DOCTYPE html>
<html> <!-- urlpostxhr.html -->
  <head>
    <title>Asynchronous Communication Example</title>
    <style>
      body { text-align:center; }
    </style>
  </head>
  <body>
    <h1>Loading a web page into a DIV</h1>
    <div id="info">This sentence will be replaced</div>

    <script>
      const xhr = new XMLHttpRequest()
      xhr.open("POST", "http://localhost/18/urlpost.php", true)
      xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")
```

```

    xhr.send("url=cnet.com")
    xhr.onload = () => {
      if (xhr.readyState === xhr.DONE && xhr.status === 200) {
        document.getElementById("info").innerHTML = xhr.responseText
      }
    };
  </script>
</body>
</html>

```

The HTML is the same as in [Example 17-1](#), except the `<script>...</script>` block. It uses `XMLHttpRequest` to send a POST request, but here we also have to set the correct Content-Type header, check the `readyState` property, and handle the load event.

When using `XMLHttpRequest`, the `xhr.responseXML` property contains the XML response, if the server sent back an XML document, parsed into a DOM tree for you if you'd ever need it. If you have a JSON response you'd have to parse it yourself using `JSON.parse()`; nothing like the `response.json()` method is offered by the Fetch API.

Using Frameworks for Asynchronous Communication

Now that you know how to code your own asynchronous routines, you might like to investigate some of the free frameworks available to make it even easier, and that offer many more advanced features. In particular, I suggest you check out [React](#), probably the fastest-growing framework, or [Axios](#). You may also encounter [jQuery](#), which is still very popular in existing applications but shouldn't be used for new development as most of the functionality is now available natively.

In [Chapter 18](#) we'll look at how to apply styling to your websites with CSS, but before moving on, you should try to answer the following questions first to repeat what you've learned in this chapter.

Questions

1. Which function can you use to conduct asynchronous communication between a web server and JavaScript client?
2. How can you send a POST request using the `fetch` function?
3. What does the `fetch` function return and how do you use it?
4. Create a function to get the response as parsed JSON that can be passed to the `then` method called on the promise object returned by `fetch`.

5. If your server returned an array with two items called `a` and `b` encoded as JSON, how can you access the fields in your JavaScript code after calling `fetch`?
6. Given the URL `https://book.example/ch18?q=6`, what's the origin part of the URL?
7. You've loaded `https://example.com/map` into your browser, and JavaScript running on that page would like to send an asynchronous request to `https://www.example.com/data`. Would that be a same-origin request? Explain why or why not.
8. What is the best way to allow JavaScript on `http://localhost/info` to access the fetch response from `https://example.com/data`?

See “Chapter 17 Answers” on page 580 in the [Appendix](#) for the answers to these questions.

Advanced CSS

The first CSS implementation was drawn up in 1996 and released in 1999; it has been supported by all browser releases since 2001. The standard for this version (CSS1) was revised in 2008. In 1998, developers began drawing up the second specification (CSS2); its standard was completed in 2007 and revised in 2009, while development for the CSS3 specification commenced in 2001, with some new features proposed in 2009 and recommendations continuing to be made.

A CSS4 was proposed by the CSS working group, but the naming convention appears to have been dropped as this is not a major leap forward. Rather, it's simply a development of one part of CSS—the selectors, and therefore mostly referred to as [Selectors Level 4](#).

Thankfully, though, the CSS working group publishes regular snapshots of the CSS modules that it considers stable, and you can see the 2023 snapshot at the [World Wide Web Consortium \(W3C\) website](#), which is the best place to gauge the current state of play in the world of CSS. You can learn more about how CSS3 has developed in practice as of 2023 (and also what's coming) at the [Chrome Developers Blog](#).

In this chapter, I'll take you through the most important CSS3 features that have been adopted by the major browsers, many of which provide functionality that previously could be attained only with JavaScript.

I recommend using CSS, instead of JavaScript, to implement dynamic features. The features CSS provides make document attributes part of the document itself, instead of being tacked on through JavaScript, providing a cleaner design.



There's an awful lot to CSS, and browsers implement the various features differently (if at all). When you want to ensure that the CSS you are creating will work in all browsers, first look at the “[Can I Use...](#)” website. It maintains a record of what features are available in which browsers, so it will always be more up-to-date than this book, which sees a new edition only every couple of years —and CSS can move a long way in that time.

Attribute Selectors

In [Supplemental Chapter 1, “Introduction to CSS”](#), (available as a bonus PDF in the [GitHub resource for this book](#)), I detail the various CSS attribute selectors, which I will now quickly recap. Selectors are used in CSS to match HTML elements, and there are 10 different types, as listed in [Table 18-1](#).

Table 18-1. CSS selectors, pseudoclasses, and pseudoelements

Selector type	Example
Universal selector	<code>* { color:#555; }</code>
Type selectors	<code>b { color:red; }</code>
Class selectors	<code>.classname { color:blue; }</code>
ID selectors	<code>#id { background:cyan; }</code>
Descendant selectors	<code>span em { color:green; }</code>
Child selectors	<code>div > em { background:lime; }</code>
Adjacent sibling selectors	<code>i + b { color:gray; }</code>
Attribute selectors	<code>a[href='info.htm'] { color:red; }</code>
Pseudoclasses	<code>a:hover { font-weight:bold; }</code>
Pseudoelements	<code>P::first-letter { font-size:300%; }</code>

The CSS designers decided that most of these selectors worked just fine the way they were, but made three enhancements so that you can more easily match elements based on the contents of their attributes. The following sections examine these.

In CSS2 you can use a selector such as `a[href='info.htm']` to match the string `info.htm` when found in an `href` attribute, but there's no way to match only a *portion* of a string. CSS3 comes to the rescue with three new operators: `^`, `$`, and `*`. If one directly precedes the `=` symbol, you can match the start, end, or any part of a string, respectively.

The ^= Operator

The ^= operator matches at the start of a string. So, for example, the following will match any href attribute whose value begins with the string http://website:

```
a[href^='http://website']
```

Therefore, the following element will match:

```
<a href='http://website.com'>
```

But this will not:

```
<a href='http://mywebsite.com'>
```

The \$= Operator

To match only at the end of a string, you can use a selector such as the following, which will match any img tag whose src attribute ends with .png:

```
img[src$='.png']
```

For example, the following will match:

```
<img src='photo.png'>
```

But this will not:

```
<img src='snapshot.jpg'>
```

The *= Operator

To match any substring anywhere in the attribute, you can use a selector such as the following, which finds any links on a page that have the string google anywhere within them:

```
a[href*='google']
```

For example, the HTML segment will match, while the segment will not.

The box-sizing Property

The W3C box model specifies that the width and height of an object should refer only to the dimensions of an element's content, ignoring any padding or border. But some web designers have expressed a desire to specify dimensions that refer to an entire element, including any padding and border. [Figure 18-1](#) shows two elements of the same width using the different box models.

To provide this feature, CSS lets you choose the box model you wish to use with the `box-sizing` property. For example, to use the total width and height of an object including padding and borders, use this declaration:

```
box-sizing : border-box;
```

Or to have an object's width and height refer only to its content, use this declaration (the default):

```
box-sizing : content-box;
```

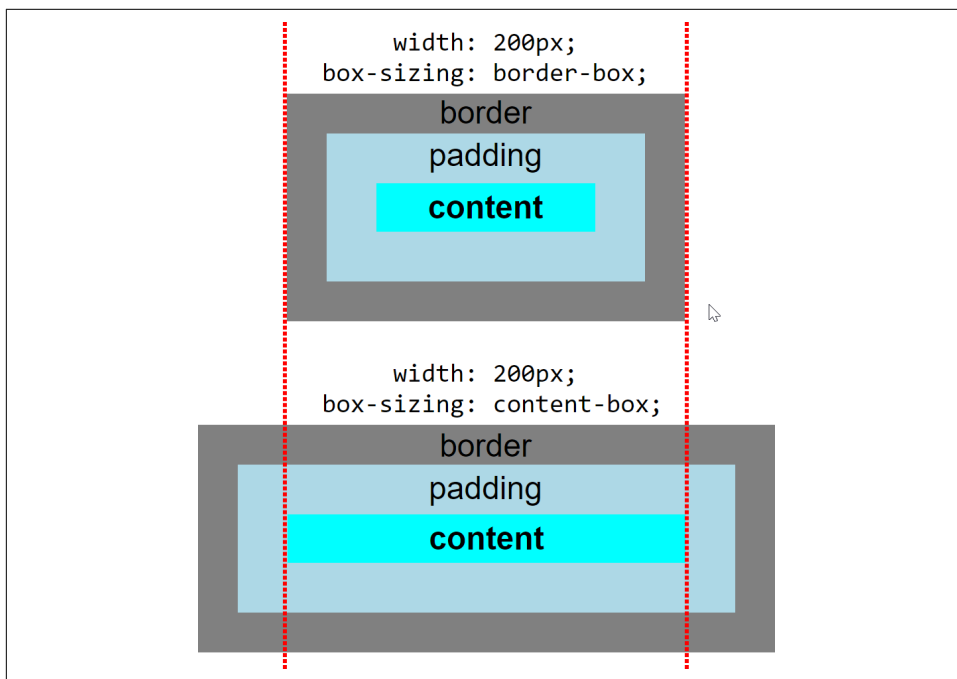


Figure 18-1. An element with `width: 200px` in the two box models

CSS Backgrounds

CSS provides two related properties: `background-clip` and `background-origin`. Between them, you can specify where a background should start within an element, and how to clip the background so that it doesn't appear in parts of the box model where you don't want it to.

To accomplish this, both properties support these values:

`border-box`

Refers to the outer edge of the border

`padding-box`

Refers to the outer edge of the padding area

`content-box`

Refers to the outer edge of the content area

The `background-clip` Property

The `background-clip` property specifies whether the background should be ignored (clipped) if it appears within either the border or padding area of an element. For example, the following declaration states that the background may display in all parts of an element, all the way to the outer edge of the border:

```
background-clip : border-box;
```

To keep the background from appearing within the border area of an element, you can restrict it to only the section of an element inside the outer edge of its padding area, like this:

```
background-clip : padding-box;
```

To restrict the background to display only within the content area of an element, use this declaration:

```
background-clip : content-box;
```

Figure 18-2 shows three rows of elements displayed in the Chrome web browser, in which the first row uses `border-box` for the `background-clip` property, the second uses `padding-box`, and the third uses `content-box`.

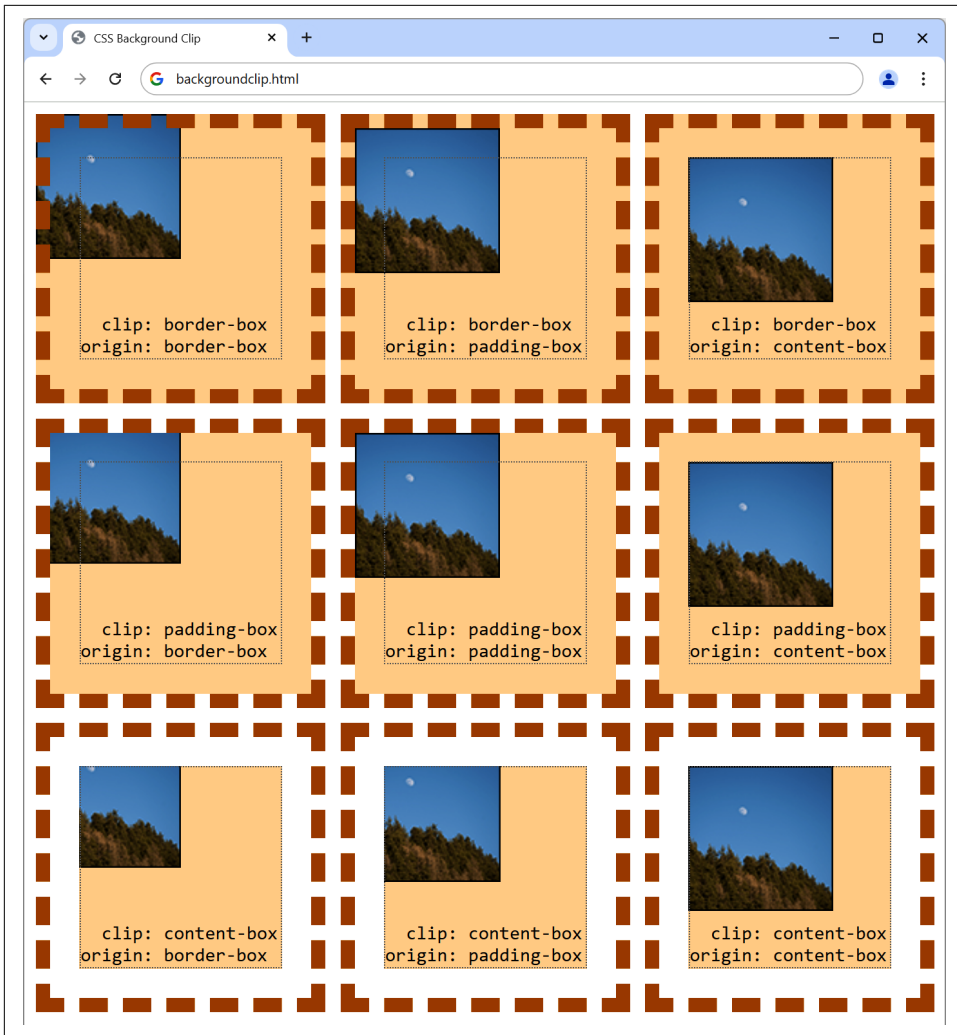


Figure 18-2. Different ways of combining CSS background properties

In the first row, the inner box (an image file that has been loaded into the top left of the element, with repeating disabled) is allowed to display anywhere in the element. You can also clearly see it displayed in the border area of the first box because the border has been set to dotted.

In the second row, neither the background image nor the background shading displays in the border area, because they have been clipped to the padding area with a background-clip property value of padding-box.

Then, in the third row, both the background shading and the image have been clipped to display only within the inner content area of each element (shown inside a light-colored, dotted box), using a `background-clip` property of `content-box`.

The background-origin Property

With the `background-origin` property, you can control where a background image will be located by specifying where the top left of the image should start. For example, the following declaration states that the background image's origin should be the top-left corner of the outer edge of the border:

```
background-origin : border-box;
```

To set the origin of an image to the top-left outer corner of the padding area, use this declaration:

```
background-origin : padding-box;
```

Or to set the origin of an image to the top-left corner of an element's inner content section, use this declaration:

```
background-origin : content-box;
```

Looking again at [Figure 18-2](#), you can see in each row the first box uses a `background-origin` property of `border-box`, the second uses `padding-box`, and the third uses `content-box`. Consequently, in each row the smaller inner box displays at the top left of the border in the first box, the top left of the padding in the second, and the top left of the content in the third box.



The only differences to note between the rows, with regard to the origins of the inner box in [Figure 18-2](#), are that in rows 2 and 3 the inner box is clipped to the padding and content areas, respectively; therefore, outside these areas no portion of the box is displayed.

The background-size Property

In the same way that you can specify the width and height of an image when used in the `<img>` tag, in the latest browser versions you can also do so for background images.

Apply the property as follows (where *ww* is the width and *hh* is the height):

```
background-size : wwpx hhpix;
```

If you prefer, you can use only one argument, and then both dimensions will be set to that value. Also, if you apply this property to a block-level element such as a `<div>` (rather than one that is inline, such as a `<span>`), you can specify the width and/or height as a percentage instead of a fixed value. Units such as *em* (relative to the font size of this element) and *rem* (relative to the font size of the root element) also can be used. The property allows two special size values, `contain` and `cover`, that specify how the image should be sized and scaled. See the [MDN page on resizing images with background-size for an example](#).

Using the auto Value

If you wish to scale only one dimension of a background image, and then have the other one scale automatically to retain the same proportions, you can use the value `auto` for the other dimension, like this:

```
background-size : 100px auto;
```

This sets the width to 100 pixels and the height to a value proportionate to the increase or decrease in width.



Different browsers may require different versions of the various background property names, so refer to the [“Can I Use...” website](#) to ensure you are applying all the versions required for the browsers you are targeting.

Multiple Backgrounds

With CSS you can attach multiple backgrounds to an element, each of which can use the previously discussed CSS background properties. [Figure 18-3](#) shows an example of this; eight different images have been assigned to the background to create the four corners and four edges of the certificate border.

To display multiple background images in a single CSS declaration, separate them with commas. [Example 18-1](#) shows the HTML and CSS used to create the background in [Figure 18-3](#).

Example 18-1. Using multiple images in a background

```
<!DOCTYPE html>
<html> <!-- backgroundimages.html -->
<head>
  <title>CSS Multiple Backgrounds Example</title>
  <style>
    .border {
      font-family: 'Times New Roman';
      font-style :italic;
      font-size  :170%;
      text-align :center;
      padding    :60px;
      width      :350px;
      height     :500px;
      background :url('b1.gif') top left no-repeat,
                  url('b2.gif') top right no-repeat,
                  url('b3.gif') bottom left no-repeat,
                  url('b4.gif') bottom right no-repeat,
                  url('ba.gif') top repeat-x,
                  url('bb.gif') left repeat-y,
                  url('bc.gif') right repeat-y,
                  url('bd.gif') bottom repeat-x
    }
  </style>
</head>
<body>
  <div class='border'>
    <h1>Employee of the month</h1>
    <h2>Awarded To:</h2>
    <h3>_____</h3>
    <h2>Date:</h2>
    <h3>___/___/____</h3>
  </div>
</body>
</html>
```

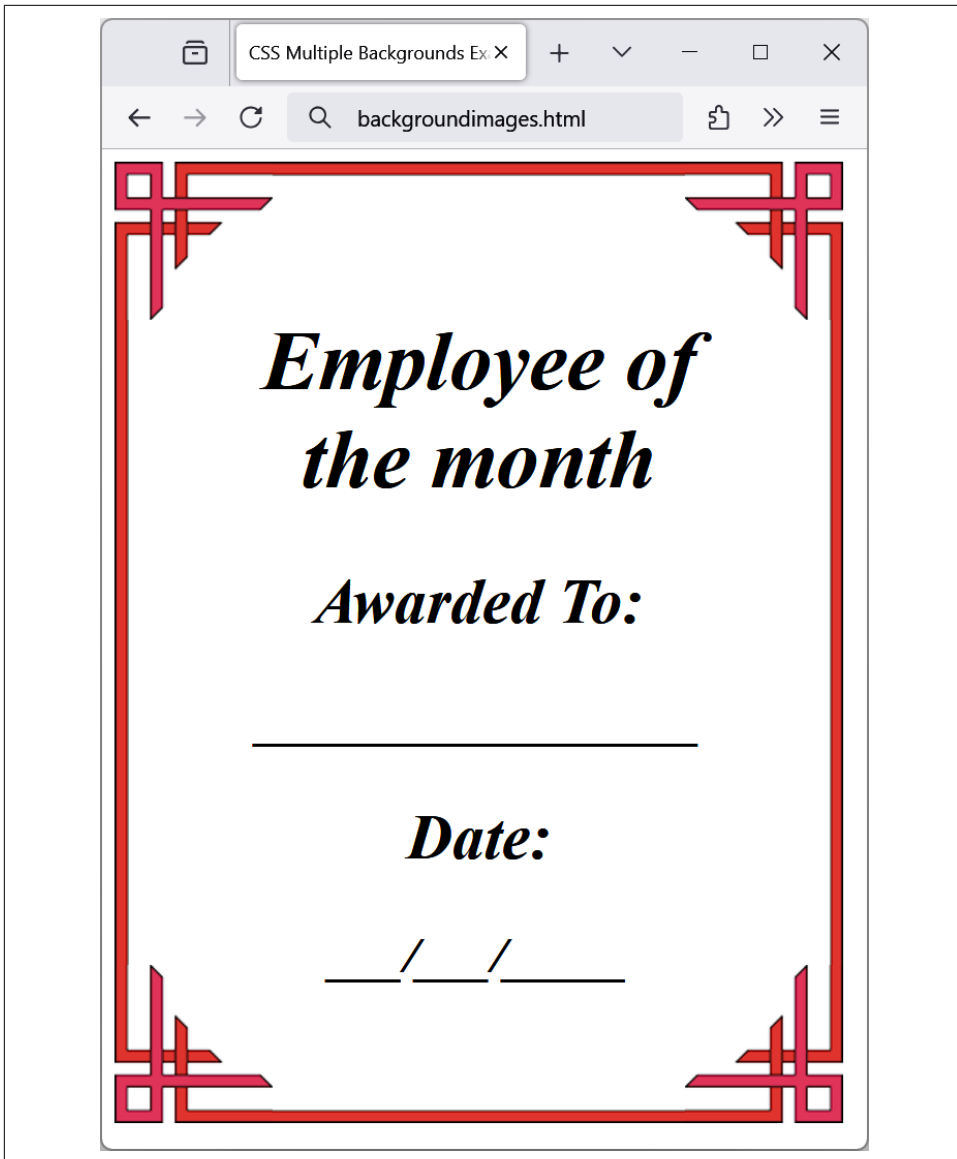


Figure 18-3. A background created with multiple images

In the CSS section, the first four lines of the background declaration place the corner images into the four corners of the element, and the final four place the edge images, which are handled last because the order of priority for background images goes from top to bottom. In other words, where they overlap, additional background images

will appear behind already placed images. If the GIFs were in the reverse order, the repeating edge images would display on top of the corners, which would be incorrect.



Using this CSS, you can resize the containing element to any dimensions, and the border will always correctly resize to fit, which is much easier than using tables or multiple elements for the same effect.

CSS Borders

CSS also brings a lot more flexibility to the way borders can be presented, by allowing you to independently change the colors of all four border edges, display images for the edges and corners, provide a radius value for applying rounded corners to borders, and place box shadows underneath elements.

The border-color Property

There are two ways to apply colors to a border. First, you can pass a single color to the property:

```
border-color : #888;
```



The shorthand color notation #303 is the same as specifying #330033, so #888 seen in the preceding code snippet is the same as #888888.

This property sets all the borders of an element to mid-gray. You can also set border colors individually, like this (which sets the border colors to various shades of gray):

```
border-top-color : #000;  
border-left-color : #444;  
border-right-color : #888;  
border-bottom-color : #ccc;
```

Or you can set all the colors individually with a single declaration:

```
border-color:#f00 #0f0 #880 #00f;
```

This declaration sets the top border color to #f00, the right one to #0f0, the bottom one to #880, and the left one to #00f (red, green, orange, and blue, respectively). You can also use color names for the arguments.

The border-radius Property

Prior to CSS3, talented web developers came up with numerous tweaks and fixes to achieve rounded borders, generally using `<table>` or `<div>` tags.

Now adding rounded borders to an element is really simple, and it works in the latest versions of all major browsers, as shown in [Figure 18-4](#), in which a 10-pixel border is displayed in different ways. [Example 18-2](#) shows the HTML for this.

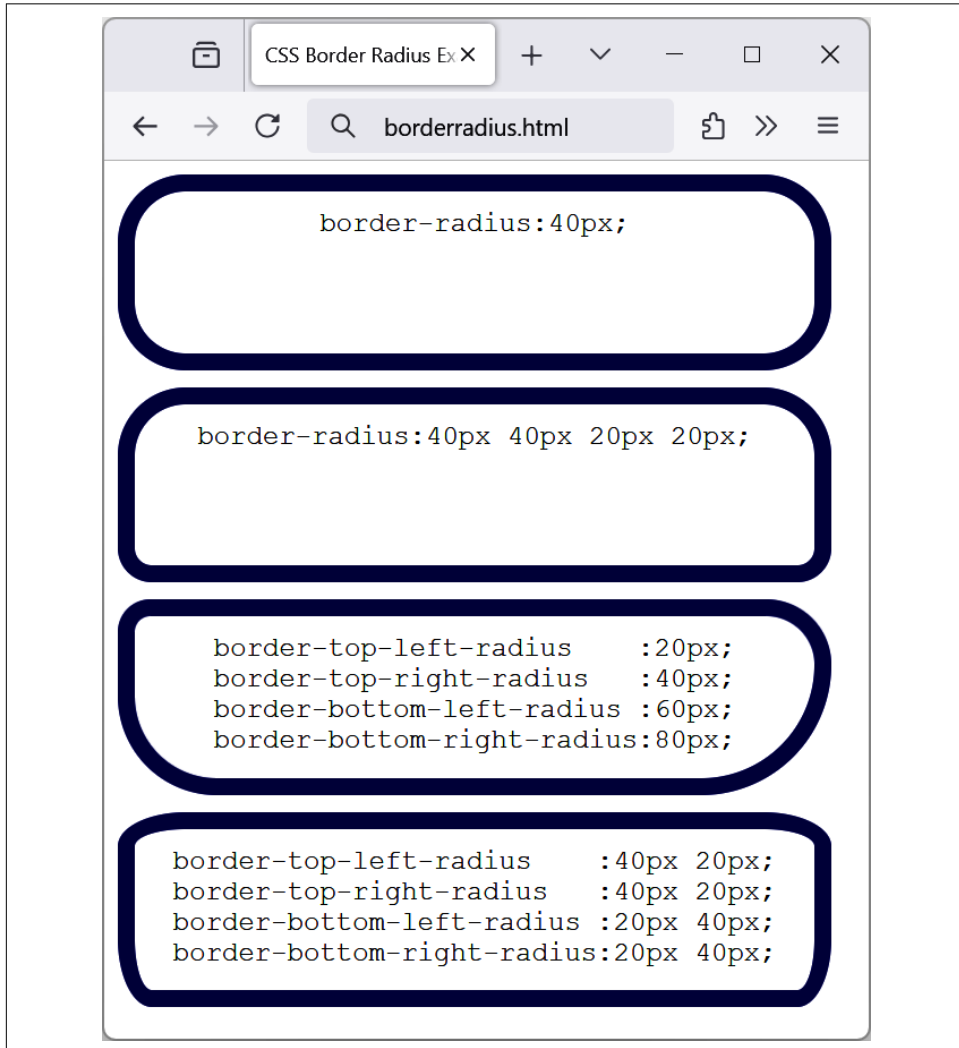


Figure 18-4. Mixing and matching various border radius properties

Example 18-2. The border-radius property

```
<!DOCTYPE html>
<html> <!-- borderradius.html -->
<head>
<title>CSS Border Radius Examples</title>
<style>
    .box {
        margin-bottom : 10px;
        font-family   : 'Courier New', monospace;
        font-size      : 1rem;
        text-align     : center;
        padding        : 10px;
        width          : 380px;
        height         : 75px;
        border         : 10px solid #006;
    }
    .b1 {
        border-radius : 40px;
    }
    .b2 {
        border-radius : 40px 40px 20px 20px;
    }
    .b3 {
        border-top-left-radius       : 20px;
        border-top-right-radius      : 40px;
        border-bottom-left-radius    : 60px;
        border-bottom-right-radius   : 80px;
    }
    .b4 {
        border-top-left-radius       : 40px 20px;
        border-top-right-radius      : 40px 20px;
        border-bottom-left-radius    : 20px 40px;
        border-bottom-right-radius   : n20px 40px;
    }
</style>
</head>
<body>
<div class='box b1'>
    border-radius : 40px;
</div>

<div class='box b2'>
    border-radius : 40px 40px 20px 20px;
</div>

<div class='box b3'>
    border-top-left-radius &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~20px;<br>
    border-top-right-radius &nbsp;&nbsp;&nbsp;&nbsp;&~40px;<br>
    border-bottom-left-radius :60px;<br>
    border-bottom-right-radius:80px;
</div>
```

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

use the following declaration:

border-radius : 20px;

clockwise direction starting from the top-left corner):

border-radius : 10px 20px 30px 40px;

If you prefer, you can also address each corner of an element individually, like this:

```
border-top-left-radius    : 20px;
border-top-right-radius   : 40px;
border-bottom-left-radius : 60px;
border-bottom-right-radius: 80px;
```

like this:

```
border-top-left-radius      : 40px 20px;
border-top-right-radius     : 40px 20px;
border-bottom-left-radius   : 20px 40px;
border-bottom-right-radius  : 20px 40px;
```

The first argument is the horizontal, and the second is the vertical radius.

Box Shadows

To apply a box shadow, specify a horizontal and vertical offset from the object, the amount of blurring to add to the shadow, and the color to use, like this:

```
box-shadow : 15px 15px 10px #888;
```

blurring, with smaller values resulting in less blurring, and #888 is the color for the shadow, which can be any valid color value. The result of this declaration can be seen in [Figure 18-5](#).



Figure 18-5. A box shadow displayed under an element

Element Overflow

In CSS2, you can indicate what to do when one element is too large to be fully contained by its parent by setting the `overflow` property to `hidden`, `visible`, `scroll`, or `auto`. But with CSS3, you can now separately apply these values in the horizontal or vertical directions, as well as with these example declarations:

```
overflow-x : hidden;  
overflow-x : visible;  
overflow-y : auto;  
overflow-y : scroll;
```

Multicolumn Layout

One of the features most requested by web developers is multiple columns, and this was realized in CSS3. Now, flowing text over multiple columns is as easy as specifying the number of columns and then (optionally) choosing the spacing between them and the type of dividing line (if any), as shown in [Figure 18-6](#) (created with [Example 18-3](#)).

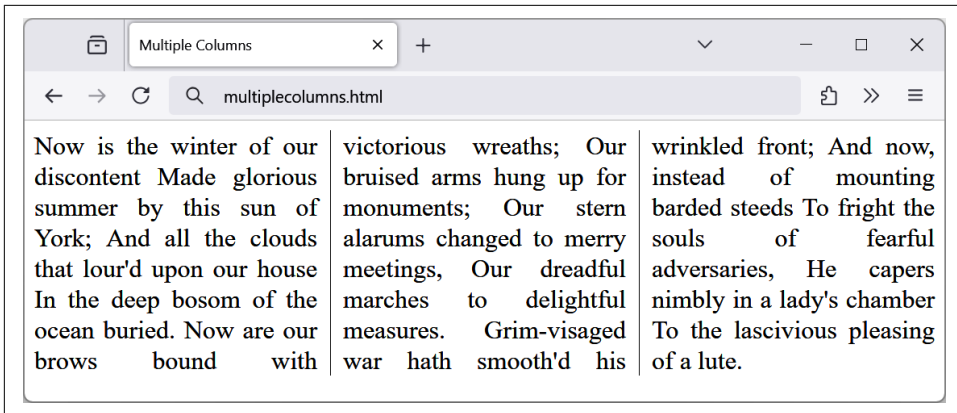


Figure 18-6. Flowing text in multiple columns

Example 18-3. Using CSS to create multiple columns

```
<!DOCTYPE html>
<html> <!-- multiplecolumns.html -->
<head>
  <title>Multiple Columns</title>
  <style>
    .columns {
      text-align : justify;
      font-size  : 1.3rem;
      column-count : 3;
      column-gap   : 1em;
      column-rule  : 1px solid black;
    }
  </style>
</head>
<body>
  <div class='columns'>
    Now is the winter of our discontent
    Made glorious summer by this sun of York;
    And all the clouds that lour'd upon our house
    In the deep bosom of the ocean buried.
    Now are our brows bound with victorious wreaths;
    Our bruised arms hung up for monuments;
    Our stern alarums changed to merry meetings,
    Our dreadful marches to delightful measures.
    Grim-visaged war hath smooth'd his wrinkled front;
    And now, instead of mounting barded steeds
    To fright the souls of fearful adversaries,
    He capers nimbly in a lady's chamber
    To the lascivious pleasing of a lute.
  </div>
</body>
</html>
```

Within the `.columns` class, the first two lines simply tell the browser to right-justify the text and to set it to a font size of `1.3rem`. These declarations aren't needed for multiple columns, but they improve the text display. The remaining lines set up the element so that, within it, text will flow over three columns, with a gap of `1em` between the columns and with a single-pixel border down the middle of each gap.

Colors and Opacity

The ways you can define colors have greatly expanded with CSS3, and you can now also use CSS functions to apply colors in the common formats RGB (red, green, and blue), RGBA (red, green, blue, and alpha), HSL (hue, saturation, and luminance), and HSLA (hue, saturation, luminance, and alpha). The alpha value specifies a color's transparency, which allows underlying elements to show through.

HSL Colors

To define a color with the `hsl` function, you must first choose a value for the hue between `0` and `359` from a color wheel. Any higher color numbers simply wrap around to the beginning again, so the value of `0` is red, and so are the values `360` and `720`.

In a color wheel, the primary colors of red, green, and blue are separated by `120` degrees, so pure red is `0`, green is `120`, and blue is `240`. The numbers between these values represent shades comprising different proportions of the primary colors on either side.

Next you need the saturation level, which is a value between `0` and `100%`. This specifies how washed out or vibrant a color will appear. The saturation values commence in the center of the wheel with a mid-gray color (a saturation of `0%`) and then become more vivid as they progress to the outer edge (a saturation of `100%`).

All that's left then is for you to decide how bright you want the color to be, by choosing a luminance value of between `0` and `100%`. A value of `50%` for the luminance gives the fullest, brightest color. Decreasing the value (down to a minimum of `0%`) darkens the color until it displays as black, and increasing the value (up to a maximum of `100%`) lightens the color until it shows as white. You can visualize this as if you are mixing levels of either black or white into the color.

Therefore, for example, to choose a fully saturated yellow color with standard percent brightness, you would use a declaration like this:

```
color : hsl(60, 100%, 50%);
```

Or, for a darker blue color, you would use a declaration like this:

```
color : hsl(240, 100%, 40%);
```

You can also use this (and all other CSS color functions) with any property that expects a color, such as `background-color` and so on.

HSLA Colors

To provide even further control over how colors appear, you can use the `hsla` function, supplying it with a fourth (alpha) level for a color, which is a floating-point value between 0 and 1. A value of 0 specifies that the color is totally transparent, while 1 means it is fully opaque.

Here's how you would choose a fully saturated yellow color with standard brightness and 30% opacity:

```
color : hsla(60, 100%, 50%, 0.3);
```

Or, for a fully saturated but lighter blue color with 82% opacity, you could use this declaration:

```
color : hsla(240, 100%, 60%, 0.82);
```

RGB Colors

You probably are more familiar with the RGB system of selecting a color, as it's similar to the `#nnnnnn` and `#nnn` color formats. For example, to apply a yellow color to a property, you can use either of the following declarations (the first supporting 16 million colors, and the second supporting 4,000):

```
color : #ffff00;  
color : #ff0;
```

You can also use the CSS `rgb` function to achieve the same result but with decimal numbers instead of hexadecimal (where 255 decimal is `ff` hexadecimal):

```
color : rgb(255, 255, 0);
```

But even better than that, you don't even have to think in amounts of up to 256 anymore, because you can specify percentage values, like this:

```
color : rgb(100%, 100%, 0);
```

In fact, you can now get very close to a desired color by thinking about its primary colors. For example, green and blue make cyan, so to create a color close to cyan, but with more blue in it than green, you could make a good first guess at 0% red, 40% green, and 60% blue and try a declaration like this:

```
color : rgb(0%, 40%, 60%);
```


RGBA Colors

As with the `hsla` function, the `rgba` function supports a fourth alpha argument, so you can, for example, apply the previous cyan-like color with an opacity of 40% by using a declaration such as this:

```
color : rgba(0%, 40%, 60%, 0.4);
```

The opacity Property

The `opacity` property provides the same alpha control as the `hsla` and `rgba` functions but lets you modify an object's opacity (or transparency, if you prefer) separately from its color, as shown in [Figure 18-7](#).

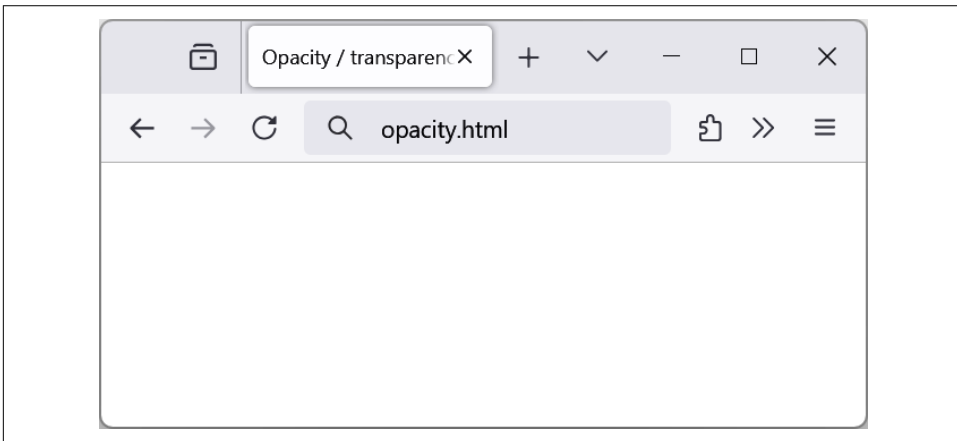


Figure 18-7. A fully transparent picture with `opacity` set to 0

To use it, apply a declaration such as the following to an element (which in this example sets the opacity to 25%, or 75% transparent):

```
opacity : 0.25;
```

Text Effects

A number of new effects can now be applied to text with the help of CSS3, including text shadows, text overlapping, and word wrapping.

The text-shadow Property

The `text-shadow` property is similar to the `box-shadow` property and takes the same set of arguments: a horizontal and vertical offset, an amount for the blurring, and the color to use. For example, the following declaration offsets the shadow by 3 pixels

both horizontally and vertically and displays the shadow in dark gray, with a blurring of 4 pixels:

```
text-shadow : 3px 3px 4px #444;
```

The result of this declaration looks like [Figure 18-8](#) and works in all recent versions of all major browsers (but not IE 9 or lower).

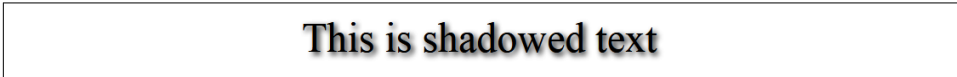


Figure 18-8. Applying a shadow to text

The text-overflow Property

When using any of the CSS overflow properties with a value of `hidden`, you can also use the `text-overflow` property to place an ellipsis (three dots) just before the cutoff to indicate that some text has been truncated, like this:

```
text-overflow : ellipsis;
```

Without this property, when the text “To be, or not to be. That is the question.” is truncated, the result will look like [Figure 18-9](#); with the declaration applied, however, the result looks like [Figure 18-10](#).

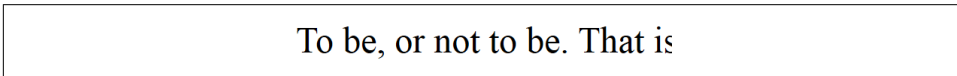


Figure 18-9. Text is automatically truncated

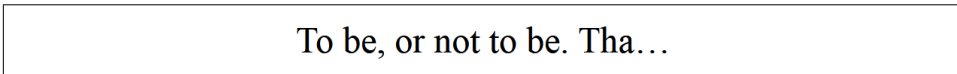


Figure 18-10. Instead of being cut off, text trails off using an ellipsis

For this to work, three things are required:

- The element should have an overflow property that is not visible, such as `overflow:hidden`.
- The element must have the `white-space:nowrap` property set to constrain the text.
- The width of the element must be less than that of the text to truncate.

The word-wrap Property

When you have a really long word that is wider than the element containing it, it will either overflow or be truncated. But as an alternative to using the `text-overflow` property and truncating text, you can use the `word-wrap` property with a value of `break-word` to wrap long lines, like this:

```
word-wrap : break-word;
```

For example, in [Figure 18-11](#) the word *Honorificabilitudinitatibus* is too wide for the containing box (whose righthand edge is shown as a solid vertical line between the letters *t* and *a*), and, because no overflow properties have been applied, it has overflowed its bounds.

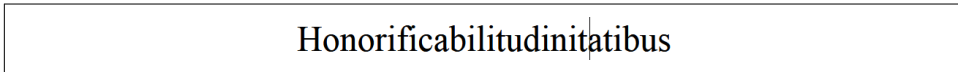


Figure 18-11. The word is too wide for its container and has overflowed

But in [Figure 18-12](#), the `word-wrap` property of the element has been assigned a value of `break-word`, so the word has neatly wrapped around to the next line.

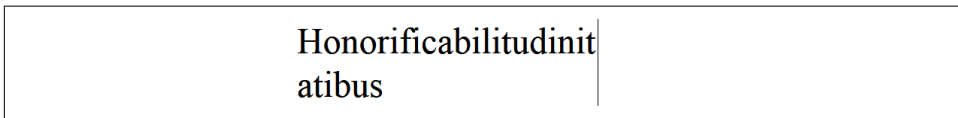


Figure 18-12. The word now wraps at the righthand edge

Web Fonts

The use of CSS web fonts vastly increases the typography available to web designers by allowing fonts to be loaded in and displayed from across the web, not just from the user's computer. To achieve this, declare a web font by using `@font-face`, like this:

```
@font-face
{
  font-family : FontName;
  src        : url('FontName.otf');
}
```

The `url` function requires a value containing the path or URL of a font. You can use either TrueType (*.ttf*) or OpenType (*.otf*) fonts.

To tell the browser the type of font, you can use the `format` function, like this for OpenType fonts:

```
@font-face
{
  font-family : FontName;
  src         : url('FontName.otf') format('opentype');
}
```

Or this for TrueType fonts:

```
@font-face
{
  font-family : FontName;
  src         : url('FontName.ttf') format('truetype');
}
```

However, because Internet Explorer accepts only EOT fonts, it ignores `@font-face` declarations that contain the `format` function.

Google Web Fonts

One of the neatest ways to use web fonts is to load them for free from Google's servers. To find out more about this, check out the [Google Fonts Website](#), seen in [Figure 18-13](#), where you can access well over one thousand fonts.



Privacy Issues Using a Third-Party Site

If you link to a Google-hosted font, be aware of the privacy and data collection that comes with using the tool; especially if your website is used in Europe you may need to obtain user consent. Consult your legal advisor and the [Google Fonts Privacy and Data Collection document](#). You can also download the selected font and serve the files from your server if you're not sure whether a Google-hosted font is the right choice for you and your users.

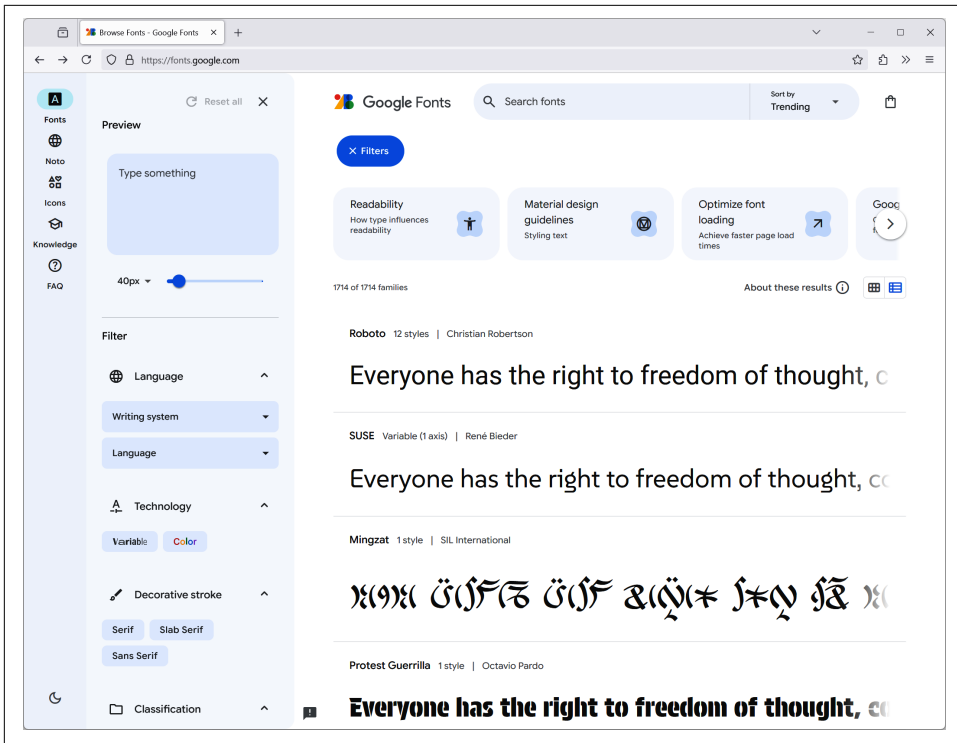


Figure 18-13. Some of Google's web fonts

To show you how easy it is to use one of these fonts, here's how to load a Google font (in this case, Lobster) into your HTML for use in `<h1>` headings:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 { font-family:'Lobster', arial, serif; }
    </style>
    <link href='http://fonts.googleapis.com/css?family=Lobster'
          rel='stylesheet'>
  </head>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

When you select a font from the website, Google provides the `<link>` tag to copy and paste into the `<head>` of your web page.

Transformations

Using transformations, you can skew, rotate, stretch, and squash elements in any of up to three dimensions. This makes it easy to create great effects by stepping out of the uniform rectangular layout of `<div>` and other elements, because now they can be shown at a variety of angles and in many different forms.

To perform a transformation, use the `transform` property. You can apply various properties to the `transform` property, starting with the value `none`, which resets an object to a nontransformed state:

```
transform: none;
```

You can supply one or more of the following functions to the `transform` property:

`matrix`

Transforms an object by applying a matrix of values to it

`translate`

Moves an element's origin

`scale`

Scales an object

`rotate`

Rotates an object

`skew`

Skews an object

The only one of these that might cause you to scratch your head is `skew`. With this function, one coordinate is displaced in one direction in proportion to its distance from a coordinate plane or axis. So, a rectangle, for example, is transformed into a parallelogram when skewed.

There are also single versions of many of these functions, such as `translateX`, `scaleY`, and so on.

So, for example, to rotate an element clockwise by 45 degrees, you could apply this declaration to it:

```
transform : rotate(45deg);
```

At the same time, you could enlarge this object, as in the following declaration, which enlarges its width by 1.5 times and its height by 2 times and then performs the rotation. **Figure 18-14** shows an object before and after the transformations are applied:

```
transform : scale(1.5, 2) rotate(45deg);
```

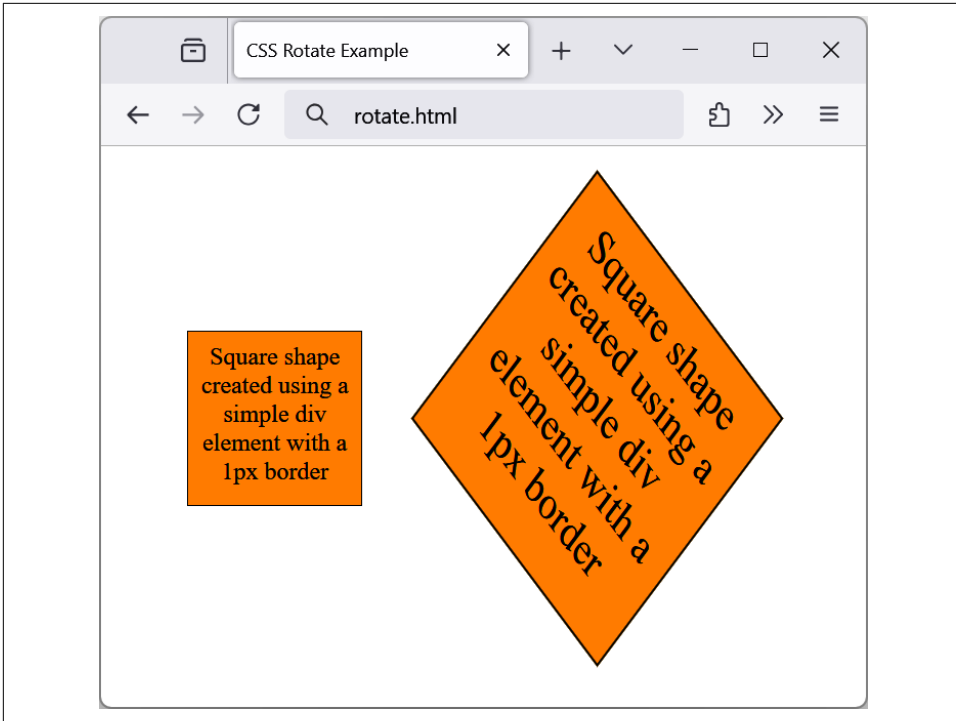


Figure 18-14. An object before and after transformation

You can also transform objects in three dimensions by using the following CSS 3D transformation features:

perspective

Releases an element from 2D space and creates a third dimension within that it can move. Required to work with 3D CSS functions.

transform-origin

Exploits perspective, setting the location at which all lines converge to a single point.

translate3d

Moves an element to another location in its 3D space.

scale3d

Rescales one or more dimensions.

rotate3d

Rotates an element around any of the x-, y-, and z-axes.

Figure 18-15 shows a 2D object that has been rotated in 3D space with a CSS rule like this:

```
transform:perspective(200px) rotateX(20deg) rotateY(40deg) rotateZ(10deg);
```

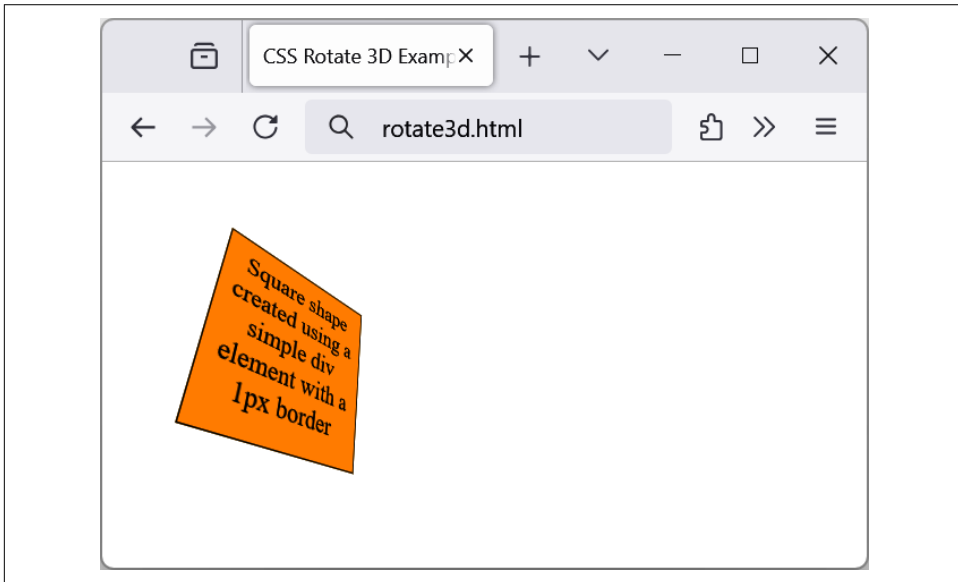


Figure 18-15. A figure rotated in 3D space

Transitions

Also appearing in all the latest major browser versions is a dynamic new feature called *transitions*. These specify an animation effect you want to occur when an element is transformed, and the browser will automatically take care of all the in-between frames for you.

The four properties to supply to set up a transition are:

```
transition-property      : property;  
transition-duration      : time;  
transition-delay         : time;  
transition-timing-function : type;
```

Properties to Transition

Transitions have properties such as height and border-color. Specify the properties you want to change in the CSS property named transition-property. (I'm using the word *property* here in two different ways: for a CSS property and for the transition properties it sets.)

You can include multiple properties by separating them with commas, like this:

```
transition-property : width, height, opacity;
```

Or if you want absolutely everything about an element to transition (including colors), use the value `all`, like this:

```
transition-property : all;
```

Transition Duration

The `transition-duration` property requires a value of 0 seconds or greater, like the following, which specifies that the transition should take 1.25 seconds to complete:

```
transition-duration : 1.25s;
```

Transition Delay

If the `transition-delay` property is given a value greater than 0 seconds (the default), it introduces a delay between the initial display of the element and the beginning of the transition. The following starts the transition after a 0.1-second delay:

```
transition-delay : 0.1s;
```

If the `transition-delay` property is given a value of less than 0 seconds (in other words, a negative value), the transition will execute the moment the property is changed but will appear to have begun execution at the specified offset, partway through its cycle.

Transition Timing

The `transition-timing` function property requires one of the following values:

`ease`

Start slowly, get faster, and then end slowly.

`linear`

Transition at constant speed.

`ease-in`

Start slowly, and then go quickly until finished.

`ease-out`

Start quickly, stay fast until near the end, and then end slowly.

`ease-in-out`

Start slowly, go fast, and then end slowly.

Using any of the values containing the word *ease* ensures that the transition looks extra fluid and natural, unlike a linear transition that seems more mechanical. And if these aren't sufficiently varied for you, you can also create your own transitions using the cubic-bezier function.

For example, here are the declarations used to create the preceding five transition types, illustrating how you can easily create your own:

```
transition-timing-function:cubic-bezier(0.25, 0.1, 0.25, 1);
transition-timing-function:cubic-bezier(0, 0, 1, 1);
transition-timing-function:cubic-bezier(0.42, 0, 1, 1);
transition-timing-function:cubic-bezier(0, 0, 0.58, 1);
transition-timing-function:cubic-bezier(0.42, 0, 0.58, 1);
```

Shorthand Syntax

You might find it easier to use the shorthand version of this property and include all the values in a single declaration like the following, which will transition all properties linearly, over a period of 0.3 seconds, after an initial (optional) delay of 0.2 seconds:

```
transition:all .3s linear .2s;
```

Doing this will save you the trouble of entering many very similar declarations, particularly if you are supporting all the major browser prefixes.

Example 18-4 illustrates how you might use transitions and transformations together. The CSS creates a square, orange element with some text in it, and a hover pseudo-class specifying that, when the mouse passes over the object, it should rotate by 180 degrees and change from orange to yellow (see [Figure 18-16](#)).

Example 18-4. A transition on hover effect

```
<!DOCTYPE html>
<html>
  <head>
    <title>Transitioning on hover</title>
    <style>
      #square {
        position      : absolute;
        top           : 50px;
        left          : 50px;
        width         : 100px;
        height        : 100px;
        padding       : 2px;
        text-align    : center;
        border-width  : 1px;
        border-style  : solid;
        background    : orange;
        transition    : all .8s ease-in-out;
```

```

    }
    #square:hover {
        background : yellow;
        transform  : rotate(180deg);
    }
</style>
</head>
<body>
    <div id='square'>
        Square shape<br>
        created using<br>
        a simple div<br>
        element with<br>
        a 1px border
    </div>
</body>
</html>

```

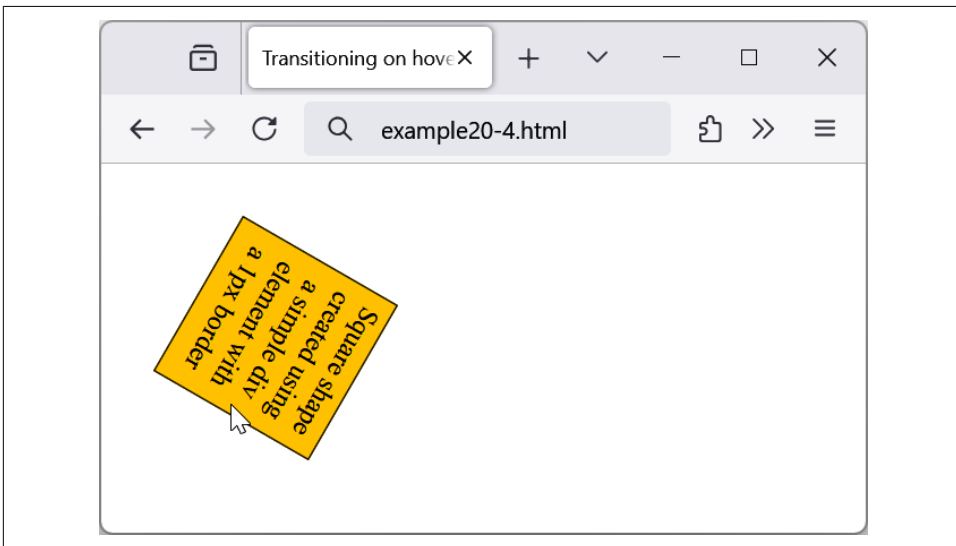


Figure 18-16. The object rotates and changes color when hovered over

The object will rotate clockwise when hovered over while slowly changing from orange to yellow.

CSS transitions are smart in that when they are canceled, they smoothly return to their original value. So, if you move the mouse away before the transition has completed, it will instantly reverse and transition back to its initial state.

Flexbox

Flexbox layout allows you to distribute space and align elements within a container element regardless of its dimensions, enabling responsive layouts that work with all viewports, by applying the value `flex` to its `display` property, like this:

```
#myelement {  
  display : flex;  
}
```

The following examples use a series of `<div>` elements, where one such element contains three `<p>` and one `<div>` element, each with a different background set, which is omitted here for brevity, but the HTML looks like this:

```
<div>  
  <p>Some long text 1 ...</p>  
  <div id="div">DIV</div>  
  <p>Longer ...</p>  
  <p>The longest ...</p>  
</div>
```

You can see the result of adding the `display: flex` property in [Figure 18-17](#).

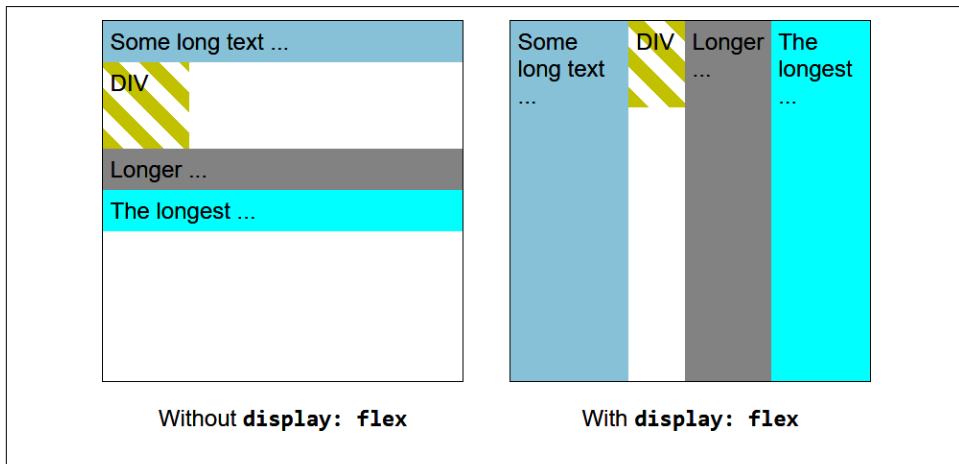


Figure 18-17. Enabling flexbox

If you're using a Chromium-based browser like Chrome or Edge, you can use the flexbox editor, which is available in developer tools and can be seen in [Figure 18-18](#). Locate the element with `display: flex` and click the editor icon next to the `display` property. A window will appear where you can set the flexbox properties and see how they change the element layout.

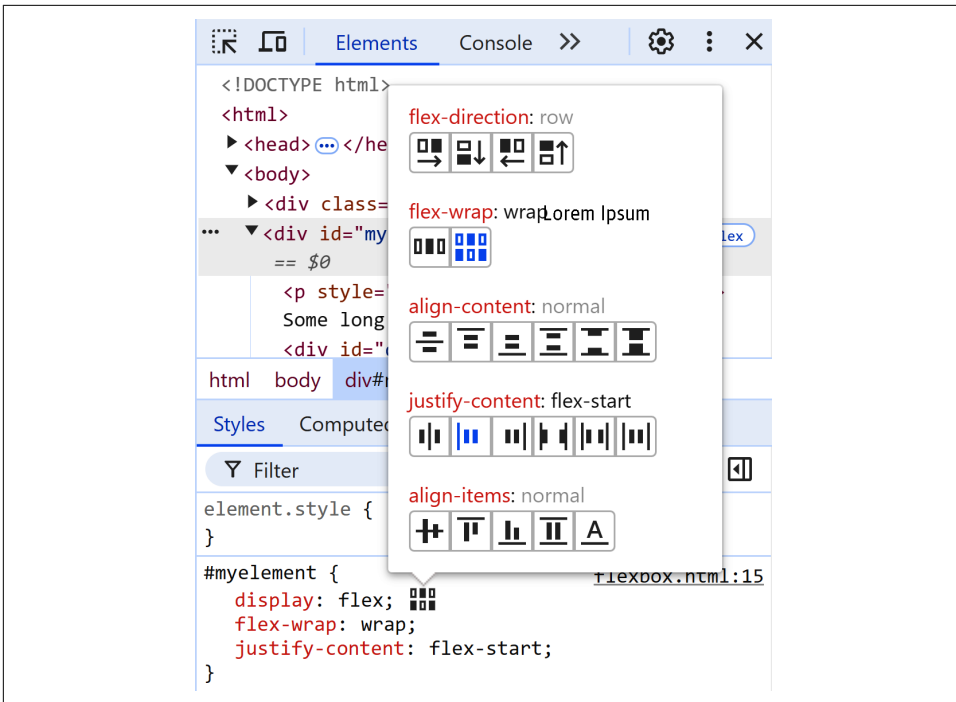


Figure 18-18. Flexbox editor in Chrome developer tools

Flex Items

Flex items are the child elements of a flex container and can be blocks, text, images, or any other HTML elements. You can apply various flex properties to them to specify how and where they should display.

Flow Direction

In a flex container, there are two primary axes or directions of flow: the main axis and the cross axis. The main axis is defined by the flex direction (either row or column), and the cross axis is perpendicular to the main axis. The `flex-direction` property sets the direction of the main axis with these supported values:

row

Items are laid out in a row from left to right (the default).

row-reverse

Items are laid out in a row from right to left.

column

Items are laid out in a column from top to bottom.

column-reverse

Items are laid out in a column from bottom to top.

The various values and the result of applying them can be seen in [Figure 18-19](#).



Figure 18-19. The *flex-direction* property

The following creates a flexbox in which items will flow from top to bottom in columns, and the columns will flow from left to right as each column fills:

```
flex-direction : column;
```

Justifying Content

The `justify-content` property determines how flex items are spaced along the main axis. It can be used to distribute space evenly, align items at the start or end of the container, or center them. Centering helps distribute free space leftover when all the flex items on a line are inflexible, or the items are flexible but have reached their maximum size. Supported values are:

`flex-start`

Items are packed toward the start (default) taking into account any reverse value.

`flex-end`

Items are packed toward the end taking into account any reverse value.

`start`

Items are packed from the start ignoring any reverse.

`end`

Items are packed to the end ignoring any reverse.

`center`

Items are centered along the axis.

`stretch`

Auto-sized items are distributed evenly.

`space-between`

Items are evenly distributed along the axis, with the first item at the start and last item at the end.

`space-around`

Items are evenly distributed along the axis with equal space around each.

`space-evenly`

Items are distributed so that the spacing between each item, and before the first and after the last item is equal.

The effect of applying some of these values to a column of items can be seen in [Figure 18-20](#).



Figure 18-20. Selected values of the *justify-content* property

The following rule equally spaces all items in rows (the default axis), with no space before the first item and none after the last item in each row:

```
justify-content : space-between;
```


Aligning Items

You can control the alignment of flex items along the cross axis of a container with the `align-items`, while `align-self` can be applied to individual elements to override the container's alignment. Supported values are:

`stretch`

Items are separated by equal spaces to fill the available space.

`flex-start`

Items are aligned flush with the cross-start edge of the line.

`flex-end`

Items are aligned flush with the cross-end edge of the line.

`baseline`

Items are aligned such that their baselines align.

`center`

Items are centered along the flow axis.

`start`

Items are aligned toward the start of the axis.

`end`

Items are aligned toward the end of the axis.

Some of the common alignment options can be seen in [Figure 18-21](#).

The following centers a container's items along the flow axis:

```
align-items : center;
```

The `align-self` property overrides the alignment of an individual flex item. The following creates a flexbox in which all its items have `center` alignment, followed by an individual item from this flexbox with its alignment overridden to `end`:

```
#myflexbox {  
  display : flex;  
  align-items : center;  
}  
#myflexitem {  
  align-self : end;  
}
```



Figure 18-21. Showing some of the `align-items` values

Aligning Content

You can use the same values supplied to `align-items` for the `align-content` property, which works similarly but acts along the opposite axis to the direction of flow. The following centers items in the default direction, stretching them along the cross axis:

```
align-items : center;
align-content : stretch;
```

See “[Flex Wrap](#)” on page 451 for an example using the `flex-wrap` property.

Resizing Items

A group of three properties control how individual flex items should expand or shrink to fit or fill the available space:

`flex-grow`

Specifies how much an item can grow relative to others. The default value is 0, which indicates the default sizing algorithm. Greater values indicate the relative amounts items can grow by to fill available free space. The higher the value the more space given.

`flex-shrink`

Determines how much an element can shrink when there's not enough space in the flexbox. The default value is 0, which indicates the default sizing algorithm. Greater values indicate the relative amounts items can shrink to fit available space. The higher the value the more that item will shrink.

`flex-basis`

Sets the initial length or height before any growth or shrinking occurs. This can be any positive CSS size or percentage, `auto`, `max-content` (preferred width), `min-content` (minimum width), `fit-content` (fit maximally), or `content` (size automatically). When omitted its value will be 0.

As a shortcut, the main flex property supports three additional values after it to set the `flex-grow`, `flex-shrink`, and `flex-basis` (in that order) of all items to the same set of values. The following will allow all elements to shrink or grow as necessary by equal amounts, starting with a size of 100px:

```
flex : 1 1 100px;
```

Flex Wrap

By default, flex items try to fit in a single line, so the `flex-wrap` property allows items to wrap onto new lines when there's not enough room. Supported values are:

`nowrap`

Forces all items into a single line, which may cause the container to overflow (the default).

`wrap`

Items break across multiple rows (or columns).

`wrap-reverse`

Same as `wrap`, but the order of items is reversed.

Figure 18-22 contains the result of wrapping items as combined with various alignment options.

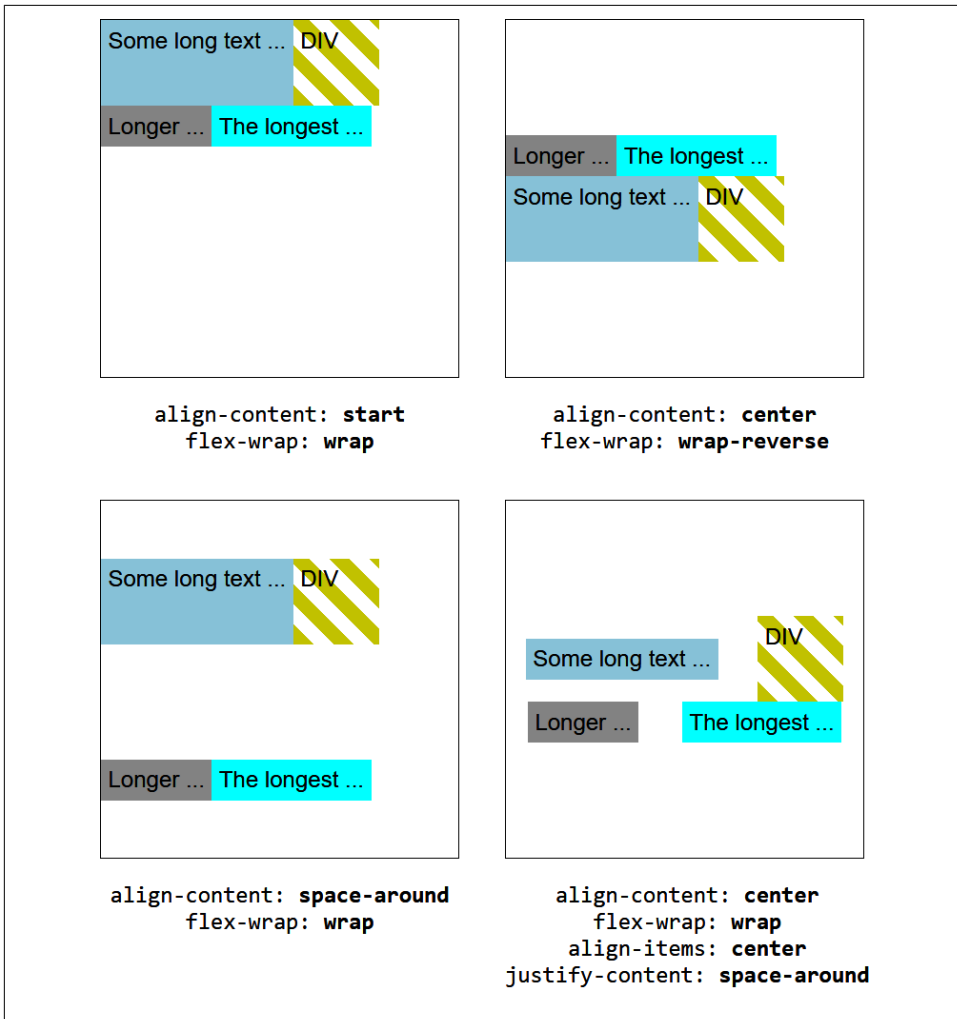


Figure 18-22. Various values of the `align-content` property with `flex-wrap` applied

The following allows items to wrap, appearing in their default order:

```
flex-wrap : wrap;
```

You can combine both `flex-direction` and `flex-wrap` in the single `flex-flow` property, as in the following, which flows a column at a time, wrapping to the next column as necessary:

```
flex-flow : column wrap;
```

Order

The `order` property lets you change the display order of individual flex items without changing their source order in the HTML. Items with lower order values appear earlier in the layout. In the following, the third item is made to appear before the first two:

```
#item1 { order: 2; }  
#item2 { order: 3; }  
#item3 { order: 1; }
```

Flex items have a default order value of 0, so items with a value greater than 0 are displayed *after* any that have not been given an explicit order value. Full details on using flexbox layout are available [in the MDN](#).

In [Figure 18-23](#) you can see how the last element will be moved to the first position when you add `order: -1` on it, and how the second item will be moved to the end when `order: 1` is applied on it.

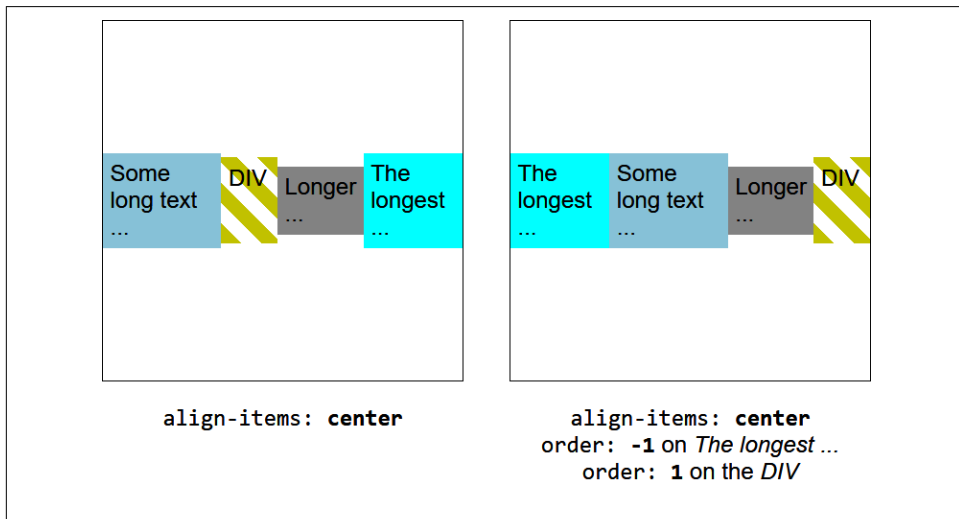


Figure 18-23. Reordered elements using the `order` CSS property

Item Gaps

To define the spacing between flex items you can use the `gap` property (you also could see the older property name `grid-gap` used extensively, but they both do the same thing), like this:

```
gap : 10px;  
grid-gap : 10px;
```

[Figure 18-24](#) shows the elements have a gap when the property is added.

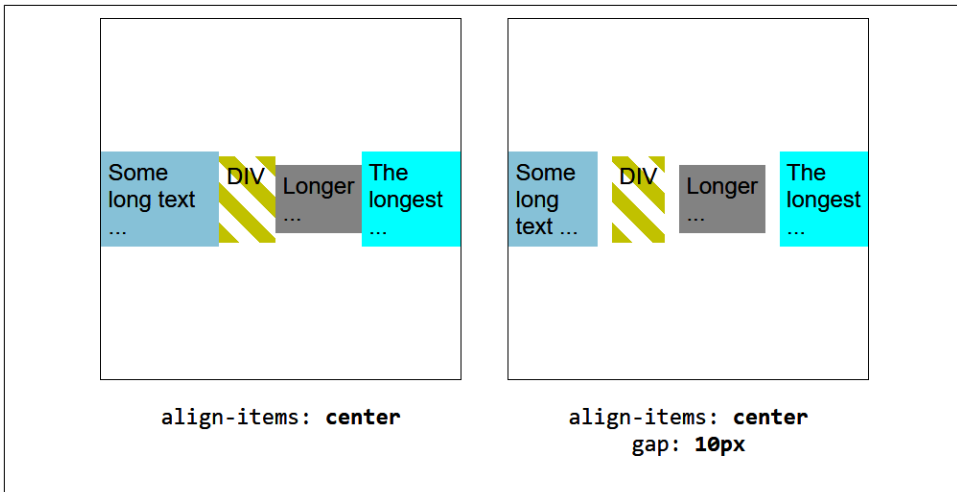


Figure 18-24. Defining the spacing between items with the `gap` property

Or you can individually set the column or row gaps with the `column-gap` and `row-gap` properties:

```
column-gap : 10px;
row-gap    : 15px;
```

Alternatively you can use `gap` as a shorthand property to set both values:

```
gap : 10px 15px;
```

CSS Grid

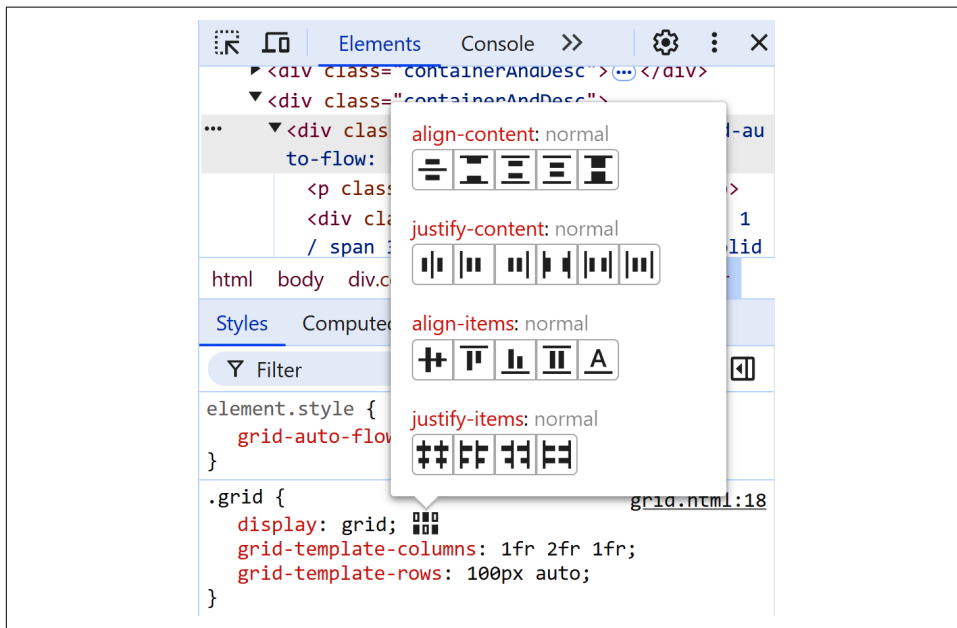
You can create two-dimensional grid layouts using CSS Grid for organizing and aligning content on a web page, providing a more flexible and precise way to design layouts compared to older methods such as floats and positioning. Each grid consists of a container and a number of grid items.

Grid Container

To create a grid layout, you first define a container element as a grid container by setting its `display` property to `grid` or `inline-grid`, like this:

```
#myelement {
  display : grid;
}
```

Similar to the flexbox editor, if you're using Chrome or a Chromium-based browser like Edge, you can locate the element in the browser developer tools and edit the properties using the grid editor, as seen in [Figure 18-25](#).



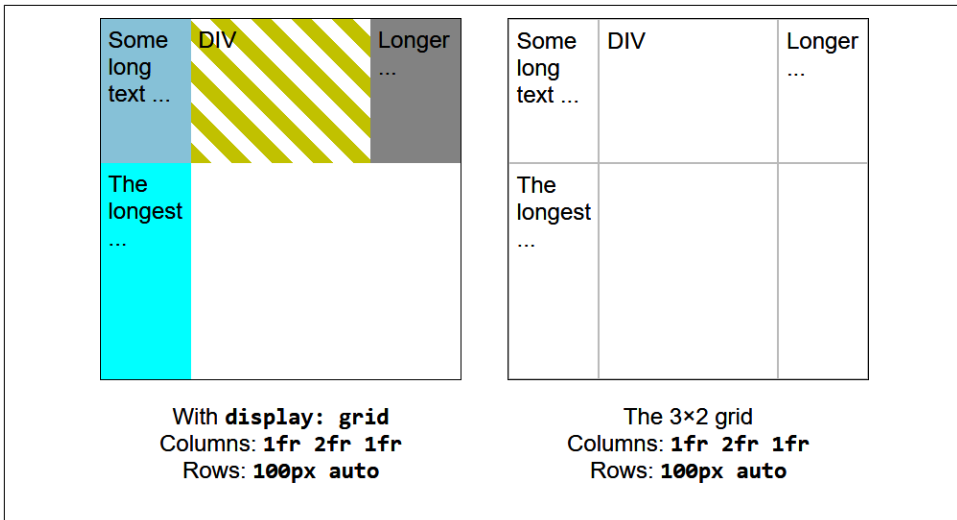


Figure 18-26. Creating a grid with the *grid-template-column* and *grid-template-rows* properties

Grid Flow

The flow direction of a grid is specified by the *grid-auto-flow* property, which supports these values:

row

Items are placed, filling each row in turn as necessary (default).

column

Items are placed, filling each column in turn as necessary.

dense

Attempts to fill in holes earlier in the grid, if smaller items come up later.

row dense

Items are placed, filling each row and filling any earlier holes.

column dense

Items are placed, filling each column and filling any earlier holes.

The difference between *row* and *column* can be seen in [Figure 18-27](#); notice how the second item *DIV* changes its position.

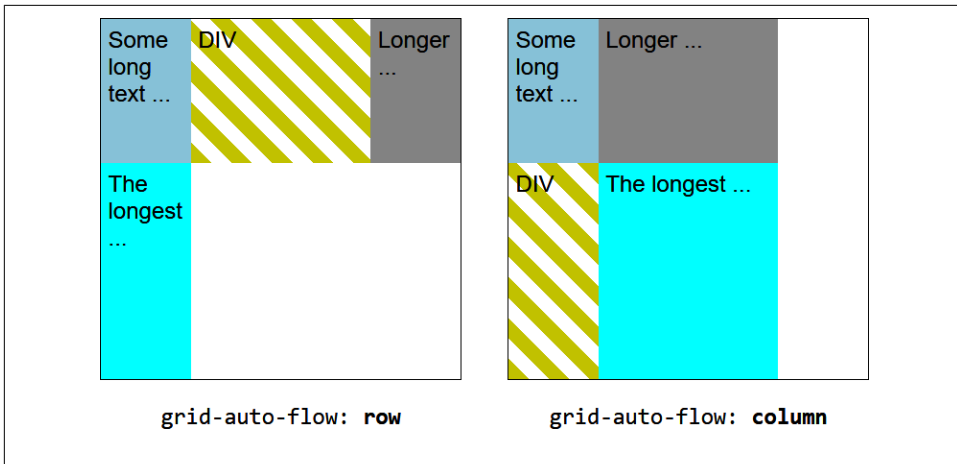


Figure 18-27. Setting the flow direction of the grid with the `grid-auto-flow` property

For example, this rule sets up a dense grid:

```
grid-auto-flow: dense;
```

Placing Grid Items

By default, all direct children of a grid container become grid items. You can have items placed automatically, or you can explicitly place them in specific grid cells using properties such as `grid-column` and `grid-row`. Figure 18-28 shows the second item placed in the first row, spanning all three columns and shifting all the other items to the second row.

The following places the item in the second column, spanning across two rows and down two columns:

```
grid-column : 2 / span 2;
grid-row    : 1 / span 2;
```

For precise item positioning you can use these four properties:

`grid-row-start`

Specifies a grid item's start position

`grid-row-end`

Specifies a grid item's end position

`grid-column-start`

Specifies a grid item's start position within the grid column

`grid-column-end`

Specifies a grid item's end position within the grid column

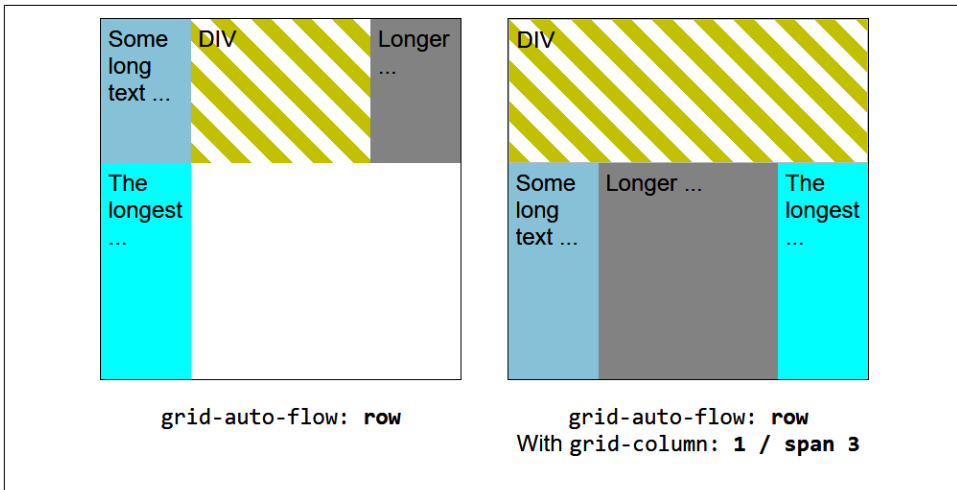


Figure 18-28. Placing the DIV into the first column, spanning all three, using the *grid-column* property

You can also combine these properties into a single `grid-area` shorthand property. For example, to make an item begin on the second row down and the second column in, and for it to extend to the fourth row down and fifth column across, you can use a rule such as this:

```
grid-area : 2 / 2 / 5 / 6;
```

Grid Gaps

To define the spacing between grid columns and rows, you can use the `gap` property (or the older alias `grid-gap`) as seen in “[Item Gaps](#)” on page 453. Figure 18-29 shows the gap applied to the row of grid items.

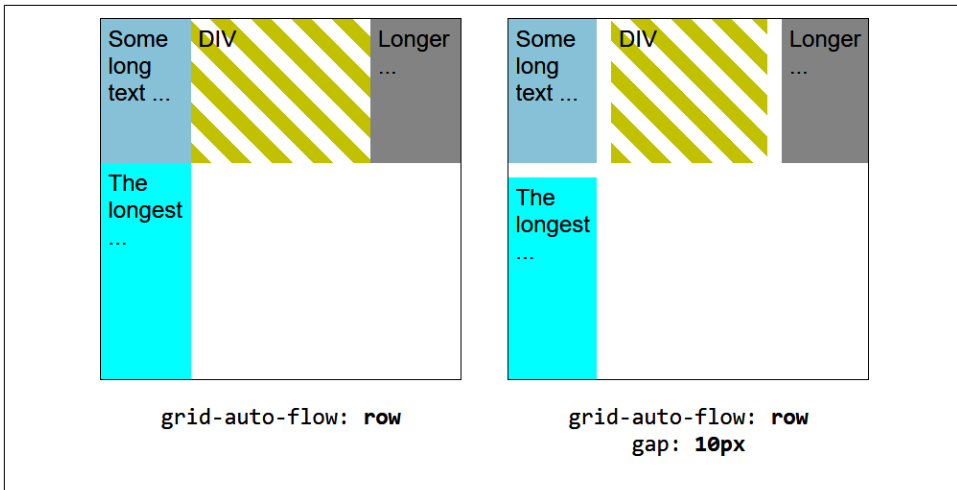


Figure 18-29. The grid item spacing set to 10 pixels with the `gap` property

Alignment

To align items both vertically and horizontally you can use the `justify-items` and `align-items` properties, which support these values:

`normal`

This is the default, as if no justification is set; it defaults to the start edge.

`start`

Items are flush to each other at the start edge of the container inline axis.

`end`

Items are flush to each other at the end edge of the container inline axis.

`center`

Items are flush to each other centered on the inline axis.

`left`

Items are flush to each other at the start edge of the container.

`right`

Items are flush to each other at the end edge of the container.

`space-between`

Items are evenly distributed along the inline axis flush to edges.

`space-around`

Items are evenly distributed along the inline axis—small edge gap.

space-evenly

Items are evenly distributed along the inline axis—full edge gap.

Figure 18-30 shows items vertically centered in two rows and horizontally centered in two columns.

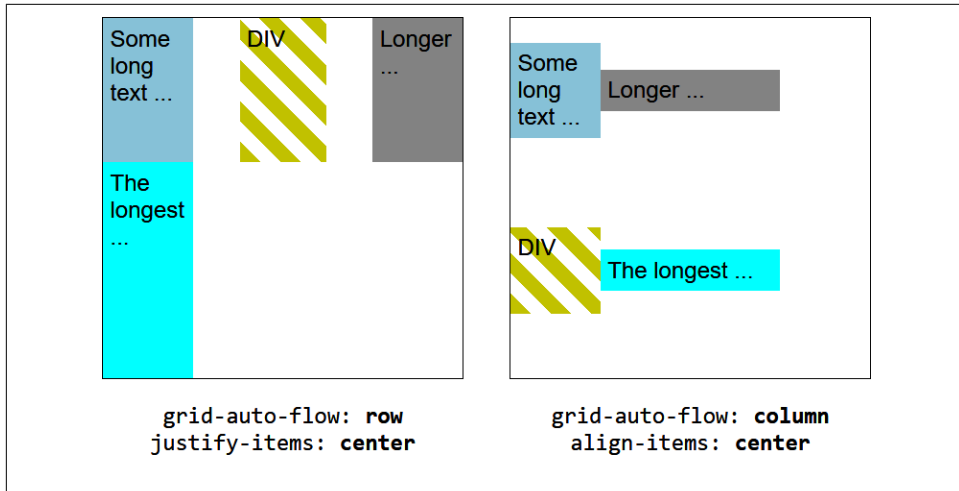


Figure 18-30. The grid items aligned with the *justify-items* and *align-items* properties

The following pair of rules will center-justify all elements of a container vertically, while left aligning them in a horizontal plane:

```
justify-items : center;  
align-items  : left;
```

Full details on using CSS grid layout are available on the [MDN website](#). In [Chapter 19](#) we'll start accessing CSS directly from JavaScript, but first try answering the following questions to test your knowledge, because it will be handy in [Chapter 19](#) as well.

Questions

1. What do the CSS attribute selector operators `^=`, `$=`, and `*=` do?
2. What property do you use to specify the size of a background image?
3. With which property can you specify the radius of a border?
4. How can you flow text over multiple columns?
5. Name the four functions you can use to specify CSS colors.
6. How would you create a gray shadow under some text, offset diagonally to the bottom right by 5 pixels, with a blurring of 3 pixels?

7. How can you indicate with an ellipsis that text is truncated?
8. How can you include a Google web font in a web page?
9. What CSS declaration would you use to rotate an object by 90 degrees?
10. How do you set up a transition on an object so that when any of its properties are changed, the change will transition linearly in a half-second?
11. How do you create a flexbox container?
12. How do you define how flex items are spaced along the main axis of a flexbox?
13. How can you control the alignment of flex items along the cross axis of a container?
14. Which two properties determine how much a flexbox element can grow and shrink?
15. How can you change the order of elements in a flexbox container?
16. How do you set up a CSS grid layout?
17. How do you specify the flow direction of a grid container?
18. Which two properties can you use to place items into a grid?
19. Which property allows you to set the gap spacing of a grid, and what is the alternate older name of this property?
20. Which two properties can you use to align grid items vertically and horizontally?

See “[Chapter 18 Answers](#)” on page 581 in the [Appendix](#) for the answers to these questions.

Accessing CSS from JavaScript

Now that you have a good understanding of the DOM and CSS, you'll learn in this chapter how to access both the DOM and CSS directly from JavaScript, enabling you to create highly dynamic, responsive websites.

I'll also show you how to use time-based events so you can create animations or provide any code that must continue running (such as a clock). Finally, I'll explain how you can add new elements to or remove existing ones from the DOM so you don't have to precreate elements in HTML just in case JavaScript might need to access them later.

Revisiting the `getElementById` Function

To help with the examples in the rest of this book, I will provide an abbreviated version of the `getElementById` function (which returns an element object when passed an ID name). This will allow for handling DOM elements and CSS styles quickly and efficiently, without needing to include a framework such as jQuery.

I've selected a name that is short to type yet it still explains what the function does: it returns an object represented by the ID passed to the function when called.



It is highly unlikely you will use the following functions in development code, because you likely will have a custom-made or third-party framework to provide this functionality, plus a whole lot more. But they serve to keep the examples in this book short and easy to follow, as well as being a simple example of how new JavaScript functions can be added.

The byId Function

You can see the bare-bones `byId` function in [Example 19-1](#). The abbreviated version alone saves 19 characters of typing each time it's called.

Example 19-1. The `byId` function

```
function byId(id)
{
    return document.getElementById(id)
}
```

The style Function

The partner function, called `style`, gives you easy access to the style (or CSS) properties of an object, and is shown in [Example 19-2](#).

Example 19-2. The `style` function

```
function style(selector)
{
    return document.querySelector(selector).style
}
```

This function performs the task of returning the `style` property (or subobject) of the element referred to. Because it uses the `document.querySelector` function which uses a CSS selector to find the element to return, you can pass an ID (for example `#myobject`), class name (for example `.myclass`), or any other valid CSS selector. If the document has more than one element that matches, for example multiple elements with `class="myclass"`, the `document.querySelector` function, and the `style` function, it returns only the first element.

Let's look at what's going on here by taking a `<div>` element with the ID of `myobj` and setting its text color to green, like this:

```
<div id='myobj'>Some text</div>

<script>
    byId('myobj').style.color = 'green'
</script>
```

The preceding code will do the job, but it's much simpler to call the new `style` function, like this:

```
style('#myobj').color = 'green'
```

Remember that you have to prefix the ID with `#`. Otherwise, the `style` function will be looking for a `<myobj>` element, which is not what you want.

The by Function

So far I've provided two simple functions that make it easy for you to access any element on a web page and any style property of an element. Sometimes, though, you will want to access more than one element at a time. You can do this by assigning a CSS class name to each such element, like in these examples, both of which employ the class `myclass`:

```
<div class='myclass'>Div contents</div>
<p class='myclass'>Paragraph contents</p>
```

If you want to access all elements on a page that use a particular class, you can use the `by` function, shown in [Example 19-3](#), which uses the `document.querySelectorAll` function to return an array containing all the objects that match the *selector* provided. Similar to the `document.querySelector` function, the selector must be a valid CSS selector and can be a class name prefixed with a dot (like `.myclass`), or a tag name (like `td`).

Example 19-3. The `by` function

```
function by(selector)
{
    return document.querySelectorAll(selector)
}
```

To use this function, simply call it as follows, saving the returned array so that you can access each of the elements individually as required or (more likely the case) en masse via a loop:

```
myarray = by('.myclass')
```

Now you can do whatever you like with the objects returned, such as (for example) setting their `textDecoration` style property to `underline`:

```
for (i = 0 ; i < myarray.length ; ++i)
    myarray[i].style.textDecoration = 'underline'
```

This code iterates through the objects in `myarray[]` and then each one's style property, setting its `textDecoration` property to `underline`.

Including the Functions

I use the `byId` and `style` functions in the examples for the remainder of this chapter, as they make the code shorter and easier to follow. Therefore, I have saved them in the file *functions.js* (along with the `by` function, which I think you'll find extremely useful) in the [Chapter 19](#) folder of the accompanying archive of examples, freely downloadable from the [book's example repository](#).

You can include these functions in any web page by using the following statement, preferably in its `<head>` section, anywhere before any script that relies on calling them:

```
<script src='functions.js'></script>
```

The contents of *functions.js* are shown in [Example 19-4](#).

Example 19-4. The functions.js file

```
function byId(id)
{
    return document.getElementById(id)
}

function style(selector)
{
    return document.querySelector(selector).style
}

function by(selector)
{
    return document.querySelectorAll(selector)
}
```

Accessing CSS Properties from JavaScript

The `textDecoration` property I used in an earlier example represents a CSS property normally hyphenated like this: `text-decoration`. But since JavaScript reserves the hyphen character for use as a mathematical operator, whenever you access a hyphenated CSS property, you must convert the name to follow the camelCase notation, that is, to omit the hyphen and set the character immediately following it to uppercase.

Another example of this is the `font-size` property, which is referenced in JavaScript as `fontSize` when placed after a period operator, like this:

```
myobject.fontSize = '16pt'
```

An alternative is to be more long-winded and use the `setAttribute` function, which *does* support (and in fact requires) standard CSS property names, like this:

```
myobject.setAttribute('style', 'font-size:16pt')
```

Some Common Properties

Using JavaScript, you can modify any property of any element in a web document, similar to using CSS. I've already shown you how to access CSS properties using either the JavaScript short form or the `setAttribute` function to use exact CSS property names, so I won't detail all of the hundreds of properties. Rather, I'd like to

show you how to access just a few of the CSS properties as an overview of some of the things you can do.

First, then, let's look at modifying a few CSS properties from JavaScript using [Example 19-5](#), which loads in the three earlier functions, creates a `<div>` element, and then issues JavaScript statements within a `<script>` section of HTML to modify some of its attributes (see [Figure 19-1](#)).

Example 19-5. Accessing CSS properties from JavaScript

```
<!DOCTYPE html>
<html>
  <head>
    <title>Accessing CSS Properties</title>
    <script src='functions.js'></script>
  </head>
  <body>
    <div id='object'>Div Object</div>

    <script>
      const o = style('#object')
      o.border    = 'solid 1px red'
      o.width     = '100px'
      o.height    = '100px'
      o.background = '#eee'
      o.color     = 'blue'
      o.fontSize  = '15pt'
      o.fontFamily = 'Helvetica'
      o.fontStyle  = 'italic'
    </script>
  </body>
</html>
```

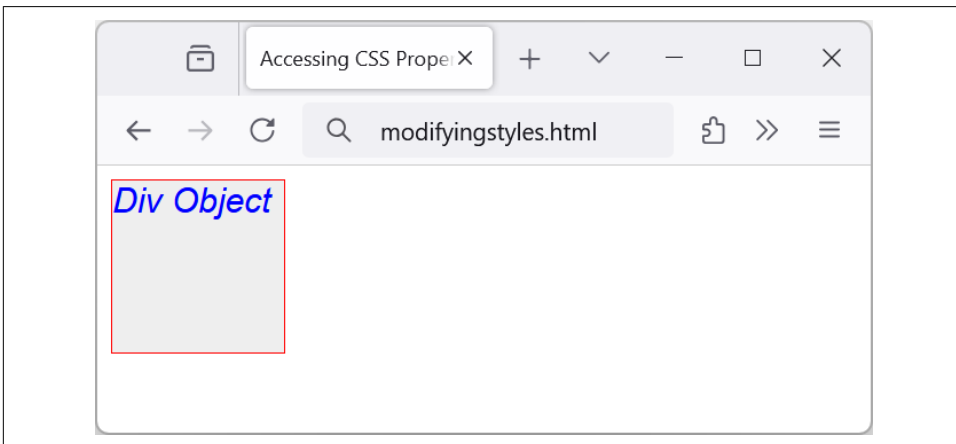


Figure 19-1. Modifying styles from JavaScript

You gain nothing by modifying properties like this, because you could just as easily have included some CSS directly, but shortly we'll be modifying properties in response to user interaction—and then you'll see the real power of combining JavaScript and CSS.

Other Properties

JavaScript also opens up access to a very wide range of other properties, such as the width and height of the browser window and in-browser windows or frames, plus handy information such as the parent window (if there is one) and the history of URLs visited in a session.

All these properties are accessed from the window object via the period operator (for example, `window.name`). [Table 19-1](#) lists some of them and their descriptions.

Table 19-1. Some of the window properties

Property	Description
<code>closed</code>	Returns a Boolean value indicating whether or not a window has been closed
<code>document</code>	Returns the document object for the window
<code>frameElement</code>	Returns the <code>iframe</code> element in which the current window is inserted
<code>frames</code>	Returns an array of all the frames and iframes in the window
<code>history</code>	Returns the history object for the window
<code>innerHeight</code>	Sets or returns the inner height of a window's content area
<code>innerWidth</code>	Sets or returns the inner width of a window's content area
<code>length</code>	Returns the number of frames and iframes in a window
<code>localStorage</code>	Allows saving of key/value pairs in a web browser
<code>location</code>	Returns the location object for the window
<code>name</code>	Sets or returns the name of a window
<code>navigator</code>	Returns the navigator object for the window
<code>opener</code>	Returns a reference to the window that created the window
<code>outerHeight</code>	Sets or returns the outer height of a window, including toolbars and scroll bars
<code>outerWidth</code>	Sets or returns the outer width of a window, including toolbars and scroll bars
<code>pageXOffset</code>	Returns the number of pixels the document has been scrolled horizontally from the left of the window
<code>pageYOffset</code>	Returns the number of pixels the document has been scrolled vertically from the top of the window
<code>parent</code>	Returns the parent window of a window
<code>screen</code>	Returns the screen object for the window
<code>screenLeft</code>	Returns the x coordinate of the window relative to the screen
<code>screenTop</code>	Returns the y coordinate of the window relative to the screen
<code>screenX</code>	Returns the x coordinate of the window relative to the screen
<code>screenY</code>	Returns the y coordinate of the window relative to the screen
<code>sessionStorage</code>	Allows saving of key/value pairs in a web browser

Property	Description
<code>self</code>	Returns the current window
<code>top</code>	Returns the top browser window

There are a few points to note about some of these properties:

- The `history` object cannot be read from (so you cannot see where your visitors have been surfing). But it supports the `length` property to determine how long the history is, and the `back`, `forward`, and `go` methods to navigate to specific pages in the history.
- When you need to know how much space is available in a current window of the web browser, just read the values in `window.innerHeight` and `window.innerWidth`. I've often used these values for centering in-browser alert or "confirm dialog" windows; today you can also use CSS Grid or flexbox to achieve the same positioning.
- The `screen` object supports the read-only properties `availHeight`, `availWidth`, `colorDepth`, `height`, `pixelDepth`, and `width` and is therefore great for determining information about the user's display.



Many of these properties are invaluable when you're targeting mobile phones and tablet devices, as they will tell you exactly how much screen space you have to work with, the type of browser being used, and more.

These few items of information will get you started and provide you with an idea of the many new and interesting things you can do with JavaScript. Far more properties and methods are available than can be covered in this chapter, but now that you know how to access and use properties, all you need is a resource listing them all. I recommend that you check out [the online docs](#) as a good starting point.

Inline JavaScript

Using `<script>` tags isn't the only way you can execute JavaScript statements; you can also access JavaScript from within HTML tags, which makes for great dynamic interactivity. For example, to add a quick effect when the mouse passes over an object, you can use code such as that in the `<img>` tag in [Example 19-6](#), which displays an apple by default but replaces it with an orange when the mouse passes over the object and restores the apple when the mouse leaves. Note the properties have a prefix before the event name, so for example when I want to add a handler for the `mouseover` event, I'll have to use the `onmouseover` property.

Example 19-6. Using inline JavaScript

```
<!DOCTYPE html>
<html>
  <head>
    <title>Inline JavaScript</title>
  </head>
  <body>
    <img src='apple.png'
      onmouseover="this.src='orange.png'"
      onmouseout="this.src='apple.png'">
  </body>
</html>
```

The this Keyword

In the preceding example, you see the `this` keyword in use. It tells JavaScript to operate on the calling object, namely the `<img>` tag. You can see the result in [Figure 19-2](#), where the mouse has yet to pass over the apple.

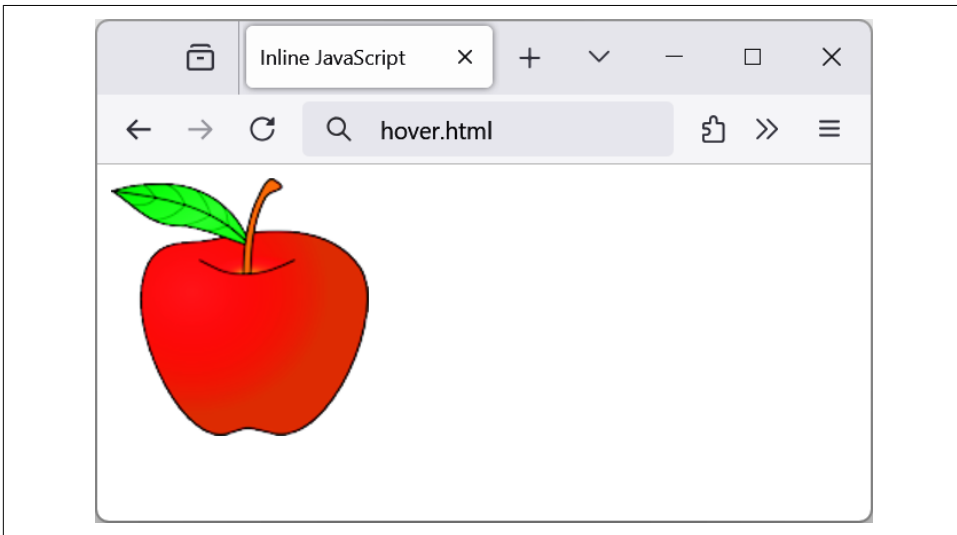


Figure 19-2. Inline mouse hover JavaScript example

Attaching Events to Objects in a Script

The preceding code is the equivalent of providing an ID to the `<img>` tag and then attaching the actions to the tag's mouse events, like in [Example 19-7](#).

Example 19-7. Noninline JavaScript

```
<!DOCTYPE html>
<html>
  <head>
    <title>Non-inline JavaScript</title>
    <script src='functions.js'></script>
  </head>
  <body>
    <img id='object' src='apple.png'>

    <script>
      const o = byId('object')
      o.onmouseover = function() { this.src = 'orange.png' }
      o.onmouseout = function() { this.src = 'apple.png' }
    </script>
  </body>
</html>
```

In the HTML section, this example gives the `<img>` element an ID of `object`, and it then proceeds to manipulate it separately in the JavaScript section by attaching anonymous functions to each event. The `on` property prefix is also used here.

Attaching to Other Events

Whether you're using inline or separate JavaScript, you can attach actions to several events, providing a wealth of additional features to offer your users. [Table 19-2](#) lists some common events and details when they will be triggered. Note that some events can be triggered on multiple different elements and on multiple occasions. See the MDN for a [full list](#).

Table 19-2. Events, elements they trigger on, and when

Event	Element	Occurs for example
abort	HTMLMediaElement	When a video's loading is stopped before completion
blur	Element	When an element loses focus ^a
change	HTMLElement	When any part of a form has changed
click	Element	When an object is clicked
dblclick	Element	When an object is double-clicked
error	Window	When a JavaScript error is encountered
focus	Element	When an element gets focus
keydown	Element	When a key is being pressed (including Shift, Alt, Ctrl, and Esc)
keypress	Element	When a key is being pressed (not including Shift, Alt, Ctrl, and Esc)
keyup	Element	When a key is released
load	HTMLMediaElement	When an object has loaded
mousedown	Element	When the mouse button is pressed over an element

Event	Element	Occurs for example
mousemove	Element	When the mouse is moved over an element
mouseout	Element	When the mouse leaves an element
mouseover	Element	When the mouse passes over an element from outside it
mouseup	Element	When the mouse button is released
reset	HTMLFormElement	When a form is reset
resize	Window	When the browser is resized
scroll	Document	When the document is scrolled
select	HTMLInputElement	When some text is selected
submit	HTMLFormElement	When a form is submitted

^a An element that has *focus* is one that has been clicked or otherwise entered into, such as an input field.



Make sure you attach events to objects that make sense. For example, an object that is not a form will not respond to the `onsubmit` event.

Adding New Elements

With JavaScript, you are not limited to manipulating the elements and objects supplied to a document in its HTML. In fact, you can create objects at will by inserting them into the DOM.

For example, suppose you need a new `<div>` element. [Example 19-8](#) shows one way you can add it to a web page.

Example 19-8. Inserting an element into the DOM

```
<!DOCTYPE html>
<head>
  <title>Adding Elements</title>
</head>
<body>
  <p>This is a document with only this text in it.</p>

  <script>
    alert('Click OK to add an element')

    const newdiv = document.createElement('div')
    newdiv.id = 'NewDiv'
    document.body.appendChild(newdiv)

    newdiv.style.border = 'solid 1px red'
    newdiv.style.width = '100px'
    newdiv.style.height = '100px'
```



```

newdiv.innerText = "I'm a new object inserted in the DOM"

setTimeout(function()
{
    alert('Click OK to remove the element')

    newdiv.parentNode.removeChild(newdiv)
}, 1000)
</script>
</body>
</html>

```

Figure 19-3 shows this code being used to add a new `<div>` element to a web document. First, the new element is created with `createElement`; then the `appendChild` function is called, and the element gets inserted into the DOM.

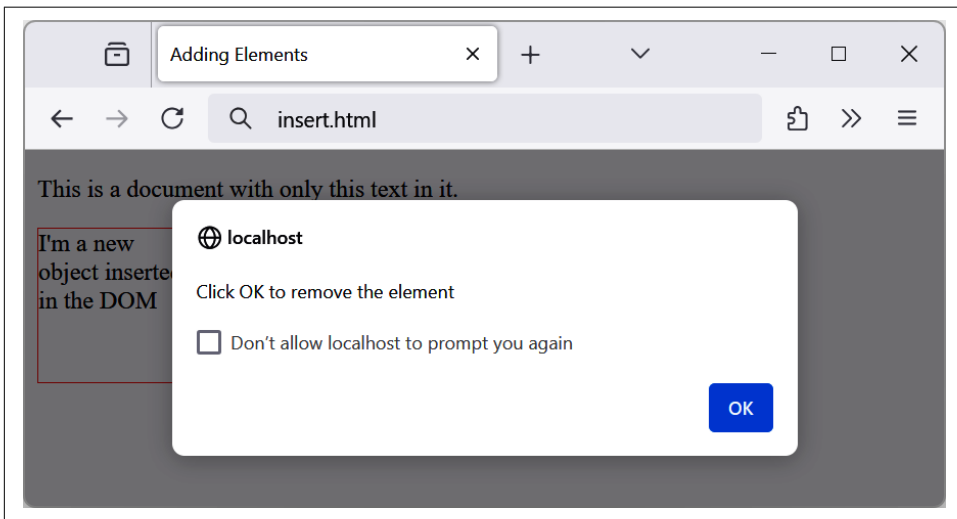


Figure 19-3. Inserting a new element into the DOM

After this, various properties are assigned to the element, including some text for its inner content. And then, to make sure the new element is instantly revealed, a timeout is set to trigger one second in the future, delaying the running of the remaining code to give the DOM time to update and display, before popping up the alert about removing the element again. See [“Using setTimeout” on page 475](#) for more on creating and using timeouts.

This newly created element is exactly the same as if it had been included in the original HTML and has all the same properties and methods available.



This is an alternative to adding an initially hidden `<div>` to your HTML for this purpose, and it may be a better option if you display multiple modals or display them infrequently.

Removing Elements

You can also remove elements from the DOM, including ones that you didn't insert using JavaScript; it's even easier than adding an element. It works like this, assuming the element to remove is in the object `element`:

```
element.parentNode.removeChild(element)
```

This code accesses the element's `parentNode` object so that it can remove the element from that node. Then it calls the `removeChild` method on that parent object, passing the object to be removed.

Alternatives to Adding and Removing Elements

Inserting an element is intended for adding totally new objects into a web page. But if all you intend to do is hide and reveal objects according to a `mouseover` or other event, don't forget there are a couple of CSS properties you can use for this purpose, without taking such drastic measures as creating and deleting DOM elements.

There are two ways to hide and unhide an object: one uses `visibility`, while the other uses the `display` property. When you want to make an element invisible but leave it in place (and with all the elements surrounding it remaining in their positions), you can simply set the object's `visibility` property to `hidden`, like this:

```
myobject.visibility = 'hidden'
```

And to redisplay the object, you can use:

```
myobject.visibility = 'visible'
```

With the `display` property, you can also collapse an element to occupy zero width and height (with all the objects around it filling in the freed-up space), like this:

```
myobject.display = 'none'
```

To restore the element to its original dimensions, you would use:

```
myobject.display = 'block' // or for example 'flex' or 'grid'
```

And, of course, there's always the `innerHTML` property, with which you can change the HTML applied to an element, like this, for example:

```
myelement.innerHTML = '<b>Replacement HTML</b>'
```

Or to use the `byId` function outlined earlier:

```
byId('someid').innerHTML = 'New contents'
```

Or you can make an element seem to disappear, like this:

```
byId('someid').innerHTML = ''
```



Don't forget the other useful CSS properties you can access from JavaScript, such as `opacity` for setting the visibility of an object to somewhere between visible and invisible, or `width` and `height` for resizing an object. And, of course, using the `position` property with values of `absolute`, `static`, `fixed`, `sticky`, or `relative`, you can even locate an object anywhere in (or outside) the browser window that you like.

Time-based Events

JavaScript provides access to time-based events by which you can ask the browser to call your code after a set period of time, or even to keep calling it at specified intervals. This gives you a means of handling background tasks such as asynchronous communications or even things like animating web elements.

There are two types: `setTimeout` and `setInterval`, which have accompanying `clearTimeout` and `clearInterval` functions for canceling them.

Using `setTimeout`

When you call `setTimeout`, you pass it a function and a value in milliseconds representing how long to wait before the code should be executed, like this:

```
setTimeout(dothis, 5000)
```

Your `dothis` function might look like:

```
function dothis()
{
  alert('This is your wakeup alert!');
}
```

Be aware that you need to pass a function to the `setTimeout` call, not the result of calling that function. Consider a code like this; you can also run it in browser console for example:

```
setTimeout(alert('Hello'), 5000)
```

The alert pop-up will appear immediately, not after 5 seconds like you'd probably expect. The reason is that `alert('Hello')` is executed immediately and the return value of the `alert` call is passed to `setTimeout` to be executed after the specified timeout, but because `alert` doesn't return anything, nothing will be executed after the 5-second interval.

This is why the previous example uses `setTimeout(dothis, 5000)`, without parentheses after `dothis`, and not `setTimeout(dothis(), 5000)`. Only when you provide a function name without parentheses will its code be executed when the timeout occurs.

Passing an arrow function

You don't need to pass only a named function. You can also provide an anonymous arrow function to the `setTimeout` function, which will not be executed until the correct time. For example:

```
setTimeout(() => alert('Hello!'), 5000)
```

In fact, you can provide as many lines of JavaScript code as you need in that arrow function, like this:

```
setTimeout(() => {  
  console.log('Starting');  
  alert('Hello!')  
}, 5000)
```



Don't Pass a String

You can also pass a string value to the `setTimeout` function, but this has been discouraged for many years as it presents an unnecessary security risk and incurs a performance penalty. Always pass a named or an arrow function, as shown in the previous examples.

Canceling a timeout

Once a timeout has been set up, you can cancel it if you previously saved the value returned from the initial call to `setTimeout`, like this:

```
timeoutID = setTimeout(dothis, 5000)
```

Armed with the value in `timeoutID` (sometimes called a *handle*), you can cancel the execution at any point until its due time:

```
clearTimeout(timeoutID)
```

When you do this, the timeout identifier is completely forgotten, and the code assigned to it will not get executed.

Using `setInterval`

An easy way to set up regular execution is to use the `setInterval` function. It works in the same way as `setTimeout`, except that having executed after the interval you specify in milliseconds, it will do so again after that interval again passes, and so on forever, until you cancel it.

Example 19-9 uses this function to display a simple clock in the browser, as shown in Figure 19-4.

Example 19-9. A clock created using `setInterval`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Using setInterval</title>
    <script src='functions.js'></script>
  </head>
  <body>
    The time is: <span id='time'>...</span><br>

    <script>
      setInterval(() => showtime(byId('time')), 1000)

      function showtime(object)
      {
        const date = new Date()
        object.innerText = date.toTimeString().substr(0,8)
      }
    </script>
  </body>
</html>
```

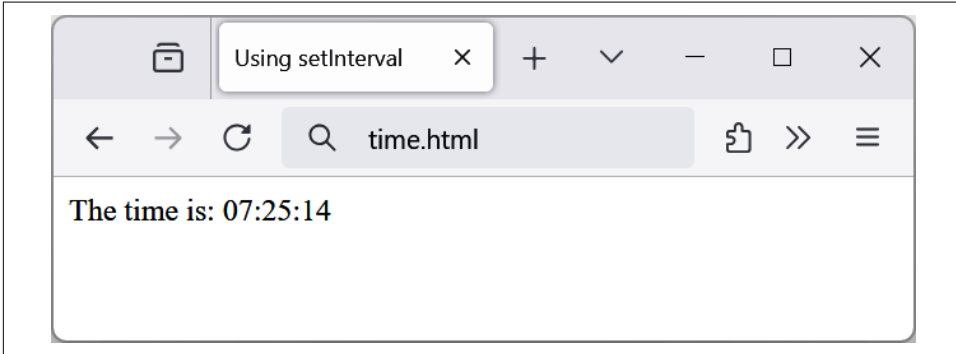


Figure 19-4. Maintaining the correct time with `setInterval`

Every time the arrow function is called, it sets the object date to the current date and time with a call to `Date`:

```
var date = new Date()
```

Then the `innerText` property of the object passed to `showtime` (namely, `object`) is set to the current time in hours, minutes, and seconds, as determined by a call to the

function `toTimeString`. This returns a string such as `09:57:17 UTC+0530`, which is then truncated to just the first eight characters with a call to the `substr` function:

```
object.innerText = date.toTimeString().substr(0,8)
```

Using the function

To use this function, you first have to create an object whose `innerText` property will be used for displaying the time, like with this HTML:

```
The time is: <span id='time'>...</span>
```

The value `...` is simply there to show where and how the time will display. It is not necessary as it will be replaced anyway. Then, from a `<script>` section of code, call the `setInterval` function, like this:

```
setInterval(() => showtime(byId('time')), 1000)
```

The script then passes an arrow function to `setInterval` containing the following statement, which is set to execute once a second (every 1,000 milliseconds):

```
showtime(byId('time'))
```

In the rare situation where somebody has disabled JavaScript (which people sometimes do for security reasons), your JavaScript will not run, and the user will just see the original `...` placeholder.

Canceling an interval

To stop a repeating interval, when you first set up the interval with a call to the function `setInterval`, you must note the interval's identifier (sometimes also called a *handle*), like this:

```
intervalID = setInterval(() => showtime(byId('time')), 1000)
```

You can stop the clock at any time by issuing this call:

```
clearInterval(intervalID)
```

You can even set up a timer to stop the clock after a certain amount of time, like this:

```
setTimeout(() => clearInterval(intervalID), 10000)
```

This statement will execute the code in 10 seconds that will clear the repeating intervals.

Using Time-Based Events for Animation

By combining a few CSS properties with a repeating code execution, you can produce all manner of animations and effects.

For example, the code in [Example 19-10](#) moves a square shape across the top of the browser window, all the time ballooning in size, as shown in [Figure 19-5](#); when `left` is reset to 0, the animation restarts.

Example 19-10. A simple animation

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Animation</title>
    <script src='functions.js'></script>
    <style>
      #box {
        position : absolute;
        background : orange;
        border    : 1px solid red;
      }
    </style>
  </head>
  <body>
    <div id='box'></div>

    <script>
      let size = 0
      let left = 0

      setInterval(animate, 30)

      function animate()
      {
        size += 10
        left += 3

        if (size === 200) size = 0
        if (left === 600) left = 0

        const b = style('#box')
        b.width  = size + 'px'
        b.height = size + 'px'
        b.left   = left + 'px'
      }
    </script>
  </body>
</html>
```

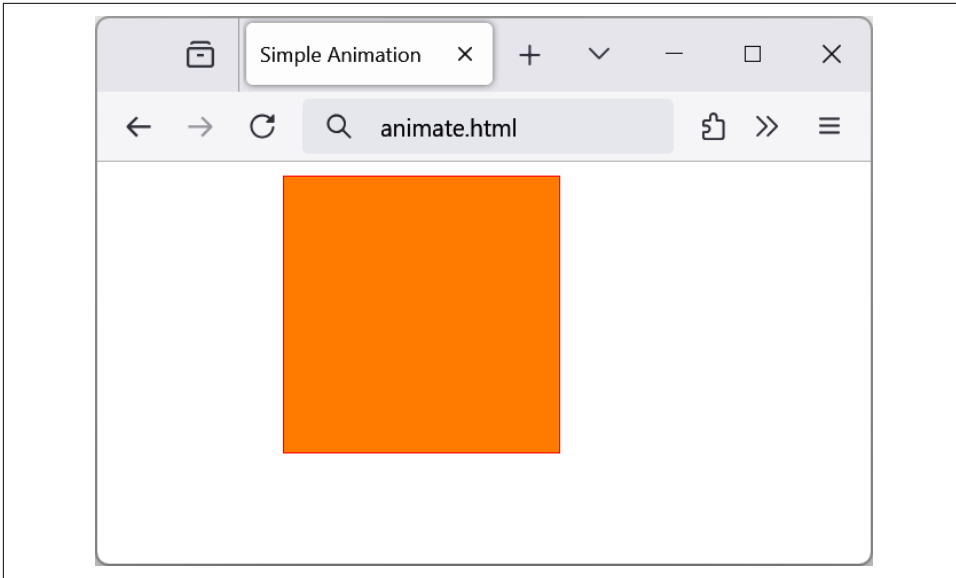


Figure 19-5. This object slides in from the left while changing size

In the `<head>` section of the document, the box object is set to a background color of orange with a border value of `1px solid red`, and its `position` property is set to `absolute` so that the animation code that follows can position it precisely.

Then, in the `animate` function, the global variables `size` and `left` are continuously updated and applied to the `width`, `height`, and `left` style attributes of the box object (with `'px'` added after each to specify that the values are in pixels), thus animating it at a frequency of once every 30 milliseconds. This results in an animation rate of 33.33 frames per second (1,000/30 milliseconds).

At this point you should be able to use JavaScript to manipulate the document and the CSS to create interactive and dynamic websites. In [Chapter 20](#) we'll introduce React, a framework that takes all this a step further. Before moving forward, try answering the following questions to refresh what you've learned in this chapter.

Questions

1. Write a function that abbreviates DOM element access by the object ID, using one of the two built-in methods.
2. Name two ways to modify a CSS attribute of an object.
3. Which properties provide the width and height available in a browser window?

4. How can you make something happen when the mouse passes both over and out of an object?
5. Which JavaScript function creates new elements, and which appends them to the DOM?
6. How can you make an element (a) invisible and (b) collapse to zero dimensions?
7. Which function creates a single event at a future time?
8. Which function sets up repeating events at set intervals?
9. What is the value of the `position` CSS property you can use to release an element from its location in a web page to enable it to be moved around?
10. What delay between events should you set (in milliseconds) to achieve an animation rate of 50 frames per second?

See “[Chapter 19 Answers](#)” on page 583 in the [Appendix](#) for the answers to these questions.

Introduction to React

When using JavaScript, HTML, and CSS to build dynamic websites, there comes a time when the creation of the code required to handle the frontend of your websites and apps can become cumbersome and overly verbose, slowing the speed of project development and potentially introducing common bugs.

This is where frameworks come in. Of course, since 2006 there's been jQuery to help us out, and consequently it's still installed on many production websites, although these days JavaScript has grown sufficiently in scope and flexibility that programmers need to rely on frameworks like jQuery a lot less. Also, the technology continually improves, and now there are a number of excellent options, such as Angular and, as discussed here, my preferred favorite, React.

jQuery was designed to simplify HTML DOM tree traversal and manipulation, as well as event handling, CSS animation, and Ajax, but some programmers, such as the development team at Google, felt it still wasn't powerful enough, and they came up with Angular JS in 2010, which evolved into Angular in 2016, and which overtook jQuery around 2019.

Angular uses a hierarchy of components as its primary architectural characteristic. Google's massive AdWords platform is powered by Angular, as are Forbes, Autodesk, Indiegogo, UPS, and many others, and it is indeed extremely powerful.

Facebook had a different vision and came up with React (also known as React JS) as its framework for the development of single-page or mobile applications, basing it around the JSX extension (which stands for JavaScript XML). The React Library (first developed in 2012) divides a web page into single components, simplifying the development of the interface required to serve all of Facebook's advertising and more, and it is now widely adopted by platforms across the web, such as Dropbox, Cloudflare, Airbnb, Netflix, the BBC, PayPal, and many more household names.

Clearly, both Angular and React were driven in their creation and design by solid commercial decisions and were built to handle complex and more sophisticated websites, where it was felt that jQuery simply did not have the oomph the developers were looking for. Interestingly, another contender, the Vue framework (released in 2014), is still around and used almost as much as Angular, so you may also encounter it on certain projects.

Which is the best choice to learn more fully? **Google Trends shows React to be way ahead** of all the others in popularity and still growing (see **Figure 20-1**), while the other main frameworks have all peaked. Therefore React is covered in this book. By the way, please don't confuse the similarly named ReactPHP with React for JavaScript, as it is an entirely separate and unconnected project.

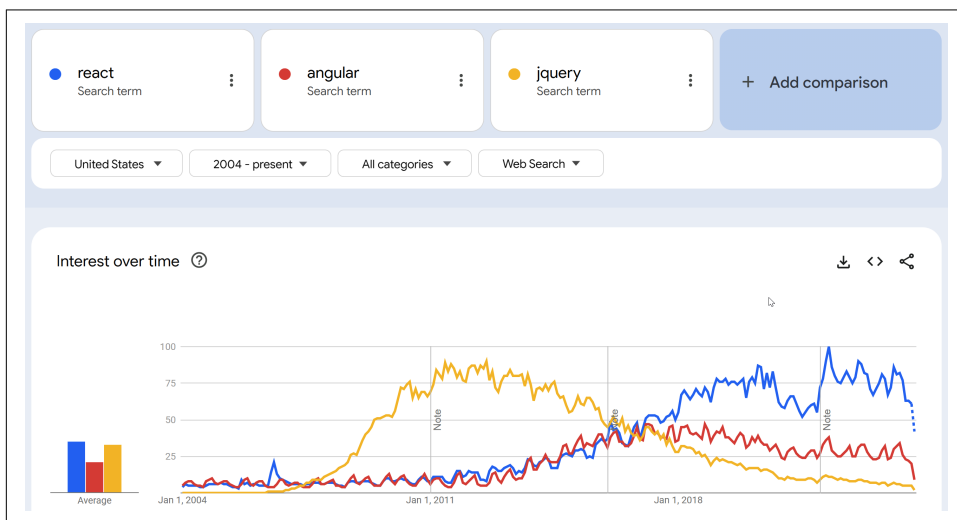


*Figure 20-1. The popularity of React compared to other frameworks 2004 to 2024 (see a larger version of this figure in color **online**)*

What Is the Point of React Anyway?

React allows developers to create large web applications that can easily handle and change data, without reloading the web page that still reflects the application's current state, even if it changes. Its main *raison d'être* are component-based architecture, scalability, and simplicity in handling the view layer of single-page web and mobile applications. It also enables the creation of reusable UI components and uses a virtual DOM to manage updates of the real DOM. Some people say you can use it as the V in the MVC (Model, View, Controller) architecture that separates applications into three components.

Instead of developers having to come up with various ways to describe transactions on interfaces, they can simply describe the interfaces in terms of a final state, such that when transactions happen to that state, React updates the UI for you. The net results are faster and less buggy development, speed, reliability, and scalability. Because React is a library and not a framework, learning it is also quick, with just a few functions to master. After that, it's all down to your JavaScript skills.

The power of a framework such as React often becomes evident only after the project gets bigger. For light projects, React may not always be the best choice, especially if you are already comfortable using jQuery (or another framework), for example. One reason is the extra lines of code needed for set up. But as soon as a project requires massive scaling, with code that many developers can instantly comprehend and work on collaboratively, and with tried, tested, and debugged modules ready to quickly import, then a framework/library such as React can become invaluable.

Accessing the React Files

React is open source and entirely free to use, and there are a number of services on the web that will serve up the latest (or any) version for you free of charge, so using it can be as easy as placing a couple of extra lines of code in your web page.

Before examining what you can do with React and how to use it, here's how you include it in a web page, pulling the files from *unpkg.com*:

```
<script
  src="https://unpkg.com/react/umd/react.development.js">
</script>
<script
  src="https://unpkg.com/react-dom/umd/react-dom.development.js">
</script>
```

Ideally, these lines should be placed within the `<head>...</head>` section of a page to ensure they are loaded before the body section. They load in the development versions of React and React DOM (a package used to access and modify the real DOM) to aid you with development and debugging. On a production website, you should replace the word `development` with `production` in these URLs, and, to speed up transfer, you can even change `development` to `production.min`, which will call up compressed versions of the files, like this:

```
<script
  src="https://unpkg.com/react/umd/react.production.min.js">
</script>
<script
  src="https://unpkg.com/react-dom/umd/react-dom.production.min.js">
</script>
```

For ease of access and to make the code as brief as possible, I have downloaded the latest (version 18 as I write) of the uncompressed development files to the

accompanying archive of examples for this book (on [GitHub](#)) so that all the examples will load locally and look like this:

```
<script src="react.development.js"></script>
<script src="react-dom.development.js"></script>
```

Now that React is available to your code, we pull in the Babel JSX extension, which allows you to include XML text directly within JavaScript, making your life much easier.



Using React Without JSX

JSX is technically not necessary for React development, but unless there is a very specific reason not to use JSX, this is not recommended and would be an advanced approach.

Including babel.js

The Babel JSX extension adds the ability for you to use XML (very similar to HTML) directly within your JavaScript, saving you from having to call a function each time. In addition, on browsers that have earlier versions of ECMAScript (the official standard of JavaScript) than 6, Babel upgrades them to handle ES6 syntax, so it provides two great benefits in one go.

Once again you can pull the file needed from the *unpkg.com* server, like this:

```
<script src="https://unpkg.com/babel-standalone/babel.min.js"></script>
```

You require only the one minimized version of the Babel code on either a development or a production server. For convenience I have also downloaded the latest version to the companion archive of example files, so examples in this book load locally and look like this:

```
<script src="babel.min.js"></script>
```

Now that we can access the React files, let's get on with doing something with them.



This chapter is intended to teach you the basics of using React to give you a clear understanding of how and why it works and to provide you with a good starting point to take your React development further. Indeed, some of the examples in this chapter are based on (or similar to) examples you can find in the official documentation at the [React website](#) so that, should you wish to learn React in greater depth, you can visit the website and will be off to a running start. You can also browse other titles on React available on the [O'Reilly Learning Platform](#).

Our First React Project

Rather than teaching you all about React and JSX before actually setting about coding, let's approach it by jumping right into our first React project, as shown in [Example 20-1](#), the result of which is to simply display the text “By Jeeves, it works!” in the browser.

Example 20-1. Our first React project

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>First React Project</title>

    <script src="react.development.js"></script>
    <script src="react-dom.development.js"></script>
    <script src="babel.min.js"></script>

    <script type="text/babel">
      function One() {
        return <p>By Jeeves, it works!</p>
      }

      ReactDOM.render(<One />, document.getElementById('div1'))
    </script>
  </head>
  <body>
    <div id="div1" style='font-family:monospace'></div>
  </body>
</html>
```

This is a standard HTML document, which loads in the two React scripts and the Babel script before opening an inline script. Here is where we first need to pay attention because, instead of not specifying a type to the script tag or using `type="application/javascript"`, the tag is given `type="text/babel"`. The browser itself will ignore the tag because it doesn't support the type. But the Babel preprocessor will run through the script, add ES6 functionality to it if necessary, and replace any XML encountered with JavaScript function calls. Only then the contents of the script, which is now a common JavaScript, will be executed.

Within the script, a new function, `One`, is created, which returns the following JSX (not a string, it's not enclosed in quotes):

```
<p>By Jeeves, it works!</p>
```

Finally, within the script, the `render` function of the `ReactDOM` class is called, passing it the name of the function that returns the JSX and the element in the body of the

document, which has been given the ID of `div1`. The result is to render the JSX into the div, which causes the browser to automatically update and display the contents, which looks like this:

By Jeeves, it works!

Immediately you should see how including the JSX within the JavaScript makes for code that is much easier and faster to write as well as easier to understand. Without the JSX extension, you would have to do all this using a sequence of JavaScript function calls.



React treats components starting with lowercase letters as DOM tags. Therefore, for example, `<div />` represents an HTML `<div>` tag, but `<One />` represents a component and requires `One` to be in scope—you cannot use `one` (with a lowercase `o`) in the previous example and expect your code to work, as the component needs to start with uppercase and so does any reference to it.

Using a Class Instead of a Function

You may encounter existing or legacy code that uses classes instead of functions as in [Example 20-2](#); however, classes are not recommended for new development. The main reasons to use functions are simplicity, ease of use, and faster development.

Example 20-2. Using a class instead of a function

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>First React Project</title>

    <script src="react.development.js"></script>
    <script src="react-dom.development.js"></script>
    <script src="babel.min.js"></script>

    <script type="text/babel">
      class Two extends React.Component
      {
        render()
        {
          return <p>And this, by Jove!</p>
        }
      }

      ReactDOM.render(<Two />, document.getElementById('div2'))
    </script>
  </head>
```



```

<body>
  <div id="div2" style='font-family:monospace'></div>
</body>
</html>

```

The class, named `Two`, extends `React.Component` and contains a method called `render` that returns the JSX. The result displayed in the browser is:

And this, by Jove!

Examples with Only the Script and Relevant Elements

In the examples that follow, for the sake of brevity and simplicity, I will show only the contents of the Babel script and the body of the document as if they were both in the body (which works just the same), but the examples in the companion archive *will* be complete. So they will look like this from now on:

```

<script type="text/babel">
  function One() {
    return <p>By Jeeves, it works!</p>
  }

  ReactDOM.render(<One/>,
    document.getElementById('div1'))
</script>

<div id="div1"></div>

```

Pure and Impure Code: A Golden Rule

When you write a normal JavaScript function, it is possible to write either *pure* or *impure* code. Pure function code does not change its inputs, doesn't modify global variables, and doesn't have any other side effects, as in the following, which returns a value calculated from its arguments:

```

function mult(m1, m2) {
  return m1 * m2
}

```

However, the following function is considered impure because it modifies an argument and absolutely should not be used as, or turned into, a React component:

```

function assign(obj, val) {
  obj.value = val
}

```

Expressed as a golden rule, all React components must act like pure functions with respect to their props, as explained in “[Props and Components](#)” on page 490.

Unfortunately, React has no mechanism to prevent you from using impure functions as components. So be careful; otherwise, the function may not work as designed.

Using Both a Class and a Function

You can, of course, use functions and classes pretty much interchangeably, as in [Example 20-3](#).

Example 20-3. Using both classes and functions

```
<script type="text/babel">
  function One() {
    return <p>By Jeeves, it works!</p>
  }

  class Two extends React.Component
  {
    render()
    {
      return <p>And this, by Jove!</p>
    }
  }

  ReactDOM.render(<One />, document.getElementById('div1'))
  ReactDOM.render(<Two />, document.getElementById('div2'))
</script>

<div id="div1" style="font-family:monospace"></div>
<div id="div2" style="font-family:monospace"></div>
```

Here there is a function named `One` and class named `Two`, which are the same as in the previous two examples. The result of running this displays in the browser as:

By Jeeves, it works!

And this, by Jove!

Props and Components

A great way to introduce you to what React calls *props* and *components* is to build a simple welcome page in which a name is passed to the script and then displayed. [Example 20-4](#) shows one way to do that. Components let you split the UI into separate, reusable parts and work with each part in isolation. They are similar to JavaScript functions and accept arbitrary inputs that are called *props* (short for *properties*), returning React elements that describe how elements should appear in the browser.

Example 20-4. Passing props to a function

```
<script type="text/babel">
  function Welcome(props) {
    return <h1>Hello, {props.name}</h1>
  }

  ReactDOM.render(<Welcome name='Robin' />,
    document.getElementById('hello'))
</script>

<div id="hello" style='font-family:monospace'></div>
```

In this example, the `Welcome` function receives an argument of `props`, which stands for properties, and within its JSX return statement a section inside curly braces fetches the `name` property from the `props` object, like this:

```
return <h1>Hello, {props.name}</h1>
```

`props` is an object in React, and next you will see one way to populate it with a property.



Curly braces let you embed expressions within JSX. In fact, you can place any JavaScript expression inside these curly braces and it will be evaluated (with the exception of `for` and `if` statements; however, they're not expressions and cannot be evaluated).

So, in place of `props.name` in this example, you could enter `76 / 13` or `"decode".substr(-4)` (which would evaluate to the string `"code"`). In this instance, however, the property name is retrieved from the `props` object and returned.

Finally, the `render` method is passed the name of the `Welcome` function, followed by assigning the string value `'Robin'` to its `name` property:

```
ReactDOM.render(<Welcome name='Robin' />, document.getElementById('hello'))
```

React then renders the `Welcome` component (which is simply the invocation of the `Welcome` function), passing it `{name: 'Robin'}` in `props`. `Welcome` then evaluates and returns `<h1>Hello, Robin</h1>` as its result, which is then rendered into the `div` called `hello` and displayed in the browser like this:

Hello, Robin

To make your code tidier you can also, if you wish, first create an element containing the XML to pass to `render`:

```
const elem = <Welcome name='Robin' />
ReactDOM.render(elem, document.getElementById('hello'))
```

The Differences Between Using a Class and a Function

The most obvious difference between using a class and a function in React is the syntax. A function is simple JavaScript (possibly incorporating JSX), which can accept a props argument and return a component element.

A class, however, is extended from `React.Component` and requires a render method to return a component. But this additional code does come with benefits, in that a class allows you to use `setState` in your component, enabling (for example) the use of timers and other stateful features. Functions in React are called *functional stateless components*.

In essence, you can do pretty much everything in React using functions, especially if you use React *hooks*, which replace classes and come with much less additional code. You can learn about hooks in the [React documentation](#).

React State and Life Cycle

Let's assume you want a ticking clock to be displayed on your web page (an ordinary digital one for simplicity's sake). If you are using stateless code this is not an easy matter, but if you set up your code to retain its state, then the clock counter can be updated once per second and the time rendered equally as often. This is where you would use a class in React rather than a function. So let's build such a clock:

```
<script type="text/babel">
  class Clock extends React.Component
  {
    constructor(props)
    {
      super(props)
      this.state = {date: new Date()}
    }

    render()
    {
      return <span> {this.state.date.toLocaleTimeString()} </span>
    }
  }

  ReactDOM.render(<Clock />, document.getElementById('the_time'))
</script>

<p style='font-family:monospace'>The time is: <span id="the_time"></span></p>
```

This code assigns the result returned from calling the `Date` function to the `state` property of the constructor's `this` object, which is `props`. The JSX content will now be rendered whenever the class's `render` function is called, and, as long as it is rendered into the same DOM node, only a single instance of the class will be used.



Did you see the call to `super` at the start of the constructor? By passing it props, it is now possible to refer to props using the `this` keyword within the constructor, without which call you could not.

However, as things stand, the time will be displayed only once, and then the code will stop running. So now we need to set up some time-based code to keep the `date` property updated, which is done by adding a *life-cycle* method to the class by mounting a timer using `componentDidMount`, like this:

```
componentDidMount()  
{  
  this.timerID = setInterval(() => this.tick(), 1000)  
}
```

We aren't quite there yet, as we still need to write the `tick` function, but first, to explain the preceding: *mounting* is the term React uses to describe the action of adding nodes to the DOM. A class's `componentDidMount` method is always called if the component is mounted successfully, so it's the ideal place to set up a timer, and, indeed, in the preceding code, `this.timerID` is assigned the ID returned by calling the `setInterval` function, passing it the method `this.tick` to be called every 1,000 milliseconds (once per second).

When a timer is mounted, we must also provide a means for it to be *unmounted*. In this case when the DOM produced by `Clock` is removed (that is, the component is unmounted), the method and code we use to stop the timer looks like this:

```
componentWillUnmount()  
{  
  clearInterval(this.timerID)  
}
```

Here, `componentWillUnmount` is called by React when the DOM is removed; thus, this is where we place the code to clear the interval stored in `this.timerID`, which instantly stops `tick` from being called.

The last piece of the puzzle is the time-based code to be called every 1,000 milliseconds, which is in the method `tick`:

```
tick()  
{  
  this.setState({date: new Date()})  
}
```

Here the React `setState` function is called to update the value in the `state` property with the latest result of calling the `Date` function once every second.

All this code together in one example is shown in [Example 20-5](#).

Example 20-5. Building a clock in React

```
<script type="text/babel">
  class Clock extends React.Component
  {
    constructor(props)
    {
      super(props)
      this.state = {date: new Date()}
    }

    componentDidMount()
    {
      this.timerID = setInterval(() => this.tick(), 1000)
    }

    componentWillUnmount()
    {
      clearInterval(this.timerID)
    }

    tick()
    {
      this.setState({date: new Date()})
    }

    render()
    {
      return <span> {this.state.date.toLocaleTimeString()} </span>
    }
  }

  ReactDOM.render(<Clock />, document.getElementById('the_time'))
</script>

<p style='font-family:monospace'>The time is: <span id="the_time"></span></p>
```

With the Clock class now complete with a constructor, a timer starter and stopper, a method to update the state property using the timer, and a render function, all that is needed is to call the render method to get the whole thing ticking away, as smooth as clockwork! The result looks like this in the browser:

The time is: 12:17:21

The clock is automatically updated to screen each time the `setState` function is called, because components are re-rendered by this function, so you don't have to worry about doing this with your code.



After the initial state setup, `setState` is the only legitimate way to update state, because simply modifying a state directly will *not* cause a component to be re-rendered. Remember that the *only* place you can assign `this.state` is in the constructor. React can bundle multiple `setState` calls into a single update.

Events in React

In React, events are named using camelCase, and you use JSX to pass a function as the event handler. Also, React events do not work in exactly the same way as native JavaScript events, in that your handlers are passed instances of a cross-browser wrapper around the browser's native event called `syntheticEvent`. The reason for this is that React normalizes events to have consistent properties across different browsers. Should you need access to the browser event, however, you can always use the `nativeEvent` attribute to reach it.

To illustrate the use of events in React, [Example 20-6](#) shows an `onClick` event that removes or redisplay some text when clicked.

Example 20-6. Setting up an event

```
<script type="text/babel">
class Toggle extends React.Component
{
  constructor(props)
  {
    super(props)
    this.state      = {isVisible: true}
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick()
  {
    this.setState(state => ({isVisible: !state.isVisible}))
  }

  render()
  {
    const show = this.state.isVisible

    return (
      <div>
        <button onClick={this.handleClick}>
          {show ? 'HIDE' : 'DISPLAY'}
        </button>
        <p>{show ? 'Here is some text' : ''}</p>
      </div>
    )
  }
}
```

```

    }
  }

  ReactDOM.render(<Toggle />, document.getElementById('display'))
</script>

<div id="display" style="font-family:monospace"></div>

```

In the constructor of a new class called `Toggle`, an `isVisible` property is set to `true` and assigned to `this.state`:

```
this.state = {isVisible: true}
```

Then an event handler called `handleClick` is attached to `this` using the `bind` method:

```
this.handleClick = this.handleClick.bind(this)
```

With the constructor finished, the `handleClick` event handler is next. This has a single-line command to toggle the state of `isVisible` between `true` and `false`:

```
this.setState(state => ({isVisible: !state.isVisible}))
```

Last, there's the call to the `render` method, which returns two elements wrapped inside a `<div>`. It's done this way because `render` can return only a single component (or XML tag), so the two elements are wrapped into a single element to satisfy that requirement.

The elements returned are a button, which displays either the text `DISPLAY` if the following text is currently hidden (that is, `isVisible` is set to `false`) or `HIDE` if `isVisible` is set to `true` and the text is currently visible. Following this button, some text is displayed underneath if `isVisible` is `true`; otherwise, nothing is shown (in fact, an empty string is returned, which is the same thing).

To decide what button text to display, or whether or not to show the text, the ternary operator is used, which you will recall follows the syntax:

```
expression ? return this if true : or this if false.
```

This is done by the single-word expression of the variable `show` (which retrieved its value from `this.state.isVisible`). If it evaluates to `true`, then the button shows `HIDE` and the text is displayed; otherwise, the button shows `DISPLAY` and the text is not displayed. When loaded into the browser, the result looks like this (where `[HIDE]` and `[DISPLAY]` are buttons):

```
[HIDE]
```

```
Here is some text
```

When the button is pressed, it changes to just the following:

```
[DISPLAY]
```




Using JSX over Multiple Lines

Although you can split your JSX across many lines to improve readability, as in the previous example, the one thing you may not do is move the parenthesis following the return command down a line (or anywhere else). It must stay in its place following return or syntax errors will be reported. The closing parenthesis may, however, appear where you wish.

Inline JSX Conditional Statements

In JSX there is a way to return only XML if a condition is true, thus enabling conditional rendering. This is achieved because `true && expression` evaluates to `expression`, while `false && expression` evaluates to `false`.

Therefore, for example, [Example 20-7](#) sets up two variables as if they are in part of a game. `this.highScore` is set to 90, and `this.currentScore` is set to 100.

Example 20-7. A conditional JSX statement

```
<script type="text/babel">
  class Setup extends React.Component
  {
    constructor(props)
    {
      super(props)
      this.highScore = 90
      this.currentScore = 100
    }

    render()
    {
      return (
        <div>
          {
            this.currentScore > this.highScore &&
            <h1>New High Score</h1>
          }
        </div>
      )
    }
  }

  ReactDOM.render(<Setup />, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>
```

In this instance, if `this.currentScore` is greater than `this.highScore`, then the `h1` element is returned; otherwise, `false` is returned. The result of the code looks like this in the browser:

New High Score

Of course, in an actual game you would then proceed to set `this.highScore` to the value in `this.currentScore` and would probably do a few other things, too, before going back to the game code.

So, wherever something should be displayed only upon a condition being true, the `&&` operator is a great way to achieve this. And, of course, you have just seen (near the end of “Events in React” on page 495) how you can also create an `if...then...else` block in JSX using a ternary (`? :`) expression.



React will not render the literal falsy values `undefined`, `null`, or `false`, but it will render the numeric value `0` even though the value `0` is also falsy. Consider the following code that will render “non-zero” when `x` is, for example, `5`:

```
{x && <p>non-zero</p>}
```

But it will render `0` when `x` is zero, which may not be what you want. If you want React to render nothing instead of `0`, the easiest way to do that is to make sure the expression evaluates to literal `false`:

```
{x !== 0 && <p>non-zero</p>}
```

Using Lists and Keys

Displaying lists using React is a breeze. In [Example 20-8](#) the array `cats` contains a list of four types of cat. This is then extracted in the following line of code using the `map` function, which iterates through the array, returning each item in turn in the variable `cat`. This results in each iteration being embedded in a pair of `<li>...</li>` tags and then appended to the `listOfCats` string.

Example 20-8. Displaying a list

```
<script type="text/babel">
  const cats = ['lion', 'tiger', 'cheetah', 'lynx']
  const listOfCats = cats.map((cat) => <li>{cat}</li>)

  ReactDOM.render(<ul>{listOfCats}</ul>, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>
```

Finally, `ReactDOM.render` is called, embedding `listOfCats` within a pair of `<ul>...</ul>` tags; the result displays like this:

- lion
- tiger
- cheetah
- lynx

Unique Keys

If your JavaScript console was open when you ran [Example 20-8](#) (press Ctrl-Shift-J or Option-Command-J on a Mac), you may have noticed the warning message “Each child in a list should have a unique ‘key’ prop.”

Although it’s not required, React works best when you supply a unique key for each sibling list item, which helps it to find references to the appropriate DOM nodes and, when you make a small change, allows for making minor adjustments to the DOM, rather than requiring re-rendering of larger sections. [Example 20-9](#) shows how to provide such a key.

Example 20-9. Using unique keys

```
<script type="text/babel">
  const cats = ['lion', 'tiger', 'cheetah', 'lynx']
  const listOfCats = cats.map((cat, i) => <li key={i}>{cat}</li>)

  ReactDOM.render(<ul>{listOfCats}</ul>, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>
```

In this example, the arrow function passed to `map` has a second parameter called `i` in which the array index will be passed and used for the item key. The displayed output is the same as the previous example, but if you would like to see the keys (just out of interest), you can change the contents of the `li` element from `{cat}` to `{i + ' ' + cat}`, and you will see the following displayed:

- 0 lion
- 1 tiger
- 2 cheetah
- 3 lynx

But what is the point of this, you may ask? Well, consider the case of the following list structure:

```
<ul> // Cities In Europe
  <li>Birmingham</li>
  <li>Paris</li>
  <li>Milan</li>
```

```

    <li>Vienna</li>
  </ul>
  <ul> // Cities in the USA
    <li>Cincinnati</li>
    <li>Paris</li>
    <li>Chicago</li>
    <li>Birmingham</li>
  </ul>

```

Here are two sets of lists, each with four unique siblings, but between the lists, both the city names of “Birmingham” and “Paris” are shared at the same level of nesting. When React performs certain reconciliation actions (after reordering or modifying an element perhaps), there are instances when you can gain speed and possibly avoid problems when sibling list items at the same level share the same values. To do this you can provide unique keys for all siblings, which, to React, looks like this:

```

<ul> // Cities In Europe
  <li key="1">Birmingham</li>
  <li key="2">Paris</li>
  <li key="3">Milan</li>
  <li key="4">Vienna</li>
</ul>
<ul> // Cities in the USA
  <li key="5">Cincinnati</li>
  <li key="6">Paris</li>
  <li key="7">Chicago</li>
  <li key="8">Birmingham</li>
</ul>

```

Now there is no possibility of confusing Paris in Europe with Paris in the USA (or at least of having to work harder to locate and possibly re-render the correct DOM node), as each array element has a different unique ID for React.



Don’t worry too much about why you are creating these unique keys. Just remember that React works best when you do so, and a good rule to follow is that elements within a `map` call will need keys. Also, you can reuse your keys for different sets of siblings that are not related in any way. However, you may not have to create your own keys because the data you are working with could well supply them for you, such as book ISBN numbers. As a last resort, you can simply use the index of an item as its key, but reorders could be slow, and you could encounter other issues, so creating your own keys to control what they contain is usually the best option.

Handling Forms

In React, `<input type='text'>`, `<textarea>`, and `<select>` all work similarly because React's internal state becomes what is known as the “source of truth,” and these components are therefore called *controlled*.

With a *controlled component*, the input's value is always driven by the React state. This does mean that you need to write a bit more code in React, but the end benefit is that you can then pass values to other UI elements or access them from event handlers.

Normally, without React or any other framework or library loaded, form elements maintain their own state, which is updated based on input received from the user. In React, the mutable state is typically kept in the `state` property of components and should only be updated using the `setState` function.

Using Text Input

Let's look at these three input types, starting with a simple text input:

```
<form>
  Name: <input type='text' name='name'>
  <input type='submit'>
</form>
```

This code requests a string of characters to be input, which is then submitted when the submit button is clicked (or the Enter or Return key pressed). Now let's change this to a controlled React component ([Example 20-10](#)).

Example 20-10. Using text input

```
<script type="text/babel">
class GetName extends React.Component
{
  constructor(props)
  {
    super(props)
    this.state = {value: ''}
    this.onChange = this.onChange.bind(this)
    this.onSubmit = this.onSubmit.bind(this)
  }

  onChange(event)
  {
    this.setState({value: event.target.value})
  }

  onSubmit(event)
  {
    alert('You submitted: ' + this.state.value)
  }
}
```

```

        event.preventDefault()
    }

    render()
    {
        return (
            <form onSubmit={this.onSubmit}>
                <label>
                    Name:
                    <input type="text" value={this.state.value}
                        onChange={this.onChange} />
                </label>
                <input type="submit" />
            </form>
        )
    }
}

ReactDOM.render(<GetName />, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>

```

Let me take you through this part by part. First we create a new class called `GetName`, which will be used to create a form that will prompt for a name to be entered. This class contains two event handlers called `onChange` and `onSubmit`. These local handlers are set to override the standard JavaScript handlers of the same named events by using the calls to `bind` in the constructor, which is also the place the value in `value` is initialized to the empty string.

When called on the change event, the new `onChange` handler calls the `setState` function to update `value` whenever the input is changed, so that `value` is always kept up-to-date with the contents in the input field.

When the submit event is triggered, it is handled by the new `onSubmit` handler, which in this instance issues a pop-up alert window so that we can see it has worked. Because *we* are dealing with the event and *not* the system, the event is then prevented from bubbling through to the system by calling `preventDefault`.

Finally, the `render` method contains all the HTML code to be rendered into the display `<div>`. Of course, we use HTML formatted as XML to do this, as that is what Babel expects (namely JSX syntax). In this instance, it simply requires the additional self-closing of the input elements with `/>`.



We have not globally overridden the `onChange` and `onSubmit` events, because we have bound events issued by the rendered code only to local event handlers within the `GetName` class, so it is safe to use the same names for our event handlers, which helps make our code's purpose more immediately obvious to other developers. But if there is ever any doubt, you might prefer to use different names for your handlers, such as `actOnSubmit`, etc.

So, as you should see by now, `this.state.value` will always reflect the state of the input field because, as noted earlier, with a controlled component, `value` is always driven by the React state.

Using textarea

One of the ideas behind using React is to maintain cross-browser control over the DOM for quick and simple access as well as to streamline and simplify the development process. By using controlled components, we are in control at all times and can make inputting data of all types work in similar ways.

In [Example 20-11](#), the previous example has been modified to use a `<textarea>` element for input.

Example 20-11. Using `textarea`

```
<script type="text/babel">
class GetText extends React.Component
{
  constructor(props)
  {
    super(props)
    this.state = {value: ''}
    this.onChange = this.onChange.bind(this)
    this.onSubmit = this.onSubmit.bind(this)
  }

  onChange(event)
  {
    this.setState({value: event.target.value})
  }

  onSubmit(event)
  {
    alert('You submitted: ' + this.state.value)
    event.preventDefault()
  }

  render()
  {
```

```

    return (
      <form onSubmit={this.onSubmit}>
        <label>
          Enter some text:<br />
          <textarea rows='5' cols='40' value={this.state.value}
            onChange={this.onChange} />
        </label><br />
        <input type="submit" />
      </form>
    )
  }
}

ReactDOM.render(<GetText />, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>

```

This code is very similar to the text input example, with a few simple changes: this class is now called `GetText`, the text input in the render method is replaced with a `<textarea>` element that has been set to 40 columns wide by 5 rows high, and a couple of `<br>` elements have been added for formatting. And that's it—nothing else has required changing to enable us to have full control over the `<textarea>` input field. As with the previous example, `this.state.value` will always reflect the state of the input field.

Of course, this type of input supports the use of Enter or Return to input carriage returns into the field, so now the input can only be submitted by clicking the button.

Using select

Before showing how to use `<select>` in React, let's first look at a typical snippet of HTML code that offers a few countries from which the user must choose, with USA being the default selection:

```

<select>
  <option value="Australia">Australia</option>
  <option value="Canada">Canada</option>
  <option value="UK">United Kingdom</option>
  <option selected value="USA">United States</option>
</select>

```

In React this needs to be handled slightly differently because it uses a `value` attribute on the `select` element instead of the `selected` attribute applied to an `option` sub-element, as in [Example 20-12](#).

Example 20-12. Using select

```
<script type="text/babel">
  class GetCountry extends React.Component
  {
    constructor(props)
    {
      super(props)
      this.state = {value: 'USA'}
      this.onChange = this.onChange.bind(this)
      this.onSubmit = this.onSubmit.bind(this)
    }

    onChange(event)
    {
      this.setState({value: event.target.value})
    }

    onSubmit(event)
    {
      alert('You selected: ' + this.state.value)
      event.preventDefault()
    }

    render()
    {
      return (
        <form onSubmit={this.onSubmit}>
          <label>
            Select a country:
            <select value={this.state.value}
              onChange={this.onChange}>
              <option value="Australia">Australia</option>
              <option value="Canada" >Canada</option>
              <option value="UK" >United Kingdom</option>
              <option value="USA" >United States</option>
            </select>
          </label>
          <input type="submit" />
        </form>
      )
    }
  }

  ReactDOM.render(<GetCountry />, document.getElementById('display'))
</script>

<div id='display' style='font-family:monospace'></div>
```

Once again you will see that very little has changed in this example other than the new class name of `GetCountry`, that `this.state.value` is assigned the default value of `'USA'`, and that the input type is now a `<select>` but without a `selected` attribute.

Just as with the previous two examples, `this.state.value` always reflects the state of the input.

React Native

React also has a companion product called React Native. With it you can create full-blown applications for both iOS and Android phones and tablets, just using the JSX extended JavaScript language and without needing to understand Java or Kotlin (for Android) or Objective-C or Swift (for iOS).

Full details and explanations of how to do all this and have your apps up and running on a wide range of mobile devices are beyond the scope of this book, but you can work through [the tutorial](#) on the [React Native website](#), noting the differences between building apps on Windows and on macOS.

You've learned the basics of how to set up and use React, but you can do a great deal more with it (especially if you intend to build React Native apps with it) that is sadly beyond the scope of this book. To continue your React journey, you can visit the [react.dev web page](#), where you can review some of the things discussed here before moving on to [applying CSS styles](#) to elements and many even more powerful features.

Remember that you can download all the samples from this chapter (and this book as a whole) on [GitHub](#).

So, with React now in your toolkit (at least enough to get you up and running), let's move to [Chapter 21](#) and all the goodies that HTML brings. But first, try answering the following questions to test your React knowledge.

Questions

1. What are the main two ways you can incorporate the React scripts in your web page?
2. How is XML incorporated into JavaScript for use with React?
3. Instead of `<script type="application/javascript">`, what value for `type` should you use for your JSX JavaScript code?
4. What are two different ways you can extend React to your code?
5. In React, what is meant by pure and impure code?
6. How does React keep track of state?
7. How can you embed an expression within JSX code?

8. How can you change the state of a value once a class has been constructed?
9. What must you first do to enable referring to props using the `this` keyword within a constructor?
10. How can you create a conditional statement in JSX?

See “Chapter 20 Answers” on page 584 in the [Appendix](#) for the answers to these questions.

Introduction to Node.js

Node.js is an open source, cross-platform JavaScript runtime environment that was built on Chrome's V8 JavaScript engine by Ryan Dahl in 2009. It allows developers to execute JavaScript code on the server side, meaning you can run JavaScript both within and outside the web browser.

According to [W3Techs](#), at the time of writing, Node.js is used by 3.5% of websites. This may not sound like much, but a year ago it was 2%, which represents a stunning growth rate of 150% in a year. What's more, it's most frequently used on very high-traffic sites such as X/Twitter, Netflix, GitHub, Spotify, TikTok, eBay, Reddit, and over 30 million other equally and less well-known properties. To make an educated guess and extrapolate this usage into potential market share over the next few years, it's possible that Node.js could be implemented on up to a quarter of all web properties by the 2030s.

Node.js uses an event-driven, nonblocking model, whereas web servers like Apache use a synchronous request-response model, which means each incoming request is processed in a separate thread, and the thread is blocked until the response is ready. Node.js handles requests asynchronously, processing connections without creating a new thread for each request. This makes it highly scalable and ideal for real-time applications, chat applications, gaming servers, and other scenarios where low latency is essential.

For example, a web server like Apache will handle a request to serve up a web page by sending the task to the computer's filesystem, then it will wait for the filesystem to return the file before opening it and sending the contents back to the requesting web browser. Node.js, on the other hand, uses events in such a way that it passes the file request off to the system and immediately goes back to listening for more incoming requests. Then, when the filesystem is ready, it uses an event to notify Node.js, which then opens the file and sends the contents to the requesting web browser.

Node.js also comes with npm (commonly understood as Node Package Manager, but officially it stands for “npm Is Not An Acronym”), a powerful program that allows developers to access and use a vast ecosystem of open source libraries and modules. This extensive collection of packages simplifies development and enables developers to leverage prebuilt solutions for common tasks, significantly speeding up the development process. The downside is that some packages are often poorly maintained, poorly documented, buggy, and may represent security threats. Blindly installing packages from npm without thinking about it can get you into trouble.



Apache Remains Relevant

While Node.js is an excellent choice for building scalable, real-time applications that require both high concurrency and low latency, Apache remains a strong choice for traditional web applications and scenarios where CPU-bound processing is predominant. In some cases, developers might even choose to use both technologies, with Node.js handling real-time aspects and Apache serving static content or acting as a reverse proxy. Another popular high-performance web server you can use with PHP instead of Apache, and also as a reverse proxy for Node.js, is nginx (pronounced “engine x”).

Installing Node.js on Windows

To use Node.js you must first install it, just as you installed AMPPS in [Chapter 2](#). You can download the latest release at the [Node.js website](#). You are recommended to install the LTS (Long Term Support) version because, as the name suggests, it will be supported for at least 18 months. You *can* always try out the latest stable version to access the newest features, but this is not recommended unless you are happy to not necessarily get long-term support for it.



Node.js on WSL

If you’re using the Windows Subsystem for Linux (WSL) you may prefer to install Node.js there instead of installing directly to Windows. This allows you to install the recommended nvm (Node Version Manager) to manage multiple active versions, for example. However, similar to the section on Linux installation later on, this is not covered here due to the many available installation options.

As of writing, the latest LTS release is version 20.17 and installers are available as msi, ZIP, or source code files. These days you are almost certain to be running a 64-bit operating system so, unless you have a good reason otherwise, Windows users should download and install the recommended msi installer, which may well be a newer version than 20.17 by the time you read this.

Once Node.js is downloaded you need to run the installer, and you should see an intro screen similar to [Figure 21-1](#). Over the lifetime of this edition of the book the installation process for Node.js may change, so use common sense to follow through the installation if it's much different than the following. For now, click Next to get started.

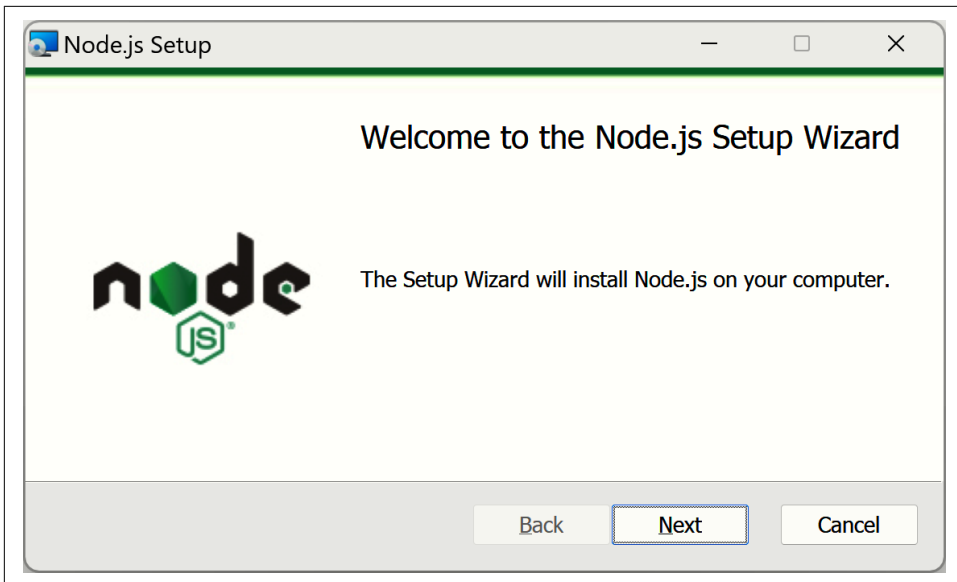


Figure 21-1. The Node.js installation wizard

Your first decision is where you would like Node.js to be installed, as shown in [Figure 21-2](#). In most cases you should accept the default directory offered and click Next.

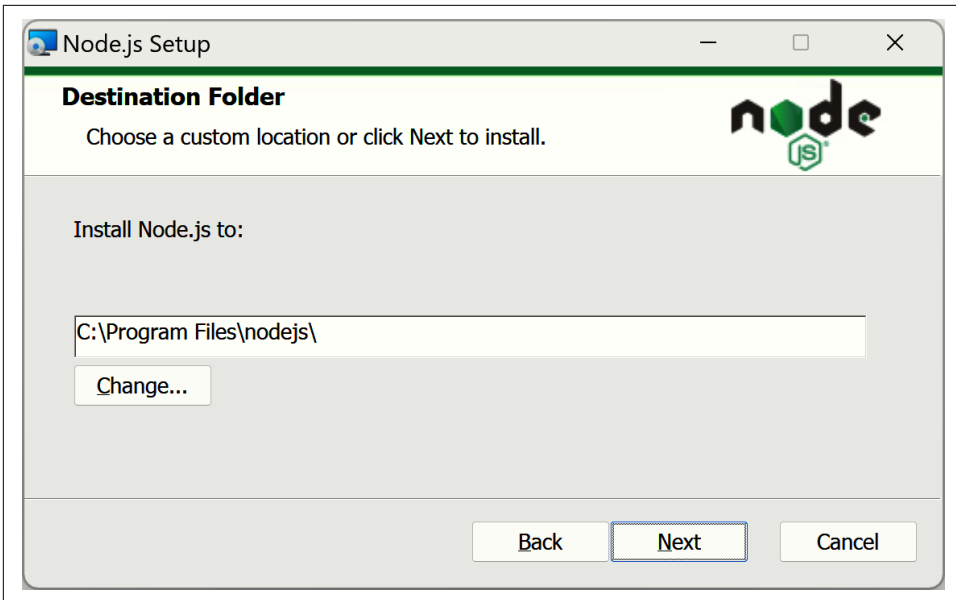


Figure 21-2. Selecting a destination installation folder

Next, you can customize the features you wish to be installed, as shown in [Figure 21-3](#). Again, unless you have good reason otherwise, just accept the defaults offered and click Next.

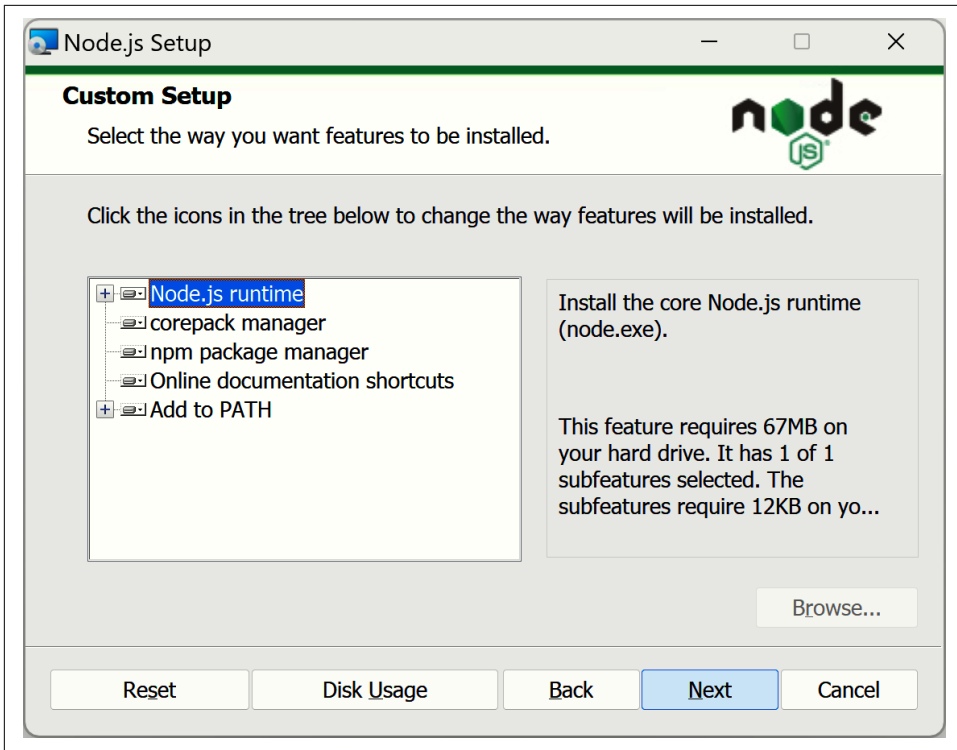


Figure 21-3. Customizing the defaults

I do recommend that you check the box enabling installation of the tools necessary for compiling native modules, as this is a lot simpler than following the set of alternative instructions linked to, as shown in [Figure 21-4](#). Whether or not you opt to enable this, click Next to continue.

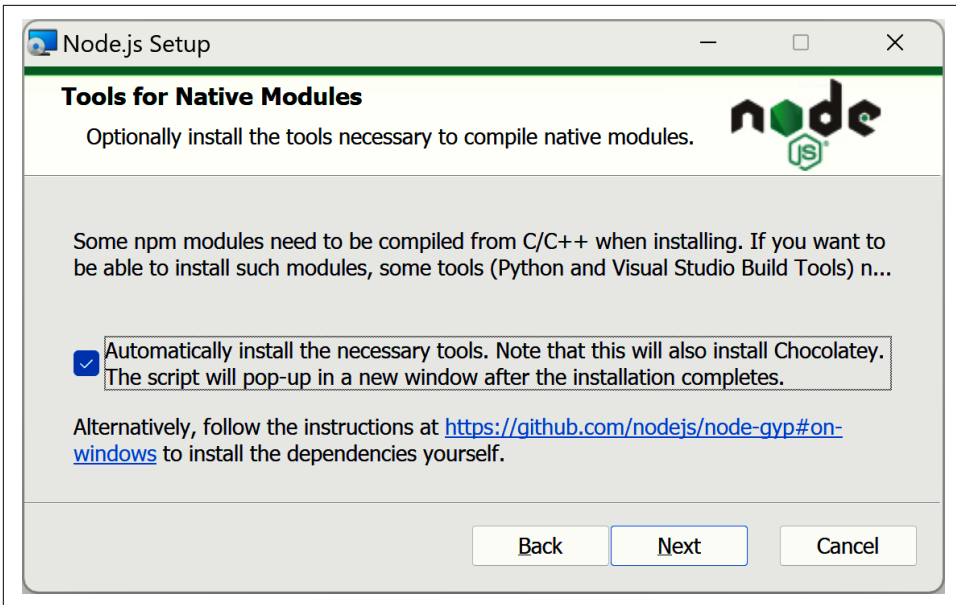


Figure 21-4. Enabling compiling of native modules

If you decide to enable the compiling of native modules then you will see a window similar to [Figure 21-5](#), which tells you what will be installed and the resources required. Press any key when you are ready to continue.

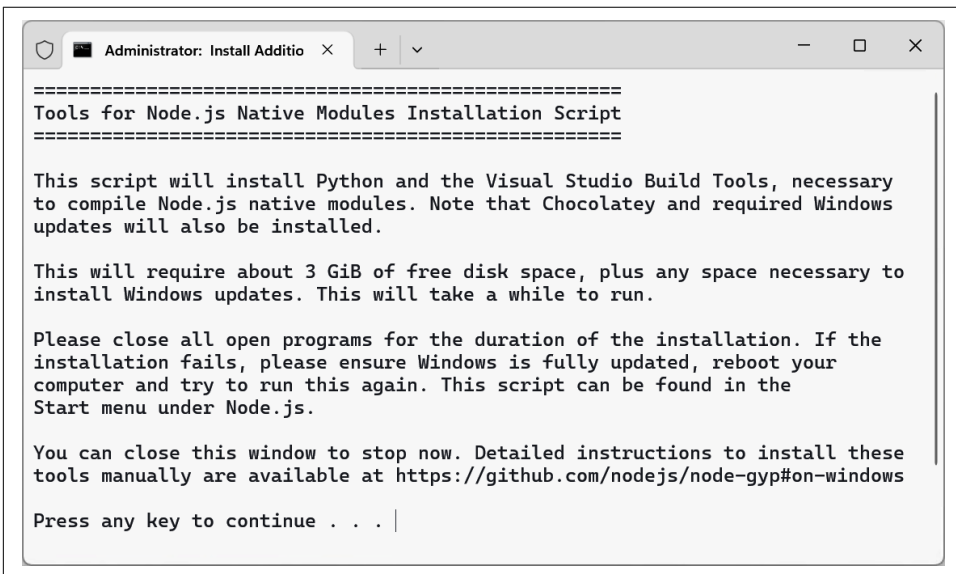
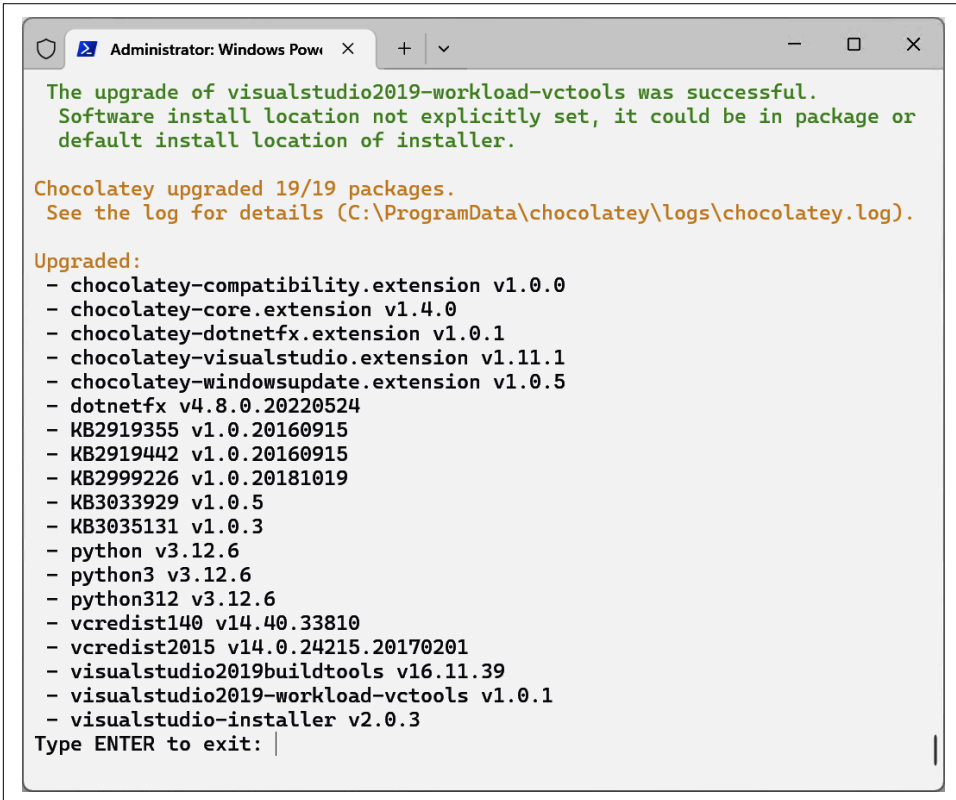


Figure 21-5. Installing the tools for Node.js

If you are installing the native module compilation support, a PowerShell window will open, as shown in [Figure 21-6](#), in which you can watch the installation process. Once the installation is finished you can press Enter and installation should be complete.



```
Administrator: Windows Powe x + v - □ x

The upgrade of visualstudio2019-workload-vctools was successful.
Software install location not explicitly set, it could be in package or
default install location of installer.

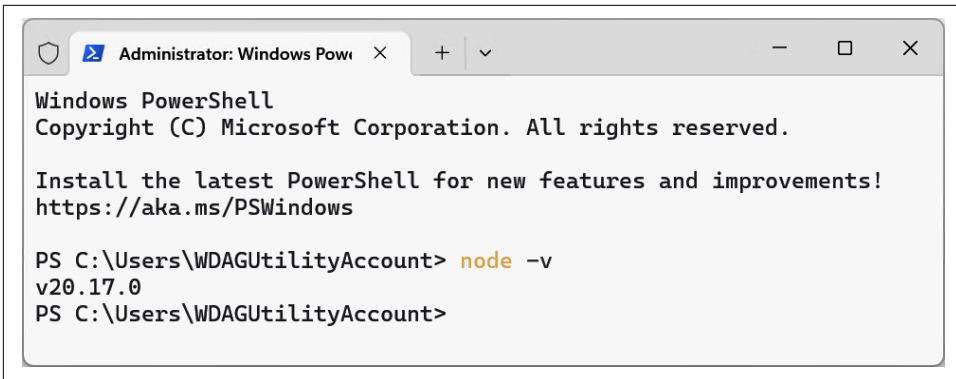
Chocolatey upgraded 19/19 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

Upgraded:
- chocolatey-compatibility.extension v1.0.0
- chocolatey-core.extension v1.4.0
- chocolatey-dotnetfx.extension v1.0.1
- chocolatey-visualstudio.extension v1.11.1
- chocolatey-windowsupdate.extension v1.0.5
- dotnetfx v4.8.0.20220524
- KB2919355 v1.0.20160915
- KB2919442 v1.0.20160915
- KB2999226 v1.0.20181019
- KB3033929 v1.0.5
- KB3035131 v1.0.3
- python v3.12.6
- python3 v3.12.6
- python312 v3.12.6
- vcredist140 v14.40.33810
- vcredist2015 v14.0.24215.20170201
- visualstudio2019buildtools v16.11.39
- visualstudio2019-workload-vctools v1.0.1
- visualstudio-installer v2.0.3
Type ENTER to exit: |
```

Figure 21-6. Various Python tools are installed

You are now ready to test your new installation of Node.js. To do this, open a PowerShell window and type the information shown in [Figure 21-7](#).

```
node -v
```



```
Administrator: Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements!
https://aka.ms/PSWindows

PS C:\Users\WDAGUtilityAccount> node -v
v20.17.0
PS C:\Users\WDAGUtilityAccount>
```

Figure 21-7. Checking that Node.js is installed

All being well the version number of Node.js will be shown. If you see anything else (such as an error message) you most likely can correct this by restarting the terminal and issuing the command again.

Installing Node.js on macOS

To use Node.js you must first install it, just as you installed AMPPS in [Chapter 2](#). You can download the latest release at the [Node.js website](#). I recommend that you install the LTS (Long Term Support) version because, as the name suggests, it will be supported for a good time to come. You *can* always try out the latest stable version to access the newest features, but this is not recommended unless you do not need to get long-term support for it.

As of writing, the latest LTS release is version 20.17 and since the recommended installer works on either Intel or ARM chips, unless you have a good reason to choose the specific installer you need, you should download and install the suggested pkg file, which may well be a newer version than 20.17 by the time you read this.

Once Node.js is downloaded you need to run the installer; you should see an intro screen similar to [Figure 21-8](#). Over the lifetime of this edition of the book the installation process for Node.js might change, so just use common sense to follow through the installation if it's much different than the following. That said, to get started click Continue.

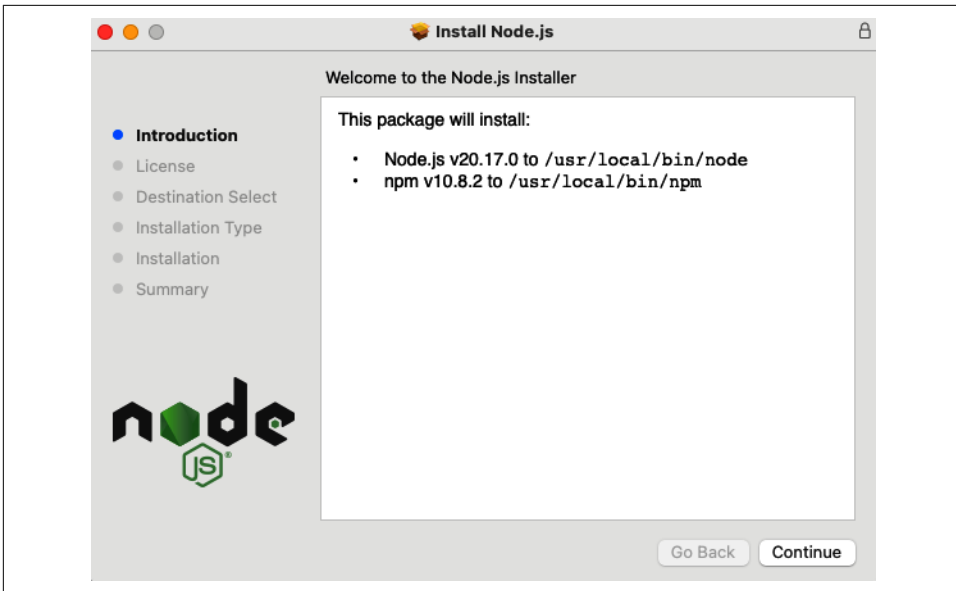


Figure 21-8. The Node.js installer

Next you will need to read and agree to the software license terms before you can proceed with installation, as shown in [Figure 21-9](#).

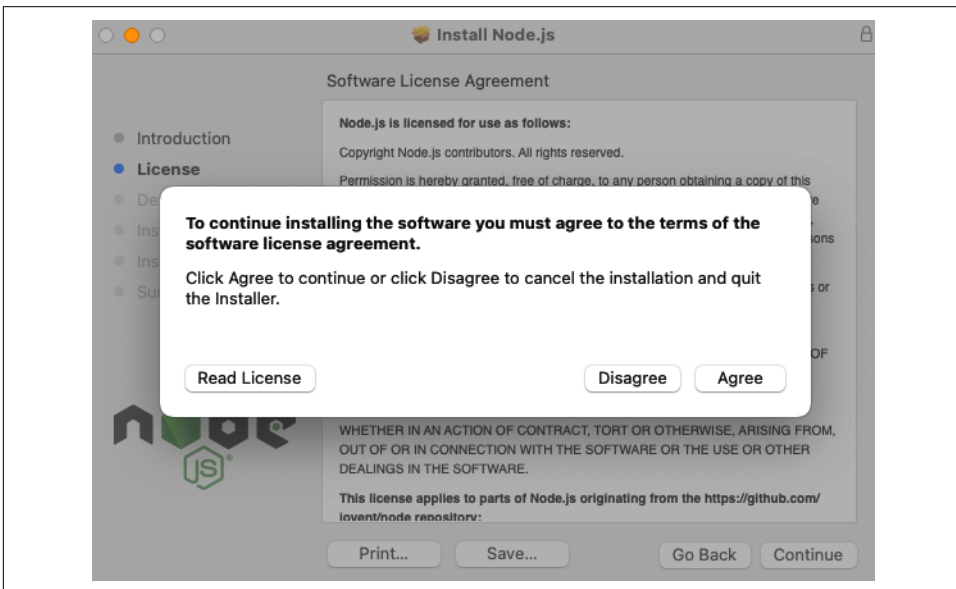


Figure 21-9. Agreeing to the software license

At this point you can choose the destination location and type for the installation. In most cases, unless you have a good reason to do otherwise, you should accept the default location and type offered, as shown in [Figure 21-10](#).

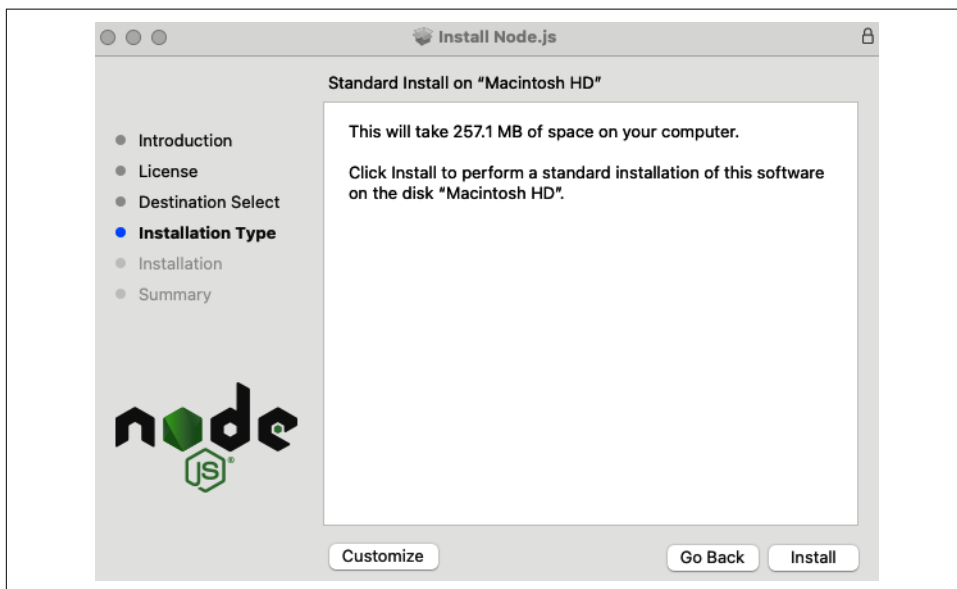


Figure 21-10. Selecting the installation destination location

For security reasons you must enter your password or use biometric identification to commence the installation, as shown in [Figure 21-11](#).

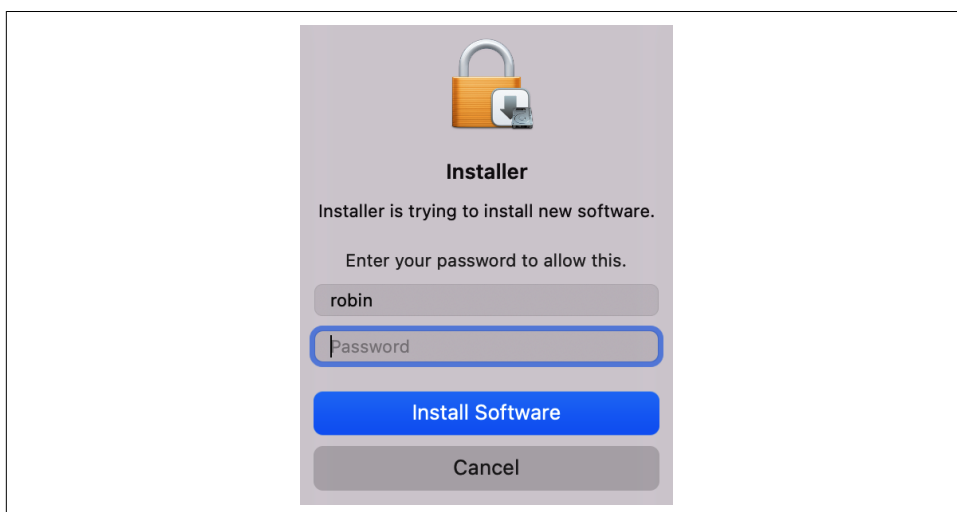


Figure 21-11. You must identify yourself before proceeding

Installation should proceed and after a short while you will be informed it has completed, as shown in [Figure 21-12](#). Click Close to finish.

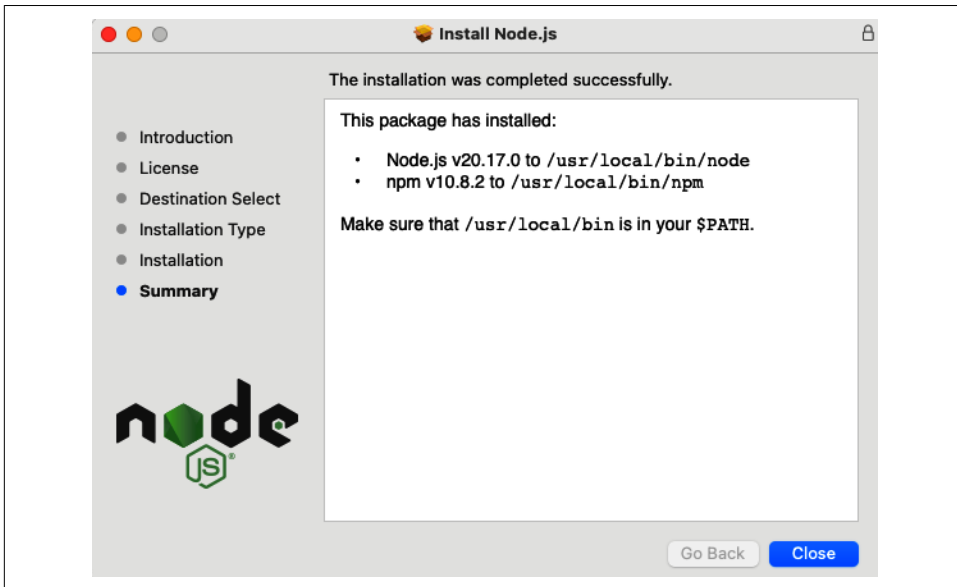


Figure 21-12. Installation is complete

You are ready to test your new installation of Node.js. To do this, open a Terminal window and type the information shown in [Figure 21-13](#).

```
node -v
```

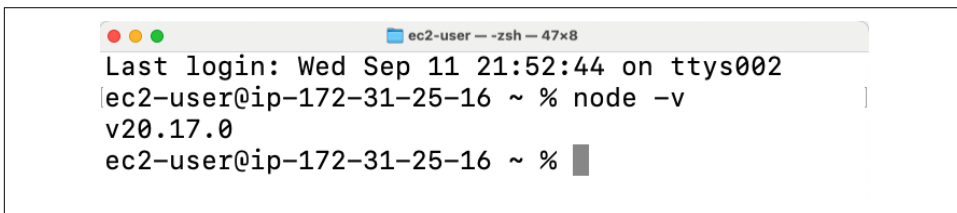


Figure 21-13. Verifying the installation

You should see the version number of the software you just installed.

Installing Node.js on Linux

You can download the latest release at the [Node.js website](#). I recommend you install the LTS (Long Term Support) version because, as the name suggests, it will be supported for a good time to come. You *can* always try out the latest stable version

to access the newest features, but this is not recommended unless you do not need to receive long-term support.

As of writing, the latest LTS release is version 20.17 and a variety of options are available, including Intel and Arm installers, the Node.js source code, a Docker image, Node Version Manager (nvm), Linux on Power LE or System z, and AIX on Power Systems.

Since these are so varied, and it is assumed that as a Linux user you will already be familiar with how to install this type of software, I leave it to you to determine the installer that is best for you and to follow the relevant instructions linked to at the bottom of the download page.

Once Node.js is installed you can verify success by typing the following in a terminal window to be told the version of the software just installed:

```
node -v
```

The result of running the command should look similar to the macOS window shown in [Figure 21-13](#).

Getting Started with Node.js

Creating your first Node.js program is a very simple, straightforward process. We'll be using *ECMAScript modules* (ECMAScript is a standardized specification of JavaScript, sometimes abbreviated as *ES*), the official standard format to write JavaScript code that's supposed to be reused. Most of the Node.js ecosystem uses these modules. They are also why the following files will use the *.mjs* extension (*m* for module) as that's the easiest way to tell Node.js you've created a module.

Let's begin with a single-mission web server that responds with the famous “Hello World” when accessed, as shown in [Examples 21-1](#) and [21-2](#).

Example 21-1. Hello World in Node.js, the function

```
function helloWorld()  
{  
  return 'Hello World'  
}  
  
export { helloWorld }
```

Type the code and save it in the current directory as *helloWorld.mjs*. The `helloWorld` function is simple; it only returns the string, while the next line exports the function from the module, so it can be imported, or reused, by some other module.

The following code represents the main module, the application itself. Save it in the current directory under the name *app.mjs*.

Example 21-2. Hello World in Node.js, the main application

```
import * as http from 'http'
import { helloWorld } from './helloWorld.mjs'

const server = http.createServer((request, response) =>
{
  response.writeHead(200, {'Content-Type': 'text/html'})
  response.end(helloWorld())
})

const port = 8000
server.listen(port, () => console.log('Server listening on port ' + port))
```

Let's work through this example. The Node.js http module is loaded, or imported, into an identifier called http using the import declaration. Node.js supports a range of modules to provide different functionality, and this one provides a set of functions to manage HTTP connections. The second line imports the helloWorld function from a module created a moment ago.

Next, another object called server is created by a call to the http object's method createServer, passing it a function that takes two arguments, request and response. The response object sets three properties: a status code of 200, an HTTP header string specifying the content type, and a string to respond with as returned by our helloWorld function, applied using the response object's methods writeHead and end.

After this a variable port is created with a port number 8000 the server will listen on. Using a different port than the standard HTTP port (80) is a common practice; it solves port clashes that would otherwise happen with the AMPPS Apache still running. Last, the server object is set to start listening with a call to the listen function, using the port specified, and outputting a string to the console (the command line) after the server starts to listen.

All you need do now is open a command prompt or terminal window, and you can run it with the following instruction:

```
node app.mjs
```

All being well you will receive the following response:

```
Server listening on port 8000
```

You are now ready to test the server with a simple call to `localhost` and the selected port `8000` from your web browser, as follows, with the result being “Hello World” displayed in your browser:

```
localhost:8000
```

To exit from a Node.js program press Ctrl-C, which you should remember to do each time you modify a program, before then rerunning it, as saving changes to a program will not have any effect until it is restarted. Do this now, as we are about to vastly improve on this server.

Building a Functioning Web Server

Now that you know how to interact with a Node.js program, let’s create a web server that delivers the basic functionality of a program such as Apache by supporting requests for multiple files using different URLs. To do this we’ll need access to a couple more modules for URL and file handling (`url`, `fs`, and `path`), as shown, following the loading of the `http` module at the head of [Example 21-3](#).

Example 21-3. A functioning web server

```
import http from 'http'
import url from 'url'
import { readFile } from 'fs/promises'
import { resolve, extname } from 'path'

const SCRIPT_DIRECTORY = new URL('./', import.meta.url).pathname

const server = http.createServer(async (request, response) => {
  const fpath = resolve('.') + url.parse(request.url).pathname
  if (!fpath.startsWith(SCRIPT_DIRECTORY)) {
    response.writeHead(400, { 'Content-Type': 'text/html' })
    response.end('400 Bad Request')
    return
  }
  if (extname(fpath) !== '.html') {
    response.writeHead(400, { 'Content-Type': 'text/html' })
    response.end('Sorry, only <code>.html</code> extension is supported')
    return
  }
  try {
    const data = await readFile(fpath, 'utf8')
    response.writeHead(200, { 'Content-Type': 'text/html' })
    response.end(data)
  } catch (err) {
    response.writeHead(404, { 'Content-Type': 'text/html' })
    response.end('404 Not Found')
  }
})
```

```
const port = 8000
server.listen(port, () => console.log('Server listening on port ' + port))
```

This is a very simple, very small, and very fast event-driven web server for either serving up an HTML file if it exists or otherwise returning a “404 Not Found” error message. It’s not very smart as it supports only HTML files located in the current directory, and it knows nothing about special files such as PHP etc. Also there is no caching of files.

Nevertheless, on a simple website with just a collection of HTML files and associated media, this web server can handle thousands of simultaneous requests, due to there being no log-jam waiting for the filesystem to return requested files, and it will be very fast and effective. Let’s save this example as *server.mjs* so that it can be run from a command prompt, like this:

```
node server.mjs
```

To accompany the program we also need a simple HTML file for the server to return, so save **Example 21-4** as *hello.html* and we’re ready to test the code.

Example 21-4. A simple HTML file

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple HTML file</title>
  </head>
  <body>
    <h1>Hello, how are you?</h1>
  </body>
</html>
```

Now you can type *localhost:8000/hello.html* into your web browser and the file will be displayed. Or you can ask for a nonexistent page such as *localhost:8000/bye.html*, in which case a “404 Not Found” error will be returned. Let’s look at how all this works.

First, after importing the required modules and functions, the variable `SCRIPT_DIRECTORY` will be set to contain the path to the current directory. It will be used later for a security check on whether the requested file is in the current directory with the following line:

```
if (!fpath.startsWith(SCRIPT_DIRECTORY)) {
  // return "Bad Request"
}
```

If the check fails, “Bad Request” will be returned instead. If there was no check like this, the application would be vulnerable to an attack called *path traversal* (sometimes

directory traversal), which would allow the attacker to request any file in any directory on the server using paths like `../../etc/passwd` and similar.

The code also checks the extension of the requested file and allows just `.html`, because otherwise anyone could download the source code by loading `localhost:8000/server.mjs`, which is not what you want. It uses the `extname` function from the built-in `path` module, like this:

```
if (extname(fpath) !== '.html') {  
  // return "Only .html supported"  
}
```

Similar to the “Hello World” example, this code creates a `server` object to process requests, and it also makes calls to the `writeHead` and `end` methods of the response object to send the status code, content type, and the page content to the calling web browser.

What’s new is how requests are dynamically processed, starting with the variable `fpath`, which is given its value by calling the `parse` method of the `url` module, passing it the `url` property of the request object to obtain `pathname`, which is then prefaced with the `.` character. The path is then resolved to an absolute path by calling `resolve` on the `path` module and compared with the `SCRIPT_DIRECTORY` variable. The result is that if you, for example, request the URL `localhost:8000/hello.html` then `fpath` will contain a full absolute path to the `hello.html` file in the current directory.

This file handling is managed with a call to the `readFile` method of the `fs` module, passing it the value in `fpath`, and requesting the contents to be delivered in `utf8` encoding, with the file data returned in the `data` variable. The `await` operator is used to wait for a fulfilled promise (these are the same objects you’ve already encountered in the information about asynchronous functions in [Chapter 17](#)). If there is an error, an exception is thrown and caught, and for the sake of brevity and simplicity in this example, it is assumed to be “Not Found,” with a status code of 404.

Of course you don’t have to only serve up preexisting files. It’s quite possible (and highly likely) you will build responses into your Node.js programs. That’s the power of it after all: the ability to write both backend and frontend code in JavaScript. So let’s modify the example one more time to show a simple way of creating a fully self-contained program that acts like a whole website of files, as in [Example 21-5](#).

Example 21-5. A self-contained server

```
import http from 'http'  
  
const server = http.createServer(async (request, response) => {  
  let status = 200  
  let output = '404 Not Found'
```

```

switch (request.url) {
  case '/hello.html': output = 'Hello there'
                    break
  case '/bye.html':  output = 'Goodbye'
                    break
  default:           status = 404
}

response.writeHead(status, { 'Content-Type': 'text/html' })
response.end(output)
})

const port = 8000
server.listen(port, () => console.log('Server listening on port ' + port))

```

Here the `fs`, `url`, and `path` modules are no longer required, and in place of file handling there is now a simple `switch` statement in which each case is handled individually by assigning a value to the variable `output`. There's also no path traversal check, because the code doesn't access any files.

You can include as many cases as you like, but this example just supports requests for *hello.html* and *bye.html* with a default `status` value of 200 (OK) then passing through to the `writeHead` and `end` methods. If neither case applies, in other words anything other than these two strings are entered in the web browser, the `status` code value is set to 404 and the default error string “404 Not Found,” previously assigned to `output`, both pass through to be returned to the browser.

Of course, your own code will be much more powerful and creative than these examples, but you are now equipped with enough Node.js knowledge to return either full files from a filesystem or construct responses on the fly (or do both), according to your needs. Next we'll look more closely at Node.js modules, how to manage them using `npm`, and how to load them into Node.js to interact with a MySQL database.

Working with Modules

Now that you know how to leverage the frontend JavaScript skills you learned earlier in this book to write backend code, you can use the number three trending language (as I write). Unfortunately, at some point, PHP as a technology (though still widely used across the internet) is no longer even in the [top ten trending IEEE languages](#).

Perhaps it's the strong relationship between PHP and MySQL that keeps PHP installed on so many web properties, but even that may change over the coming years because Node.js can work with MySQL databases too, just by importing a module. In fact, with over two million packages available for Node.js, it offers support for just about any application you can think of.

The way this is achieved is via npm, the Node.js package manager, which is installed when you install Node.js itself. But first let's examine the modules that come with Node.js.

Built-in Modules

Node.js comes with a number of built-in modules that you can access immediately through the `import` declaration without having to install them using npm, a few of which you have already encountered. Here's a list of the most commonly used modules:

- `crypto` handles encrypted data.
- `dns` handles name resolution.
- `fs` accesses the local filesystem.
- `http` transfers data over HTTP.
- `https` transfers secure data over HTTP.
- `net` transfers servers and clients.
- `os` obtains information about the operating system.
- `path` works with directory and filepaths.
- `querystring` parses URL query strings.
- `url` parses and resolves URL strings.
- `util` accesses various utility functions.

Please see the [official Node.js documentation](#) for full details on using Node.js modules.

Installing Modules with npm

To import a module to Node.js that is not built-in, use the npm program, which stands for “npm Is Not An Acronym,” although commonly understood as Node Package Manager. For example, to install the `mysql2` module, which offers some additional features over the older `mysql` module, such as prepared statements and placeholders, enter the following at a command prompt:

```
npm install mysql2
```

Go ahead and do this now as we'll soon use this module to connect to the database created in [Chapter 8](#) just as easily as we did using PHP. In a few seconds you should see something like:

```
added 13 packages in 2s
```

You can now access this module as described in [Chapter 20](#) using the `import` method.

You can also use npm to create your own packages, although how to do this is beyond the scope of this book. For more information, see the [full npm documentation](#), or you can access a treasure trove of ready-made packages at the [npm website](#).

Accessing MySQL

If Node.js is truly to be able to replace a stack such as AMPPS then it must be able to access databases and, as you'd expect, it does, with great ease and simplicity. It has support for PostgreSQL, DynamoDB, MongoDB, and many other SQL and NoSQL databases. But since this is a book focusing on MySQL, let's work with MySQL via the `mysql2` package you downloaded during “[Installing Modules with npm](#)” on page 526.

I recommend you add a new user for all your Node.js accesses by calling up MySQL like this, on a PC with AMPPS installed:

```
C:\Program Files\Amps\mysql\bin\mysql" -u root -pmysql
```

Or like this on a Mac:

```
/Applications/ampps/mysql/bin/mysql -u root -pmysql
```

Or on Linux:

```
mysql -u root -p
```

Once you are at the MySQL prompt you can create a new user with the name *node* and password *letmein* like this:

```
CREATE USER 'node'@'localhost' IDENTIFIED BY 'letmein';  
GRANT ALL ON publications.* TO 'node'@'localhost';
```

You might want to change this user's name or password later, but for the purposes of the following example these are the details we'll work with. Now you can exit from MySQL with the following command:

```
quit;
```

Now, let's write a simple Node.js program to log in to the *publications* database we created in [Chapter 8](#) and extract some data from it, as in [Example 21-6](#). Save the file as *mysql.mjs*.

Example 21-6. Querying a MySQL database

```
import mysql from 'mysql2/promise'  
  
const connection = await mysql.createConnection({  
  host: 'localhost',  
  user: 'node',  
  password: 'letmein',
```

```

    database: 'publications'
  })

  try {
    const query = 'SELECT * FROM classics WHERE author = ?'
    const [results, fields] = await connection.execute(
      query,
      ['Jane Austen']
    )

    console.log('Results:', results.length)
    console.log('Data returned:', results)
    console.log('Author:', results[0].author)
    console.log('Title:', results[0].title)
    console.log('Category:', results[0].category)
    console.log('Year:', results[0].year)
    console.log('ISBN:', results[0].isbn)
  } catch (error) {
    console.log(error)
  }

  connection.end()

```

In this example, the first line fetches the `mysql2` module and creates the object `mysql`, creating a matching object called `connection` from the access details provided. The access details like username and password should not be stored directly in your production code; instead, you should use a special file called `.env`, or `config.env` to store them.



Built-in .env Support

Starting with version 20.6, Node.js has a built-in `.env` support. You create a file named, for example, `config.env`, which follows the INI format with key=value lines like this:

```

USER=node
PASSWORD=letmein

```

Execute Node.js with a new command-line parameter:

```
node --env-file=config.env mysql.mjs
```

Then you can access the details with `process.env`, like this:

```

const connection = await mysql.createConnection({
  host: 'localhost',
  user: process.env.USER,
  password: process.env.PASSWORD,
  database: 'publications'
})

```


Next the variable `query` is assigned a MySQL query string, which is using a placeholder `?` instead of a variable (or hardcoded string) to prevent SQL injection, and which when executed with the parameter will return all fields from any rows in which the `author` field contains “Jane Austen.” The query is passed to MySQL via the `execute` method of the connection object together with the parameter array. Two objects are returned: `results` for the results and `fields` for the fields accessed.

If an error occurred, an exception is thrown and a suitable message logged; otherwise, the returned data is examined. First, the number of results returned is obtained from `results.length` then, just to show you what the returned data object looks like, `results` is displayed in its entirety. After that the contents of each of the five fields is individually displayed.

Finally, the connection to the database is closed by calling the `end` method of the connection object. Unlike a server, the program then ends, returning access to the command prompt, although you will probably include such MySQL accessing code within a server-style program.

The result of running this code should look something like:

```
C:\nodesql> node mysql.mjs
Results: 1
Data returned: [
  {
    author: 'Jane Austen',
    title: 'Pride and Prejudice',
    category: 'Classic Fiction',
    year: 1811,
    isbn: '9780582506206'
  }
]
Author: Jane Austen
Title: Pride and Prejudice
Category: Classic Fiction
Year: 1811
ISBN: 9780582506206
```

You now have the means to construct queries, pass them to the database, and retrieve the results.

To further your knowledge in this area, comprehensive documentation on the `mysql2` module is available at the [npm website](#).

Further Information

Of course we can only scratch the surface about software with the power of a program such as Node.js. Nevertheless it's a testament to the simplicity and robustness of Node.js that it has been possible to show you how to create functional web servers

that can interact with complex third-party software such as the MySQL relational database.

You are well on your way to creating small, fast, functional, and high-traffic Node.js web properties. To further your knowledge you can browse other titles on the subject available from resources such as the [O'Reilly Learning Platform](#). And you can read the official documentation at the [Node.js website](#).

In [Chapter 22](#) all the technologies covered in this book are brought together to create a simple social network application. But first, try answering the following questions to test your Node.js knowledge.

Questions

1. After receiving a request for a file from a web browser and passing the request off to the filesystem, what does Node.js do?
2. What is the method called for including Node.js prewritten modules?
3. Which three modules does Node.js use to manage HTTP communication, parse URLs, and access the local filesystem?
4. What is the default HTTP port a server listens to?
5. Which method of the `http` Node.js module is used to create a new server object?
6. Which part of the response must be sent back to a web browser first, before returning data (or an error)?
7. How do you end a connection to a web browser?
8. How do you start a Node.js server?
9. How do you manually terminate a Node.js server?
10. How do you write messages to the terminal window command line from Node.js?
11. How do you add external Node.js modules to a project?
12. How do you use Node.js to access a MySQL database?
13. How do you create a connection to MySQL in Node.js?
14. How do you query a MySQL database with Node.js?
15. How do you terminate a connection to a MySQL database?

See [“Chapter 21 Answers” on page 585](#) in the [Appendix](#) for the answers to these questions.

Bringing It All Together

Now that you've reached the end of this book, your first milestone along the path of the hows, whys, and wherefores of dynamic web programming, I want to leave you with a real example. In fact, it's a collection of examples, because I've put together a simple social networking project comprising all the main features you'd expect from such a site, or more to the point, such a web app.

Across the various files, there are examples of MySQL table creation and database access, CSS, file inclusion, session control, asynchronous calls, event and error handling, file uploading, image manipulation, and a whole lot more.

Each example file is complete and self-contained yet works with all the others to build a fully working social networking site, even including a stylesheet you can modify to completely change the project's look and feel.

The small, light end product is particularly usable on mobile platforms such as a smartphone or tablet but will run equally well on a full-size desktop computer. To exercise your skills, you may wish to adapt the code further, perhaps using React in some way.

I have tried to keep this code as concise as possible so it's easy to follow. Consequently, a great deal of improvement could be made to it, such as smoother handling of some of the transitions between being logged on and off—but let's leave those as the exercises for the reader, particularly since there are no questions at the end of this chapter. *Well...just the one!*

I leave it up to you to take any pieces of this code you think you can use and expand on them for your own purposes. You might even build on these files to create a social networking site of your own.

Designing a Social Networking App

Before writing any code, I thought about several things that were essential for a social networking application:

- A signup process
- A login form
- A logout facility
- Session control
- User profiles with uploaded thumbnails
- A member directory
- Adding members as friends
- Public and private messaging between members
- Project styling

I named the project *Robin's Nest*; if you use this code, you will need to modify the name and logo in the *index.php* and *header.php* files.

Online Repository

All the examples in this chapter are available online in a [repository at GitHub](#), where you can download the archive file *lpmj7examples.zip*, which you should extract to a suitable location on your computer (such as the document root of the AMPPS web server), where it can be easily accessed from your browser.

Of particular interest to this chapter, within the file, you will find a folder called *robinsnest*, in which all the examples from this chapter have been saved. Once these files are set up (as detailed next), you should be able to type the following into your browser to run the application:

localhost/robinsnest

functions.php

Let's jump right into the project, starting with [Example 22-1](#), *functions.php*, the included file for the main functions. This file contains a little more than just the functions, though, because I have added the database login details here instead of using a separate file. The first four lines of code define the host and name of the database to use, as well as the username and password.

By default, in this file the MySQL username is set to *robinsnest*, and the database used by the program is also called *robinsnest*. [Chapter 8](#) provides detailed instructions on

how to create a new user and/or database, but to recap, first create a new database called *robinsnest* by entering a MySQL command prompt and typing this:

```
CREATE DATABASE robinsnest;
```

Then you can create a user called *robinsnest* capable of accessing this database like this:

```
CREATE USER 'robinsnest'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON robinsnest.* TO 'robinsnest'@'localhost';
```

Obviously you would use a much more secure password for this user than *password*, but for the sake of simplicity, this is the password used in these examples—just make sure you change it if you use any of this code on a production site.

The project uses two main functions:

`destroySession`

Destroys a PHP session and clears its data to log users out.

`showProfile`

Looks for an image of the name `<user.jpg>` (where `<user>` is the username of the current user) and, if it finds it, displays it. It also displays any “about me” text the user may have saved.

I have ensured that error handling is in place for all the functions that need it so that they can catch any typographical or other errors you may introduce and generate error messages. However, if you use any of this code on a production server, you will want to provide your own error-handling routines to make the code more user-friendly.

So, type in [Example 22-1](#) and save it as *functions.php* (or download it from the [companion website](#)), and you’ll be ready to move to “[header.php](#)” on [page 535](#).

Example 22-1. functions.php

```
<?php // Example 01: functions.php
$dbhost = 'localhost'; // Change as necessary
$db      = 'robinsnest'; // Change as necessary
$dbuser  = 'robinsnest'; // Change as necessary
$dbpass  = 'password';   // Change as necessary
$charset = 'utf8mb4';
$dbattr  = "mysql:host=$dbhost;dbname=$db;charset=$charset";
$opts    =
[
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES  => false,
];
```

```

try {
    $pdo = new PDO($dbattr, $dbuser, $dbpass, $opts);
} catch (\PDOException $e) {
    throw new \PDOException($e->getMessage(), (int)$e->getCode());
}

function destroySession()
{
    $_SESSION=array();

    if (session_id() !== "" || isset($_COOKIE[session_name()]))
        setcookie(session_name(), '', time()-2592000, '/');

    session_destroy();
}

function showProfile($user, $pdo)
{
    if (file_exists("$user.jpg"))
        echo "<img src='$user.jpg' style='float:left;'>";

    $stmt = $pdo->prepare("SELECT * FROM profiles WHERE user=?");
    $stmt->execute([$user]);

    $row = $stmt->fetch();
    if ($row)
        echo htmlentities($row['text']) . "<br style='clear:left;'><br>";
    else
        echo "<p>Nothing to see here, yet</p><br>";
}
?>

```



If you've read previous editions of this book, in which these examples used the old `mysql` extension, and later on `mysqli`, you will see I have moved on to the best solution so far, which is `PDO`. I'm also using placeholders and prepared statements to protect the application against SQL injection attacks.

To reference the MySQL database using `PDO`, the `showProfile` function has a `$pdo` parameter you need to pass when calling the function.

header.php

For uniformity, each page of the project needs to have access to the same set of features. Therefore, I placed these in [Example 22-2](#), *header.php*. This is the file that is actually included by the other files. It includes *functions.php*. This means only a single `require_once` is needed in each file.

header.php starts a session by calling the function `session_start`. As you'll recall from [Chapter 12](#), this sets up a session that will remember certain values we want stored across different PHP files. In other words, it represents a visit by a user to the site, and it can time out if the user ignores the site for a period of time.

With the session started, the file of functions (*functions.php*) is included, and the default string of "Welcome Guest" is assigned to `$userstr`.

Next, the code checks whether the session variable `user` is currently assigned a value. If so, a user has already logged in, so the variable `$loggedin` is set to `TRUE` and the username is retrieved from the session variable `user` into the PHP variable `$user_html_entities` (sanitized string for output), with `$userstr` updated appropriately. If the user has not yet logged in, then `$loggedin` is set to `FALSE`.

The program then outputs the HTML needed to set up each web page, including loading stylesheets, Bootstrap Icons, the logo, and the greeting.



Bootstrap Icons

The app uses icons from the Bootstrap Icons collection (the latest version 1.11.3 as I write), a free, open source icon library. It can be installed with npm but I'm loading it from the internet for the sake of simplicity. To add an icon, you can use the `<i>` tag with the class that specifies the icon you want, for example like this for a checkmark icon:

```
<i class="bi-check"></i>
```

There are multiple usage methods and many icons to choose from. Visit the [Bootstrap Icons website](#) to learn more.

After this, using the value of `$loggedin`, an `if` block displays one of two sets of menus. The non-logged-in set simply offers options of Home, Sign Up, and Log In, whereas the logged-in version offers full access to the app's features.

The additional styling applied to this file is in the file *styles.css* ([Example 22-13](#), detailed at the end of this chapter).

Example 22-2. header.php

```
<?php // Example 02: header.php
session_start();
require_once 'functions.php';

$userstr = 'Welcome Guest';

if (isset($_SESSION['user'])) {
    $user_html_entities = htmlentities($_SESSION['user']);
    $loggedin = TRUE;
    $userstr = "Logged in as: $user_html_entities";
}
else
    $loggedin = FALSE;
?>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="styles.css">
        <script src="javascript.js"></script>
        <link rel="stylesheet" href="
https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.min.css
">
        <title>Robin's Nest: <?php echo $userstr; ?></title>
    </head>
    <body>
        <div>
            <div>
                <div id="logo"
                    class="center">bin's Nest</div>
                <div class="username"><?php echo $userstr; ?></div>
            </div>
            <div class="content">

<?php
    if ($loggedin) {
?>
        <div class="center">
            <a class="button"
                href="members.php?view=<?php echo $user_html_entities; ?>">
                <i class="bi-house-door-fill"></i> Home</a>
            <a class="button" href="members.php">
                <i class="bi-person-fill"></i> Members</a>
            <a class="button" href="friends.php">
                <i class="bi-heart-fill"></i> Friends</a><br>
            <a class="button" href="messages.php">
                <i class="bi-envelope-fill"></i> Messages</a>
            <a class="button" href="profile.php">
                <i class="bi-pencil-fill"></i> Edit Profile</a>
        </div>
    }
}
```



```

        <a class="button" href="logout.php">
            <i class="bi-door-closed-fill"></i> Log out</a>
    </div>
<?php
} else {
?>
    <div class="center">
        <a class="button" href="index.php">
            <i class="bi-house-door-fill"></i> Home</a>
        <a class="button" href="signup.php">
            <i class="bi-plus-circle-fill"></i> Sign Up</a>
        <a class="button" href="login.php">
            <i class="bi-check-circle-fill"></i> Log In</a>
    </div>
    <p class="info">(You must be logged in to use this app)</p>
<?php
}
?>

```

setup.php

With the pair of included files written, it's time to set up the MySQL tables they will use. We do this with [Example 22-3](#), *setup.php*, which you should type and load into your browser before calling up any other files; otherwise, you'll get numerous MySQL errors.

The tables created are short and sweet, and they have the following names and columns:

members

username *user* (indexed), password *pass* (to store a password hash)

messages

ID *id* (indexed), author *auth* (indexed), recipient *recip*, message type *pm*, message *message*

friends

username *user* (indexed), friend's username *friend*

profiles

username *user* (indexed), "about me" *text*

Because the SQL query first checks whether a table already exists, this program can be safely called multiple times without generating any errors.

You very likely will need to add many more columns to these tables if you choose to expand this project. If so, remember that you may need to issue a MySQL `DROP TABLE` command before re-creating a table.

Example 22-3. setup.php

```
<!DOCTYPE html> <!-- Example 03: setup.php -->
<html>
  <head>
    <title>Setting up database</title>
  </head>
  <body>
    <h3>Setting up...</h3>

<?php
  require_once 'functions.php';

  $pdo->query('CREATE TABLE IF NOT EXISTS members (
    user VARCHAR(16),
    pass VARCHAR(255),
    INDEX(user(6))
  )');

  $pdo->query('CREATE TABLE IF NOT EXISTS messages (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    auth VARCHAR(16),
    recip VARCHAR(16),
    pm CHAR(1),
    time INT UNSIGNED,
    message VARCHAR(4096),
    INDEX(auth(6)),
    INDEX(recip(6))
  )');

  $pdo->query('CREATE TABLE IF NOT EXISTS friends (
    user VARCHAR(16),
    friend VARCHAR(16),
    INDEX(user(6)),
    INDEX(friend(6))
  )');

  $pdo->query('CREATE TABLE IF NOT EXISTS profiles (
    user VARCHAR(16),
    text VARCHAR(4096),
    INDEX(user(6))
  )');
?>

    <br>...done.
  </body>
</html>
```



For this example to work, you must first ensure that you have created the database specified in the variable `$db` in [Example 22-1](#) and granted access to the user given the name in `$dbuser`, with the password in `$dbpass`.

index.php

The *index.php* file just displays a simple welcome message, but it is necessary to give the project a home page. In a finished application, this would be where you sell the virtues of your site to encourage signups.

Incidentally, because we have already set up all the MySQL tables and created the included files, you can now load [Example 22-4](#), *index.php*, into your browser to get your first peek at the new application. It should look like [Figure 22-1](#).

Example 22-4. index.php

```
<?php // Example 04: index.php
require_once 'header.php';

echo "<div class='center'>Welcome to Robin's Nest,";

if ($loggedin) {
    $user_html_entities = htmlentities($_SESSION['user']);
    echo " $user_html_entities, you are logged in";
} else
    echo ' please sign up or log in';
?>

</div>
</div>
<h4 id="footer" class="center">Web App from <i>
    <a href="https://github.com/RobinNixon/lpmj7" target="_blank">
        Learning PHP MySQL & JavaScript</a>
    </i></h4>
</body>
</html>
```

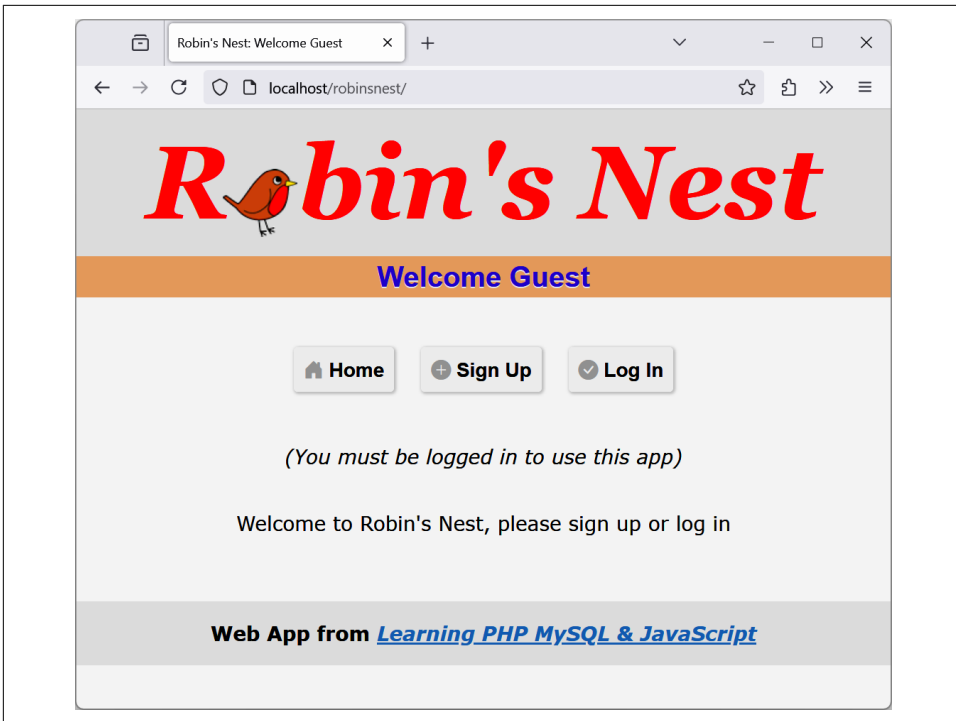


Figure 22-1. The main page of the app

signup.php

Now we need a module to enable users to join our new social network, and that's **Example 22-5**, *signup.php*. This is a slightly longer program, but you've seen all its parts before.

Let's start by looking at the end block of HTML. This is a simple form that allows a username and password to be entered. But note the use of the empty `<span>` given the id of used. This will be the destination of the asynchronous call in this program that checks whether a desired username is available. See **Chapter 17** for a complete description of how this works.

Checking for Username Availability

At the end of the HTML you'll see a block of JavaScript. It contains an anonymous arrow function that is called on the JavaScript `blur` event when focus is removed from the username field of the form. The function makes a request to the program *checkuser.php*, which reports whether the username in `user` is available. The returned result of the asynchronous call (performed using the `fetch` function), a friendly message, is then placed in the used `<span>`.

Go back to the program start: there's some PHP code you should recognize from the discussion of form validation in [Chapter 16](#). This section also uses placeholders and prepared statements when looking up the username in the database and, if it's not already taken, inserting the new username and password. The password is not stored in the clear, as that would be a huge security risk. Instead its one-way hash is used (see [Chapter 12](#) for more details).

Logging In

Upon successfully signing up, the user is then prompted to log in. A more fluid response at this point might be to automatically log in a newly created user, but because I don't want to overly complicate the code, I have kept the signup and login modules separate. You can easily implement this if you want to, however.

When loaded into a browser (and in conjunction with *checkuser.php*, shown later), this program will look like [Figure 22-2](#), where you can see that the asynchronous call has identified that the username *Robin* is available. If you would like the password field to show only asterisks, change its type from `text` to `password`.

Remember that you must run *setup.php* before you can run any of these other PHP program files.

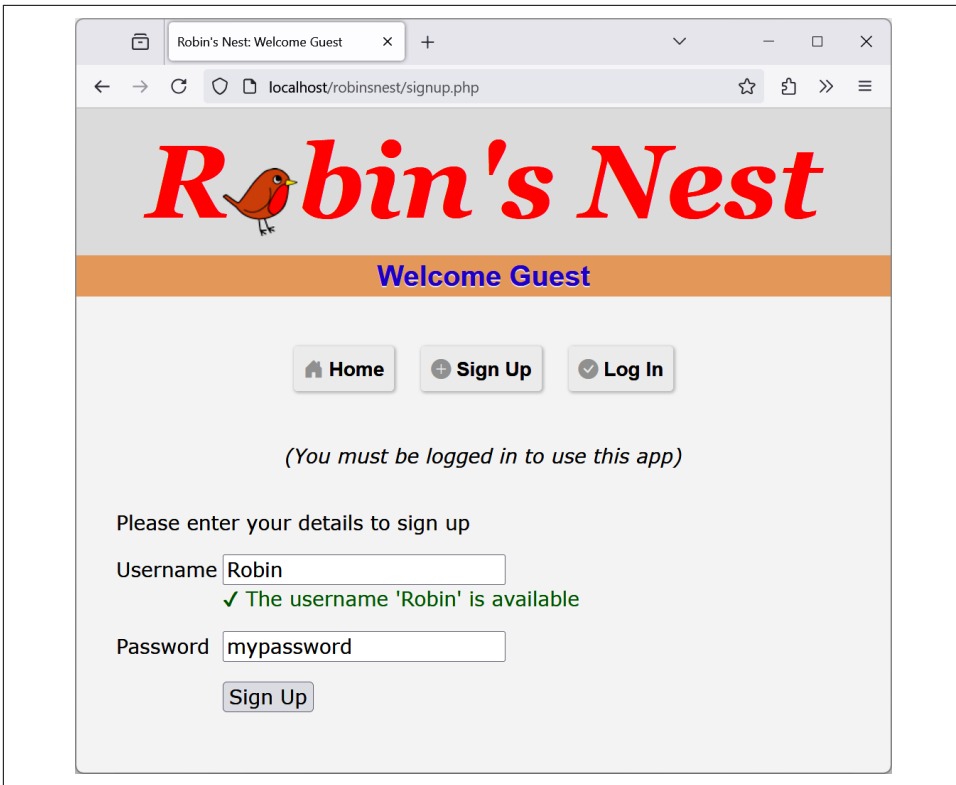


Figure 22-2. The signup page

Example 22-5. *signup.php*

```
<?php // Example 05: signup.php
require_once 'header.php';

$error = $user = "";
if (isset($_SESSION['user']))
    destroySession();

if (isset($_POST['user'])) {
    $user = $_POST['user'];
    if ($_POST['user'] === "" || $_POST['pass'] === "")
        $error = 'Not all fields were entered';
    else {
        $stmt = $pdo->prepare('SELECT * FROM members WHERE user=?');
        $stmt->execute([$user]);
        if ($stmt->rowCount())
            $error = 'That username already exists<br><br>';
        else {
            $stmt = $pdo->prepare('INSERT INTO members VALUES(?, ?)');
```

```

        $stmt->execute([$user, password_hash($_POST['pass'], PASSWORD_DEFAULT)]);
        die('<h4>Account created</h4>Please Log in.</div></body></html>');
    }
}
}
$error_html_entities = htmlentities($error);
$user_html_entities = htmlentities($user);
?>

<form method="post" action="signup.php">
  <p class="error">
    <?php echo $error_html_entities; ?>
  </p>
  <p>Please enter your details to sign up</p>
  <p>
    <label>Username</label>
    <input type="text" maxlength="16" name="user" id="username"
      value="<?php echo $user_html_entities; ?>"><br>
    <label></label><span id="used">&nbsp;</span>
  </p>
  <p>
    <label>Password</label>
    <input type="text" name="pass">
  </p>
  <p>
    <label></label>
    <input type="submit" value="Sign Up">
  </p>
</form>
<script>
  const field = byId('username');
  field.onblur = () => {
    if (field.value === '')
      return
    const data = new FormData()
    data.set('user', field.value)
    fetch('checkuser.php', { method: 'post', body: data })
      .then(response => response.text())
      .then(text => byId('used').innerHTML = text)
  }
</script>
</div>
</body>
</html>

```

checkuser.php

To go with *signup.php*, here's **Example 22-6**, *checkuser.php*, which looks up a username in the database and returns a string indicating whether it has already been taken. Because it relies on the `$pdo` variable to use prepared statements, the program first includes the file *functions.php*.

Then, if the `$_POST` variable `user` has a value, the function looks it up in the database and, depending on whether it exists as a username, outputs either “Sorry, the username ‘*user*’ is taken” or “The username ‘*user*’ is available.” Just checking the value returned by the function call to `$stmt->rowCount` is sufficient for this, as it will return 0 if the name is not found or 1 if it is found.

The HTML entities `✘` and `✔` are also used to preface the string with either a cross or a checkmark, and the string will be displayed in either red for the class `taken` or green for the class `available`, as defined in *styles.css*, shown later in this chapter.

Example 22-6. *checkuser.php*

```
<?php // Example 06: checkuser.php
require_once 'functions.php';

if (isset($_POST['user'])) {
    $stmt = $pdo->prepare('SELECT * FROM members WHERE user=?');
    $stmt->execute([$_POST['user']]);

    $user_html_entities = htmlentities($_POST['user']);
    if ($stmt->rowCount())
        echo "<span class='taken'>&#x2718; " .
            "The username '$user_html_entities' is taken</span>";
    else
        echo "<span class='available'>&#x2714; " .
            "The username '$user_html_entities' is available</span>";
}
?>
```

login.php

With users now able to sign up on the site, [Example 22-7](#), *login.php*, provides the code needed to let them log in. Like the signup page, it features a simple HTML form and some basic error checking, and it uses prepared statements and placeholders to query the MySQL database.

Two things to note here: first is that to verify a password stored as a one-way hash, the row with the hash needs to be queried from the database by the username. Then the hash from the database, together with the password from the login field, is passed to the `password_verify` function, which returns true if the password from the form matches the stored hash. This is a bit more complicated than just comparing two strings, so we’ll leave it all to `password_verify`.

Second, upon successful verification of the username and password, the session variable `user` is given the username. As long as the current session remains active, this variable will be accessible by all the programs in the project, allowing them

to automatically provide access to logged-in users. Storing the password or the password hash in the session is not needed and would be a security risk as it would be stored in the clear in the session data.

The header function, upon successfully logging in, redirects the user to the home page once logged in. The function sends a special HTTP header `Location` that will cause the redirection in the browser, followed by a URL, in this case the *members.php* filename and the username in the view parameter.

When you call this program up in your browser, it should look like [Figure 22-3](#). Note how the input type of password has been used here to mask the password with asterisks to prevent it from being viewed by anyone looking over the user's shoulder.

Example 22-7. *login.php*

```
<?php // Example 07: login.php
require_once 'header.php';
$error = $user = "";

if (isset($_POST['user'])) {
    $user = $_POST['user'];
    if ($user === "" || $_POST['pass'] === "")
        $error = 'Not all fields were entered';
    else {
        $stmt = $pdo->prepare('SELECT user,pass FROM members WHERE user=?');
        $stmt->execute([$user]);
        $result = $stmt->fetchAll();

        if (count($result) === 0
            || !password_verify($_POST['pass'], $result[0]['pass']))
        {
            $error = "Invalid login attempt";
        } else {
            $_SESSION['user'] = $user;
            header('Location: members.php?view=' . $user);
        }
    }
}
$error_html_entities = htmlentities($error);
$user_html_entities = htmlentities($user);
?>

<form method="post" action="login.php">
    <p class="error">
        <?php echo $error_html_entities; ?>
    </p>
    <p>
        Please enter your details to log in
    </p>
    <p>
        <label>Username</label>
```

```

        <input type="text" maxlength="16" name="user"
            value="<?php echo $user_html_entities; ?>">
    </p>
    <p>
        <label>Password</label>
        <input type="password" name="pass">
    </p>
    <p>
        <label></label>
        <input type="submit" value="Login">
    </p>
</form>
</div>
</body>
</html>

```

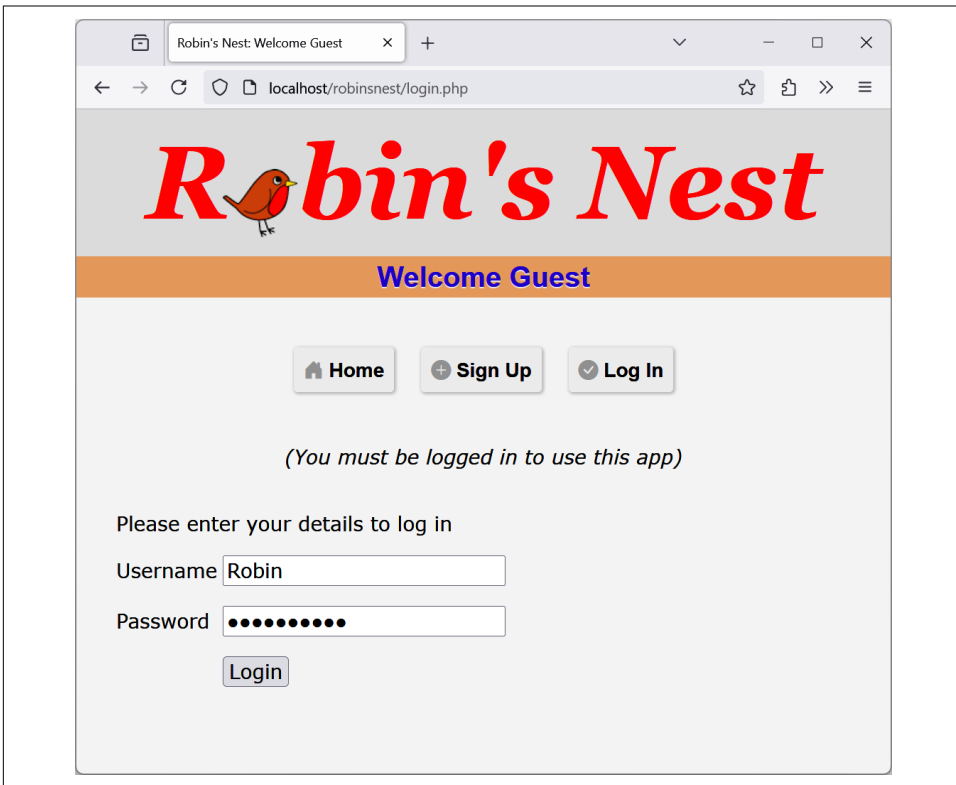


Figure 22-3. The login page

profile.php

One of the first things that new users may want to do after signing up and logging in is to create a profile, which can be done via [Example 22-8](#), *profile.php*. There's some interesting code here, such as routines to upload, resize, and sharpen images.

Let's start by looking at the main HTML at the end of the code. This is like the forms you've just seen, but this time it has the parameter `enctype="multipart/form-data"`. This allows us to send more than one type of data at a time, enabling the posting of an image as well as text. There's also an input type of file, which creates a Browse button that a user can click to select a file to be uploaded.

When the form is submitted, the code at the start of the program is executed. The first thing it does is ensure that a user is logged in before allowing program execution to proceed. Only then is the page heading displayed.

Adding the “About Me” Text

Next, the `$_POST` variable `text` is checked to see whether some text was posted to the program. If so, all long whitespace sequences (including carriage returns and line feeds) are replaced with single spaces. This program checks that the user's profile already exists in the database and if it does, the text that is the user's “about me” will be updated; otherwise a new profile text is inserted. Notice how two different queries are prepared, but `execute` is called only once, because while the queries are different, both use the same data, `$text` and `$user`.

If no text was posted, the database is queried to see whether any text already exists to prepopulate the `<textarea>` for the user to edit it. `htmlspecialchars` is used to sanitize the output against XSS attacks, similar to other outputs in this web application.

Adding a Profile Image

Next we move on to the section where the `$_FILES` system variable is checked to see whether an image has been uploaded. If so, a string variable called `$saveto` is created, based on the user's username followed by the extension `.jpg`. For example, a user called *Jill* will cause `$saveto` to have the value *Jill.jpg*. This is the file where the uploaded image will be saved for use in the user's profile.

Following this, the uploaded image type is examined and is accepted only if it is a `.jpeg`, `.png`, or `.gif` image. Besides the image's dimensions, the function `getimagesize` also returns the image type (as one of the `IMAGETYPE_XXX` constants, see the [PHP manual](#) for the list) by examining the file itself, as used here.

We store the image's dimensions in `$w`, `$h`, and the `$type` using the following statement, which is a quick way of assigning values from an array to separate variables:

```
list($w, $h, $type) = getimagesize($saveto);
```



Don't use the `$_FILES` array to get the image type. That may not be correct because that type comes from the browser and could, for example, be modified by the attacker.

Upon success, the variable `$src` is populated with the uploaded image using one of the `imagecreatefrom` functions, according to the image type uploaded. The image is now in a raw format that PHP can process. If the image is not of an allowed type, the flag `$typeok` is set to `FALSE`, preventing the final section of image upload code from being processed.

Processing the Image

After the correct file is uploaded, using the value of `$max` (which is set to 100), we calculate new dimensions that will result in a new image of the same ratio but with no dimension greater than 100 pixels. This results in giving the variables `$tw` and `$th` the new values needed. If you want smaller or larger thumbnails, simply change the value of `$max` accordingly.

Next, the function `imagecreatetruecolor` is called to create a new, blank canvas `$tw` wide and `$th` high in `$tmp`. Then `imagecopyresampled` is called to resample the image from `$src` to the new `$tmp`. Sometimes resampling images can result in a slightly blurred copy, so the next piece of code uses the `imageconvolution` function to sharpen the image a bit.

Finally, the image is saved as a `.jpeg` file in the location defined by the variable `$saveto`, after which we remove both the original and the resized image canvases from memory using the `imagedestroy` function, returning the memory that was used.

Displaying the Current Profile

Last but not least, so that the user can see what the current profile looks like before editing it, the `showProfile` function from *functions.php* is called prior to outputting the form HTML. If no profile exists yet, nothing will be displayed.

When a profile image is displayed, CSS is applied to it to provide a border, a shadow, and a margin to its right, to separate the profile text from the image. The result of loading [Example 22-8](#) into a browser is shown in [Figure 22-4](#), where you can see that the `<textarea>` has been prepopulated with the “about me” text.

Example 22-8. profile.php

```
<?php // Example 08: profile.php
require_once 'header.php';

if (!$loggedin)
    die("</div></body></html>");
$user = $_SESSION['user'];

echo "<h3>Your Profile</h3>";

$stmt = $pdo->prepare('SELECT * FROM profiles WHERE user=?');
$stmt->execute([$user]);

if (isset($_POST['text'])) {
    $text = preg_replace('/\s\s+/', ' ', $_POST['text']);
    $text_html_entities = htmlentities($_POST['text']);

    if ($stmt->rowCount()) {
        $stmt2 = $pdo->prepare('UPDATE profiles SET text=:text WHERE user=:user');
    } else
        $stmt2 = $pdo->prepare('INSERT INTO profiles VALUES(:user, :text)');
    $stmt2->execute([':text' => $text, ':user' => $user]);
} else {
    if ($stmt->rowCount()) {
        $row = $stmt->fetch();
        $text_html_entities = htmlentities($row['text']);
    }
    else $text_html_entities = "";
}

if (isset($_FILES['image']['name'])) {
    $saveto = "$user.jpg";
    move_uploaded_file($_FILES['image']['tmp_name'], $saveto);
    $typeok = TRUE;

    $info = getimagesize($saveto);
    if ($info) {
        list($w, $h, $type) = $info;
        switch ($type) {
            case IMAGETYPE_GIF: $src = imagecreatefromgif($saveto); break;
            case IMAGETYPE_JPEG: $src = imagecreatefromjpeg($saveto); break;
            case IMAGETYPE_PNG: $src = imagecreatefrompng($saveto); break;
            default: $typeok = FALSE; break;
        }
    }
}
```

```

else
    $typeok = FALSE;

if ($typeok) {
    $max = 100;
    $tw = $w;
    $th = $h;

    if ($w > $h && $max < $w) {
        $th = $max / $w * $h;
        $tw = $max;
    } elseif ($h > $w && $max < $h) {
        $tw = $max / $h * $w;
        $th = $max;
    } elseif ($max < $w) {
        $tw = $th = $max;
    }

    $tmp = imagecreatetruecolor($tw, $th);
    imagecopyresampled($tmp, $src, 0, 0, 0, 0, $tw, $th, $w, $h);
    imageconvolution($tmp, array(array(-1, -1, -1),
        array(-1, 16, -1), array(-1, -1, -1)), 8, 0);
    imagejpeg($tmp, $saveto);
    imagedestroy($tmp);
    imagedestroy($src);
}
}

showProfile($user, $pdo);
?>
<form method="post"
    action="profile.php" enctype="multipart/form-data">
<h3>Enter or edit your details and/or upload an image</h3>
<textarea
    name="text" cols="50"><?php echo $text_html_entities; ?></textarea>
<p>Image: <input type="file" name="image" size="14"></p>
<input type="submit" class="button" value="Save Profile">
</form>
</div>
</body>
</html>

```

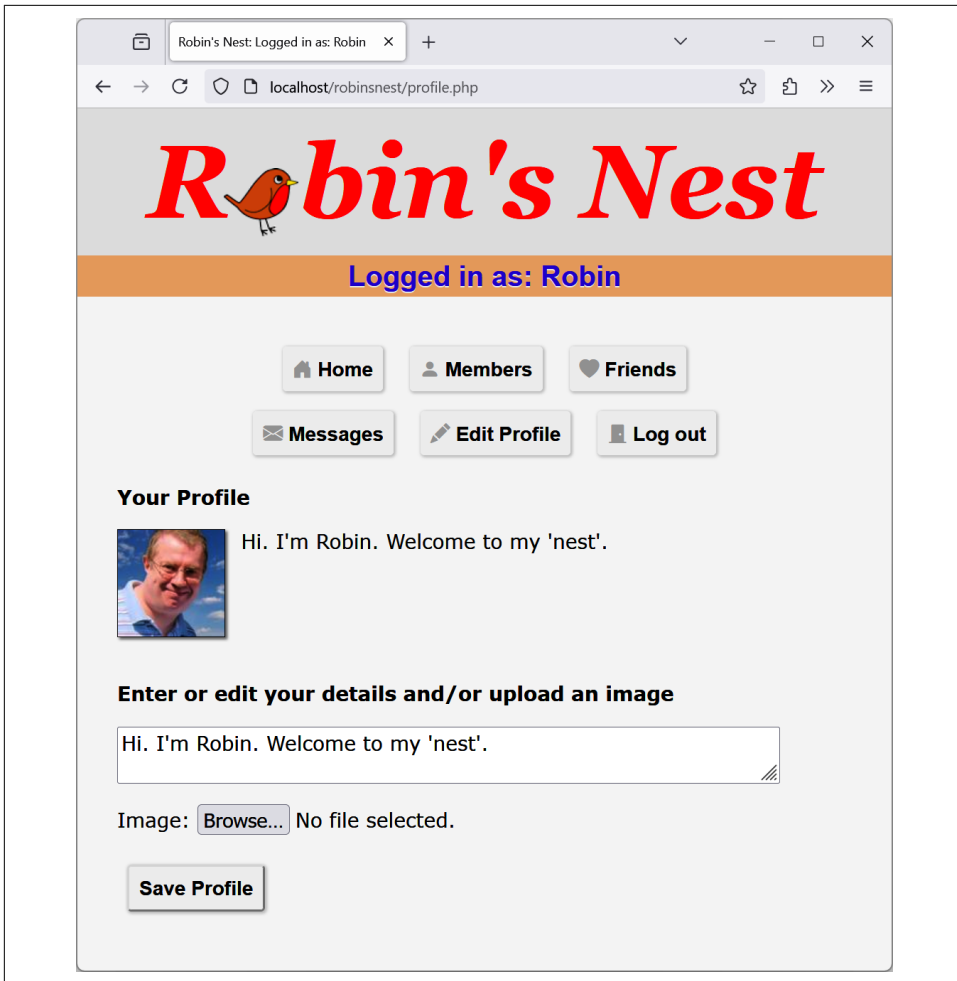


Figure 22-4. Editing a user profile

members.php

Using [Example 22-9](#), *members.php*, your users will be able to find other members and choose to add them as friends (or drop them if they are already friends). This program has two modes. The first lists all members and their relationships to you, and the second shows a user's profile.

Viewing a User's Profile

The code for the latter mode comes first, where a test is made for the variable `view`, retrieved from the `$_GET` array. If it exists, a user wants to view someone's profile, so the program does that using the `showProfile` function, along with providing a couple of links to the user's friends and messages, sanitizing the URL parameter against malicious input.

Adding and Dropping Friends

After that, the two `$_GET` parameters `add` and `remove` are tested. If one or the other has a value, it will be the username of a user to either add or drop as a friend. We achieve this by looking up the user in the MySQL *friends* table and either inserting the username or removing it from the table.

And, of course, every database query is done using placeholders and prepared statements to ensure that the data is safe to use with MySQL.

Listing All Members

The final section of code issues an SQL query to list all usernames. A `while` loop then iterates through every member, fetching their details and then looking them up in the *friends* table to see if they are being followed by or are following the user. If someone is both a follower and a followee, they are classed as a mutual friend.

The variable `$t1` is nonzero when the user is following another member, and `$t2` is nonzero when another member is following the user. Depending on these values, text is displayed after each username, showing the relationship (if any) to the current user.

Icons are also displayed to show the relationships. A double-pointing arrow means that the users are mutual friends, a left-pointing arrow indicates the user is following another member, and a right-pointing arrow indicates that another member is following the user.

Finally, depending on whether the user is following another member, a link is provided to either add or drop that member as a friend.

When you call up [Example 22-9](#) in a browser, it will look like [Figure 22-5](#). Note how the user is invited to “follow” a nonfollowing member, but if the member is already following the user, a “recip” link to reciprocate the friendship is offered. In the case of a user already following another member, the user can select “drop” to end the following.

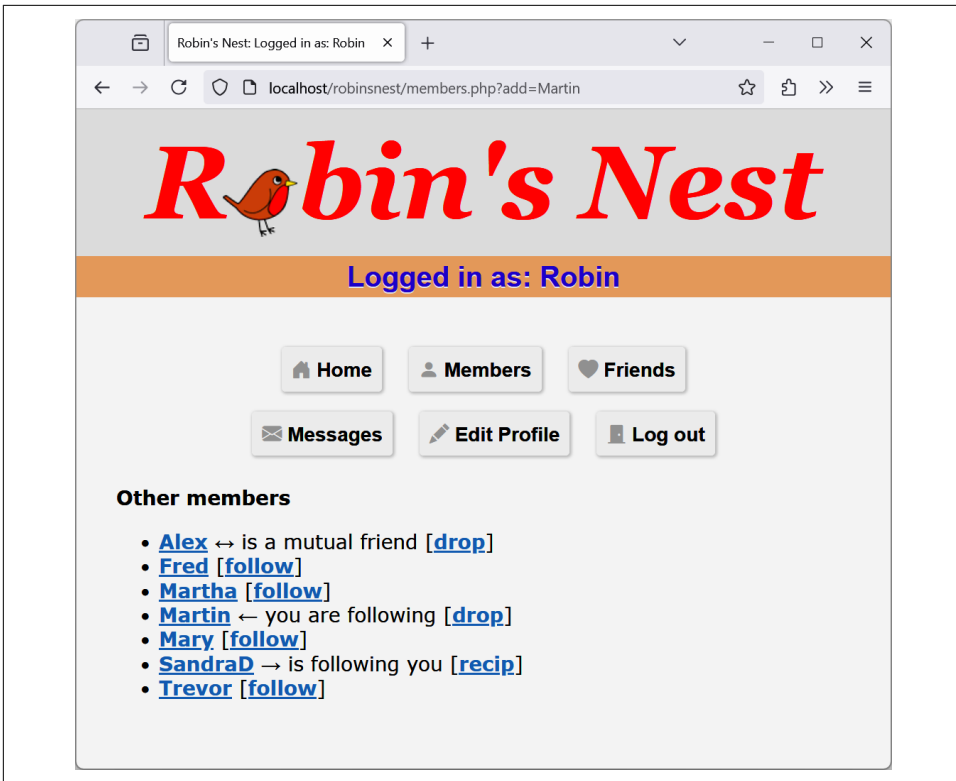


Figure 22-5. Using the members module



On a production server, there could be thousands or even millions of users, so you would substantially modify this program to include support for searching the “about me” text, paging the output a screen at a time, and so on.

Example 22-9. members.php

```
<?php // Example 09: members.php
require_once 'header.php';

if (!$loggedin)
    die("</div></body></html>");
$user = $_SESSION['user'];

if (isset($_GET['view'])) {
    $view = $_GET['view'];
    $view_html_entities = htmlentities($view);

    if ($_GET['view'] === $user)
```

```

        $name = "Your";
    else
        $name = "$view_html_entities's";

    echo "<h3>$name Profile</h3>";
    showProfile($view, $pdo);
    echo "<a class='button'
        href='messages.php?view=$view_html_entities'>View $name messages</a>";
    die("</div></body></html>");
}

if (isset($_GET['add'])) {
    $stmt = $pdo->prepare('SELECT * FROM friends WHERE user=? AND friend=?');
    $stmt->execute([$_GET['add'], $user]);
    if (!$stmt->rowCount()) {
        $stmt = $pdo->prepare("INSERT INTO friends VALUES (?, ?)");
        $stmt->execute([$_GET['add'], $user]);
    }
} elseif (isset($_GET['remove'])) {
    $stmt = $pdo->prepare('DELETE FROM friends WHERE user=? AND friend=?');
    $stmt->execute([$_GET['remove'], $user]);
}
?>

<p><strong>Other members</strong></p>
<ul>

<?php
    $stmt = $pdo->prepare("SELECT user FROM members ORDER BY user");
    $stmt->execute();
    if (!$stmt->rowCount()) {
        echo '<li>No other members</li>';
    }
    while ($row = $stmt->fetch()) {
        if ($row['user'] === $user)
            continue;
        $rowuser_html_entities = htmlentities($row['user']);

        echo "<li><a
            href='members.php?view=$rowuser_html_entities'>$rowuser_html_entities</a>";
        $follow = "follow";

        $stmt2 = $pdo->prepare('SELECT * FROM friends WHERE user=? AND friend=?');
        $stmt2->execute([$row['user'], $user]);
        $t1 = $stmt2->rowCount();

        $stmt2->execute([$user, $row['user']]);
        $t2 = $stmt2->rowCount();

        if (($t1 + $t2) > 1)
            echo " &harr; is a mutual friend";
        elseif ($t1)
            echo " &larr; you are following";
        elseif ($t2) {

```

```

        echo " &rarr; is following you";
        $follow = "recip";
    }

    if (!$t1)
        echo " [<a href='members.php?add=$rowuser_html_entities'>$follow</a>]";
    else
        echo " [<a href='members.php?remove=$rowuser_html_entities'>drop</a>]";
    }
?>
    </ul>
</div>
</body>
</html>

```

friends.php

The module that shows a user's friends and followers is [Example 22-10](#), *friends.php*. This interrogates the *friends* table just like the *members.php* program but only for a single user. It then shows all of that user's mutual friends and followers along with the people they are following.

All the followers are saved into an array called `$followers`, and all the people being followed are placed in an array called `$following`. Then a neat piece of code is used to extract all of those who are both following and followed by the user, like this:

```
$mutual = array_intersect($followers, $following);
```

The `array_intersect` function extracts all members common to both arrays and returns a new array containing only those people. This array is then stored in `$mutual`. Now it's possible to use the `array_diff` function for each of the `$followers` and `$following` arrays to keep only those people who are *not* mutual friends, like this:

```

$followers = array_diff($followers, $mutual);
$following = array_diff($following, $mutual);

```

This results in the array `$mutual` containing only mutual friends, `$followers` containing only followers (and no mutual friends), and `$following` containing only people being followed (and no mutual friends).

With these arrays, it's a simple matter to separately display each category of members, as in [Figure 22-6](#). The PHP `sizeof` function (alias of the `count` function) returns the number of elements in an array; here I use it just to trigger code when the size is nonzero (that is, when friends of that type exist). Note how, by using the variables `$name1`, `$name2`, and `$name3` in the relevant places, the code can tell when you're looking at your own friends list, using the words *Your* and *You are*, instead of simply

displaying the username. The commented line can be uncommented if you wish to display the user's profile information on this screen.

Example 22-10. friends.php

```
<?php // Example 10: friends.php
require_once 'header.php';

if (!$loggedin)
    die("</div></body></html>");
$user = $_SESSION['user'];

if (isset($_GET['view'])) {
    $view = $_GET['view'];
} else {
    $view = $user;
}
$view_html_entities = htmlentities($view);

if ($view === $user) {
    $name1 = $name2 = "Your";
    $name3 = "You are";
} else {
    $name1 = "<a
        href='members.php?view=$view_html_entities'>$view_html_entities</a>'s";
    $name2 = "$view_html_entities's";
    $name3 = "$view_html_entities is";
}

// Uncomment this line if you wish the user's profile to show here
// showProfile($view);

$followers = $following = [];

$stmt = $pdo->prepare('SELECT * FROM friends WHERE user=?');
$stmt->execute([$view]);
while ($row = $stmt->fetch()) {
    $followers[] = $row['friend'];
}

$stmt = $pdo->prepare('SELECT * FROM friends WHERE friend=?');
$stmt->execute([$view]);
while ($row = $stmt->fetch()) {
    $following[] = $row['user'];
}

$mutual = array_intersect($followers, $following);
$followers = array_diff($followers, $mutual);
$following = array_diff($following, $mutual);
$friends = FALSE;
```

```

echo "<br>";

if (sizeof($mutual)) {
    echo "<span class='subhead'>$name2 mutual friends</span><ul>";
    foreach ($mutual as $friend) {
        $fr_html_entities = htmlentities($friend);
        echo "<li><a
            href='members.php?view=$fr_html_entities'>$fr_html_entities</a>";
    }
    echo "</ul>";
    $friends = TRUE;
}

if (sizeof($followers)) {
    echo "<span class='subhead'>$name2 followers</span><ul>";
    foreach ($followers as $friend)
        $fr_html_entities = htmlentities($friend);
    echo "<li><a
        href='members.php?view=$fr_html_entities'>$fr_html_entities</a>";
    echo "</ul>";
    $friends = TRUE;
}

if (sizeof($following)) {
    echo "<span class='subhead'>$name3 following</span><ul>";
    foreach ($following as $friend)
        $fr_html_entities = htmlentities($friend);
    echo "<li><a
        href='members.php?view=$fr_html_entities'>$fr_html_entities</a>";
    echo "</ul>";
    $friends = TRUE;
}

if (!$friends)
    echo "<br>You don't have any friends yet.";
?>
</div>
</body>
</html>

```

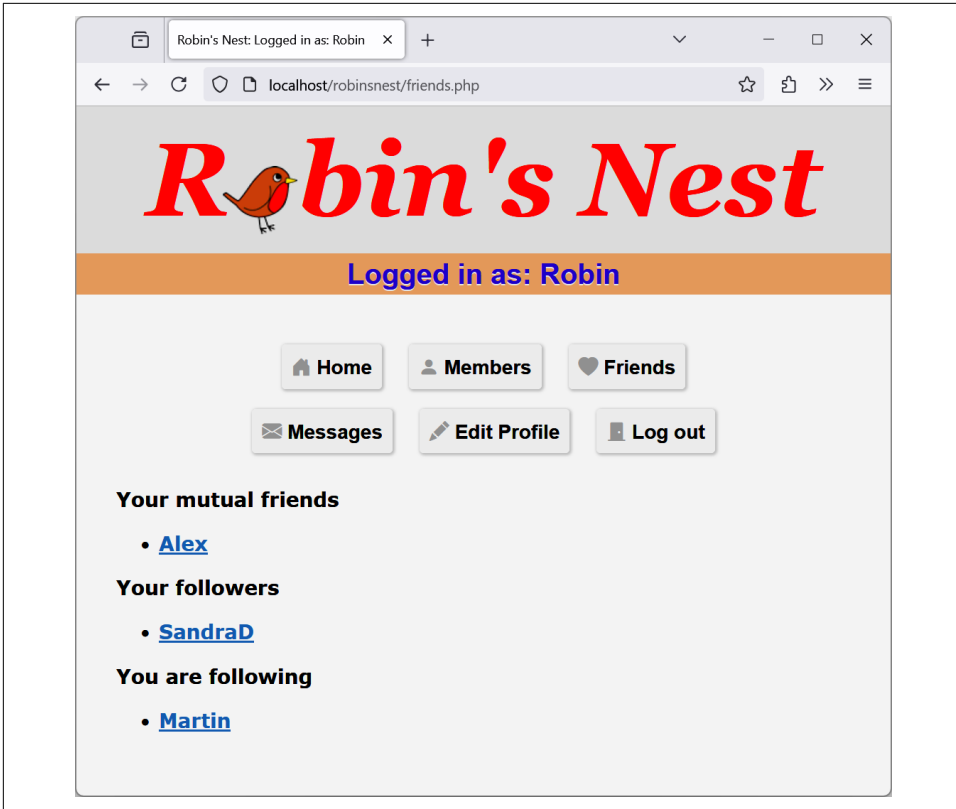


Figure 22-6. Displaying a user's friends and followers

messages.php

The last of the main modules is **Example 22-11**, *messages.php*. The program starts by checking whether a message has been posted in the variable `text`. If so, it is inserted into the *messages* table. At the same time, the value of `pm` is also stored. This indicates whether a message is private or public. A `0` represents a public message, and `1` is private.

Next, the user's profile and a form for entering a message are displayed, along with radio buttons to choose between a private or public message. After this, all the messages are shown, depending on whether they are private or public. If they are public, all users can see them, but private messages are visible only to the sender and recipient. This is all handled by a couple of queries to the MySQL database. Additionally, when a message is private, it is introduced by the word *whispered* and shown in italic.

Finally, the program displays a couple of links to refresh the messages (in case another user has posted one in the meantime) and to view the user's friends. The trick using the variables `$name1` and `$name2` is used again so that when you view your own profile, the word *Your* is displayed instead of the username.

Example 22-11. messages.php

```
<?php // Example 11: messages.php
require_once 'header.php';

if (!$loggedin)
    die("</div></body></html>");
$user = $_SESSION['user'];

if (isset($_GET['view'])) {
    $view = $_GET['view'];
} else {
    $view = $user;
}
$view_html_entities = htmlentities($view);

if (isset($_POST['text']) && $_POST['text'] !== "") {
    $stmt = $pdo->prepare('INSERT INTO messages VALUES(NULL, ?, ?, ?, ?, ?)');
    $stmt->execute([$user, $view, (int)$_POST['pm'], time(), $_POST['text']]);
}

if ($view !== "") {
    if ($view === $user)
        $name1 = $name2 = "Your";
    else {
        $name1 = "<a
            href='members.php?view=$view_html_entities'>$view_html_entities</a>'s";
        $name2 = "$view_html_entities's";
    }

    echo "<h3>$name1 Messages</h3>";
    showProfile($view, $pdo);
?>

<form method="post"
    action="messages.php?view=<?php echo $view_html_entities; ?>">
    <p>Type here to leave a message</p>
    <p>
        <input type="radio" name="pm" id="public"
            value="0" checked="checked">
        <label for="public">Public</label>
        <input type="radio" name="pm" id="private" value="1">
        <label for="private">Private</label>
    </p>
    <textarea name="text" cols="50"></textarea>
    <br>
    <input type="submit" class="button" value="Post Message">

```

```

        </form><br>
<?php
    date_default_timezone_set('UTC');

    if (isset($_GET['erase'])) {
        $stmt = $pdo->prepare('DELETE FROM messages WHERE id=? AND recip=?');
        $stmt->execute([(int)$_GET['erase'], $user]);
    }

    $stmt = $pdo->prepare('SELECT * FROM messages
        WHERE recip=? ORDER BY time DESC');
    $stmt->execute([$view]);
    $num = $stmt->rowCount();

    while ($row = $stmt->fetch()) {
        $pm = $row['pm'] === '1';
        $auth_html_entities = htmlentities($row['auth']);
        $message_html_entities = htmlentities($row['message']);
        $id_html_entities = htmlentities($row['id']);

        if (!$pm || $row['auth'] === $user || $row['recip'] === $user) {
            echo date('M jS \y g:ia:', $row['time']);
            echo " <a href='messages.php?view=$auth_html_entities'>
                $auth_html_entities</a> ";

            if (!$pm)
                echo "wrote: &quot;$message_html_entities&quot; ";
            else
                echo "whispered: <span class='whisper'>
                    &quot;$message_html_entities&quot;</span> ";

            if ($row['recip'] === $user)
                echo "[<a href='messages.php?view=$view_html_entities' .
                    "&erase=$id_html_entities'>erase</a>]";

            echo "<br>";
        }
    }
}

if (!$num)
    echo "<br><span class='info'>No messages yet</span><br><br>";

echo "<br><a class='button'
    href='messages.php?view=$view_html_entities'>Refresh messages</a>";
?>

</div>
</body>
</html>

```


You can see the result of viewing this program with a browser in [Figure 22-7](#). Note how users viewing their own messages are provided with links to erase any they don't want to keep.

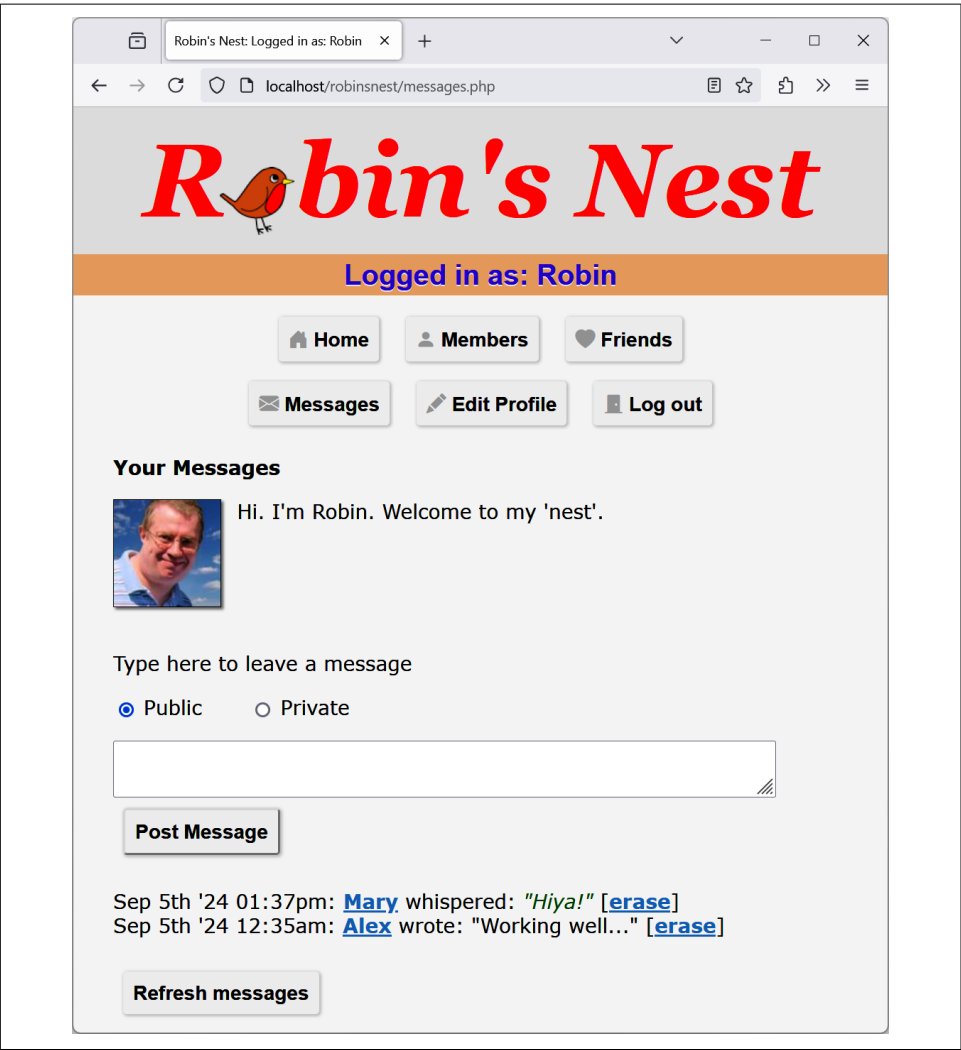


Figure 22-7. The messaging module

logout.php

The final ingredient in our social networking recipe is [Example 22-12](#), *logout.php*, the logout page that closes a session and deletes any associated data and cookies. The result of calling up this program is an HTTP redirect to *index.php* and, unless you're not logged in, there's no other output.

Example 22-12. logout.php

```
<?php // Example 12: logout.php
require_once 'header.php';

if (isset($_SESSION['user'])) {
    destroySession();
    header('Location: index.php');
} else
    echo "<div class='center'>You cannot log out because
        you are not logged in</div>";
?>

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

styles.css

The stylesheet used for this project is shown in [Example 22-13](#). Most of the sets of declarations should be clear, but these might require some explanation:

*

Sets the default font family and size for the project using the universal selector.

.content

Sets the maximum width for the page content block which, unlike setting width, allows the content to be smaller if the device screen is smaller. It also centers the block by setting the left and right margins to auto.

.button

The properties set here make sure that links with this CSS class appear as buttons.

.button:hover

If you move your mouse cursor over such button, the background color of the button changes to the color specified here.

.button:focus

Clicking the button will add a blue glow around it, as specified by this declaration.

`.button i`

The color for the icons (from the Bootstrap Icons collection) in the buttons is set here.

`#robin`

Aligns the image of the robin in the page title and removes the shadow that's otherwise added to other pictures.

`#used`

Ensures the element populated by the *checkuser.php* asynchronous call if a user-name is already taken is not too close to the field above it.

Example 22-13. styles.css

```
/* Example 13: styles.css */

* {
  font-family: Verdana, sans-serif;
  font-size: 14pt;
}

body {
  margin: 0;
  padding: 0;
}

html {
  background: #f8f8f8
}

img {
  border: 1px solid black;
  margin-right: 15px;
  box-shadow: 2px 2px 2px #888;
}

a {
  color: #38c;
  font-weight: bold;
}

label {
  display: inline-block;
  width: 5rem;
  text-align: left;
}

textarea {
  max-width: 100%;
}
```

```

.content {
  padding :2rem;
  margin :0 auto;
  max-width:700px;
}

.button {
  font-family :Arial, sans-serif;
  background-color:#f3f3f3;
  padding :0.5rem;
  margin :0.5rem;
  text-decoration :none;
  color :black;
  border-radius :0.2rem;
  box-shadow :1px 1px 3px 1px #d3d3d3;
  font-weight :bold;
  white-space :nowrap;
  display :inline-block;
}

.button:hover {
  background-color: #e3e3e3;
}

.button:focus {
  box-shadow:0 0 12px #38c;
}

.button i {
  color: #b5b5b5;
}

.username {
  text-align :center;
  background :#eb8;
  color :#40d;
  font-family:helvetica;
  font-size :20pt;
  padding :4px;
  text-shadow:0 1px 0 #eee;
  font-weight:bold;
}

.info {
  font-style :italic;
  margin :40px 0px;
  text-align :center;
}

.center {
  text-align:center;
}

```

```

}

.subhead {
  font-weight:bold;
}

.taken, .error {
  color      :red;
  font-weight:bold;
}

.available {
  color:green;
}

.whisper {
  font-style:italic;
  color      :#006600;
}

#logo {
  font-family:Georgia;
  font-weight:bold;
  font-style :italic;
  font-size  :97px;
  color      :red;
  padding    : 0.75rem;
  background-color: #e9e9e9;
}

#robin {
  position      :relative;
  border        :0px;
  margin-left   : -6px;
  margin-right  :0px;
  top           :17px;
  box-shadow    :0px 0px 0px;
}

#used {
  margin-top: 0.2rem;
}

#footer {
  padding: 1rem;
  background-color: #e9e9e9;
}

```

javascript.js

Finally, there's the JavaScript file (see [Example 22-14](#)), which contains the `byId`, `style`, and `by` functions used throughout this book.

Example 22-14. javascript.js

```
function byId(id)
{
    return document.getElementById(id)
}

function style(selector)
{
    return document.querySelector(selector).style
}

function by(selector)
{
    return document.querySelectorAll(selector)
}
```

And that, as they say, is that. If you write anything based on this code or any other examples in this book, or have gained in any other way from it, then I am glad to have been of help, and I thank you for reading this book.

Questions

1. Have you enjoyed learning from this book?

See “[Chapter 22 Answers](#)” on page 586 in the [Appendix](#) for the answer to this question.

Solutions to the Chapter Questions

Chapter 1 Answers

1. A web server (such as Apache), a server-side scripting language (PHP), a database (MySQL), and a client-side scripting language (JavaScript).
2. *HyperText Markup Language*: the web page itself, including text and markup tags.
3. Like nearly all SQL-based database engines, MySQL accepts commands in *Structured Query Language* (SQL). SQL is the way that every user (including a PHP program) communicates with MySQL.
4. PHP runs on the server, whereas JavaScript runs on the client. PHP can communicate with the database to store and retrieve data, but it can't alter the user's web page quickly and dynamically. JavaScript has the opposite benefits and drawbacks. With Node.js, JavaScript can also be used on the server.
5. *Cascading Style Sheets*: styling and layout rules applied to the elements in an HTML document.
6. Probably the most interesting new elements in HTML5 are `<audio>`, `<video>`, and `<canvas>`, although there are many others, such as `<article>`, `<summary>`, `<footer>`.
7. Some of these technologies are controlled by companies that accept bug reports and fix the errors like any software company. But open source software also depends on a community, so your bug report may be handled by any user who understands the code well enough. You may someday fix bugs in an open source tool yourself.

8. It allows developers to concentrate on building the core functionality of a website or web app, passing on to the framework the task of making sure it always looks and runs its best, regardless of the platform (whether Linux, macOS, Windows, iOS, or Android), the dimensions of the screen, or the browser it runs.
9. The event-driven model of Node.js is superior to the Apache web server mainly because it is nonblocking, and therefore a substantially greater number of connections can be supported.

Chapter 2 Answers

1. WAMP stands for *Windows, Apache, MySQL, PHP*. The *M* in MAMP stands for *Mac* instead of Windows, and the *L* in LAMP stands for *Linux*. They all refer to a complete solution for hosting dynamic web pages.
2. SFTP stands for *SSH File Transfer Protocol* (sometimes *Secure File Transfer Protocol*). An SFTP program, similar to an FTP program, is used to transfer files back and forth between a client and a server but unlike FTP, SFTP is secure.
3. Transferring files to a remote server to update them can substantially increase development time if this action is carried out many times and the internet connection is slow.
4. Dedicated code editors are smart and can highlight problems in your code before you run it.

Chapter 3 Answers

1. The tag used is `<?php...?>`. It can be shortened to `<?...?>`, but that is not recommended practice.
2. You can use `//` for a single-line comment or `/*...*/` to span multiple lines.
3. All PHP statements must end with a semicolon (`;`).
4. With the exception of constants, all PHP variables must begin with `$`.
5. A variable holds a value that can be a string, a number, or other data.
6. `$variable = 1` is an assignment statement, whereas the `==` in `$variable == 1` and the `===` in `$variable === 1` is a comparison operator. Use `$variable = 1` to set the value of `$variable`. Use `$variable === 1` to find out later in the program whether `$variable` equals the number 1. If you use `$variable == 1`,

it returns true even if `$variable` is a string "1", which can cause unexpected bugs. If you mistakenly use `$variable = 1` where you meant to do a comparison, it will do two things you probably don't want: set `$variable` to 1 and return a true value all the time, no matter its previous value.

7. In PHP, the hyphen is reserved for the subtraction, decrement, and negation operators. A construct like `$current-user` would be harder to interpret if hyphens were also allowed in variable names, and in any case would lead programs to be ambiguous.
8. Yes, variable names are case-sensitive. `$This_Variable` is not the same as `$this_variable`.
9. You cannot use spaces in variable names, as this would confuse the PHP parser. Instead, try using the `_` (underscore), or use `camelCase` notation.
10. You can use type casting to convert the type, for example like `$number = (int)$string`. However, if type declarations are not used and you want to convert one variable type to another, reference it and PHP will automatically convert it for you.
11. There is no difference between `++$j` and `$j++` unless the value of `$j` is being tested, assigned to another variable, or passed as a parameter to a function. In such cases, `++$j` increments `$j` before the test or other operation is performed, whereas `$j++` performs the operation and then increments `$j`.
12. Generally, the operators `&&` and `and` are interchangeable except where precedence is important, in which case `&&` has a high precedence, while `and` has a low one.
13. You can use multiple lines within quotation marks or the `<<<_END..._END;` construct to create a multiline echo or assignment. In the latter case, the closing tag must be on a line by itself with nothing before or after it.
14. You cannot redefine constants because, by definition, once defined they retain their value until the program terminates.
15. You can use `\'` or `\"` to escape either a single or double quote.
16. The `echo` and `print` commands are similar in that they are both constructs, except that `print` behaves like a PHP function and takes a single argument, while `echo` can take multiple arguments.
17. The purpose of functions is to separate discrete sections of code into their own self-contained sections that can be referenced by a single function name.

18. You can make a variable accessible to all parts of a PHP program by declaring it as `global`. However this is not a recommended approach in production code.
19. If you generate data within a function, you can convey the data to the rest of the program by returning a value or modifying a global variable.
20. When you combine a string with a number, the result is another string.

Chapter 4 Answers

1. When converting to string, `TRUE` is represented as the string value `"1"`, and `FALSE` is represented as an empty string, so `"1"` and `""` will be printed instead.
2. The simplest forms of expressions are literals (such as numbers and strings) and variables, which simply evaluate to themselves.
3. The difference between unary, binary, and ternary operators is the number of operands each requires (one, two, and three, respectively).
4. The best way to force your own operator precedence is to place parentheses around subexpressions that you wish to give high precedence.
5. Operator associativity refers to the direction of processing (left to right or right to left).
6. Use the identity operator when you wish to avoid PHP's automatic operand type changing (also called *type casting*), and the bugs it could introduce.
7. The three conditional statement types are `if`, `switch`, and the `?:` operator.
8. To skip the current iteration of a loop and move to the next one, use a `continue` statement.
9. The difference between `for` and `while` is that loops using `for` statements support two additional parameters to control the loop handling.
10. Most conditional expressions in `if` and `while` statements are literals (or Booleans) and therefore trigger execution when they evaluate to `TRUE`. Numeric expressions trigger execution when they evaluate to a nonzero value. String expressions trigger execution when they evaluate to a nonempty string. A `NULL` value is evaluated as false and therefore does not trigger execution.

Chapter 5 Answers

1. Using functions prevents the need to copy or rewrite similar code sections many times over by combining sets of statements so they can be called by a simple name.
2. By default, a function can return a single value. But by utilizing arrays, references, and global variables, it can return any number of values.
3. When you use a variable by name, such as by assigning its value to another variable or by passing its value to a function, its value is copied. The original does not change when the copy is changed. But if you reference a variable, only a pointer (or reference) to its value is used so that a single value is referenced by more than one name. Changing the value of the reference will change the original as well.
4. Scope refers to which parts of a program can access a variable. For example, a variable of global scope can be accessed by all parts of a PHP program.
5. To incorporate one file within another, you can use the `include` or `require` directives or their safer variants, `include_once` and `require_once`.
6. A function is a set of statements referenced by a name that can receive and return values. An object may contain zero or many functions (which are then called methods) as well as variables (which are called properties), all combined in a single unit.
7. To create a new object in PHP, use the `new` keyword like this:

```
$object = new Class;
```
8. To create a subclass, use the `extends` keyword with syntax such as this:

```
class SubClass extends ParentClass { ... }
```
9. To cause an object to be initialized when you create it, you can call a piece of initializing code by creating a constructor method called `__construct` within the class and place your code there.
10. Using properties not explicitly declared emits a deprecation notice starting with PHP 8.2. Previously, they were implicitly declared upon first use. But declaring them has always been considered good practice as it helps with code readability and debugging and is especially useful to other people who may have to maintain your code.

Chapter 6 Answers

1. A numeric array can be indexed numerically using numbers or numeric variables. An associative array uses alphanumeric identifiers to index elements.
2. The main benefit of the `array` keyword is that it enables you to assign several values at a time to an array without repeating the array name.
3. Besides using the `foreach` loop to walk through an associative array, you can also use functions like `key`, `current`, and `next` to code your own way of walking through an array. Calling `next` modifies the internal array pointer, but using `foreach` does not.
4. To create a multidimensional array, you need to assign additional arrays to elements of the parent array.
5. You can use the `count` function to count the number of elements in an array.
6. The purpose of the `explode` function is to extract sections from a string that are separated by an identifier, such as extracting words separated by spaces within a sentence.
7. To reset PHP's internal pointer into an array back to the first element, call the `reset` function.

Chapter 7 Answers

1. The conversion specifier you would use to display a floating-point number is `%f`.
2. To take the input string "Happy Birthday" and output the string "**Happy", you could use a `printf` statement such as this:

```
printf("%'*7.5s", "Happy Birthday");
```

3. To send the output from `printf` to a variable instead of to a browser, you would use `sprintf` instead.
4. To create a Unix timestamp for 7:11 a.m. on May 2, 2025, you could use this command:

```
$timestamp = mktime(7, 11, 0, 5, 2, 2025);
```

5. You would use the `w+` file access mode with `fopen` to open a file in write and read mode, with the file truncated and the file pointer at the start.
6. The PHP command for deleting the file *file.txt* is:

```
unlink('file.txt');
```

7. The PHP function `file_get_contents` is used to read in an entire file in one go. It will also read a file from across the internet if provided with a URL.
8. The PHP superglobal associative array `$_FILES` contains the details about uploaded files.
9. The PHP `exec` function enables running system commands.
10. To turn any special characters returned by the system into ones that HTML can understand and properly display, use the `htmlspecialchars` function.

Chapter 8 Answers

1. The semicolon in MySQL separates or ends commands. If you forget to enter it, MySQL will issue a prompt and wait for you to enter the semicolon.
2. To see the available databases, type `SHOW databases`. To see tables within a database that you are using, type `SHOW tables`. (These commands are case-insensitive.)
3. To create this new user, use the `GRANT` command like this:

```
GRANT PRIVILEGES ON newdatabase.* TO 'newuser'@'localhost'  
IDENTIFIED BY 'newpassword';
```
4. To view the structure of a table, type `DESCRIBE tablename`.
5. The purpose of a MySQL index is to substantially decrease database access times by adding metadata to the table about one or more key columns, which can then be quickly searched to locate rows within a table.
6. A `FULLTEXT` index enables natural-language queries to find keywords, wherever they are in the `FULLTEXT` column(s), in much the same way as using a search engine.
7. A *stopword* is a word that is so common that it is considered not worth including in a `FULLTEXT` index or using in searches. However, it is included in searches when it is part of a larger string bounded by double quotes.
8. `SELECT DISTINCT` essentially affects only the display, choosing a single row and eliminating all the duplicates. `GROUP BY` does not eliminate rows but combines all the rows that have the same value in the column. Therefore, `GROUP BY` is useful for performing an operation such as `COUNT` on groups of rows. `SELECT DISTINCT` is not useful for that purpose.

9. To return only those rows containing the word *Langhorne* somewhere in the column *author* of the table *classics*, use a command such as this:

```
SELECT * FROM classics WHERE author LIKE "%Langhorne%";
```

10. When you're joining two tables together, they must share at least one common column, such as an ID number or, as in the case of the *classics* and *customers* tables, the *isbn* column.

Chapter 9 Answers

1. The term *relationship* refers to the connection between two pieces of data that have some association, such as a book and its author or a book and the customer who bought the book. A relational database such as MySQL specializes in storing and retrieving such relationships.
2. The process of removing duplicate data and optimizing tables is called *normalization*.
3. The three rules of First Normal Form are:
 - There should be no repeating columns containing the same kind of data.
 - All columns should contain a single value.
 - There should be a primary key to uniquely identify each row.
4. To satisfy Second Normal Form, columns whose data repeats across multiple rows should be removed to their own tables.
5. In a one-to-many relationship, the primary key from the table on the “one” side must be added as a separate column (a foreign key) to the table on the “many” side.
6. To create a database with a many-to-many relationship, you create an intermediary table containing keys from two other tables. The other tables can then reference one another via the third.
7. To initiate a MySQL transaction, use the `BEGIN` or `START TRANSACTION` command. To terminate a transaction and cancel all actions, issue a `ROLLBACK` command. To terminate a transaction and commit all actions, issue a `COMMIT` command.
8. To examine how a query will work in detail, you can use the `EXPLAIN` command.
9. To back up the database *publications* to a file called *publications.sql*, you would use a command such as:

```
mysqldump -u user -ppassword publications > publications.sql
```

Chapter 10 Answers

1. To connect to a MySQL database with PDO, create a new object of the PDO class, passing the attributes, username, password, and options. A connection object will be returned on success.
2. To submit a query to MySQL using PDO, ensure you have first created a connection object to a database, and then call its query method, passing the query string.
3. The PDO: :FETCH_NUM style of the fetch method can be used to return a row as an array indexed by column number.
4. You can manually close a PDO connection by assigning the value null to the PDO object used to connect to the database.
5. When adding a row to a table with an AUTO_INCREMENT column, you should pass the value null to that column.
6. To escape special characters in strings, you can call the quote method of a PDO connection object, passing it the string to be escaped. Of course, for security, using prepared statements will serve you best.
7. The best way to ensure database security when accessing it is to use placeholders and prepared statements.

Chapter 11 Answers

1. The associative arrays used to pass submitted form data to PHP are \$_GET for the GET method and \$_POST for the POST method.
2. The difference between a text box and a text area is that although they both accept text for form input, a text box is a single line, whereas a text area can be multiple lines and include word wrapping.
3. To offer three mutually exclusive choices in a web form, you should use radio buttons, because checkboxes allow multiple selections.
4. You can submit a group of selections from a web form using a single field name by using an array name with square brackets, such as choices[], instead of a regular field name. Each value is then placed into the array, whose length will be the number of elements submitted.
5. To submit a form field without the user seeing it, place it in a hidden field using the attribute type="hidden".

6. You can encapsulate a form element and supporting text or graphics, making the entire unit selectable with a mouse click, by using the `<label>` and `</label>` tags.
7. To prevent XSS attacks, that is, to convert HTML into a format that can be displayed but will not be interpreted as HTML by a browser, use the PHP `htmlspecialchars` or `htmlspecialchars` function.
8. You can help users complete fields with data they may have submitted elsewhere by using the `autocomplete` attribute, which prompts the user with possible values.
9. To ensure that a form is not submitted with missing data, you can apply the `required` attribute to essential inputs.

Chapter 12 Answers

1. Cookies should be transferred before a web page's HTML because they are sent as part of the headers.
2. To store a cookie in a web browser, use the `setcookie` function.
3. To destroy a cookie, reissue it with `setcookie`, but set its expiration date in the past.
4. Using HTTP authentication, the username and password are stored in `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`.
5. The `password_hash` function is a powerful security measure because it is a one-way function that converts a string to a long hexadecimal string of numbers that cannot be converted back quickly, and is therefore very hard to crack as long as strong passwords are required from users (for example, at least eight characters in length, including randomly placed numbers and punctuation marks).
6. When a string is salted, extra characters known only by the web server (or by the programmer if they are self-salting the code) are added to it before hash conversion (this should normally be left up to PHP to handle for you, though, don't add any extra salt when you're using the `password_hash` function for example). This ensures that users with the same password will not have the same hash and prevents the use of precomputed hash tables.
7. A PHP session is a group of variables unique to the current user, stored on the server, passed along with successive requests so that the variables remain available as the user visits different pages.

8. To initiate a PHP session, use the `session_start` function.
9. Session hijacking is where a hacker somehow discovers an existing session ID and attempts to take it over.
10. Session fixation is when an attacker attempts to force a user to log in using a valid session ID the attacker has obtained earlier, thus compromising the connection's security.

Chapter 13 Answers

1. To enclose JavaScript code, you use `<script>` and `</script>` tags.
2. You can include JavaScript code from other files in your documents by either copying and pasting them or, more commonly, including them as part of a `<script src='filename.js'>` tag.
3. The equivalent of the `echo` and `print` commands used in PHP for quick output are the JavaScript functions `console.log`, `alert`, `document.write`, or writing directly into elements.
4. To create a comment in JavaScript, preface it with `//` for a single-line comment or surround it with `/*` and `*/` for a multiline comment.
5. The JavaScript string concatenation operator is the `+` symbol.
6. Within a JavaScript function, you can define a variable that has local scope by preceding it with the `let` or `var` keywords upon first assignment.
7. To display the URL assigned to the link with an id of `thislink` in all main browsers, you can use either of these commands:

```
console.log(document.getElementById('thislink').href)
console.log(thislink.href)
```

8. The commands to change to the previous page in the browser's history array are:

```
history.back()
history.go(-1)
```

9. To replace the current document with the main page at the [O'Reilly website](http://oreilly.com), you could use this command:

```
document.location.href = 'http://oreilly.com'
```

Chapter 14 Answers

1. The most noticeable difference between Boolean values in PHP and JavaScript is that PHP recognizes the keywords `TRUE`, `true`, `FALSE`, and `false`, whereas only `true` and `false` are supported in JavaScript. Additionally, in PHP when converted to string, `TRUE` has a value of `"1"`, and `FALSE` is an empty string; in JavaScript they are represented by string values `"true"` and `"false"`.
2. Unlike PHP, no character (such as `$`) is used to define a JavaScript variable name. JavaScript variable names can start with and contain uppercase and lowercase letters as well as underscores; names can also include digits but not as the first character.
3. The difference between unary, binary, and ternary operators is the number of operands each requires (one, two, and three, respectively).
4. The best way to force your own operator precedence is to surround the parts of an expression to be evaluated first with parentheses.
5. Use the identity operator when you wish to avoid JavaScript's automatic operand type changing and the bugs it could introduce.
6. The simplest forms of expressions are literals (such as numbers and strings) and variables, which simply evaluate to themselves.
7. The three conditional statement types are `if`, `switch`, and the `?:` operator.
8. Most conditional expressions in `if` and `while` statements are literals or Booleans and therefore trigger execution when they evaluate to `true`. Numeric expressions trigger execution when they evaluate to a nonzero value. String expressions trigger execution when they evaluate to a nonempty string. A `NULL` value is evaluated as `false` and therefore does not trigger execution.
9. Loops using `for` statements give you extensive control over the loop as they support two additional parameters to control the loop handling. You can use `while` loops if you have a single variable to control the loop.
10. In JavaScript, to cast one type to another you can use one of the built-in functions such as `parseInt` or `parseFloat`.

Chapter 15 Answers

1. JavaScript functions and variable name are case-sensitive. The variables `Count`, `count`, and `COUNT` are all different.
2. To write a function that accepts and processes an unlimited number of parameters, use the *rest parameter* syntax `...params`, or as the less preferred option, access parameters through the `arguments` array, which is a member of all functions.
3. One way to return multiple values from a function is to place them all inside an array and return the array.
4. When defining a class, use the `this` keyword to refer to the current object.
5. Yes, if using the recommended `class` syntax. If functions are used to create the object, the methods do not have to be defined within the class definition. If a method is defined outside the constructor, the method name must be assigned to the `this` object within the class definition.
6. New objects are created via the `new` keyword.
7. To create a multidimensional array, place subarrays inside the parent array.
8. In JavaScript, objects can be used to emulate “associative arrays,” so you use the object syntax (*key* : *value*, within curly braces) to create such “array,” as in:

```
assocarray = {  
  forename : "Paul",  
  surname  : "McCartney",  
  group    : "The Beatles"  
}
```

9. A statement to sort an array of numbers into descending numerical order would look like this, using an anonymous arrow function:

```
numbers.sort((a, b) => b - a)
```

Chapter 16 Answers

1. You can send a form for validation prior to submitting it by attaching the `submit` event handler to the form using the `addEventListener` method. To prevent the form values being submitted on any validation error, use `event.preventDefault()`.
2. To match a string against a regular expression in JavaScript, use the `test` method.

3. Regular expressions to match characters not in a word could be any of `/[^\\w]/`, `/[\\W]/`, `/^\\w/`, `/\\W/` `/[^a-zA-Z0-9_]/`, and so on.
4. A regular expression to match either of the words *fox* or *fix* could be `/f[oi]x/`.
5. A regular expression to match any single word followed by any nonword character could be `/\\w+\\W/g`.
6. A JavaScript function using regular expressions to test whether the word *fox* exists in the string *The quick brown fox* could be:

```
console.log(/fox/.test("The quick brown fox"))
```
7. A PHP function using a regular expression to replace all occurrences of the word *the* in *The cow jumps over the moon* with the word *my* could be as follows:

```
$s = preg_replace("/the/i", "my", "The cow jumps over the moon");
```
8. The HTML attribute used to precomplete form fields with a value is `value`, which is placed within an `<input>` tag and takes the form `value="value"`.

Chapter 17 Answers

1. The `fetch` function can be used for conducting asynchronous communication between a server and JavaScript client.
2. You can pass an options object with `method`: "POST" and `body`: data properties as the second parameter when calling `fetch`.
3. The `fetch` function returns an object of class `Promise`, which represents an asynchronous operation. You can use its `then` method to pass a code, as an arrow function for example, that will be called when the operation completed successfully.
4. The function to get JSON data may be an arrow function and can be as simple as this (it will return a promise object):

```
response => response.json()
```
5. After calling the `fetch` function that returns a promise, pass an arrow function that returns the JSON data as the parameter to the `then` method. The `json` method also returns a promise. Then pass another arrow function to the `then` call on the promise, which can look like this:

```
data => {  
  // do something with data.a and data.b  
}
```

6. The origin of the URL `https://book.example/ch18?q=6` is `https://book.example` as origin consists of schema + domain + port if specified, which is not the case here.
7. If a JavaScript code running on `https://example.com/map` would like to send an asynchronous request to `https://www.example.com/data`, then it would be a cross-origin request, not a same-origin one, because the origin of `https://example.com/map` (which is `https://example.com`) doesn't match the origin of `https://www.example.com/data` (which is `https://www.example.com`). Note that unlike the former URL, the latter one uses the `www` subdomain.
8. The best way for JavaScript on `http://localhost/info` to access the response from `https://example.com/data` would be to add one of the two following HTTP headers to the response:

```
Access-Control-Allow-Origin: http://localhost
Access-Control-Allow-Origin: *
```

Chapter 18 Answers

1. The CSS operators `^=`, `$=`, and `*=` match the start, end, or any portion of a string, respectively.
2. The property you use to specify the size of a background image is `background-size`, like this:

```
background-size:800px 600px;
```

3. You can specify the radius of a border using the `border-radius` property:

```
border-radius:20px;
```

4. To flow text over multiple columns, use the `column-count`, `column-gap`, and `column-rule` properties, like this:

```
column-count:3;
column-gap :1em;
column-rule :1px solid black;
```

5. The four functions you can use to specify CSS colors are `hsl`, `hsla`, `rgb`, and `rgba`. For example:

```
color:rgba(0%,60%,40%,0.4);
```

6. To create a gray shadow under some text, offset diagonally to the bottom right by 5 pixels, with a blurring of 3 pixels, use this declaration:

```
text-shadow:5px 5px 3px #888;
```

7. You can indicate that text is truncated with an ellipsis by using this declaration:

```
text-overflow:ellipsis;
```

8. To include a Google web font such as Lobster in a web page, first select it from the [Google Fonts website](https://fonts.google.com/), then copy the provided <link> tag into the <head> of your HTML document. It will look something like this:

```
<link href='https://fonts.googleapis.com/css?family=Lobster'  
      rel='stylesheet'>
```

You can then refer to the font in a CSS declaration:

```
h1 { font-family:'Lobster', arial, serif; }
```

9. The CSS declaration you would use to rotate an object by 90 degrees is:

```
transform:rotate(90deg);
```

10. To set up a transition on an object so that when any of its properties are changed the change will transition immediately and linearly over the course of half a second, use this declaration:

```
transition:all .5s linear;
```

11. To create a flexbox container set its display property to flex.
12. To define how flex items are spaced along the main axis of a flexbox, use the justify-content property of the container.
13. To control the alignment of flex items along the cross axis of a flexbox container use the align-items property of the container.
14. The two properties that determine how much a flexbox element can grow and shrink are flex-grow and flex-shrink.
15. To change the order of elements in a flexbox container, assign numeric values to the order properties of the individual elements, with lower values appearing first.
16. To set up a CSS grid layout, set the display property of the container object to either grid or inline-grid.
17. To specify the flow direction of a grid container, use the container's grid-auto-flow property.
18. You can place items into a grid by using the items' grid-column and grid-row properties.

19. You can set the spacing of a grid with the `gap` property, which also has an older property name of `grid-gap`; both work the same way.
20. You can align grid items vertically and horizontally using the `justify-items` and `align-items` properties.

Chapter 19 Answers

1. The function to abbreviate DOM access uses the `getElementById` method, like this:

```
function getById(id)
{
    return document.getElementById(id)
}
```

Or it can use the `querySelector` method, for example like this:

```
function getBy(selector)
{
    return document.querySelector(selector)
}
```

Given an element with an ID `box`, you'd then call the functions as follows:

```
getById('box')
getBy('#box') // note the # prefix
```

2. You can modify a CSS attribute of an object using the `setAttribute` function, like this:

```
myobject.setAttribute('font-size', '16pt')
```

You can also (usually) modify an attribute directly (using slightly modified property names where required), like this:

```
myobject.fontSize = '16pt'
```

3. The properties that provide the width and height available in a browser window are `window.innerHeight` and `window.innerWidth`.
4. To make something happen when the mouse passes over and out of an object, attach the `mouseover` and `mouseout` events.
5. To create a new element, use code such as:

```
elem = document.createElement('span')
```

To add the new element to the DOM, use code such as:

```
document.body.appendChild(elem)
```

6. To make an element invisible, set its `visibility` property to `hidden` (set it to `visible` to restore it again). To collapse an element's dimensions to zero, set its `display` property to `none` (setting this property to `block` is one way to restore it to its original dimensions).
7. To set up a single event at a future time, call the `setTimeout` function, passing it the code or function name to execute and the time delay in milliseconds.
8. To set up repeating events at regular intervals, use the `setInterval` function, passing it the code or function name to execute and the time delay between repeats in milliseconds.
9. To release an element from its location in a web page to enable it to be moved around, set its `position` property to `relative`, `absolute`, or `fixed`. To restore it to its original place, set the property to `static`.
10. To achieve an animation rate of 50 frames per second, you should set a delay between interrupts of 20 milliseconds. To calculate this value, divide 1,000 milliseconds by the desired frame rate.

Chapter 20 Answers

1. You can incorporate the React scripts in your web page by downloading the files and serving them from your own web server, or by using a CDN such as *unpkg.com*. Then load the scripts using `script` tags in your HTML document.
2. To incorporate XML into your React JavaScript, you first need to load the Babel extension, either locally or from a CDN, using a `script` tag.
3. JSX JavaScript `<script>` sections of code require `type="text/babel"` to work.
4. You can extend React to your code either as a class using `class Name extends React.Component` or simply by returning code to be rendered by a function's return statement. In both cases, `ReactDOM.render` must be called to initiate the rendering.
5. In React, pure code doesn't change its inputs, whereas code that does change inputs is considered impure.
6. React keeps track of state with the props object and its attributes.
7. To embed an expression within JSX code, you place it within curly braces, like this: `Hello {props.name}`.
8. Once a class has been constructed, you can change the state of a value only by using the `setState` function.

9. To enable referring to props using the `this` keyword within a constructor, you must first call the `super` method, passing it props, like this: `super(props)`.
10. You can create a conditional statement in JSX using the `&&` operator after the expression. For an `if...then...else` statement, you can use the ternary `?:` operator.

Chapter 21 Answers

1. After receiving a request for a file from a web browser and passing the request to the filesystem, Node.js returns to listening for new browser requests. Only when the filesystem sends an event to indicate the requested file is ready does it send the file contents back to the browser.
2. To include a prewritten module in Node.js, load it using the `import` method.
3. Node.js uses the `http` module to manage HTTP communications, the `url` module for URL parsing, and the `fs` module for accessing the local filesystem.
4. The default HTTP port a server listens to is port 80. To avoid port clashes multiple servers may use different port numbers, such as port 8000, which is often used for proxies. Note that the default port for HTTPS is 443.
5. The `createServer` method of an `http` module is called to create a new server object.
6. Headers, if any, must be sent before returning data to a web browser. One way to do this is by calling the `writeHead` method of the server object.
7. To terminate a Node.js connection with a web browser, call the `end` method of the response object passed as a parameter to the `createServer` function of the `http` module.
8. To start a Node.js server, from the command line type `node server.js`, where `server.js` is the filename of the server to run. Use the `.mjs` file extension to run a module.
9. To manually terminate a Node.js server, press Ctrl-C at the command line.
10. To write from Node.js to the terminal window command line, pass a string to the `console.log` or `console.error` functions.
11. To add external Node.js modules to a project, use the `npm` program, like this: `npm install packagename`.

12. To access a MySQL database with Node.js, first install the `mysql2` package using `npm` with the command `npm install mysql2`, then import it into a project using the `import` method:

```
import mysql from 'mysql2/promise'
```

13. To create a connection to MySQL (once you have installed and imported the module) call the `createConnection` method of the `mysql` object, like this:

```
const connection = await mysql.createConnection({  
  host: 'localhost',  
  user: 'node',  
  password: 'letmein',  
  database: 'publications'  
})
```

14. To use Node.js to query a MySQL database, call the `execute` method of an already created connection object, passing it the query string with placeholders.
15. To terminate a connection to a MySQL database, call the `end` method of its connection object.

Chapter 22 Answers

1. The answer to this question is entirely up to you. If you have enjoyed this book, please tell your friends and leave a review at an online bookstore. If you have any questions, comments, suggestions, or addenda, [please visit the book's O'Reilly catalog page](#) and leave them there. Thanks for reading!

Symbols

- ! (not) logical operator
 - JavaScript, 321, 340
 - PHP, 45, 72-74
- != (not equal to) operator
 - JavaScript, 320, 337
 - PHP, 44, 72
- !== (not identical to) operator
 - JavaScript, 320, 337
 - PHP, 44
- " (quotation mark, double)
 - JavaScript escape character, 322
 - PHP
 - printf parameter string, 135
 - variable value in string, 48
- "> MySQL prompt, 167
- #id (ID) selectors (CSS), 416
- \$ (dollar) symbol
 - JavaScript variable and function names, 317
 - PHP variables, 37
 - porting into JavaScript, 317
 - regular expressions, 389
- \$= operator (CSS), 417
- \$GLOBALS array (PHP), 60
- \$_COOKIE array (PHP), 60
- \$_ENV array (PHP), 60
- \$_FILES array (PHP), 60, 154
 - social networking site profile image, 547
 - do not use for image type, 548
- \$_GET array (PHP), 60, 246
 - asynchronous program
 - GET method, 409-410
 - JSON requests, 410-412
 - sanitizing via htmlentities function, 61, 264, 283
 - social networking site
 - adding or dropping friends, 552
 - viewing a user profile, 552
- \$_POST array (PHP), 60, 246
 - asynchronous program POST method, 404-407
 - sanitizing via htmlentities function, 61, 264, 283
 - social networking site
 - About Me text, 547
 - username check, 543
- \$_REQUEST array (PHP), 60
- \$_SERVER array (PHP), 60
 - HTTP authentication, 293
 - example program, 298-301
 - htmlspecialchars function, 294
 - starting a session, 302
 - validating username and password, 295
- \$_SESSION array (PHP), 60, 302-304
 - IP address for session security, 306
 - user-agent string for session security, 307
- % (modulus) operator
 - JavaScript, 319
 - PHP, 43
- % printf specifier (PHP), 135
- % wildcard (MySQL), 193
- %= (modulus assignment) operator
 - JavaScript, 320
 - PHP, 43
- && (And) logical operator
 - JavaScript, 321, 340
 - short-circuit evaluation, 341

- PHP, 45
 - higher precedence than and, or, 73
- ' (quotation mark, single)
 - JavaScript escape character, 322
- PHP
 - literal strings, 48
 - printf argument string, 136
- '> MySQL prompt, 167
- ()
 - functions
 - JavaScript, 324, 372, 373
 - PHP, 94, 96
 - regular expression grouping, 385
- * (asterisk) in regular expressions, 383
 - HTML tag match expressions, 384
- * (multiplication) operator
 - JavaScript, 319
 - PHP, 43
- * (universal) selector (CSS), 416
- *= (multiplication and assignment) operator
 - JavaScript, 320
 - PHP, 43
- *= (substring) operator (CSS), 417
- + (addition) operator
 - JavaScript, 319
 - PHP, 43
- + (plus sign) in regular expressions, 383
 - HTML tag match expressions, 384
- + (string concatenation) operator (JavaScript), 321
- + with MATCH...AGAINST Boolean mode, 197
- ++ (increment) operator
 - JavaScript, 319, 321, 335
 - PHP, 43, 46
- += (addition and assignment) operator
 - JavaScript, 320, 321
 - string concatenation, 322
 - PHP, 43, 46
- (hyphen) for range in regular expressions, 386
- (subtraction) operator
 - JavaScript, 319
 - PHP, 43
- with MATCH...AGAINST Boolean mode, 197
- (decrement) operator
 - JavaScript, 319, 321, 335
 - PHP, 43, 46
- = (subtraction and assignment) operator
 - JavaScript, 320, 321
- PHP, 43, 46
 - > for object access (PHP), 105, 109
 - > MySQL prompt, 167
- . (dot) in regular expressions, 384
 - HTML tag match expressions, 384
 - matching the dot character, 385
- . (period) in JavaScript, 328
 - window object properties, 468
- . (string concatenation) operator (PHP), 47
- ... (rest parameter) syntax (JavaScript), 354
 - example fixNames function, 356
- ... (spread) syntax (JavaScript), 368
- .= (concatenation assignment) operator (PHP), 43, 47
- / (division) operator
 - JavaScript, 319
 - PHP, 43
- / (forward slash)
 - /* and */ for multiline comments
 - JavaScript, 316
 - PHP, 36
 - // for single line comments
 - JavaScript, 316
 - PHP, 35
- /*> MySQL prompt, 167
- /= (division and assignment) operator
 - JavaScript, 320
 - PHP, 43
- : bind variable prefix (PHP), 262
- : in associative arrays (JavaScript), 364
- :: (scope resolution) operator (PHP), 111, 113
- ;(semicolon)
 - JavaScript, 313, 317
 - MySQL, 167
 - none for PHP accessing MySQL, 239
 - \c after a semicolon, 168
- PHP, 36
 - for loops, 88
- < (less than) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- <<< (heredoc) operator (PHP), 49
 - example of form code, 243, 247
- <<< (nowdoc) operator (PHP), 51
- <= (less than or equal to) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- <> (not equal to) operator (PHP), 44
- <?php and ?> (PHP), 7, 33

- <? and ?> deprecated, 34
- heredoc operator alternative, 248
- mysql login file, 236
- omitting closing tag, 35
- <noscript> and </noscript> tags (HTML), 313
- <pre> and </pre> tags (HTML), 104, 127
- <script> and </script> tags (HTML), 9, 313
 - <script src= to load from file, 315
 - example of use, 466
 - inline JavaScript instead, 469
- <select> in forms (HTML), 277-279
- <style> and </style> (CSS), 10
- = (assignment) operator
 - JavaScript, 320
 - not confusing with == operator, 44, 71
 - PHP, 43
- == (equality) operator
 - JavaScript, 320, 337
 - not confusing with = operator, 44, 71
 - PHP, 44, 71-72
- === (strict equality) operator
 - JavaScript, 320, 337
 - PHP, 44, 71-72
 - validating username and password, 295
- => (array assignment) operator (PHP), 122
- => (arrow) functions (JavaScript), 373
 - setTimeout calling, 476
- > (greater than) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- > (redirect), 229
 - mysqldump redirected to a file, 230
- >= (greater than or equal to) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- ? operator (PHP), 55, 82
 - as ternary operator, 66, 82
- ? placeholder (MySQL), 261-264
- ? ternary operator
 - JavaScript, 336, 347
 - PHP, 66, 82
- @font-face (CSS), 435
- [] (square brackets)
 - arrays
 - JavaScript, 363-366
 - PHP, 120, 127, 128
 - regular expressions
 - character classes, 386
 - negation of a character class, 386
- \ (backslash)
 - JavaScript
 - characters tab, newline, return, 322
 - escaping characters in strings, 322
 - PHP
 - characters tab, newline, return, 48
 - escaping characters in strings, 48
 - regular expressions escaping characters, 385
 - \ " (quote mark, double) escape character, 322
 - \ ' (quote mark, single) escape character, 322
 - \? (HELP) command (MySQL), 168
 - \b (backspace) character (JavaScript), 322
 - \c to cancel input (MySQL), 168
 - \d for digit in regular expressions, 386
 - \f (form feed) character (JavaScript), 322
 - \h (HELP) command (MySQL), 168
 - \n (newline) character, 48, 322
 - . wildcard in regular expressions, 384
 - \r (carriage return) character, 48, 322
 - \s (STATUS) command (MySQL), 168
 - \t (tab) character, 48, 322
 - \W (nonword character) match (regular expressions), 390
 - \w (word character) match (regular expressions), 390
 - \\ (backslash) escape character, 322
 - ^ beginning of the line (regular expressions), 389
 - ^ character class negation (regular expressions), 386
 - ^= (start of string) operator (CSS), 417
 - __ (double underscore) in PHP, 53, 109
 - __CLASS__ constant (PHP), 54
 - __DIR__ constant (PHP), 54
 - __FILE__ constant (PHP), 54
 - __FUNCTION__ constant (PHP), 54
 - __LINE__ constant (PHP), 54
 - __METHOD__ constant (PHP), 54
 - __NAMESPACE__ constant (PHP), 54
 - `> MySQL prompt, 167
 - { } (curly braces)
 - JavaScript
 - associative arrays, 364
 - classes, 359
 - const keyword, 328
 - functions, 353, 354, 359, 372
 - if statements, 344
 - let keyword, 327
 - objects, 359

- PHP
 - classes, 104
 - do...while loops, 86
 - for loops, 87
 - functions, 55, 96
 - if statements, 75
 - switch statements, 80
 - switch statements, alternative syntax, 81
 - while loops, 84
- || (Or) logical operator
 - JavaScript, 321, 340
 - PHP, 45
 - higher precedence than and, or, 73
- A**
- addition (+) operator
 - JavaScript, 319
 - PHP, 43
- addition and assignment (+=) operator
 - JavaScript, 320, 321
 - string concatenation, 322
 - PHP, 43, 46
- adjacent sibling selectors (CSS), 416
- Ajax (Asynchronous JavaScript and XML)
 - about Ajax, 403
 - about JavaScript, 10
 - cross-origin resource sharing, 407-409
 - Fetch API
 - about, 403
 - asynchronous program GET method, 409-410
 - asynchronous program POST method, 404-407
 - JSON requests, 410-412
 - link to standard, 403
 - XMLHttpRequest object in JavaScript, 412
- alert pop-up window for JavaScript output, 312
 - centering in browser window, 469
 - error handling for input validation, 379
- aliases in MySQL via AS keyword, 204
- ALTER command (MySQL), 168, 178
 - adding a FULLTEXT index, 189
 - adding a new column, 181
 - primary key column, 187
 - adding an auto-incrementing column, 178
 - adding an index to a column, 184
 - changing column data type, 181
 - removing a column, 179, 182
 - renaming a column, 182
 - renaming a table, 181
- AMPPS
 - macOS installation, 27
 - document root, 28
 - document root Hello World, 28
 - PHP version, 28
 - serving pages from document root versus filesystem, 26, 28
- Windows installation, 18-23
 - alternative WAMPs, 26
 - AMPPS documentation, 22, 27
 - configuration, 24
 - document root described, 25
 - document root Hello World, 25
 - document root viewed, 24
 - Microsoft Visual C++ Redistributable, 21
 - PHP version, 23
 - testing the installation, 23-24
- And (&&) logical operator
 - JavaScript, 321, 340
 - short-circuit evaluation, 341
 - PHP, 45
 - higher precedence than and, or, 73
- and logical operator (PHP), 45, 72-74
 - lower precedence than && and ||, 73
- AND operator in MySQL WHERE queries, 204
- Android applications via React Native, 506
- Angular, 483
- animation via time-based events, 478-480
- anonymous functions (JavaScript), 372
- Apache web server
 - about, 12
 - Node.js as alternative to, 13, 509
 - Apache still relevant, 510
 - open source, 14
 - secure web server documentation online, 306
 - server setup (see development server setup)
- appendChild function (JavaScript), 472
- arguments array (JavaScript), 355
- arithmetic operators
 - JavaScript, 319, 335
 - PHP, 42
- arrays
 - index starting from zero, 42
 - JavaScript, 318, 363-372
 - associative arrays, 364
 - creating a new array, 363, 364

- element values assigned, 363
 - functions returning arrays, 358
 - methods, 366-372
 - multidimensional arrays, 318, 365
 - passed to functions by reference, 325
 - split function, 351
 - spread syntax, 368
- PHP, 39-42
 - \$GLOBALS array, 60
 - \$_COOKIE array, 60
 - \$_ENV array, 60
 - \$_FILES array, 60, 154, 547, 548
 - \$_GET array, 60, 246
 - \$_POST array, 60, 246
 - \$_REQUEST array, 60
 - \$_SERVER array, 60, 293, 294, 295, 298-301, 302
 - \$_SESSION array, 60, 302-304, 306, 307
 - array functions, 129-133
 - assignment (=) operator, 122
 - assignment via array keyword, 122
 - associative arrays, 121, 242
 - creating a new array, 39
 - foreach...as loops, 123-125, 132
 - functions returning arrays, 98
 - multidimensional arrays, 125-128
 - numerically indexed arrays, 119-121
 - superglobal variables, 60
 - two-dimensional arrays, 40-42
- array_intersect function (PHP), 555
- arrow (=>) functions (JavaScript), 373
 - setTimeout calling, 476
- AS keyword (MySQL), 204
- assignment (=) operator
 - JavaScript, 320
 - not confusing with == operator, 44, 71
 - PHP, 43
- assignment operators
 - JavaScript, 320, 335
 - shorthand assignments, 321
- PHP, 43
 - multiple-assignment statement, 70
 - variable assignment, 46-47
- associative arrays
 - JavaScript, 364
 - PHP, 121
 - assignment via array keyword, 122
 - column reference easier than numeric, 242
 - foreach...as loops, 124
 - multidimensional arrays, 126
- associativity of operators
 - JavaScript, 337
 - PHP, 69
- asterisk (*) in regular expressions, 383
 - HTML tag match expressions, 384
- asynchronous communication
 - about, 403
 - Ajax in JavaScript, 10
 - about Ajax, 403
 - cross-origin resource sharing, 407-409
 - example of username check, 14-16
 - social networking site, 540
 - Fetch API in JavaScript
 - about, 403
 - asynchronous program GET method, 409-410
 - asynchronous program POST method, 404-407
 - cross-origin resource sharing, 407-409
 - JSON requests, 410-412
 - link to standard, 403
 - frameworks React, Axios, jQuery, 413
 - JavaScript handling, 5
 - Node.js, 509
 - XMLHttpRequest object in JavaScript, 412
- attribute selectors (CSS), 416
- authentication (HTTP), 292-301
 - example program, 298-301
 - htmlspecialchars function, 294, 299-301
 - storing usernames and passwords, 296-298
 - documentation online, 297
 - size of storage for hashes, 297
 - verifying password against hash, 297, 544
 - validating username and password, 295, 544
- autocomplete attribute in forms (HTML), 279
 - documentation online, 280
- autofocus attribute in forms (HTML), 280
- AUTO_INCREMENT attribute (MySQL), 177-179
 - as primary key, 187, 211
 - scripting, 256-257
- Axios, 413

B

- Babel JSX extension, 486
- backgrounds (CSS), 418-425

- about, 418
 - auto keyword for scaling, 422
 - background-clip property, 419-421
 - about, 418
 - background-origin property, 421
 - about, 418
 - background-size property, 421
 - units em and rem, 422
 - multiple backgrounds, 422-425
 - backing up, 228-232
 - all databases backed up, 232
 - mysqldump, 228
 - dumping data into CSV, 233
 - redirecting data to a file, 230
 - planning your backups, 233
 - restoring from a backup file, 232
 - test restoring periodically, 234
 - single table backup, 231
 - backslash (\)
 - JavaScript
 - characters tab, newline, return, 322
 - escaping characters in strings, 322
 - PHP
 - characters tab, newline, return, 48
 - escaping characters in strings, 48
 - regular expressions escaping characters, 385
 - backslash (\) escape character, 322
 - BACKUP command (MySQL), 168
 - Berners-Lee, Tim, 1, 2
 - BIGINT data type (MySQL), 176
 - binary operators, 66, 336
 - BINARY versus VARBINARY data types (MySQL), 174
 - bind variable (:) prefix (PHP), 262
 - bindParam method (PHP), 262
 - bitwise operators
 - JavaScript, 335
 - PHP, 66
 - BLOB data types (MySQL), 175
 - book code examples on GitHub, 35
 - book supplemental material, [xxi](#)
 - CSS selectors, 416
 - book web page, [xxiii](#)
 - Boole, George, 63
 - Boolean expressions, 64
 - Boolean function (JavaScript), 351
 - Boolean operators, 72
 - (see also logical operators)
 - Boolean values, 64
 - truthy and falsy values in JavaScript, 339
 - Bootstrap icon library, 535
 - borders (CSS), 425-428
 - border-color property, 425
 - border-radius property, 426-428
 - box shadows (CSS), 428
 - box-sizing property (CSS), 417
 - break command
 - JavaScript
 - looping, 350
 - switch statements, 346
 - PHP
 - loops, 88
 - nested loops, 89
 - switch statements, 80
 - browsers (see web browsers)
 - BrowserStack, 18
- ## C
- cache, 2
 - GET requests cached, 409
 - cachebusting, 409
 - POST requests never cached, 409
 - cachebusting, 409
 - camelCase names, 354
 - caret (^)
 - CSS start of string (^=) operator, 417
 - regular expressions
 - beginning of the line, 389
 - character class negation, 386
 - carriage return (\r) character, 48, 322
 - case of characters
 - camelCase, 354
 - constants in uppercase, 53
 - file handling and case sensitivity, 143
 - form filename data to lowercase, 156
 - global variables in uppercase, 59
 - JavaScript
 - classes, instances, properties, methods, 360
 - function names case-sensitive, 354
 - true and false, 334
 - variable names case-sensitive, 317
 - MySQL
 - commands case-insensitive, 169
 - FULLTEXT indexes case-insensitive, 196
 - table names, 169
 - PHP TRUE and FALSE, 64

- centering in-browser alerts or dialog windows, 469
- CERN in early history of the web, 1
- CGI (Common Gateway Interface), 5
- CHAR data type (MySQL), 173
 - VARCHAR versus, 173
- character classes of regular expressions, 386
- character sets, 173
- checkboxes in forms (HTML), 272-274
- checkdate function (PHP), 143
- child selectors (CSS), 416
- Chrome Developers Blog, 415
- __CLASS__ constant (PHP), 54
- class selectors (CSS), 416
- classes
 - about objects, 102
 - inheritance, 103
 - object-oriented programming terminology, 102
 - properties and methods, 102, 105
 - class composition, 114
- JavaScript
 - class constructor, 359
 - declaring a class, 359
 - instance created, 360
 - instances of classes, 359
 - legacy object-functions, 361
 - static methods and properties, 361
 - this keyword, 360
 - this with constructor and instance, 359, 360
- PHP
 - class constructor, 105, 108
 - creating an object, 105
 - declaring a class, 104
 - declaring constants, 111
 - inheritance, 103, 114-118
 - instance created, 104
 - instances of classes, 102
 - static methods, 110
 - static properties, 113
- clearInterval function (JavaScript), 478
- clearTimeout function (JavaScript), 476
- click event (JavaScript), 342
- clients and servers, 2
 - PHP, MySQL, JavaScript, CSS, HTML, 5
 - request/response process, 3-5
 - cookies, 290
 - working remotely, 29
 - logging in, 29
 - transferring files, 30
- clocks
 - React state, 492-495
 - setInterval function, 477
- cloning objects (PHP), 106
- Codd, E. F., 211
- code editors, 31
- code examples on GitHub, 35
- color hex code shorthand (CSS), 425
- color in forms (HTML), 282
- colors and opacity (CSS), 431-433
 - hex code shorthand, 425
 - HSL colors, 431
 - HSLA colors, 432
 - opacity property, 433
 - RGB colors, 432
 - RGBA colors, 433
- column of table, 162
 - adding a new column, 181
 - primary key column, 187
 - changing data type, 181
 - joining tables, 201-204
 - JOIN...ON, 204
 - NATURAL JOIN, 203
 - name alias via AS keyword, 204
 - removing, 179, 182
 - renaming, 182
 - rows unique via AUTO_INCREMENT, 177-179
 - primary key, 187, 211
- command line MySQL
 - about, 162
 - starting up
 - Linux, 165
 - macOS, 164
 - remote server, 166
 - Windows, 163
 - using, 167
 - logging in as root, 163
 - logging in as user, 170
 - prompts, 167
 - semicolon, 167
 - \c to cancel input, 168
- commands commonly used in MySQL, 168
 - adding a new column, 181
 - primary key column, 187
 - adding data to a table, 179, 203
 - in a script, 253

- case-insensitive, 169
- changing column data type, 181
- creating a database, 169, 533
- creating a table, 171-172
 - auto-incrementing column, 179
 - indexes created with table, 186
 - numeric field, 176
 - primary key created with table, 188
 - scripting, 250
- creating users, 169-171, 533
- deleting a table, 183
 - in a script, 252
- logging in as root, 163
- logging in as user, 170
- renaming a column, 182
- renaming a table, 181
- comments
 - JavaScript, 316
 - MySQL and canceling input, 168
 - PHP, 35
- COMMIT a transaction (MySQL), 225
- Common Gateway Interface (CGI), 5
- compact array function (PHP), 131
- comparison operators
 - JavaScript, 335, 338
 - short-circuit evaluation, 341
 - PHP, 44
- concat array method (JavaScript), 368
- concatenation (.) of strings (PHP), 47
- concatenation assignment (.=) operator (PHP), 43, 47
- conditionals
 - JavaScript, 344-347
 - else statement, 344
 - if statement, 344
 - switch statement, 345-347
 - JSX inline conditional statements, 497
 - PHP, 74-81
 - ? operator, 55, 66, 82
 - else statement, 76-78
 - elseif statement, 78
 - if statement, 75-76
 - switch statement, 79-81
- console.log for JavaScript output, 312
 - Hello World, 313
 - log method of console object, 329
- const keyword (JavaScript), 327
 - creating an object, 359
- constants
 - JavaScript, 327
 - arrays and objects can be modified, 328
 - names in uppercase letters, 53
 - PHP, 53
 - constants inside classes, 111
 - predefined constants, 53
 - superglobal variables versus, 61
 - __construct function (PHP), 108
- constructors
 - PHP, 105, 108
 - subclasses calling parent constructors, 116
- continue statement
 - JavaScript, 351
 - PHP, 89
- cookies in PHP, 289-292
 - about cookies, 289
 - third-party cookies, 289
 - third-party cookies being phased out, 301
 - cookie-only sessions, 307
 - deleting, 292
 - disabled in web browser, 290, 301
 - editing cookies within browser, 291
 - reading a cookie, 292
 - session ID, 301
 - setting a cookie, 291
- copy function (PHP), 146
- CORS (see cross-origin resource sharing)
- count array function (PHP), 129
- CREATE command (MySQL), 168
 - creating a database, 169
 - social networking site, 533
 - creating a table, 171-172
 - auto-incrementing column, 179
 - indexes created with table, 186
 - numeric field specified, 176
 - primary key created with table, 188
 - creating a table in a script, 250
 - creating an index, 186
 - creating users, 169-171
 - social networking site, 533
- createServer method (Node.js http object), 521
- creating a file (PHP), 144
- cross-origin resource sharing (CORS), 407-409
 - origin, 407
 - same-origin and cross-origin requests, 408
- cross-site scripting (XSS) attack, 241, 264

- article by OWASP online, 284
- innerHTML property risks, 405
- sanitizing input, 283
- CSS
 - about using, 10
 - Can I Use... website, 416, 422
 - CSS instead of JavaScript, 415
 - CSS3 covered in book, 415
 - example of username check, 14-16
 - putting it all together (see social networking site)
 - backgrounds, 418-425
 - about, 418
 - background-clip property, 419-421
 - background-origin property, 421
 - background-size property, 421
 - multiple backgrounds, 422-425
 - benefits of, 5
 - borders, 425-428
 - border-color property, 425
 - border-radius property, 426-428
 - box shadows, 428
 - box-sizing property, 417
 - units em and rem, 422
 - color hex code shorthand, 425
 - colors and opacity, 431-433
 - HSL colors, 431
 - HSLA colors, 432
 - opacity property, 433
 - RGB colors, 432
 - RGBA colors, 433
 - development history, 415
 - 2023 snapshot of stable modules online, 415
 - further information online, 415
 - Selectors Level 4, 415
 - flexbox layout, 444-454
 - aligning content, 450
 - aligning items, 449
 - browser-based editor, 444
 - flex items, 445
 - flex wrap, 451
 - flow direction, 445
 - item gaps, 453
 - justifying content, 447
 - order property, 453
 - resizing items, 451
 - grid layout, 454-460
 - alignment, 459
 - browser-based editor, 455
 - columns and rows, 455
 - grid container, 454
 - grid flow, 456
 - grid gaps, 458
 - placing grid items, 457
 - JavaScript accessing
 - by, byID, and style functions created, 463-466
 - byID and style functions in GitHub, 465
 - CSS properties, 466-469
 - getElementById function, 463-466
 - hiding and showing elements, 474
 - name hyphenation to camelCase, 466
 - multicolumn layout, 429
 - overflow property of elements, 429
 - pseudoclasses, 416
 - hover pseudoclass script, 442
 - pseudoelements, 416
 - selectors, 416
 - matching part of a string, 416-417
 - supplemental material online, 416
 - supplemental material online, xxi, 11
 - selectors, 416
 - text effects, 433-435
 - text-overflow property, 434
 - text-shadow property, 433
 - word-wrap property, 435
 - transformations, 438-440
 - script with transition, 442
 - transitions, 440
 - delay, 441
 - duration, 441
 - properties, 440
 - returning to initial state, 443
 - script with transformation, 442
 - shorthand syntax, 442
 - timing, 441
 - units em and rem, 422
 - web fonts, 435
 - Google web fonts, 436
 - Google web fonts privacy page, 436
 - Google web fonts website, 436
 - specifying for browser, 436
- CSS3 covered in book, 415
 - (see also CSS)
- CSV data dump, 233
- Ctrl-C (EXIT) command
 - MySQL, 168

- Node.js, 522
- curly braces ({})
- JavaScript
 - associative arrays, 364
 - classes, 359
 - const keyword, 328
 - functions, 353, 354, 359, 372
 - if statements, 344
 - let keyword, 327
 - objects, 359
- PHP
 - classes, 104
 - do...while loops, 86
 - for loops, 87
 - functions, 55, 96
 - if statements, 75
 - switch statements, 80
 - switch statements, alternative syntax, 81
 - while loops, 84

D

- Dahl, Ryan, 13, 509
- data dumped into CSV file, 233
- data types in MySQL, 173
 - AUTO_INCREMENT attribute, 177-179
 - as primary key, 211
 - BINARY versus VARBINARY, 174
 - BLOB types, 175
 - changing column data type, 181
 - CHAR versus VARCHAR, 173
 - character sets, 173
 - DATE and TIME types, 177
 - numeric types, 176
 - signed versus unsigned numbers, 176
 - UNSIGNED qualifier, 176
 - AUTO_INCREMENT attribute
 - as primary key, 187
 - TEXT types, 175
- databases
 - backing up, 228-232
 - all databases, 232
 - mysqldump, 228
 - mysqldump redirected to a file, 230
 - restoring from a backup file, 232
 - single table backup, 231
 - basics, 161
 - definition of database, 161, 162
 - design, 209
 - key terms, 162

- MySQL
 - about MySQL, 8
 - creating a database, 169, 533
 - creating users, 169-171, 533
 - joining tables, 201-204
 - normalization, 211-219
 - placeholders, 261-264
 - slow database responsiveness, 189
 - SQL as Structured Query Language, 161
- privacy and, 223
- querying, 190-201
 - AS keyword, 204
 - DELETE a row, 193
 - GROUP BY keywords, 201
 - LIKE keyword, 193
 - LIMIT keyword, 194
 - logical operators, 204
 - MATCH...AGAINST, 196-198
 - MATCH...AGAINST in Boolean mode, 197
 - ORDER BY keywords, 200
 - phpMyAdmin tool for MySQL, 205
 - scripting, 254, 257
 - SELECT, 190
 - SELECT COUNT, 191
 - SELECT DISTINCT, 191
 - UPDATE...SET, 199
 - WHERE keyword, 193
- relationships, 219-223
 - many-to-many, 221
 - one-to-many, 220
 - one-to-one, 220
 - privacy and, 223
- transactions, 223-228
 - about, 223
 - BEGIN, 225
 - COMMIT, 225
 - START TRANSACTION, 225
 - transaction storage engines, 223
- date and time functions (MySQL) documenta-
tion online, 205
- date and time functions (PHP), 140-143
 - 2038 as end of time, 140
 - date constants, 142
 - validity check via checkdate, 143
- date and time pickers in forms (HTML), 282
- DATE data type (MySQL), 177
- date function (PHP), 141
 - date constants, 142

- DATETIME data type (MySQL), 177
- DATE_ATOM constant (PHP), 142
- DATE_COOKIE constant (PHP), 142
- DATE_RSS constant (PHP), 142
- DATE_W3C constant (PHP), 142
- debugging
 - compact array function for, 132
 - file locking, 150
 - JavaScript, 316
 - onerror event, 342
 - short-circuit evaluation, 341
 - var for redeclaring variables, 326
 - MySQL
 - DESCRIBE command, 172
 - EXPLAIN for query taking too long, 228
 - slow database responsiveness, 189
 - PHP
 - > not requiring \$, 109
 - encapsulation of classes, 103
 - global variables, 58, 99
 - if statements without curly braces, 75
 - inheritance, 114
 - magic constants for, 54
 - nested multiline comments, 36
 - object destruction, 108
 - object properties implicitly declared, 110
 - or operator second operand evaluation, 73
 - semicolons, 37
 - variable scope, 58
- decrement (--) operator
 - JavaScript, 319, 321, 335
 - PHP, 43, 46
- define function (PHP), 53
- DELETE row command (MySQL), 168, 193
- deleting a cookie, 292
- deleting a database record (PHP), 249
 - in a script, 255
- deleting a file (PHP), 147
- deleting a table (MySQL), 183
 - in a script, 252
- derived class in inheritance, 103
- descendent selectors (CSS), 416
- DESCRIBE command (MySQL), 168
 - checking indexes added, 185
 - checking table alteration, 181
 - checking table creation, 172, 179, 183
 - script use of, 251
- designing a database, 209
- __destruct function (PHP), 108
- destructors (PHP), 108
- development server setup
 - about development servers, 17
 - code editors, 31
 - Linux, 29
 - document root check, 29
 - macOS installation of AMPPS, 27
 - document root, 28
 - document root Hello World, 28
 - PHP version, 28
 - serving pages from document root ver-
sus filesystem, 28
 - Node.js alternative to Apache, 522-525
 - port number, 39
 - WAMP, MAMP, or LAMP, 18
 - Windows installation of AMPPS, 18-23
 - alternative WAMPs, 26
 - AMPPS documentation, 22, 27
 - configuration, 24
 - document root, 25
 - document root Hello World, 25
 - document root viewed, 24
 - Microsoft Visual C++ Redistributable, 21
 - PHP version, 23
 - serving pages from document root ver-
sus filesystem, 26
 - testing the installation, 23-24
 - working remotely, 29
 - logging in, 29
 - SSH for MySQL, 29
 - transferring files, 30
- dictionaries (see associative arrays)
- die function (PHP), 144
 - file closed as part of termination, 144, 150
 - file unlocked as part of termination, 150
 - message instead, 146
- digit (\d) in regular expressions, 386
- __DIR__ constant (PHP), 54
- directories (PHP)
 - rename function, 147
 - system call to view contents, 157
- directory traversal attack, 523
- display property (CSS)
 - flex value, 444-454
 - aligning content, 450
 - aligning items, 449
 - browser-based editor, 444

- flex items, 445
- flex wrap, 451
- flow direction, 445
- item gaps, 453
- justifying content, 447
- order property, 453
- resizing items, 451
- grid value, 454-460
 - alignment, 459
 - browser-based editor, 455
 - columns and rows, 455
 - grid container, 454
 - grid flow, 456
 - grid gaps, 458
 - justifying content, 459
 - placing grid items, 457
- division (/) operator
 - JavaScript, 319
 - PHP, 43
- division and assignment (/=) operator
 - JavaScript, 320
 - PHP, 43
- DNS (Domain Name System), 4
- do...while loops
 - JavaScript, 348
 - PHP, 86
- Document Object Model (DOM)
 - about, 328
 - hierarchy of objects, 329
 - JavaScript design, 328-330
 - adding new elements, 472
 - document object URL specified, 331
 - hierarchy of parent and child objects, 329
 - history of JavaScript, 311
 - length property, 331
 - links object, 330
 - removing elements, 474
 - using the DOM, 330
 - jQuery simplifying traversal and manipulation, 483
 - React updates via virtual DOM, 484
- document root
 - about, 25
 - Linux, 28
 - macOS, 28
 - Hello World file, 28
 - serving pages from document root versus filesystem, 26, 28
- Windows
 - command for viewing on local server, 24
 - Hello World file, 25
- document.write function (JavaScript), 9-10, 312
- dollar (\$) symbol
 - JavaScript variable and function names, 317
 - PHP variables, 37
 - porting into JavaScript, 317
 - regular expressions, 389
- DOM (see Document Object Model)
- Domain Name System (DNS), 4
- DOMDocument class loadHTML method (PHP), 384
- DOMPurify library, 284
- dot (.) in regular expressions, 384
 - HTML tag match expressions, 384
 - matching the dot character, 385
- DOUBLE or REAL data type (MySQL), 176
- double quote mark (see quotation mark, double (""))
- DROP command (MySQL), 168
 - deleting a table, 183
 - in a script, 252
 - irreversible, 183
 - removing a column, 179, 182
- drop-down list (HTML), 277
- dumping data into CSV, 233
- dynamic web design
 - Apache web server, 12
 - (see also Apache web server; Node.js)
 - basics of HTTP and HTML, 2-5
 - HTML5, 11
 - HTTP GET and POST requests, 152
 - history, 1
 - MariaDB, 6
 - multicolumn layout, 429
 - open source technologies, 14
 - PHP, MySQL, JavaScript, CSS, HTML, 5
 - asynchronous communication example, 14-16
 - CSS instead of JavaScript, 415
 - WAMP, MAMP, or LAMP, 18
 - putting it all together (see social networking site)
 - request/response process, 3-5
 - cookies, 290
 - server setup (see development server setup)
 - supplemental material online, [xxi](#)
 - using CSS, 10

- using JavaScript, 9
 - using MySQL, 8
 - using PHP, 7, 12
 - modularization of PHP code, 91
- ## E
- EasyPHP, 26
 - echo command versus print (PHP), 54
 - PHP output via echo, 312
 - ECMAScript modules, 520
 - editors for coding, 31
 - element overflow (CSS), 429
 - else statement
 - JavaScript, 344
 - PHP, 76-78
 - elseif statement (PHP), 78
 - email address validation, 382
 - encapsulation of objects, 102
 - encoding as multipart/form-data, 152
 - end array function (PHP), 133
 - ENGINE command (MySQL)
 - creating a table, 171, 224
 - InnoDB storage engine, 171, 224
 - equality (==) operator
 - JavaScript, 320, 337
 - not confusing with = operator, 44, 71
 - PHP, 44, 71-72
 - errors
 - JavaScript
 - displayed in browser console, 316
 - error handling in user input validation, 376, 378
 - onerror for error handling, 342
 - trapping with try...catch, 343
 - Uncaught TypeError, 327
 - PHP
 - error messages trapped, 239, 247
 - Parse error, 37, 51
 - TypeError, 52
 - Undefined variable, 57
 - variable scope, 58
 - trapping with try...catch
 - JavaScript, 343
 - PHP, 239, 247
 - escape characters (\)
 - JavaScript
 - characters tab, newline, return, 322
 - escaping characters in strings, 322
 - PHP
 - characters tab, newline, return, 48
 - escaping characters in strings, 48
 - escapeshellcmd function (PHP), 158
 - events
 - JavaScript
 - about, 342
 - click, 342
 - mouseover, 469
 - objects attached to, 470
 - onerror for error handling, 342
 - table of events and their triggers, 471
 - time-based, 475-480
 - validation of user input, 376, 379
 - React, 495-496
 - names in camelCase, 495
 - every array method (JavaScript), 366
 - exceptions (see errors)
 - exclusive or (xor) logical operator (PHP), 45, 46, 72-74
 - exec function (PHP), 157-158
 - cautions, 158
 - escapeshellcmd function, 158
 - placeholders, 262
 - colon prefix for values optional, 263
 - EXECUTE command (MySQL), 261
 - execute function (PDO), 262
 - HTTP authentication, 298-301
 - starting a session, 302-304
 - execution operators (PHP), 66
 - caution, 67
 - EXIT (Ctrl-C) command (MySQL), 168
 - EXPLAIN a transaction (MySQL), 226
 - explicit casting of variables
 - JavaScript, 351
 - PHP, 90
 - explode array function (PHP), 130
 - exponentiation (**) operator (PHP), 43
 - expressions
 - definition, 63
 - JavaScript, 333-335
 - literals and variables, 334
 - multiple properties and methods in one expression, 357
 - PHP, 63-66
 - Boolean expressions, 64
 - literals and variables, 65
 - statements from, 66
 - extract array function (PHP), 131

F

- F12 key for browser console, 316
- FALSE with value of "" (PHP), 64
 - NULL different, 64
- false with value of "false" (JavaScript), 334
- falsy and truthy values (JavaScript), 339
- fclose function (PHP), 144, 145
- Fetch API (JavaScript)
 - about, 403
 - asynchronous program GET method, 409-410
 - asynchronous program POST method, 404-407
 - fetch function returning a Promise, 405
 - JSON requests, 410-412
 - link to standard, 403
- fetch method (PDO), 239-241
 - data style options, 241
 - documentation online, 241
 - scripting, 254
 - HTTP authentication example, 298-301
- fgets function (PHP)
 - reading from a file, 145
 - updating a file, 148
- fields in records, 162
 - UPDATE...SET to update, 199
- __FILE__ constant (PHP), 54
- file handle for opening a file (PHP), 144
- file handling (PHP)
 - about case sensitivity, 143
 - always open, write/read, close, 144
 - copying a file, 146
 - creating a file, 144
 - deleting a file, 147
 - die function replaced with message, 146
 - error trapping break example, 88
 - file exists check, 143
 - file handle, 144
 - file pointer, 148
 - locking files for multiple accesses, 149
 - as advisory lock, 150
 - not supported on all systems, 150
 - moving a file, 147
 - opening a file, 144
 - modes, 145
 - reading from a file, 145
 - HTML fetch and display, 151
 - reading an entire file, 151
 - updating a file, 148
 - uploading a file, 152-157
 - \$_FILES, 154
 - internet media content types, 154
- file pointer (PHP), 148
- File Transfer Protocol SSL (FTPS), 30
 - not FTP, 30
- files to include directive (PHP), 99
 - files required to be included, 101
- FileZilla for SFTP, 30
- FileZilla Wiki, 30
- open source, 30
- file_exists function (PHP), 143
- file_get_contents function (PHP), 151
 - HTML fetch and display, 151
 - asynchronous program, 406
- filter array method (JavaScript), 367
- final methods in inheritance, 117
- flexbox layout (CSS), 444-454
 - aligning content, 450
 - aligning items, 449
 - browser-based editor, 444
 - flex items, 445
 - flex wrap, 451
 - flow direction, 445
 - item gaps, 453
 - justifying content, 447
 - order property, 453
 - resizing items, 451
- FLOAT data type (MySQL), 176
- flock function (PHP), 149
 - die function unlocks a lock, 150
 - not supported on all systems, 150
- fonts loaded from web, 435
 - Google web fonts, 436
 - Google Fonts Website, 436
 - privacy information online, 436
 - specifying for browser, 436
- fontSize property (JavaScript), 466
- fopen function (PHP), 144
 - creating a text file, 144
 - locking a file for multiple accesses, 149
 - modes, 145
 - reading from a file, 145
 - updating a file, 148
- for loops
 - JavaScript, 349
 - for...in loops, 365
 - PHP, 86-88
 - arrays, 120

- for...of loop (JavaScript), 350
- forEach array method (JavaScript), 368
- foreach...as loops (PHP), 123-125, 132
- foreign keys (MySQL), 214
- form attribute in forms (HTML), 281
- form feed (\f) character (JavaScript), 322
- formatted output to string via sprintf (PHP), 139
- formatted output via printf (PHP), 135
- forms (HTML)
 - about, 267
 - building, 267
 - default values, 270
 - example of PHP integrating with forms, 284-287
 - filename data to lowercase, 156
 - GET method of submission, 246
 - sanitizing via htmlentities function, 61, 264, 283
 - htmlspecialchars function in PHP, 241, 247, 294
 - image and text posted, 547
 - image uploader, 152
 - POST method of submission, 152
 - input types, 271-283
 - <select>, 277-279
 - autocomplete attribute, 279
 - autofocus attribute, 280
 - checkboxes, 272-274
 - color, 282
 - date and time pickers, 282
 - form attribute, 281
 - hidden fields, 276
 - labels, 274
 - list attribute, 282
 - min and max attributes, 282
 - number and range, 282
 - override attributes, 281
 - placeholder attribute, 280
 - radio buttons, 275
 - required attribute, 280
 - step attribute, 281
 - submit button, 279
 - text areas, 271
 - text boxes, 271
 - width and height attributes, 281
 - PHP querying MySQL database, 243-246
 - POST method of submission, 246
 - \$_POST array, 246
 - image uploader, 152
 - PHP querying MySQL database, 243
 - sanitizing via htmlentities function, 61, 264, 283
 - retrieving submitted data, 269-284
 - default values, 270
 - DOMPurify library, 284
 - hacking prevention via placeholders, 261-264
 - sanitizing input, 283
 - sanitizing via htmlentities function, 61, 264, 283
 - social networking site, 540, 544
 - photo and text, 547
 - validation, 154-156
 - user input using JavaScript, 375-383
 - user input via PHP, 393-399
- forms (React), 501-506
 - controlled components, 501
 - select, 504
 - text area, 503
 - text input, 501-503
- forward slash (/)
 - /* and */ for multiline comments
 - JavaScript, 316
 - PHP, 36
 - // for single line comments
 - JavaScript, 316
 - PHP, 35
 - self-closing elements in HTML, 12
- frameworks, 483
 - asynchronous communication, 413
 - React as, 485
- fread function (PHP), 146
- fseek function (PHP)
 - reseek at each file access, 150
 - updating a file, 148
- FTPS (File Transfer Protocol SSL), 30
 - not FTP, 30
- FULLTEXT indexes (MySQL), 188
 - case-insensitive, 196
 - facts to know, 189
 - MATCH...AGAINST command, 196-198
 - Boolean mode, 197
 - stopwords, 189
- __FUNCTION__ constant (PHP), 54
- functions
 - definition, 93
 - JavaScript, 324

- anonymous functions, 372
 - arguments array, 355
 - arrays passed by reference, 325
 - arrow functions, 373
 - defining a function, 324, 353-356
 - legacy object-functions, 361
 - pure versus impure code, 489
 - rest parameter syntax, 354
 - returning a value, 356
 - returning an array, 358
 - setTimeout calling, 475
 - local variables
 - JavaScript, 325
 - PHP, 56-58, 99
 - MySQL documentation online, 205
 - PHP, 55, 94-95
 - defining a function, 96
 - expression evaluation, 97
 - function_exists function, 101
 - returning a value, 96-98
 - returning an array, 98
 - returning global variables, 98
 - static variables in PHP, 59, 99
 - function_exists function (PHP), 101
 - fwrite function (PHP)
 - creating a file, 144
 - locking a file for multiple accesses, 149
 - updating a file, 148
- ## G
- GD (Graphics Draw) library (PHP), 12
 - GET method (HTTP)
 - asynchronous program, 409-410
 - browsers may cache requests, 409
 - first asynchronous program
 - JSON requests, 410-412
 - forms, 246
 - sanitizing via htmlentities function, 61, 264, 283
 - image uploader, 152
 - getCode function (PHP), 239
 - getElementById function (JavaScript)
 - called via \$, 330
 - CSS from JavaScript, 463-466
 - byID and style functions in GitHub, 465
 - getimagesize function (PHP), 547
 - getMessage function (PHP), 239
 - GitHub repo for this book
 - book supplemental material, xxi
 - CSS selectors, 416
 - book's examples, 35
 - byID and style functions, 465
 - database login.php, 236
 - input validation via JavaScript, 376
 - input validation via PHP, 393
 - React, 506
 - social networking site, 532
 - users table and accounts in PHP, 298
 - DOMPurify library, 284
 - Introduction to CSS, 11
 - Introduction to HTML5, xxi, 11
 - React development files, 485
 - global variables
 - JavaScript, 324
 - names in uppercase letters, 59
 - PHP, 58, 99
 - functions returning, 98
 - globally unique identifiers (GUIDs), 211
 - Google Maps as asynchronous communication, 403
 - Google Privacy Sandbox, 301
 - Google V8 JavaScript engine, 13
 - Google web fonts, 436
 - Google Fonts Website, 436
 - privacy information online, 436
 - script that loads a font, 437
 - GRANT command (MySQL), 168
 - creating users, 169-171
 - only privileges you already have, 171
 - social networking site, 533
 - documentation online, 171
 - graphical user interface access to MySQL, 163
 - Graphics Draw (GD) library (PHP), 12
 - greater than (>) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
 - greater than or equal to (>=) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
 - grid layout (CSS), 454-460
 - alignment, 459
 - browser-based editor, 455
 - columns and rows, 455
 - grid container, 454
 - grid flow, 456
 - grid gaps, 458
 - placing grid items, 457
 - GUIDs (globally unique identifiers), 211

H

- hacking prevention, 259-265
 - (see also security)
 - about the risks, 259
 - article by OWASP online, 284
 - cross-site scripting (XSS) attack, 241, 264
 - article by OWASP online, 284
 - sanitizing input, 283
 - JavaScript injection into HTML, 264
 - path traversal attack, 523
 - PDO quote method, 260-261
 - placeholders in MySQL, 261-264
 - SQL injection attack, 259
 - steps to take, 260-261
- height attribute in forms (HTML), 281
- Hello World
 - document root file
 - macOS, 28
 - Windows, 25
 - JavaScript, 313
 - Node.js, 520
 - PHP, 34
- HELP (\h, \?) command (MySQL), 168
- heredoc (<<<) operator (PHP), 49
 - example of form code, 243, 247
- hex code shorthand for colors (CSS), 425
- hexadecimal (\xXX) escape character, 322
- hidden fields in forms (HTML), 276
 - not secure, 277
- history object in web browsers (JavaScript), 331, 469
 - go, back, and forward methods, 331
 - pushState method, 331
 - replaceState method, 331
- hover pseudoclass script (CSS), 442
- HSL colors (CSS), 431
- HTML (Hypertext Markup Language)
 - about HTML5, 11
 - (see also HTML5)
 - basics of, 2-5
 - benefits of, 5
 - client/server request/response process, 3-5
 - cookies, 290
 - closing / character, 12
 - CSS crucial to, 10
 - (see also CSS)
 - Document Object Model
 - adding new elements via JavaScript, 472
 - hierarchy of objects, 329
 - jQuery simplifying traversal and manipulation, 483
 - removing elements via JavaScript, 474
 - events, 342
 - (see also events)
 - file_get_contents fetch and display, 151
 - forms
 - about, 267
 - building, 267
 - default values, 270
 - DOMPurify library, 284
 - example of PHP integrating with forms, 284-287
 - example of username check, 14-16
 - GET method of submission, 246
 - hacking prevention via placeholders, 261-264
 - htmlspecialchars function in PHP, 241, 247, 294
 - image and text posted, 547
 - image uploader, 152
 - input types, 271-283
 - PHP querying MySQL database, 243-246
 - POST method of submission, 152, 246
 - retrieving submitted data, 269-284
 - sanitizing input, 283
 - sanitizing input via htmlentities function, 61, 264, 283
 - social networking site, 540, 544
 - social networking site photo and text, 547
 - validation, 154-156
 - validation of input using JavaScript, 375-383
 - validation of input via PHP, 393-399
 - history, 1
 - JavaScript added to web page, 313-316
 - <script> and </script> tags, 313, 334
 - inline JavaScript, 469
 - pulling JavaScript code from files, 315
 - scripts within document head, 315
 - PHP incorporated into, 33-35
 - React added to web page, 485
 - regular expressions not used for parsing, 384
 - online post explaining, 384
 - what to use for parsing, 384
- HTML Living Standard, 11
- HTML5

- about, 11
- closing / character, 12
- supplemental material online, [xxi](#), 11
- htmlentities function (PHP), [61](#), [264](#), [283](#)
- htmlspecialchars function (PHP), [157](#)
 - directory listing output, [157](#)
 - HTTP authentication output, [294](#), [299-301](#)
 - PHP fetching data from MySQL, [241](#), [247](#)
- HTTP (Hypertext Transfer Protocol)
 - authentication, [292-301](#)
 - example program, [298-301](#)
 - hash storage documentation online, [297](#)
 - htmlspecialchars function, [294](#)
 - size of storage for hashes, [297](#)
 - storing usernames and passwords, [296-298](#)
 - validating username and password, [295](#), [544](#)
 - verifying password against hash, [297](#), [544](#)
 - basics of, [2-5](#)
 - client/server request/response process, [3-5](#)
 - cookies, [290](#)
 - GET and POST requests, [152](#)
 - history, 1
 - HTTPS instead, [306](#)
 - localhost hostname to view document root
 - about document root, [25](#)
 - Linux, [29](#)
 - macOS, [28](#)
 - phpMyAdmin tool for MySQL, [205](#)
 - Windows, [24](#)
 - Node.js http module, [521](#)
 - HTTPS instead of HTTP, [306](#)
 - hyphen (-) for range in regular expressions, [386](#)
- I
 - ID (#id) selectors (CSS), [416](#)
 - identity operator (see strict equality (===) operator)
 - if statement
 - JavaScript, [344](#)
 - PHP, [75-76](#)
 - image uploader, [152-157](#)
 - form in HTML, [152](#)
 - POST method of submission, [152](#)
 - validation, [154-156](#)
 - image/jpeg content type, [157](#)
 - imageconvolution function (PHP), [548](#)
 - imagecopyresampled function (PHP), [548](#)
 - imagecreatetruecolor function (PHP), [548](#)
 - implicit casting of PHP variables, [90](#)
 - include directive (PHP), [99](#)
 - includes array method (JavaScript), [367](#)
 - include_once directive (PHP), [100](#)
 - increment (++) operator
 - JavaScript, [319](#), [321](#), [335](#)
 - PHP, [43](#), [46](#)
 - indexes (MySQL), [184-205](#)
 - about, [184](#), [214](#)
 - slow database responsiveness, [189](#)
 - creating an index
 - ALTER TABLE, [184](#)
 - CREATE INDEX, [186](#)
 - creating a FULLTEXT index, [188](#)
 - creating when creating table, [186](#)
 - foreign keys, [214](#)
 - primary keys, [186-188](#), [210](#)
 - normalization, [211-219](#)
 - indexOf array method (JavaScript), [367](#)
 - inequality (!=) operator
 - JavaScript, [320](#), [337](#)
 - PHP, [44](#), [72](#)
 - inheritance from a class, [103](#)
 - caution about, [114](#)
 - PHP, [114-118](#)
 - final methods, [117](#)
 - parent operator, [115](#)
 - ini_get function (PHP), [306](#)
 - ini_set function (PHP), [306](#)
 - cookie-only sessions, [307](#)
 - session fixation, [308](#)
 - shared server, [309](#)
 - innerHTML property (JavaScript), [474](#)
 - example of use, [404](#)
 - risks of, [405](#)
 - innerText property (JavaScript), [477](#)
 - InnoDB storage engine, [171](#)
 - searching for “and”, [196](#)
 - tables can use FULLTEXT indexes, [189](#)
 - transaction storage engine, [223](#)
 - specifying InnoDB, [223](#)
 - input types in HTML forms, [271-283](#)
 - input validation via JavaScript, [375-383](#)
 - INSERT command (MySQL), [168](#)
 - adding data to a table, [179](#)
 - in a script, [253](#), [256](#)
 - inserting multiple rows of data, [203](#)

- in a script, 253
- instances of classes, 102, 359
 - JavaScript creating an instance, 360
 - PHP
 - cloning objects, 106
 - creating an object, 105
- INT or INTEGER data type (MySQL), 176
- interface of an object, 102
- internet
 - DNS, 4
 - history, 1
 - request/response process, 3-5
 - cookies, 290
- internet media content types, 154
 - image/jpeg, 157
- intval function (PHP), 91
- iOS applications via React Native, 506
- IP address
 - DNS, 4
 - request/response process, 3-5
 - session security, 306
- isset function (PHP), 292
 - HTTP authentication, 293, 295, 298-301
 - HTTP authentication before session, 302
 - variable scope, 325
- is_array function (PHP), 129

J

- JavaScript
 - about JavaScript, 311
 - history, 311
 - about using, 9
 - CSS instead of JavaScript, 415
 - example of username check, 14-16
 - putting it all together (see social networking site)
 - adding to web page, 313-316
 - <noscript> and </noscript> tags, 313
 - <script src=, 315, 466
 - <script> and </script> tags, 313, 334
 - inline JavaScript, 469
 - pulling JavaScript code from files, 315
 - scripts within document head, 315
 - Ajax, 10, 403
 - (see also Ajax (Asynchronous JavaScript and XML))
 - arrays, 363-372
 - associative arrays, 364
 - creating a new array, 363, 364

- element values assigned, 363
- methods, 366-372
- multidimensional arrays, 365
- passed to functions by reference, 325
- spread syntax, 368
- asynchronous communication, 5
- benefits of, 5
- conditionals, 344-347
 - else statement, 344
 - if statement, 344
 - switch statement, 345-347
- constants, 327
 - arrays and objects can be modified, 328
- CSS accessed
 - by, byID, and style functions created, 463-466
 - byID and style functions in GitHub, 465
 - CSS properties, 466-469
 - getElementById function, 463-466
 - hiding and showing elements, 474
 - name hyphenation to camelCase, 466
- debugging, 316
- Document Object Model basis, 328-330
 - adding new elements, 472
 - document object URL specified, 331
 - hierarchy of objects, 329
 - length property, 331
 - links object, 330
 - objects, properties, methods, 328
 - removing elements, 474
 - using the DOM, 330
- error handling
 - onerror, 342
 - trapping with try...catch, 343
 - user input validation, 376, 378
- events
 - about, 342
 - click, 342
 - mouseover, 469
 - objects attached to, 470
 - onerror, 342
 - table of events and their triggers, 471
 - time-based, 475-480
- expressions, 333-335
 - literals and variables, 334
- functions, 324
 - anonymous functions, 372
 - arguments array, 355
 - arrays passed by reference, 325

- arrow functions, 373
- defining a function, 324, 353-356
- multiple properties and methods in one expression, 357
- pure versus impure code, 489
- rest parameter syntax, 354
- returning a value, 356
- returning an array, 358
- setTimeout calling, 475
- getElementById function
 - byID and style functions in GitHub, 465
 - called via \$, 330
 - CSS from JavaScript, 463-466
- Google V8 JavaScript engine, 13
- JSX extension and closing character, 12
- looping, 347-351
 - break command, 350
 - continue statement, 351
 - do...while loops, 348
 - for loops, 349
 - for...in loops, 365
 - for...of loop, 350
 - while loops, 348
- Node.js full-stack development, 13
- objects
 - about objects, 359
 - accessing objects, 360
 - associative array structure from, 364
 - class declaration, 359
 - creating an object, 359
 - display of object turned off, 474
 - Document Object Model basis, 328-330
 - events attached to, 470
 - for...in loop iterating through, 365
 - instance created, 360
 - instances of classes, 359
 - legacy object-functions, 361
 - static methods and properties, 361
 - visibility, 474
 - window object properties, 468
 - window object property documentation online, 469
- operators, 319-322, 335-341
 - about, 319, 335
 - arithmetic, 319, 335
 - assignment, 320, 335
 - associativity, 337
 - binary operators, 336
 - bitwise, 335
 - comparison, 320, 335, 338
 - incrementing, decrementing, 321, 335
 - logical, 321, 335, 340
 - precedence, 336
 - precedence documentation online, 336
 - relational, 337-341
 - string, 335
 - ternary operator, 336, 347
 - truthy and falsy values, 339
 - unary operators, 336
- outputting results, 312
 - alert pop-up window, 312
 - console.log, 312
 - document.write function, 9-10, 312
 - writing directly into HTML elements, 312
- React library, 6
 - (see also React JavaScript library)
- regular expressions, 391
- request/response sequence, 5
- semicolons, 313, 317
- URL reference in, 329
- variables, 317-319
 - about, 317
 - arrays, 318
 - explicit casting, 351
 - global variables, 324
 - let keyword, 326
 - local variables, 325
 - loosely typed, 322
 - naming rules, 317
 - numeric variables, 318
 - objects versus, 359
 - string variables, 318
 - TypeScript for types, 323
 - variable typing, 322-324
 - variable typing documentation online, 324
- XML via Babel JSX extension, 486
- JavaScript Object Notation (see JSON (JavaScript Object Notation))
- join array method (JavaScript), 369
- joining tables (MySQL), 201-204
 - JOIN...ON, 204
 - NATURAL JOIN, 203
- jQuery
 - \$ prefix for aliases, 317
 - about, 483
 - asynchronous communication, 6

- as framework, 413, 483
- supplemental material online, xxi
- jQuery Mobile supplemental material online, xxi
- JSDoc comments, 317
- JSON (JavaScript Object Notation) asynchronous communication, 410-412
- JSX (JavaScript XML)
 - closing character, 12
 - inline conditional statements, 497
 - multiple lines, 497
 - React based around, 483
- justifying content
 - flexbox layout, 447
 - grid layout, 459

K

- keys (see indexes (MySQL))

L

- labels in forms (HTML), 274
- LAMP (Linux, Apache, MySQL, PHP), 18
 - checking for preinstalled web server, 29
 - MySQL
 - command line interface startup, 165
 - table names case-sensitive, 169
 - pager less; command to page output, 190
- length property (JavaScript), 331
 - arrays, 363
 - window object, 468
- less than (<) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- less than or equal to (<=) operator
 - JavaScript, 320, 338
 - PHP, 44, 72
- let keyword (JavaScript), 326
- LIKE keyword (MySQL), 193
 - % before or after text, 193
- LIMIT keyword (MySQL), 194
 - caution about zero-indexed offsets, 195
- __LINE__ constant (PHP), 54
- links object (JavaScript), 330
- Linux installation of Node.js, 519
- Linux, Apache, MySQL, PHP (LAMP), 18
 - checking for preinstalled web server, 29
 - MySQL
 - command line interface startup, 165
 - table names case-sensitive, 169

- pager less; command to page output, 190
- list attribute in forms (HTML), 282
- list function (PHP), 124
- lists and keys (React), 498
- literals
 - definition of literal, 65
 - in expressions
 - JavaScript, 334
 - PHP, 65
 - literal strings in PHP, 48
 - non-literals versus, 65
- loadHTML method (PHP), 384
- local variables
 - JavaScript, 325
 - PHP, 56-58, 99
 - static variables in PHP, 59
- localhost hostname to view document root
 - about document root, 25
 - Linux, 29
 - macOS, 28
 - Hello World file into document root, 28
 - Node.js, 522
 - web server, 523
 - phpMyAdmin tool for MySQL, 205
 - serving pages from document root versus filesystem, 26, 28
 - social networking site, 532
 - Windows, 24
 - Hello World file into document root, 25
- LOCK command (MySQL), 168
 - before running mysqldump, 228
- locking files for multiple accesses (PHP), 149
 - as advisory lock, 150
 - response time and, 150
- logging in
 - HTTP authentication, 292-301
 - example program, 298-301
 - hash storage documentation online, 297
 - htmlspecialchars function, 294
 - storing usernames and passwords, 296-298
 - validating username and password, 295, 544
 - verifying password against hash, 297, 544
 - login file for PHP querying MySQL, 236-238
 - MySQL
 - logging in as root, 163

- logging in as user, 170
 - social networking site, 541, 544
 - working remotely, 29
- logical operators
 - JavaScript, 321, 335, 340
 - short-circuit evaluation, 341
 - MySQL queries, 204
 - PHP, 44, 72-74
 - table of inputs and results, 74
- LONGLOB data type (MySQL), 175
- LONGTEXT data type (MySQL), 175
- looping
 - about, 83
 - JavaScript, 347-351
 - break command, 350
 - continue statement, 351
 - do...while loops, 348
 - for loops, 349
 - for...in loops, 365
 - for...of loop, 350
 - while loops, 348
 - PHP, 83-90
 - breaking out of loop, 88
 - breaking out of nested loops, 89
 - continue statement, 89
 - do...while loops, 86
 - for loops, 86-88
 - foreach...as loops, 123-125, 132
 - while loops, 84-86

M

- Mac, Apache, MySQL, PHP (MAMP), 18
 - installing AMPPS on macOS, 27
 - document root, 28
 - document root Hello World, 28
 - PHP version, 28
 - serving pages from document root ver-
sus filesystem, 28
- MySQL
 - command line interface startup, 164
 - table names case-sensitive, 169
- macOS installation of Node.js, 516-519
- magic constants of PHP, 53
- magic quotes feature removed (PHP), 260
- MAMP (Mac, Apache, MySQL, PHP), 18
 - installing AMPPS on macOS, 27
 - document root, 28
 - document root Hello World, 28
 - PHP version, 28
 - serving pages from document root ver-
sus filesystem, 28
- MySQL
 - command line interface startup, 164
 - table names case-sensitive, 169
- many-to-many relationships, 221
- many-to-one relationships, 220
- map array method (JavaScript), 367
- MariaDB as clone of MySQL, 6
 - open source, 6
- MATCH...AGAINST command (MySQL),
196-198
 - Boolean mode, 197
 - double quote marks for exact phrase, 197
- max attribute in forms (HTML), 282
 - step attribute, 281
- MEDIUMBLOB data type (MySQL), 175
- MEDIUMINT data type (MySQL), 176
- MEDIUMTEXT data type (MySQL), 175
- members of objects, 112
 - scope of members, 112
 - static members, 113
- metacharacters in regular expressions, 383
- __METHOD__ constant (PHP), 54
- methods of objects, 102, 359
 - JavaScript, 328
 - method chaining, 357
 - PHP, 105
 - final methods in inheritance, 117
 - scope of object members, 112
 - static methods, 110
 - writing methods, 109
- Microsoft Visual C++ Redistributable, 21
- MIME (Multipurpose Internet Mail Extension)
 - content type, 154
- min attribute in forms (HTML), 282
 - step attribute, 281
- minlength attribute in forms (HTML), 381
- .mjs file extension, 520
- mktime function (PHP), 140
- mobile phones
 - React Native, 506
 - screen space information, 469
- modularization of PHP code, 91
- modulus (%) operator
 - JavaScript, 319
 - PHP, 43
- modulus assignment (%=) operator
 - JavaScript, 320

- PHP, 43
- moving a file (PHP), 147
- multicolumn layout (CSS), 429
- multidimensional arrays
 - JavaScript, 318, 365
 - PHP, 125-128
- multiline strings (PHP), 49-51
 - browser handling of, 51
- multipart/form-data encoding, 152
- multiplication (*) operator
 - JavaScript, 319
 - PHP, 43
- multiplication and assignment (*=) operator
 - JavaScript, 320
 - PHP, 43
- MyISAM storage engine
 - FULLTEXT indexes and tables, 189
 - searching for “and”, 196
- MySQL
 - about, 161
 - default user and password, 163
 - default user and password risks, 259
 - scalability benchmarks online, 161
 - slow database responsiveness, 189
 - SQL as Structured Query Language, 161
 - ways of interacting with, 162
 - about using, 8
 - example of username check, 14-16
 - putting it all together (see social net-working site)
 - accessing via phpMyAdmin, 205
 - backing up, 228-232
 - all databases backed up, 232
 - dumping data into CSV, 233
 - mysqldump, 228
 - mysqldump redirected to a file, 230
 - planning your backups, 233
 - restore tested periodically, 234
 - restoring from a backup file, 232
 - single table backup, 231
 - basics, 161
 - database terms, 162
 - benefits of, 5
 - command line interface startup, 162
 - Linux, 165
 - macOS, 164
 - remote server, 166
 - Windows, 163
 - command line use, 167
 - logging in as root, 163
 - logging in as user, 170
 - prompts, 167
 - semicolon, 167
 - \c to cancel input, 168
 - commands commonly used, 168
 - adding a new column, 181
 - adding a new column for primary key, 187
 - adding data to a table, 179, 203
 - adding data to a table in a script, 253
 - case-insensitive, 169
 - changing column data type, 181
 - creating a database, 169, 533
 - creating a table, 171-172, 250
 - creating a table with an auto-incrementing column, 179
 - creating a table with indexes, 186
 - creating a table with primary key, 188
 - creating users, 169-171, 533
 - deleting a table, 183
 - deleting a table in a script, 252
 - logging in as root, 163
 - logging in as user, 170
 - renaming a column, 182
 - renaming a table, 181
 - data types, 173
 - AUTO_INCREMENT attribute, 177-179, 187
 - BINARY versus VARBINARY, 174
 - BLOB types, 175
 - changing column data type, 181
 - CHAR versus VARCHAR, 173
 - character sets, 173
 - DATE and TIME types, 177
 - numeric types, 176
 - TEXT types, 175
 - database design, 209
 - functions documentation online, 205
 - hacking prevention via placeholders, 261-264
 - indexes, 184-205
 - about, 184
 - creating a FULLTEXT index, 188
 - creating via ALTER TABLE, 184
 - creating via CREATE INDEX, 186
 - creating when creating table, 186
 - primary keys, 186-188, 210
 - slow database responsiveness, 189

- InnoDB storage engine, 171
- input on single or multiple lines, 172
- joining tables, 201-204
- MariaDB clone, 6
- Node.js working with, 525, 527-529
 - MySQL module install, 526
- normalization, 211-219
 - duplicates risking data, 211
 - First Normal Form, 212-214
 - Second Normal Form, 214-216
 - Third Normal Form, 217
 - when not to use normalization, 219
- open source, xix, 6, 14
- PHP querying a database, 235-265
 - about, 235
 - about the process, 235
 - connecting to database, 238
 - connection closed, 242
 - deleting a record, 249
 - error messages trapped, 239
 - fetching a result, 239-241
 - fetching a row, 241
 - login file created, 236-238
 - practical example, 243-246
 - query built and executed, 239
- placeholders, 261-264
- privacy and databases, 223
- prompts, 167
- querying a database, 190-201
 - AS keyword, 204
 - DELETE a row, 193
 - GROUP BY keywords, 201
 - LIKE keyword, 193
 - LIMIT keyword, 194
 - logical operators, 204
 - MATCH...AGAINST, 196-198
 - ORDER BY keywords, 200
 - phpMyAdmin tool for MySQL, 205
 - scripting, 254, 257
 - SELECT, 190
 - SELECT COUNT, 191
 - SELECT DISTINCT, 191
 - UPDATE...SET, 199
 - WHERE keyword, 193
- relationships, 219-223
 - about relational databases, 219
 - many-to-many, 221
 - one-to-many, 220
 - one-to-one, 220

- privacy and, 223
- request/response sequence, 4-5
- supplemental material online, xxi
- transactions, 223-228
 - about, 223
 - EXPLAIN, 226
 - ROLLBACK, 225
 - transaction storage engines, 223
- triggers for automatic changes, 219
- version, 171
- working remotely via SSH, 29

mysql commands

- logging in as root, 163
- logging in as user, 170
- login file created, 236-238
- restoring from a backup file, 232

mysql> prompt, 167

mysqldump for backing up, 228

- dumping data into CSV, 233
- redirecting data to a file, 230
- single table backup, 231

N

names

- camelCase, 354
- constants in uppercase letters, 53
- global variables in uppercase letters, 59
- JavaScript
 - classes, instances, properties, methods, 360
 - CSS hyphenation to camelCase, 466
 - functions, 317, 353
 - period separating objects from properties, methods, 328
 - variables, 317
 - window object name property, 468
- MySQL
 - aliases via AS keyword, 204
 - lowercase for table names, 169
- PHP
 - about case sensitivity, 143
 - arrays and variables cannot share names, 129
 - functions, 96
 - variable naming rules, 42
 - __ (double underscore), 53, 109
 - React events in camelCase, 495
 - __NAMESPACE__ constant (PHP), 54
- NaN (Not a Number), 334

- NATURAL JOIN keywords, 203
- negation of a character class in regular expressions, 386
- new keyword for creating an object
 - JavaScript, 360
 - PHP, 105
- newline (\n) character, 48, 322
 - . wildcard in regular expressions, 384
- nginx, 510
- Node.js
 - about, 509
 - Apache alternative, 13, 509
 - open source, 509
 - building a web server, 522-525
 - localhost, 523
 - .env support, 528
 - MySQL username and password, 528
 - further information, 529
 - Node.js website, 530
 - getting started, 520-522
 - Ctrl-C to exit a program, 522
 - ECMAScript modules, 520
 - HTTP connections, 521
 - localhost, 522
 - port number, 521
 - installing
 - Linux, 519
 - macOS, 516-519
 - testing via node -v, 515, 519, 520
 - Windows, 510-516
 - npm, 510
 - documentation online, 527
 - installing modules with, 526
 - website, 527
 - working with modules, 525-529
 - about, 525
 - built-in modules, 526
 - .env support, 528
 - installing with npm, 526
 - MySQL module, 527-529
 - MySQL module install, 526
- non-literals versus literals, 65
- nonword character (\W) in regular expressions, 390
- normalization, 211-219
 - duplicates risking data, 211
 - First Normal Form, 212-214
 - Second Normal Form, 214-216
 - Third Normal Form, 217
 - when not to use normalization, 219
- not (!) logical operator
 - JavaScript, 321, 340
 - PHP, 45, 72-74
- not equal to (!=) operator
 - JavaScript, 320, 337
 - PHP, 44, 72
- not identical to (!==) operator
 - JavaScript, 320, 337
 - PHP, 44, 72
- NOT operator in MySQL WHERE queries, 204
- nowdoc (<<<) operator (PHP), 51
- npm (Node.js), 510
 - documentation online, 527
 - installing modules with, 526
 - Bootstrap icon library, 535
 - MySQL module, 526
 - website, 527
- NULL (PHP), 64
 - FALSE different, 64
- null as falsy (JavaScript), 339
- number and range in forms (HTML), 282
- numeric data types (MySQL)
 - numeric types, 176
 - signed versus unsigned numbers, 176
 - UNSIGNED qualifier, 176
- numeric variables
 - JavaScript, 318
 - PHP, 39
- numerically indexed arrays (PHP), 119-121
 - assignment via array keyword, 122
 - foreach...as loops, 123
 - multidimensional arrays, 127

0

- objects
 - about object-oriented programming, 93, 102
 - destructors, 108
 - members, 112
 - terminology, 102, 359
 - Document Object Model in JavaScript
 - design, 328-330
 - (see also Document Object Model (DOM))
 - JavaScript
 - about objects, 359
 - accessing objects, 360
 - associative array structure from, 364
 - class declaration, 359

- creating an object, 359
- display of object turned off, 474
- Document Object Model basis, 328-330
- events attached to, 470
- for...in loop iterating through, 365
- instance created, 360
- instances of classes, 359
- legacy object-functions, 361
- period separating from properties, 328
- static methods and properties, 361
- visibility, 474
- window object properties, 468
- window object property documentation
 - online, 469
- methods, 102, 359
 - interface of the object, 102
 - JavaScript, 328
- PHP
 - about objects, 102
 - accessing objects, 105
 - cloning objects, 106
 - constructors, 105, 108
 - creating an object, 105
 - declaring a class, 104
 - declaring constants, 111
 - declaring properties, 110
 - destructors, 108
 - inheritance, 114-118
 - inheritance caution, 114
 - methods, 109
 - passed by reference, 106
 - print_r function for human-readable,
 - 104, 106
 - scope of properties and methods, 112
 - static methods, 110
 - static properties, 113
 - \$this, 109
- properties, 102, 359
 - JavaScript, 328
 - JavaScript inline, 469
- occurrences (see instances of classes)
- octal (\XXX) escape character, 322
- one-to-many relationships, 220
- one-to-one relationships, 220
- one-way function for password, 296
- onerror event for error handling (JavaScript),
 - 342
- syntax errors caught, 343
- online resources (see resources online)
- onmouseover property (JavaScript), 469
- opacity property (CSS), 433
- open source
 - about, 14
 - Apache as, 14
 - Bootstrap icon library as, 535
 - FileZilla as, 30
 - MariaDB as, 6
 - MySQL as, xix, 6, 14, 161
 - Node.js as, 509
 - PHP as, xix, 6, 14
 - React as, 485
- Open Worldwide Application Security Project (OWASP) article on XSS prevention, 284
- opening a file (PHP), 144
 - file handle, 144
 - modes, 145
- OpenType (.otf) fonts, 435
- operating system calls (PHP), 157-158
 - cautions, 158
 - escapeshellcmd function, 158
- operators
 - about, 42
 - JavaScript, 319-322, 335-341
 - about, 319, 335
 - arithmetic, 319, 335
 - assignment, 320, 321, 335
 - associativity, 337
 - binary operators, 336
 - bitwise, 335
 - comparison, 320, 335, 338
 - equality, 320
 - incrementing, decrementing, 321, 335
 - logical, 335
 - precedence, 336
 - precedence documentation online, 336
 - relational, 337-341
 - string, 335
 - ternary operator, 336, 347
 - unary operators, 336
 - PHP, 42-46, 66
 - arithmetic, 42, 66
 - assignment, 43, 66
 - associativity, 69
 - binary operators, 66
 - bitwise, 66
 - comparison, 44, 66, 72
 - concatenation, 47, 66
 - equality, 44

- execution, 66, 67
 - increment/decrement, 46, 66
 - logical, 44, 66, 72-74
 - precedence table of operators, 68
 - relational, 70
 - ternary operator, 66
 - unary operators, 66
 - variable assignment, 46-47
 - precedence, 45, 67
 - Or (||) logical operator
 - JavaScript, 321, 340
 - PHP, 45
 - higher precedence than and, or, 73
 - or logical operator (PHP), 45, 72-74
 - lower precedence than && and ||, 73
 - OR operator in MySQL WHERE queries, 204
 - ORDER BY keywords (MySQL), 200
 - ASC for default ascending order, 201
 - DESC for descending order, 200
 - origin of URL, 407
 - same-origin and cross-origin requests, 408
 - .otf (OpenType) fonts, 435
 - overflow property of elements (CSS), 429
 - override attributes in forms (HTML), 281
 - OWASP (Open Worldwide Application Security Project) article on XSS prevention, 284
 - O'Reilly Learning Platform, xxii
 - Node.js, 530
 - React, 486
- ## P
- packet sniffing, 306
 - pager less; command to page output (Linux), 190
 - nopager; to restore standard output, 190
 - parent class in inheritance, 103
 - parent operator in inheritance, 115
 - parentheses
 - functions
 - JavaScript, 324, 372, 373, 476
 - PHP, 94, 96
 - regular expression grouping, 385
 - Parse error and semicolons (PHP), 37, 51
 - parseFloat function (JavaScript), 351
 - parseInt function (JavaScript), 351
 - partitions of tables (MySQL), 227
 - password displayed as asterisks, 541
 - password_hash function (PHP), 297
 - example program, 298-301
 - password_verify function (PHP), 297
 - example program, 298-301
 - social networking site, 544-545
 - path traversal attack, 523
 - PDO (PHP Data Objects)
 - about, 236
 - fetch method data styles, 241
 - documentation online, 241
 - PHP querying MySQL database
 - connecting to MySQL server, 238
 - connection closed, 242
 - database login file, 236
 - fetching a result, 239-241
 - fetching a row, 241
 - practical example, 243-246
 - query built and executed, 239
 - prepare method, 262
 - quote method to sanitize, 246, 260-261
 - social networking site functions.php, 533
 - period (.) in JavaScript, 328
 - multiple properties and methods in one expression, 357
 - window object properties, 468
 - Perl versus PHP, 5
 - PHP
 - <?php and ?>, 7, 33
 - <? and ?> deprecated, 34
 - Hello World, 34
 - heredoc operator alternative, 248
 - omitting closing tag, 35
 - about structure of
 - basic syntax, 36
 - comments, 35
 - operators, 42-46
 - variable assignment, 46-47
 - variables, 37-42
 - about using, 7, 12
 - example of username check, 14-16
 - first PHP program, 38
 - putting it all together (see social networking site)
 - arrays
 - \$GLOBALS array, 60
 - \$_COOKIE array, 60
 - \$_ENV array, 60
 - \$_FILES array, 60, 154, 547, 548
 - \$_GET array, 60, 246
 - \$_POST array, 60, 246
 - \$_REQUEST array, 60

- `$_SERVER` array, 60, 293, 294, 295, 298-301, 302
- `$_SESSION` array, 60, 302-304, 306, 307
- array functions, 129-133
- assignment (`=>`) operator, 122
- assignment via array keyword, 122
- associative arrays, 121, 242
- creating a new array, 39
- `foreach...as` loops, 123-125, 132
- multidimensional arrays, 125-128
- numerically indexed arrays, 119-121
- superglobal variables, 60
- benefits of, 5
- conditionals, 74-81
 - `?` operator, 55, 66, 82
 - `else` statement, 76-78
 - `elseif` statement, 78
 - `if` statement, 75-76
 - `switch` statement, 79-81
- constants, 53
 - constants inside classes, 111
 - predefined constants, 53
- data objects (see PDO (PHP Data Objects))
- date and time functions, 140-143
 - 2038 as end of time, 140
 - date constants, 142
 - validity check via `checkdate`, 143
- documentation online
 - `date` function, 141
 - variable type conversion rules, 53
- dynamic output from server, 7, 33
- error messages trapped, 239, 247
- expressions, 63-66
 - Boolean expressions, 64
 - definition, 63
 - literals and variables, 65
 - statements from, 66
- file handling (see file handling (PHP))
- files included into current file, 99
 - files required to be included, 101
- functions, 55, 94-95
 - defining a function, 96
 - expression evaluation, 97
 - `function_exists` function, 101
 - returning a value, 96-98
 - returning an array, 98
 - returning global variables, 98
- GD library, 12
- HTML and PHP, 33-35
- looping, 83-90
 - about, 83
 - breaking out of loop, 88
 - breaking out of nested loops, 89
 - `continue` statement, 89
 - `do...while` loops, 86
 - for loops, 86-88
 - `foreach...as` loops, 123-125, 132
 - while loops, 84-86
- modularization, 91
- MySQL database queries, 235-265
 - about, 235
 - about the process, 235
 - connecting to database, 238
 - connection closed, 242
 - deleting a record, 249
 - error messages trapped, 239
 - fetching a result, 239-241
 - fetching a row, 241
 - login file created, 236-238
 - practical example, 243-246
 - query built and executed, 239
- Node.js not able to run scripts, 13
- objects
 - about objects, 102
 - accessing objects, 105
 - cloning objects, 106
 - constructors, 105, 108
 - creating an object, 105
 - declaring a class, 105
 - declaring constants, 111
 - declaring properties, 110
 - destructors, 108
 - inheritance, 114-118
 - inheritance caution, 114
 - methods, 109
 - passed by reference, 106
 - `print_r` function for human-readable, 104, 106
 - scope of properties and methods, 112
 - static methods, 110
 - static properties, 113
 - `$this`, 109
- open source, xix, 6, 14
- operators, 42-46, 66
 - arithmetic, 42, 66
 - assignment, 43, 66
 - associativity, 69
 - binary operators, 66

- bitwise, 66
- comparison, 44, 66, 72
- concatenation, 47, 66
- equality, 44, 71-72
- execution, 66, 67
- increment/decrement, 46, 66
- logical, 44, 66, 72-74
- precedence, 45, 67
- precedence table of operators, 68
- relational, 70
- ternary operator, 66
- unary operators, 66
- variable assignment, 46-47
- output via echo and print, 312
 - echo versus print command, 54
- phpMyAdmin tool for MySQL, 205
- regular expressions, 392
 - (see also regular expressions)
- request/response sequence, 4-5
- supplemental material online, xxi
 - book's examples, 35
- system calls, 157-158
- validation of user input, 393-399
- variables, 37-42
 - about, 37
 - arrays, 39-42
 - arrays, two-dimensional, 40-42
 - dollar (\$) symbol, 37
 - explicit casting, 90
 - global variables, 58, 99
 - implicit casting, 90
 - incrementing and decrementing, 46
 - local variables, 56-58, 99
 - loosely typed, 52, 71, 90
 - naming rules, 42
 - numeric, 39
 - scope, 56-61, 99
 - static variables, 59, 99
 - string, 37-39
 - superglobal variables, 60
 - \$this, 109
 - variable typing, 52
- version, 94, 101
 - checking if a function exists, 101
- PHP Data Objects (see PDO (PHP Data Objects))
- .php file extension, 7
- phpinfo function, 94
- phpMyAdmin tool for MySQL, 205
 - documentation online, 206
- placeholder attribute in forms (HTML), 280
- placeholders preventing hacking (MySQL), 261-264
 - social networking site functions.php, 533
- plus sign (+) in regular expressions, 383
 - HTML tag match expressions, 384
- pop array method (JavaScript), 369-371
- port number for web server, 39
 - Node.js, 521
- POST method (HTTP)
 - browsers never cache, 409
 - first asynchronous program, 404-407
 - XMLHttpRequest object, 412
 - forms, 246
 - sanitizing via htmlentities function, 61, 264, 283
 - image uploader, 152
 - PHP querying MySQL database, 243
- power (**) operator (PHP), 43
- precedence of operators, 67
 - JavaScript, 336
 - associativity, 337
 - documentation online, 336
 - table of operator precedence, 336
 - PHP, 45
 - associativity of operators, 69
 - table of operator precedence, 68
- PREPARE command (MySQL), 261
- prepare method (PDO), 262
 - colon prefix for values, 263
- HTTP authentication, 298-301
 - starting a session, 302
- social networking site functions.php, 533
- primary keys (MySQL), 186-188, 210
 - normalization, 211-219
- print command (PHP)
 - echo command versus, 54
 - expression evaluation, 97
 - PHP output via print, 312
 - TRUE, FALSE, NULL, 64
- printf function (PHP), 135
 - precision setting, 137
 - numeric padding setting, 137
 - string padding, 138
- print_r function (PHP)
 - arrays, 120
 - for loop and echo instead, 120
 - objects, 104, 106

- privacy
 - databases and, [223](#)
 - history object in web browsers, [331](#)
 - third-party site use, [436](#)
- Privacy Sandbox (Google), [301](#)
- private members, [112](#)
- privileges (see GRANT command (MySQL))
- prompts for MySQL, [167](#)
- properties of objects, [102](#), [359](#)
 - JavaScript, [328](#)
 - accessing CSS properties, [466](#)
 - inline JavaScript, [469](#)
 - window object properties, [468](#)
 - window object property documentation, [469](#)
- PHP, [105](#)
 - declaring, [110](#)
 - scope of object members, [112](#)
 - static properties, [113](#)
 - \$this, [109](#)
- React props, [490](#)
- protected members, [112](#)
- pseudoclasses (CSS), [416](#)
 - script with hover pseudoclass, [442](#)
- pseudoelements (CSS), [416](#)
- public members, [112](#)
- push array method (JavaScript), [363](#), [369-371](#)
- putting it all together (see social networking site)
- PuTTY for remote work login, [29](#)

Q

- querying a database (MySQL), [190-201](#)
 - AS keyword, [204](#)
 - DELETE a row, [193](#)
 - GROUP BY keywords, [201](#)
 - LIMIT keyword, [194](#)
 - MATCH...AGAINST, [196-198](#)
 - Boolean mode, [197](#)
 - ORDER BY keywords, [200](#)
 - phpMyAdmin tool for MySQL, [205](#)
 - scripting, [254](#), [257](#)
 - SELECT, [190](#)
 - SELECT COUNT, [191](#)
 - SELECT DISTINCT, [191](#)
 - UPDATE...SET, [199](#)
 - WHERE keyword, [193](#)
 - LIKE keyword, [193](#)
 - logical operators, [204](#)

- question mark (?) operator (PHP), [55](#), [82](#)
- question mark (?) placeholder (MySQL), [261-264](#)
- question mark (?) ternary operator
 - JavaScript, [336](#), [347](#)
 - PHP, [66](#)
- QUIT (\q) command (MySQL), [168](#)
- quotation mark, double (")
 - JavaScript escape character, [322](#)
 - PHP
 - printf parameter string, [135](#)
 - variable value in string, [48](#)
- quotation mark, single (')
 - JavaScript escape character, [322](#)
 - PHP
 - literal strings, [48](#)
 - printf argument string, [136](#)
- quote method (PDO), [246](#), [260-261](#)

R

- radio buttons in forms (HTML), [275](#)
- range and number in forms (HTML), [282](#)
- React JavaScript library
 - about, [6](#), [483](#)
 - library not framework, [485](#)
 - open source, [485](#)
 - purpose, [484](#)
 - adding to web page, [485](#)
 - Babel JSX extension, [486](#)
 - development files on GitHub, [485](#)
 - asynchronous communication framework, [413](#), [483](#)
 - components, [6](#)
 - controlled components, [501](#)
 - documentation online, [486](#)
 - events, [495-496](#)
 - names in camelCase, [495](#)
 - first project, [487-495](#)
 - code using a class versus a function, [492](#)
 - code using class and function, [490](#)
 - code using class instead of function, [488](#)
 - component description, [490](#)
 - component mounting, [493](#)
 - component name capitalization, [488](#)
 - component unmounting, [493](#)
 - props, [490](#)
 - pure versus impure code, [489](#)
 - forms, [501-506](#)
 - controlled components, [501](#)

- select, 504
 - text area, 503
 - text input, 501-503
- JSX extension
 - closing character, 12
 - events in React, 495
 - inline conditional statements, 497
 - multiple lines, 497
- life-cycle, 493-495
- lists and keys, 498
 - unique keys, 499
- React Native, 506
 - tutorial online, 506
 - website, 506
- react.dev web page, 506
- state, 492-495
 - setState function, 493
- reading a cookie, 292
- reading from files (PHP), 145
 - file pointer, 148
 - reading an entire file, 151
- REAL or DOUBLE data type (MySQL), 176
- records
 - database definition, 161
 - deleting a record in PHP, 249
 - in a script, 255
 - row of table as record, 162
- redirect (>), 229
 - mysqldump redirected to a file, 230
- regular expressions
 - * metacharacter, 383
 - + metacharacter, 383
 - for a range, 386
 - . wildcard, 384
 - /g for global matching, 391
 - /i for case-insensitivity, 391
 - /m for multiline mode, 391
 - both JavaScript and PHP, 383
 - character classes, 386
 - negation, 386
 - examples, 387-389
 - general modifiers, 391
 - grouping with parentheses, 385
 - HTML should not be parsed with, 384
 - online post explaining, 384
 - what to use for parsing, 384
 - HTML tag match expressions, 384
 - JavaScript, 391
 - metacharacters for matching, 383
 - summary of, 389
 - negative lookahead documentation online, 387
 - \d indicating digit, 386
 - \W indicating nonword character, 390
 - \w indicating word character, 390
 - ^ for beginning of the line, 389
 - ^ for character class negation, 386
- relational operators
 - JavaScript, 337-341
 - PHP, 70
- relationships among databases, 219-223
 - many-to-many, 221
 - one-to-many, 220
 - one-to-one, 220
 - privacy and, 223
- remote server (see working remotely)
- removeChild function (JavaScript), 472, 474
- RENAME command (MySQL), 168
- rename function (PHP), 147
- renaming a column (MySQL), 182
- renaming a table (MySQL), 181
- request/response process, 3-5
 - cookies, 290
- require directive (PHP), 101
- required attribute in forms (HTML), 280
- require_once directive (PHP), 101, 238
- reset array function (PHP), 132
- resources online
 - AMPPS Windows installation documentation, 22, 27
 - Apache secure web server documentation, 306
 - book supplemental material, xxi
 - CSS selectors, 416
 - book web page, xxiii
 - book's examples, 35
 - byID and style functions, 465
 - database login.php, 236
 - input validation via JavaScript, 376
 - input validation via PHP, 393
 - React, 506
 - social networking site, 532
 - users table and accounts in PHP, 298
 - Chrome Developers Blog, 415
 - CSS
 - Can I Use... website, 416, 422
 - development information, 415
 - snapshot of stable modules, 415

- DOMPurify library, 284
- Fetch standard, 403
- FileZilla Wiki, 30
- Google Fonts Website, 436
- hash storage documentation, 297
- HTML
 - autocomplete attribute in forms, 280
 - not parsing with regular expressions, 384
- JavaScript
 - precedence documentation, 336
 - variable typing documentation, 324
- MySQL
 - GRANT command documentation, 171
 - REVOKE command documentation, 171
 - scalability benchmarks, 161
- Node.js
 - Linux download site, 519
 - macOS download site, 516
 - Node.js website, 530
 - npm documentation, 527
 - npm website, 527
 - Windows download site, 510
- PDO fetch data styles, 241
- PHP documentation
 - date function, 141
 - phpMyAdmin, 206
 - variable type conversion, 53
- React
 - Babel JSX extension, 486
 - development files, 485
 - documentation, 486
 - React Native tutorial, 506
 - React Native website, 506
 - react.dev web page, 506
- regular expression negative lookahead documentation, 387
- XSS prevention article by OWASP, 284
- rest parameter (...) syntax (JavaScript), 354
 - example fixNames function, 356
- restoring from a backup file, 232
 - test restoring periodically, 234
- return (\r) character, 48, 322
- reverse array method (JavaScript), 371
- REVOKE command (MySQL), 171
 - documentation online, 171
- RGB colors (CSS), 432
- RGBA colors (CSS), 433
- ROLLBACK a transaction (MySQL), 225
- root login at mysql prompt, 163
- row of table, 162
 - DELETE command, 193
 - LIMIT keyword, 194
 - unique via AUTO_INCREMENT attribute
 - as primary key, 187
 - unique via AUTO_INCREMENT attribute, 177-179
 - as primary key, 211
 - WHERE keyword, 193

S

- scalability
 - MySQL, 161
 - normalization and, 219
 - Node.js, 509
- scope of object members (PHP), 112
- scope of variables
 - JavaScript, 324-326, 328
 - var keyword, 326
 - PHP, 56-61
- scope resolution (::) operator (PHP), 111, 113
- screen object for user display information, 469
- scripting
 - asynchronous communication
 - first asynchronous program GET method, 409-410
 - first asynchronous program POST method, 404-407
 - JSON requests, 410-412
 - XMLHttpRequest object, 412
- commands commonly used in MySQL
 - adding data to a table, 253
 - AUTO_INCREMENT for ID, 256-257
 - creating a table, 250
 - deleting data, 255
 - describing a table, 251
 - dropping a table, 252
 - retrieving data, 254, 257
 - subquery, 257
 - updating data, 255
- CSS
 - hover, transition, transformation, 442
 - multiple column layout, 429
- example of PHP integrating with forms, 284-287
- Google font loaded, 437
- HTTP authentication, 298-301
- JavaScript
 - accessing CSS properties, 467

- adding new elements to DOM, 472
 - animation via time-based event, 478-480
 - clock created via setInterval function, 477
 - multiple column layout via CSS, 429
 - onerror event for error handling, 342
 - PHP querying MySQL database, 235-265
 - practical example, 243-246
 - validation of user input
 - JavaScript, 376-383
 - PHP, 393-399
- security
 - AMPPS password in Windows installation, 23
 - development server for, 17
 - directory traversal attack, 523
 - “goto fail” bug in SSL, 76
 - hacking prevention, 259-265
 - about the risks, 259
 - article by OWASP online, 284
 - cross-site scripting (XSS) attack, 241
 - JavaScript injection into HTML, 264
 - PDO quote method, 260-261
 - placeholders in MySQL, 261-264
 - sanitizing input, 283
 - SQL injection attack, 259
 - steps to take, 260-261
 - HTML
 - cross-site scripting (XSS) attack, 241, 264
 - form validation, 154-156
 - hidden fields in forms, 277
 - htmlspecialchars protection, 157, 241, 247
 - sanitizing input, 283
 - MySQL
 - default user and password risks, 259
 - error message contents, 238
 - root access, 169
 - Node.js, 523
 - password_hash function, 297
 - example program, 298-301
 - password_verify function, 297
 - example program, 298-301
 - social networking site, 544-545
 - path traversal attack, 523
 - PDO quote method, 246, 260-261
 - phpinfo function, 94
 - privacy
 - databases and, 223
 - history object in web browsers, 331
 - third-party site use, 436
 - sessions, 306-309
 - about, 306
 - cookie-only sessions, 307
 - IP address to prevent hijacking, 306
 - session fixation, 308
 - shared server, 309
 - SQL injection attack, 259
 - superglobal variables and, 61
 - htmlentities function for sanitizing, 61
 - validated input still insecure, 375
 - web server performing, 9
 - XSS (cross-site scripting) attack, 241, 264
 - article by OWASP online, 284
 - innerHTML property risks, 405
 - sanitizing input, 283
- SELECT command (MySQL), 190
 - checking data added to table, 179
 - pager less; command to page output, 190
 - SELECT COUNT, 191
 - SELECT DISTINCT, 191
- selectors (CSS), 416
- self keyword (PHP), 114
 - referencing class constants, 111
- semicolon (;)
 - JavaScript, 313, 317
 - MySQL, 167
 - none for PHP accessing MySQL, 239
 - \c after a semicolon, 168
 - PHP, 36
 - for loops, 88
- server setup (see development server setup)
- server-side scripting, 5
 - Google V8 JavaScript engine for, 13
- servers and clients, 2
 - (see also web servers)
 - PHP, MySQL, JavaScript, CSS, HTML, 5
- request/response process, 3-5
 - cookies, 290
- working remotely, 29
 - logging in, 29
 - transferring files, 30
- sessions, 301-309
 - about, 301
 - ending a session, 304
 - security, 306-309
 - about, 306

- cookie-only sessions, 307
 - IP address to prevent hijacking, 306
 - session fixation, 308
 - shared server, 309
 - setting a timeout, 305
 - starting a session, 302-304
- session_destroy function (PHP), 304
- session_get_cookie_params function (PHP), 304
- session_start function (PHP), 302-304
- SET command (MySQL), 261
 - placeholder variables, 261
- setcookie function (PHP), 291
 - deleting a cookie, 292
- setInterval function (JavaScript), 476-478
 - canceling an interval, 478
 - clock created via, 477
- setState function (React), 493
 - class versus function, 492
- setTimeout function (JavaScript), 475
 - canceling a timeout, 476
 - example of use, 472
 - strings not passed to, 476
- setting a cookie, 291
- SFTP (SSH/Secure File Transfer Protocol), 30
 - not FTP, 30
 - programs supporting, 30
- short-circuit evaluation, 341
- SHOW command (MySQL), 168
 - SHOW databases
 - Linux command line startup, 165
 - macOS command line startup, 164
 - semicolon, 167
 - Windows command line startup, 163
 - SHOW tables to check table deletion, 183
- shuffle array function (PHP), 130
- signed versus unsigned numbers (MySQL), 176
 - UNSIGNED qualifier, 176
- single quote mark (see quotation mark, single ('))
- SMALLINT data type (MySQL), 176
- social networking site, 531-566
 - about, 531
 - checkuser.php, 543
 - styles.css marking if available, 544
 - CREATE DATABASE, 533
 - CREATE USER, 533
 - designing the app, 532
 - friends.php, 555
 - intersection of following and followed, 555
 - functions.php, 532-534
 - GitHub for code, 532
 - header.php, 535
 - styles.css, 535
 - index.php, 539
 - javascript.js, 566
 - login.php, 544
 - form for username and password, 544
 - logout.php, 562
 - members.php, 551-553
 - adding and dropping friends, 552
 - listing all members, 552
 - viewing a user profile, 552
 - messages.php, 558-561
 - profile.php, 547-549
 - adding a profile image, 547
 - adding “About Me”, 547
 - displaying current profile, 548
 - form for photo and text, 547
 - processing the profile image, 548
 - setup.php, 537
 - logging in, 541
 - password as asterisks, 541
 - signup.php, 540-541
 - asynchronous call destination, 540
 - form for username and password, 540
 - username availability check, 540
 - styles.css, 562
 - header.php, 535
 - marking if username available, 544
- some array method (JavaScript), 366
- sort array function (PHP), 129
- sort array method (JavaScript), 371
- sorting queries via ORDER BY (MySQL), 200
- SOURCE command (MySQL), 168
- split function (JavaScript), 351
- spread (...) syntax (JavaScript), 368
- sprintf function (PHP), 139
- SQL (Structured Query Language), 161
 - (see also MySQL)
- SQL injection attack, 259
- square brackets ([])
 - arrays
 - JavaScript, 363-366
 - PHP, 120, 127, 128
 - regular expressions
 - character classes, 386

- negation of a character class, 386
- SSH
 - preinstalled in Windows and macOS, 29
 - working remotely
 - macOS logging in via Terminal, 29
 - MySQL, 29
 - Windows logging in via PuTTY, 29
- statements
 - expressions building, 66
 - flow control as, 335
- static methods
 - JavaScript, 361
 - PHP, 110
- static properties
 - JavaScript, 361
 - PHP, 113
- static variables (PHP), 59, 99
- STATUS (\s) command (MySQL), 168
- step attribute in forms (HTML), 281
- stopwords, 189, 196, 198
- storage engines (see InnoDB storage engine; MyISAM storage engine)
- strict equality (===) operator
 - JavaScript, 320, 337
 - PHP, 44, 71-72
 - validating username and password, 295
- String function (JavaScript), 351
- string functions
 - JavaScript
 - example returning one string from many, 356
 - substring, 356
 - toLowerCase, 356
 - toUpperCase, 356
 - MySQL documentation online, 205
 - PHP
 - sprintf, 139
 - strrev, 95
 - strtolower, 96
 - strtoupper, 95
 - str_repeat, 95
 - ucfirst, 96
- strings
 - CSS selectors, 416
 - matching part of a string, 416-417
 - supplemental material online, 416
 - JavaScript
 - += assignment operator, 320
 - array to strings via join method, 369
 - concatenation, 321
 - escaping characters, 322
 - string variables, 318
 - MySQL
 - canceling input in midst of, 168
 - LIKE keyword in queries, 193
 - LIKE keyword wildcard %, 193
 - PHP
 - concatenation, 47
 - concatenation assignment operator, 43, 47
 - double quote mark for variable value, 48
 - escaping characters, 48
 - exploding into an array, 130
 - multiline strings, 49-51
 - multiline strings in browsers, 51
 - single quote mark literal strings, 48
 - special characters tab, newline, return, 48
 - string variables, 37-39
 - strip_tags (HTML) and XSS attacks, 284
 - Structured Query Language (SQL), 161 (see also MySQL)
 - subclass in inheritance, 103
 - final methods, 117
 - parent constructors called, 116
 - submit button in forms (HTML), 279
 - substring method (JavaScript), 356
 - subtraction (-) operator
 - JavaScript, 319
 - PHP, 43
 - subtraction and assignment (-=) operator
 - JavaScript, 320, 321
 - PHP, 43, 46
 - superclass in inheritance, 103
 - final methods, 117
 - superglobal variables (PHP), 60
 - constants versus, 61
 - security and, 61
 - htmlentities function for sanitizing, 61
 - switch statement
 - JavaScript, 345-347
 - break command, 346
 - default action, 347
 - PHP, 79-81
 - alternative syntax, 81
 - break command, 80
 - default action, 81
 - syntax errors caught by onerror event, 343

system calls (PHP), 157-158
cautions, 158
escapeshellcmd function, 158

T

tab (\t) character, 48, 322

tables

basics, 161

definition of table, 162
names in lowercase, 169

data dumped into CSV file, 233

MySQL

adding a FULLTEXT index, 189
adding a new column, 181
adding a new column for primary key, 187
adding data, 179, 203
adding data in a script, 253
changing column data type, 181
creating a table, 171-172
creating a table in a script, 250, 537
creating a table with a numeric field, 176
creating a table with a primary key, 188
creating a table with an auto-incrementing column, 179
creating a table with indexes, 186
database design, 209
DELETE a row, 193
deleting a table, 183
deleting a table in a script, 252
DESCRIBE command, 172, 179, 181
DESCRIBE command in a script, 251
FULLTEXT indexes and, 189
joining tables, 201-204
locking tables, 168
locking tables before backup, 228
name aliases via AS keyword, 204
normalization, 211-219
removing a column, 179, 182
renaming a column, 182
renaming a table, 181
rows unique via AUTO_INCREMENT, 177-179, 187

partitions, 227

tablets

React Native, 506
screen space information, 469

Telnet protocol versus SSH, 29

Terminal for macOS SSH, 29

ternary operator, 66, 336, 347

testing

AMPPS installation

macOS, 28

Windows, 23-24

development server for, 17

restore of backup tested periodically, 234

web browsers for, 17

text areas in forms (HTML), 271

text boxes in forms (HTML), 271

TEXT data types (MySQL), 175

VARCHAR versus, 175

text effects (CSS), 433-435

text-overflow property, 434

text-shadow property, 433

word-wrap property, 435

textDecoration property (JavaScript), 465

third-party cookies, 289

Google phasing out, 301

this keyword (JavaScript), 360, 470

class constructor and instance, 359

example of use, 469

\$this variable (PHP), 109

time and date functions (MySQL) documentation online, 205

time and date functions (PHP), 140-143

2038 as end of time, 140

date constants, 142

validity check via checkdate, 143

time and date pickers in forms (HTML), 282

TIME data type (MySQL), 177

time function (PHP), 140

time-based events, 475-480

about, 475

animation, 478-480

setInterval function, 476-478

setTimeout function, 475

timeout for session, 305

TIMESTAMP data type (MySQL), 177

TINYBLOB data type (MySQL), 175

TINYINT data type (MySQL), 176

TINYTEXT data type (MySQL), 175

TLS (Transport Layer Security), 306

transactions, 223-228

about, 223

BEGIN, 225

COMMIT, 225

EXPLAIN, 226

ROLLBACK, 225

- START TRANSACTION, 225
 - transaction storage engines, 223
- transformations (CSS), 438-440
 - script with transitions, 442
- transitions (CSS), 440
 - delay, 441
 - duration, 441
 - properties, 440
 - returning to initial state, 443
 - script with transformation, 442
 - shorthand syntax, 442
 - timing, 441
- Transport Layer Security (TLS), 306
- triggers for automatic changes (MySQL), 219
- troubleshooting (see debugging)
- TRUE with value of 1 (PHP), 64
- true with value of "true" (JavaScript), 334
- TrueType (.ttf) fonts, 435
- TRUNCATE command (MySQL), 168
- truthy and falsy values (JavaScript), 339
- try...catch commands
 - JavaScript, 343
 - syntax errors need onerror, 343
 - PHP, 239
- .ttf (TrueType) fonts, 435
- type selectors (CSS), 416
- TypeError (PHP), 52
- typeof operator (JavaScript), 324
 - testing variable scope, 325
 - typeof null and typeof [], 324
- TypeScript as JavaScript with variable types, 323

U

- unary operators, 66, 336
- Undefined variable error (PHP), 57
- Unicode (\uXXXX) escape character, 322
- universal (*) selector (CSS), 416
- Universally Unique Identifiers (UUIDs), 211
- Unix epoch for timestamps, 140
- unlink function (PHP), 147
- UNLOCK command (MySQL), 168
- unsigned versus signed numbers (MySQL), 176
 - UNSIGNED qualifier, 176
- UPDATE command (MySQL), 168
 - in a script, 255
- UPDATE...SET command (MySQL), 199
- updating files (PHP), 148
- uploading a file (PHP), 152-157

- \$_FILES, 154
 - internet media content types, 154
 - image uploader, 152-157
- URL origin, 407
- URL reference in JavaScript, 329
 - replacing currently loaded with specified URL, 331
- USE command (MySQL), 168, 169
- user-agent string, 307
- username availability check, 14-16, 540, 543
- users created, 169-171
 - social networking site, 533
- UUIDs (Universally Unique Identifiers), 211

V

- validation
 - about security, 375
 - forms in HTML, 154-156
 - user input using JavaScript, 375-383
 - about, 375
 - setting up form and events, 376-379
 - validate.html online, 376
 - validation, 379-383
 - user input via PHP, 393-399
- var keyword (JavaScript)
 - global variables, 324
 - legacy replaced with let, 326
 - local variables, 325
- VARBINARY versus BINARY data types (MySQL), 174
- VARCHAR data type (MySQL), 173
 - CHAR versus, 173
 - TEXT versus, 175
- variables
 - in expressions
 - JavaScript, 334
 - PHP, 65
 - global variables
 - JavaScript, 324
 - PHP, 58, 98
 - JavaScript, 317-319
 - about, 317
 - arrays, 318
 - explicit casting, 351
 - global variables, 324
 - let keyword, 326
 - local variables, 325
 - loosely typed, 322
 - naming rules, 317

- numeric variables, 318
 - objects versus, 359
 - string variables, 318
 - TypeScript for types, 323
 - variable typing, 322-324
 - variable typing documentation online, 324
 - local variables
 - JavaScript, 325
 - PHP, 56-58, 99
 - PHP, 37-42
 - about, 37
 - arrays, 39-42
 - arrays, two-dimensional, 40-42
 - dollar (\$) symbol, 37
 - explicit casting, 90
 - expressions, 65
 - implicit casting, 90
 - incrementing and decrementing, 46-47
 - loosely typed, 52, 71, 90
 - naming rules, 42
 - numeric, 39
 - scope, 56-61, 99
 - string, 37-39
 - \$this, 109
 - variable typing, 52
 - placeholder variables, 261-264
 - static variables in PHP, 59, 99
 - string concatenation, 47
 - superglobal variables in PHP, 60
 - version of PHP to use, 94
 - checking if a function exists, 101
 - Visual C++ Redistributable (Microsoft), 21
 - Visual Studio Code (VSC), 31
 - Vue, 484
- ## W
- WAMP (Windows, Apache, MySQL, PHP), 18
 - installing AMPPS on Windows, 18-23
 - alternative WAMPs, 26
 - AMPPS documentation, 22, 27
 - configuration, 24
 - document root described, 25
 - document root Hello World, 25
 - document root viewed, 24
 - Microsoft Visual C++ Redistributable, 21
 - PHP version, 23
 - serving pages from document root versus filesystem, 26
 - testing the installation, 23-24
 - MySQL
 - command line interface startup, 163
 - table names case-insensitive, 169
 - WAMPServer, 26
 - web browsers
 - <pre> and </pre> tags, 104, 127
 - autofocus, 280
 - console
 - JavaScript errors displayed, 316
 - opening, 316
 - Safari Develop menu enabled, 316
 - cookies
 - disabled, 290, 301
 - editing, 291
 - reading, 292
 - request/response process, 290
 - CSS
 - Can I Use... website, 416, 422
 - development, 415
 - flexbox editor, 444
 - font specification, 436
 - grid editor, 455
 - support of, 416
 - events normalized by React, 495
 - Fetch API, 403
 - (see also Fetch API (JavaScript))
 - htmlspecialchars protection, 157, 241, 247, 294
 - JavaScript running within, 311
 - errors displayed in browser console, 316
 - history object, 331
 - JavaScript engine required, 313
 - running outside of browser via Node.js, 509
 - multiline string output, 51
 - request/response process, 3-5
 - cookies, 290
 - Safari browser Develop menu enabled, 316
 - testing development work, 17
 - uploading via multipart/form-data encoding, 152
 - user-agent string, 307
 - web design (see dynamic web design)
 - web fonts (CSS), 435
 - Google web fonts, 436
 - privacy information online, 436

- website, 436
- specifying for browser, 436
- Web Hypertext Application Technology Working Group (WHATWG), 11
- web page for book, xxiii
- web servers
 - Apache, 12
 - (see also Apache web server; Node.js)
 - dynamic output via PHP, 7, 33
 - (see also PHP)
 - file_get_contents function in PHP, 151
 - HTTP authentication, 292-301
 - example program, 298-301
 - hash storage documentation online, 297
 - htmlspecialchars function, 294
 - installed on server, 292
 - size of storage for hashes, 297
 - storing usernames and passwords, 296-298
 - validating username and password, 295
 - verifying password against hash, 297, 544
 - Node.js alternative to Apache, 13
 - building a Node.js web server, 522-525
 - building an Apache web server (see development server setup)
 - port number, 39
 - request/response process, 3-5
 - cookies, 290
 - security via, 9
 - serving pages from document root versus filesystem, 26, 28
 - sessions, 301-309
 - about, 301
 - ending a session, 304
 - security, 306-309
 - setting a timeout, 305
 - starting a session, 302-304
 - uploading a file, 152-157
 - working remotely, 29
 - logging in, 29
 - transferring files, 30
- WHATWG (Web Hypertext Application Technology Working Group), 11
- WHERE keyword (MySQL), 193
 - deleting a row, 193
 - LIKE keyword, 193
 - % before or after text, 193
- logical operators, 204
- while loops
 - JavaScript, 348
 - PHP, 84-86
- width attribute in forms (HTML), 281
- wildcard (.) in regular expressions, 384
- window object (JavaScript)
 - centering in-browser alerts or dialogs, 469
 - properties, 468
 - documentation online, 469
- Windows installation of Node.js, 510-516
- Windows PowerShell
 - MySQL command line, 163
 - MySQL errors, 231
- Windows Subsystem for Linux (WSL) and Node.js, 510
- Windows, Apache, MySQL, PHP (WAMP), 18
 - installing AMPPS on Windows, 18-23
 - alternative WAMPs, 26
 - AMPPS documentation, 22, 27
 - configuration, 24
 - document root described, 25
 - document root Hello World, 25
 - document root viewed, 24
 - Microsoft Visual C++ Redistributable, 21
 - PHP version, 23
 - serving pages from document root versus filesystem, 26
 - testing the installation, 23-24
 - MySQL
 - command line interface startup, 163
 - table names case-insensitive, 169
- WinSCP for SFTP, 31
- word character (\w) in regular expressions, 390
 - nonword character (\W), 390
- word-wrap property (CSS), 435
- working remotely
 - about, 29
 - code editors, 31
 - logging in, 29
 - MySQL command line interface startup, 166
 - transferring files, 30
- World Wide Web
 - history, 1
 - Web 1.0, 1
 - Web 1.1, 5
 - Web 2.0, 11
- writing to files (PHP)
 - file pointer, 148

X

- documentation online, [412](#)
- xor (exclusive or) logical operator (PHP), [45](#), [46](#), [72-74](#)
- XSS (cross-site scripting) attack, [241](#), [264](#)
 - article by OWASP online, [284](#)
 - innerHTML property risks, [405](#)
 - sanitizing input, [283](#)

Y

626 | Index

About the Author

Robin Nixon is a broadcaster and author who has over 40 years of experience with writing software, developing websites and apps, and managing teams of developers. He also has an extensive history of writing about computers and technology, with a portfolio of over 500 published magazine articles and over 40 books, many of which have been translated into other languages.

Robin started his computing career in the Cheshire homes for disabled people, where he was responsible for setting up computer rooms in a number of residential homes, and for evaluating and tailoring hardware and software so that disabled people could use the new technology, sometimes by means of only a single switch operated by mouth or finger. Robin's first computer was a Tandy TRS 80 Model 1 with a massive 4KB of RAM!

Robin eventually went on to work for some of the UK's top-selling IT magazine publishers, where he held several roles including editorial, promotions, and cover disc editing.

With the dawn of the internet in the 1990s, Robin helped spearhead many new web developments such as the first internet radio station, one of the first webmail services, the first fully interactive chat service (before the development of Ajax), and the first widespread use of pop-up window technology.

Colophon

The animals on the cover of *Learning PHP, MySQL & JavaScript* are sugar gliders (*Petaurus breviceps*). Sugar gliders are small, gray-furred creatures that grow to an adult length of 6 to 7.5 inches. Their tails, distinguished by a black tip, are usually as long as their bodies. Membranes extend between their wrists and ankles and provide an aerodynamic surface that helps them glide between trees.

Sugar gliders are native to Australia and Tasmania. They prefer to live in the hollow parts of eucalyptus and other large trees with several other adult sugar gliders and their own children.

Though sugar gliders reside in groups and defend their territory together, they don't always live in harmony. One male will assert his dominance by marking the group's territory with his saliva and marking group members with a distinctive scent produced from his forehead and chest glands. This ensures that members of the group know when an outsider approaches.

Sugar gliders make popular pets because they are inquisitive and playful, and because many think they are cute. However, because they are exotic animals, sugar gliders need specialized, complicated diets; healthy housing requires a cage or space no less than the size of an aviary; it's not uncommon for them to lose control of their bowels while playing or eating; and in some states and countries, it is illegal to own sugar gliders as household pets.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Johnson's Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation

Try the O'Reilly learning platform free for 10 days.

