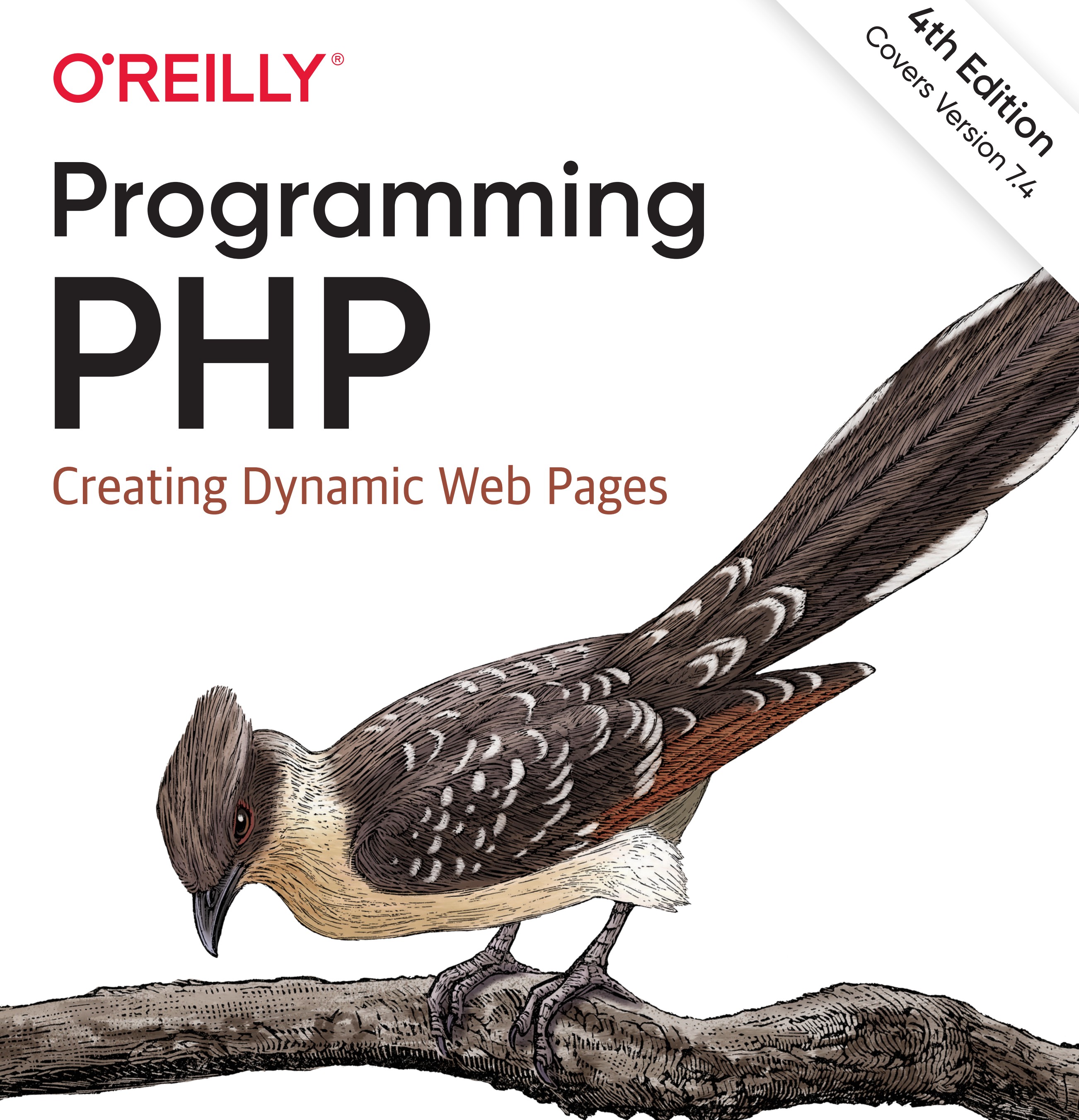
O'REILLY®

4th Edition
Covers Version 7.4

# Programming PHP

## Creating Dynamic Web Pages



Kevin Tatroe &
Peter MacIntyre
Foreword by Michael Stowe

# Programming PHP

Why is PHP the most widely used programming language on the web? This updated edition teaches everything you need to know to create effective web applications using the latest features in PHP 7.4. You'll start with the big picture and then dive into language syntax, programming techniques, and other details, using examples that illustrate both correct usage and common idioms.

If you have a working knowledge of HTML, authors Kevin Tatroe and Peter MacIntyre provide many style tips and practical programming advice in a clear and concise manner to help you become a top-notch PHP programmer.

- Understand what's possible when you use PHP programs
- Learn language fundamentals, including data types, variables, operators, and flow control statements
- Explore functions, strings, arrays, and objects
- Apply common web application techniques, such as form processing, data validation, session tracking, and cookies
- Interact with relational databases like MySQL or NoSQL databases such as MongoDB
- Generate dynamic images, create PDF files, and parse XML files
- Learn secure scripts, error handling, performance tuning, and other advanced topics
- Get a quick reference to PHP core functions and standard extensions

"PHP 7 has rejuvenated the PHP ecosystem, providing a powerful mix of world-class performance and highly anticipated features. If you're after the book that would help you unlock this potential, look no further than the new edition of *Programming PHP*!"

—**Zeev Suraski**
Cocreator of PHP

**Kevin Tatroe** has worked as an engineer on web stacks and Apple platforms for nearly 30 years, developing websites and mobile, desktop, and TV apps. He's attracted to technologies that allow for rapid iteration, experimentation, and highly opinionated architecture.

**Peter MacIntyre** has over 30 years of IT experience, primarily in PHP and web technologies. He's the author of *PHP: The Good Parts* (O'Reilly), *Pro PHP Programming* (APress), *WordPress Development In Depth* (PHP|Architect), and several other publications.


PROGRAMMING LANGUAGES / PHP

US $59.99 CAN $79.99
ISBN: 978-1-492-05413-9

Twitter: @oreillymedia
facebook.com/oreilly

# Praise for the 4th Edition of *Programming PHP*

PHP 7 has rejuvenated the PHP ecosystem, providing a powerful mix of world-class performance and highly anticipated features. If you're after the book that would help you unlock this potential, look no further than the new edition of *Programming PHP*!

*—Zeev Suraski, Cocreator of PHP*

By selecting *Programming PHP* you have taken that first step not only into PHP and its basics but into the future of website and web application development. With a firm understanding of the PHP programming language, and the tools available to you, the only limitations will be your imagination and your willingness to continue to grow and immerse yourself in the community.

*—Michael Stowe, Author, Speaker, and Technologist*

Covers all the details you'd expect in a programming language book and gets into more advanced topics that seasoned veterans would find interesting.

*—James Thoms, Senior Developer at ClearDev*

FOURTH EDITION

# Programming PHP

*Creating Dynamic Web Pages*

*Kevin Tatroe and Peter MacIntyre*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**Programming PHP**

by Kevin Tatroe and Peter MacIntyre

Copyright © 2020 Kevin Tatroe and Peter MacIntyre. 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 Editor:** Jennifer Pollock
**Development Editor:** Angela Rufino
**Production Editor:** Christopher Faucher
**Copyeditor:** Rachel Monaghan
**Proofreader:** Tom Sullivan
**Indexer:** Potomac Indexing, LLC
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

| | |
|---|---|
| March 2002: | First Edition |
| April 2006: | Second Edition |
| February 2013: | Third Edition |
| March 2020: | Fourth Edition |

**Revision History for the Fourth Edition**

2020-03-12: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492054139* for release details.

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

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-492-05413-9

[LSI]

*To Jenn*

*—KT*

*I would like to dedicate my portions of this book to my still wonderful wife, Dawn Etta Riley. I love you!*

*—PBM*

# Table of Contents

**Foreword** ........ xvii

**Preface** ........ xix

**1. Introduction to PHP** ........ 1
What Does PHP Do? 1
A Brief History of PHP 2
The Evolution of PHP 2
The Widespread Use of PHP 7
Installing PHP 7
A Walk Through PHP 8
Configuration Page 9
Forms 10
Databases 11
Graphics 13
What's Next 14

**2. Language Basics** ........ 15
Lexical Structure 15
Case Sensitivity 15
Statements and Semicolons 16
Whitespace and Line Breaks 16
Comments 17
Literals 20
Identifiers 20
Keywords 21
Data Types 22

Integers 22
Floating-Point Numbers 23
Strings 24
Booleans 25
Arrays 26
Objects 27
Resources 28
Callbacks 29
NULL 29
Variables 30
Variable Variables 30
Variable References 30
Variable Scope 31
Garbage Collection 33
Expressions and Operators 35
Number of Operands 36
Operator Precedence 36
Operator Associativity 37
Implicit Casting 37
Arithmetic Operators 38
String Concatenation Operator 39
Auto-Increment and Auto-Decrement Operators 40
Comparison Operators 41
Bitwise Operators 42
Logical Operators 44
Casting Operators 45
Assignment Operators 46
Miscellaneous Operators 48
Flow-Control Statements 49
if 49
switch 51
while 53
for 55
foreach 57
try...catch 57
declare 58
exit and return 58
goto 59
Including Code 59
Embedding PHP in Web Pages 61
Standard (XML) Style 62

SGML Style 63
Echoing Content Directly 63
What's Next 64

3. Functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
Calling a Function 65
Defining a Function 67
Variable Scope 69
Global Variables 69
Static Variables 70
Function Parameters 71
Passing Parameters by Value 71
Passing Parameters by Reference 71
Default Parameters 72
Variable Parameters 73
Missing Parameters 74
Type Hinting 74
Return Values 75
Variable Functions 76
Anonymous Functions 77
What's Next 79

4. Strings. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
Quoting String Constants 81
Variable Interpolation 81
Single-Quoted Strings 82
Double-Quoted Strings 83
Here Documents 83
Printing Strings 85
echo 85
print() 86
printf() 86
print_r() and var_dump() 88
Accessing Individual Characters 89
Cleaning Strings 90
Removing Whitespace 90
Changing Case 91
Encoding and Escaping 91
HTML 92
URLs 94
SQL 96

C-String Encoding 96
Comparing Strings 97
Exact Comparisons 97
Approximate Equality 99
Manipulating and Searching Strings 100
Substrings 100
Miscellaneous String Functions 101
Decomposing a String 102
String-Searching Functions 104
Regular Expressions 106
The Basics 107
Character Classes 108
Alternatives 109
Repeating Sequences 109
Subpatterns 110
Delimiters 110
Match Behavior 111
Character Classes 111
Anchors 112
Quantifiers and Greed 113
Noncapturing Groups 114
Backreferences 114
Trailing Options 114
Inline Options 116
Lookahead and Lookbehind 116
Cut 118
Conditional Expressions 118
Functions 119
Differences from Perl Regular Expressions 124
What's Next 124

**5. Arrays. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125**
Indexed Versus Associative Arrays 125
Identifying Elements of an Array 126
Storing Data in Arrays 127
Appending Values to an Array 128
Assigning a Range of Values 128
Getting the Size of an Array 129
Padding an Array 129
Multidimensional Arrays 129
Extracting Multiple Values 130

Slicing an Array 131
Splitting an Array into Chunks 131
Keys and Values 132
Checking Whether an Element Exists 132
Removing and Inserting Elements in an Array 133
Converting Between Arrays and Variables 134
Creating Variables from an Array 135
Creating an Array from Variables 135
Traversing Arrays 135
The foreach Construct 136
The Iterator Functions 136
Using a for Loop 138
Calling a Function for Each Array Element 138
Reducing an Array 139
Searching for Values 140
Sorting 142
Sorting One Array at a Time 142
Natural-Order Sorting 144
Sorting Multiple Arrays at Once 145
Reversing Arrays 145
Randomizing Order 146
Acting on Entire Arrays 147
Calculating the Sum of an Array 147
Merging Two Arrays 147
Calculating the Difference Between Two Arrays 147
Filtering Elements from an Array 148
Using Arrays to Implement Data Types 149
Sets 149
Stacks 149
Implementing the Iterator Interface 151
What's Next 153

**6. Objects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155**
Objects 155
Terminology 156
Creating an Object 157
Accessing Properties and Methods 157
Declaring a Class 159
Declaring Methods 159
Declaring Properties 162
Declaring Constants 163

Inheritance 164
Interfaces 165
Traits 165
Abstract Methods 168
Constructors 169
Destructors 170
Anonymous Classes 171
Introspection 171
Examining Classes 171
Examining an Object 173
Sample Introspection Program 174
Serialization 177
What's Next 180

**7. Dates and Times. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181**
What's Next 185

**8. Web Techniques. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187**
HTTP Basics 187
Variables 188
Server Information 189
Processing Forms 191
Methods 191
Parameters 192
Self-Processing Pages 194
Sticky Forms 196
Multivalued Parameters 197
Sticky Multivalued Parameters 199
File Uploads 200
Form Validation 202
Setting Response Headers 204
Different Content Types 205
Redirections 205
Expiration 205
Authentication 206
Maintaining State 207
Cookies 208
Sessions 212
Combining Cookies and Sessions 215
SSL 216
What's Next 216

**9. Databases. . . . . . . . . . 217**
Using PHP to Access a Database 217
Relational Databases and SQL 218
PHP Data Objects 219
MySQLi Object Interface 223
Retrieving Data for Display 225
SQLite 226
Direct File-Level Manipulation 229
MongoDB 237
Retrieving Data 241
Inserting More Complex Data 242
What's Next 245

**10. Graphics. . . . . . . . . . 247**
Embedding an Image in a Page 247
Basic Graphics Concepts 248
Creating and Drawing Images 249
The Structure of a Graphics Program 250
Changing the Output Format 251
Testing for Supported Image Formats 252
Reading an Existing File 252
Basic Drawing Functions 253
Images with Text 254
Fonts 255
TrueType Fonts 256
Dynamically Generated Buttons 258
Caching the Dynamically Generated Buttons 259
A Faster Cache 260
Scaling Images 262
Color Handling 264
Using the Alpha Channel 265
Identifying Colors 266
True Color Indexes 267
Text Representation of an Image 268
What's Next 269

**11. PDF. . . . . . . . . . 271**
PDF Extensions 271
Documents and Pages 271
A Simple Example 272
Initializing the Document 272

Outputting Basic Text Cells 273
Text 273
Coordinates 273
Text Attributes 276
Page Headers, Footers, and Class Extension 278
Images and Links 280
Tables and Data 283
What's Next 285

**12. XML. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 287**
Lightning Guide to XML 287
Generating XML 289
Parsing XML 291
Element Handlers 291
Character Data Handler 292
Processing Instructions 293
Entity Handlers 293
Default Handler 295
Options 296
Using the Parser 297
Errors 298
Methods as Handlers 299
Sample Parsing Application 300
Parsing XML with the DOM 304
Parsing XML with SimpleXML 305
Transforming XML with XSLT 306
What's Next 308

**13. JSON. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 309**
Using JSON 309
Serializing PHP Objects 310
Options 312
What's Next 313

**14. Security. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 315**
Safeguards 315
Filtering Input 316
Escaping Output Data 318
Security Vulnerabilities 322
Cross-Site Scripting 322
SQL Injection 323

Filename Vulnerabilities 324
Session Fixation 326
File Upload Traps 327
Unauthorized File Access 328
PHP Code Issues 331
Shell Command Weaknesses 332
Data Encryption Concerns 333
Further Resources 333
Security Recap 333
What's Next 334

**15. Application Techniques. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335**
Code Libraries 335
Templating Systems 336
Handling Output 339
Output Buffering 339
Output Compression 341
Performance Tuning 342
Benchmarking 343
Profiling 344
Optimizing Execution Time 346
Optimizing Memory Requirements 346
Reverse Proxies and Replication 347
What's Next 349

**16. Web Services. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 351**
REST Clients 351
Responses 353
Retrieving Resources 354
Updating Resources 355
Creating Resources 356
Deleting Resources 356
XML-RPC 357
Servers 357
Clients 359
What's Next 360

**17. Debugging PHP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 361**
The Development Environment 361
The Staging Environment 362
The Production Environment 363

php.ini Settings 363
Error Handling 365
Error Reporting 365
Exceptions 366
Error Suppression 367
Triggering Errors 367
Defining Error Handlers 368
Manual Debugging 371
Error Logs 373
IDE Debugging 374
Additional Debugging Techniques 376
What's Next 376

**18. PHP on Disparate Platforms. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 377**
Writing Portable Code for Windows and Unix 377
Determining the Platform 378
Handling Paths Across Platforms 378
Navigating the Server Environment 378
Sending Mail 379
End-of-Line Handling 380
End-of-File Handling 380
Using External Commands 381
Accessing Platform-Specific Extensions 381
Interfacing with COM 381
Background 381
PHP Functions 383
API Specifications 384

**Function Reference. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385**

**Index. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 493**

# Foreword

It's hard to believe that nearly 20 years ago I picked up my first PHP book. I had an interest in programming, extending beyond Netscape Composer and static HTML. I knew PHP would enable me to create dynamic, smarter websites—and to store and fetch data to create interactive web applications.

What I didn't know was the journey that unlocking these new capabilities with PHP would take me on, or how PHP would evolve 20 years later to become the programming language powering roughly 80% of the web, and backed by one of the nicest, friendliest, and most engaging communities.

A journey of a thousand miles begins with a single step. By selecting *Programming PHP* by Peter MacIntyre and Kevin Tatroe, you have taken that first step not only into PHP and its basics, but also into the future of website and web application development. With the available tools and a firm understanding of the PHP programming language, the only limitation will be your imagination and your willingness to continue to grow and immerse yourself in the community. The journey is yours, the possibilities endless, and the future for you to define.

As you get ready to begin this journey, I would like to share a couple tidbits of advice. First, take each chapter and put it into practice, try different things, and don't be afraid of breaking something or failing. While *Programming PHP* will establish a strong foundation, it's up to you to explore the language and find new and creative ways to pull together all of these components.

My second piece of advice: be an active part of the PHP community. Take advantage of online communities, user groups, and PHP conferences as you are able. As you try new things, share them with the community for their feedback and advice.

Not only are you sure to find a community of support—a group of some of the nicest people, who want you to succeed and are more than happy to take their time to help you through your journey—but you'll also establish a baseline of continuous learning, helping you grasp the core skills of PHP more quickly and keeping you up to date on

new programming theories, technologies, tools, and changes. Not to mention, you'll encounter an onslaught of terrible puns (including from yours truly).

With that, I would like to be among the first to welcome you and to wish you the very best on your journey—a journey that couldn't start off better than with this book!

— Michael Stowe, author, speaker, and technologist<br>San Francisco, California, Winter 2020

# Preface

Now more than ever, the web is a major vehicle for corporate and personal communications. Websites carry satellite images of Earth in its entirety; search for life in outer space; house personal photo albums, business shopping carts, and product lists; and so much more! Many of those websites are driven by PHP, an open source scripting language primarily designed for generating HTML content.

Since its inception in 1994, PHP has swept the web and continues its phenomenal growth today. The millions of websites powered by PHP are testament to its popularity and ease of use. Everyday people can learn PHP and build powerful dynamic websites with it.

The core PHP language (version 7+) features powerful string- and array-handling facilities, as well as greatly improved support for object-oriented programming. With the use of standard and optional extension modules, a PHP application can interact with a database such as MySQL or Oracle, draw graphs, create PDF files, and parse XML files. You can run PHP on Windows, which lets you control other Windows applications (such as Word and Excel with COM) or interact with databases using ODBC.

This book is a guide to the PHP language. When you finish it (we won't tell you how it ends!), you will know how the PHP language works, how to use the many powerful extensions that come standard with PHP, and how to design and build your own PHP web applications.

## Audience

PHP is a melting pot of cultures. Web designers appreciate its accessibility and convenience, while programmers appreciate its flexibility, power, diversity, and speed. Both cultures need a clear and accurate reference to the language. If you are a (web) programmer, then this book is for you. We show the big picture of the PHP language, and then discuss the details without wasting your time. The many examples clarify

the textual explanations; the practical programming advice and many style tips will help you become not just a PHP programmer, but a *good* PHP programmer.

If you're a web designer, you will appreciate the clear and useful guides to specific technologies, such as JSON, XML, sessions, PDF generation, and graphics. And you'll be able to quickly get the information you need from the language chapters, which explain basic programming concepts in simple terms.

This edition has been fully revised to cover the latest features of PHP version 7.4.

## Assumptions This Book Makes

This book assumes you have a working knowledge of HTML. If you don't know HTML, you should gain some experience with simple web pages before you try to tackle PHP. For more information on HTML, we recommend *HTML & XHTML: The Definitive Guide* by Chuck Musciano and Bill Kennedy (O'Reilly).

## Contents of This Book

We've arranged the material in this book so that you can either read it from start to finish or jump around to hit just the topics that interest you. The book is divided into 18 chapters and 1 appendix, as follows:

*Chapter 1, Introduction to PHP*
: Talks about the history of PHP and gives a lightning-fast overview of what is possible with PHP programs.

*Chapter 2, Language Basics*
: Is a concise guide to PHP program elements such as identifiers, data types, operators, and flow-control statements.

*Chapter 3, Functions*
: Discusses user-defined functions, including scope, variable-length parameter lists, and variable and anonymous functions.

*Chapter 4, Strings*
: Covers the functions you'll use when building, dissecting, searching, and modifying strings in your PHP code.

*Chapter 5, Arrays*
: Details the notation and functions for constructing, processing, and sorting arrays in your PHP code.

*Chapter 6, Objects*
: Covers PHP's updated object-oriented features. In this chapter, you'll learn about classes, objects, inheritance, and introspection.

*Chapter 7, Dates and Times*
: Discusses date and time manipulations like time zones and date math.

*Chapter 8, Web Techniques*
: Talks about techniques most PHP programmers eventually want to use, including processing web form data, maintaining state, and dealing with SSL.

*Chapter 9, Databases*
: Discusses PHP's modules and functions for working with databases, using MySQL database as examples. Also, SQLite and PDO database interface are covered. NoSQL concepts are also covered here.

*Chapter 10, Graphics*
: Demonstrates how to create and modify image files in a variety of formats from within PHP.

*Chapter 11, PDF*
: Explains how to create dynamic PDF files from a PHP application.

*Chapter 12, XML*
: Introduces PHP's extensions for generating and parsing XML data.

*Chapter 13, JSON*
: Covers JavaScript Object Notation (JSON), a standardized data-interchange format designed to be extremely lightweight and human-readable.

*Chapter 14, Security*
: Provides valuable advice and guidance for programmers creating secure scripts. You'll learn programming best practices to help you avoid mistakes that can lead to disaster.

*Chapter 15, Application Techniques*
: Talks about coding techniques like implementing code libraries, dealing with output in unique ways, and error handling.

*Chapter 16, Web Services*
: Describes techniques for dealing with external communication via REST tools and cloud connections.

*Chapter 17, Debugging PHP*
: Discusses techniques for debugging PHP code and for writing debuggable PHP code.

*Chapter 18, PHP on Disparate Platforms*
: Discusses the tricks and traps of the Windows port of PHP. It also discusses some of the features unique to Windows, such as COM.

*Appendix*
: Serves as a handy quick reference to all core functions in PHP.

## Conventions Used in This Book

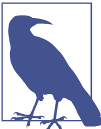
The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, general note, warning, or caution.

## 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, conferences, 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, please visit *http://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
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

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/programming-PHP-4e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

## Acknowledgments

### Kevin Tatroe

Once again, thanks to every individual who ever committed code to PHP, contributed to the vastness that is the PHP ecosystem, or wrote a line of PHP. You all made PHP what it was, is, and will continue to be.

To my parents, who once purchased a small LEGO set for a long and frightening plane trip, beginning an obsession with creativity and organization that continues to relax and inspire me to this day.

Finally, a heaping fourth spoonful of gratitude to Jenn and Hadden for helping inspire and encourage me through each and every day.

### Peter MacIntyre

I would like to praise the Lord of Hosts who gives me the strength to face each day! He created electricity through which I make my livelihood; thanks and praise to Him for this totally unique and fascinating portion of His creation!

To Kevin, who is once again my main coauthor on this edition, thanks for the effort and once again staying focused on this project to its publication.

To the technical editors who sifted through our code examples and tested them to make sure we were "telling the truth"—Lincoln, Tanja, Jim, and James—thanks!

And finally to all those at O'Reilly who so often go unmentioned—I don't know all your names, but I know what you have to do to get a project like this finally "out the door." The editing, graphics work, layout, planning, marketing, and so on all has to be done, and I certainly appreciate all your hard work toward this end.

# CHAPTER 1
# Introduction to PHP

PHP is a simple yet powerful language designed for creating HTML content. This chapter covers essential background on the PHP language. It describes the nature and history of PHP, which platforms it runs on, and how to configure it. This chapter ends by showing you PHP in action, with a quick walkthrough of several PHP programs that illustrate common tasks, such as processing form data, interacting with a database, and creating graphics.

## What Does PHP Do?

PHP can be used in two primary ways:

*Server-side scripting*
: PHP was originally designed to create dynamic web content, and it is still best suited for that task. To generate HTML, you need the PHP parser and a web server through which to send the coded document files. PHP has also become popular for generating dynamic content via database connections, XML documents, graphics, PDF files, and so much more.

*Command-line scripting*
: PHP can run scripts from the command line, much like Perl, awk, or the Unix shell. You might use the command-line scripts for system administration tasks, such as backup and log parsing; even some CRON job–type scripts can be done this way (as nonvisual PHP tasks).

In this book, however, we concentrate on the first item: using PHP to develop dynamic web content.

PHP runs on all major operating systems, from Unix variants (including Linux, FreeBSD, Ubuntu, Debian, and Solaris) to Windows and macOS. It can be used with

all leading web servers, including the Apache, Nginx, and OpenBSD servers, to name a few; even cloud environments like Azure and Amazon are on the rise.

The language itself is extremely flexible. For example, you aren't limited to outputting just HTML or other text files—any document format can be generated. PHP has built-in support for generating PDF files and GIF, JPEG, and PNG images.

One of PHP's most significant features is its wide-ranging support for databases. PHP supports all major databases (including MySQL, PostgreSQL, Oracle, Sybase, MS-SQL, DB2, and ODBC-compliant databases), and even many obscure ones. Even the more recent NoSQL-style databases like CouchDB and MongoDB are also supported. With PHP, creating web pages with dynamic content from a database is remarkably simple.

Finally, PHP provides a library of PHP code to perform common tasks, such as database abstraction, error handling, and so on, with the PHP Extension and Application Repository (PEAR). PEAR is a framework and distribution system for reusable PHP components.

# A Brief History of PHP

Rasmus Lerdorf first conceived of PHP in 1994, but the PHP that people use today is quite different from the initial version. To understand how PHP got where it is now, it is useful to know the historical evolution of the language. Here's that story, with ample comments and emails from Rasmus himself.

## The Evolution of PHP

Here is the PHP 1.0 announcement that was posted to the Usenet newsgroup (*comp.infosystems.www.authoring.cgi*) in June 1995:

```
From: rasmus@io.org (Rasmus Lerdorf)
Subject: Announce: Personal Home Page Tools (PHP Tools)
Date: 1995/06/08
Message-ID: <3r7pgp$aa1@ionews.io.org>#1/1
organization: none
newsgroups: comp.infosystems.www.authoring.cgi

Announcing the Personal Home Page Tools (PHP Tools) version 1.0.

These tools are a set of small tight cgi binaries written in C.
They perform a number of functions including:

. Logging accesses to your pages in your own private log files
. Real-time viewing of log information
. Providing a nice interface to this log information
. Displaying last access information right on your pages
. Full daily and total access counters
```

```
. Banning access to users based on their domain
. Password protecting pages based on users' domains
. Tracking accesses ** based on users' e-mail addresses **
. Tracking referring URL's - HTTP_REFERER support
. Performing server-side includes without needing server support for it
. Ability to not log accesses from certain domains (ie. your own)
. Easily create and display forms
. Ability to use form information in following documents

Here is what you don't need to use these tools:

. You do not need root access - install in your ~/public_html dir
. You do not need server-side includes enabled in your server
. You do not need access to Perl or Tcl or any other script interpreter
. You do not need access to the httpd log files

The only requirement for these tools to work is that you have
the ability to execute your own cgi programs. Ask your system
administrator if you are not sure what this means.

The tools also allow you to implement a guestbook or any other
form that needs to write information and display it to users
later in about 2 minutes.

The tools are in the public domain distributed under the GNU
Public License. Yes, that means they are free!

For a complete demonstration of these tools, point your browser
at: http://www.io.org/~rasmus

--
Rasmus Lerdorf
rasmus@io.org
http://www.io.org/~rasmus
```

Note that the URL and email address shown in this message are long gone. The language of this announcement reflects the concerns that people had at the time, such as password-protecting pages, easily creating forms, and accessing form data on subsequent pages. The announcement also illustrates PHP's initial positioning as a framework for a number of useful tools.

The announcement talks only about the tools that came with PHP, but behind the scenes the goal was to create a framework to make it easy to extend PHP and add more tools. The business logic for these add-ons was written in C; a simple parser picked tags out of the HTML and called the various C functions. It was never really part of the plan to create a scripting language.

So what happened?

Rasmus started working on a rather large project for the University of Toronto that needed a tool to pull together data from various places and present a nice web-based

administration interface. Of course, he used PHP for the task, but for performance reasons, the various small tools of PHP 1.0 had to be brought together better and integrated into the web server.

Initially, some hacks to the NCSA web server were made, to patch it to support the core PHP functionality. The problem with this approach was that as a user, you had to replace your web server software with this special, hacked-up version. Fortunately, Apache was also starting to gain momentum around this time, and the Apache API made it easier to add functionality like PHP to the server.

Over the next year or so, a lot was done and the focus changed quite a bit. Here's the PHP 2.0 (PHP/FI) announcement that was sent out in April 1996:

```
From: rasmus@madhaus.utcs.utoronto.ca (Rasmus Lerdorf)
Subject: ANNOUNCE: PHP/FI Server-side HTML-Embedded Scripting Language
Date: 1996/04/16
Newsgroups: comp.infosystems.www.authoring.cgi

PHP/FI is a server-side HTML embedded scripting language. It has built-in
access logging and access restriction features and also support for
embedded SQL queries to mSQL and/or Postgres95 backend databases.

It is most likely the fastest and simplest tool available for creating
database-enabled web sites.

It will work with any UNIX-based web server on every UNIX flavour out
there. The package is completely free of charge for all uses including
commercial.

Feature List:

. Access Logging
Log every hit to your pages in either a dbm or an mSQL database.
Having hit information in a database format makes later analysis easier.
. Access Restriction
Password protect your pages, or restrict access based on the refering URL
plus many other options.
. mSQL Support
Embed mSQL queries right in your HTML source files
. Postgres95 Support
Embed Postgres95 queries right in your HTML source files
. DBM Support
DB, DBM, NDBM and GDBM are all supported
. RFC-1867 File Upload Support
Create file upload forms
. Variables, Arrays, Associative Arrays
. User-Defined Functions with static variables + recursion
. Conditionals and While loops
Writing conditional dynamic web pages could not be easier than with
the PHP/FI conditionals and looping support
. Extended Regular Expressions
```

```
Powerful string manipulation support through full regexp support
. Raw HTTP Header Control
Lets you send customized HTTP headers to the browser for advanced
features such as cookies.
. Dynamic GIF Image Creation
Thomas Boutell's GD library is supported through an easy-to-use set of
tags.

It can be downloaded from the File Archive at: <URL:http://www.vex.net/php>

--
Rasmus Lerdorf
rasmus@vex.net
```

This was the first time the term *scripting language* was used. PHP 1.0's simplistic tag-replacement code was replaced with a parser that could handle a more sophisticated embedded tag language. By today's standards, the tag language wasn't particularly sophisticated, but compared to PHP 1.0 it certainly was.

The main reason for this change was that few people who used PHP 1.0 were actually interested in using the C-based framework for creating add-ons. Most users were much more interested in being able to embed logic directly in their web pages for creating conditional HTML, custom tags, and other such features. PHP 1.0 users were constantly requesting the ability to add the hit-tracking footer or send different HTML blocks conditionally. This led to the creation of an `if` tag. Once you have `if`, you need `else` as well, and from there it's a slippery slope to the point where, whether you want to or not, you end up writing an entire scripting language.

By mid-1997, PHP version 2.0 had grown quite a bit and had attracted a lot of users, but there were still some stability problems with the underlying parsing engine. The project was also still mostly a one-man effort, with a few contributions here and there. At this point, Zeev Suraski and Andi Gutmans in Tel Aviv, Israel, volunteered to rewrite the underlying parsing engine, and we agreed to make their rewrite the base for PHP version 3.0. Other people also volunteered to work on other parts of PHP, and the project changed from a one-person effort with a few contributors to a true open source project with many developers around the world.

Here is the PHP 3.0 announcement from June 1998:

```
June 6, 1998 -- The PHP Development Team announced the release of PHP 3.0,
the latest release of the server-side scripting solution already in use on
over 70,000 World Wide Web sites.

This all-new version of the popular scripting language includes support
for all major operating systems (Windows 95/NT, most versions of Unix,
and Macintosh) and web servers (including Apache, Netscape servers,
WebSite Pro, and Microsoft Internet Information Server).

PHP 3.0 also supports a wide range of databases, including Oracle,
```

```
Sybase, Solid, MySQ, mSQL, and PostgreSQL, as well as ODBC data sources.

New features include persistent database connections, support for the
SNMP and IMAP protocols, and a revamped C API for extending the language
with new features.

"PHP is a very programmer-friendly scripting language suitable for
people with little or no programming experience as well as the
seasoned web developer who needs to get things done quickly. The
best thing about PHP is that you get results quickly," said
Rasmus Lerdorf, one of the developers of the language.

"Version 3 provides a much more powerful, reliable, and efficient
implementation of the language, while maintaining the ease of use and
rapid development that were the key to PHP's success in the past,"
added Andi Gutmans, one of the implementors of the new language core.

"At Circle Net we have found PHP to be the most robust platform for
rapid web-based application development available today," said Troy
Cobb, Chief Technology Officer at Circle Net, Inc. "Our use of PHP
has cut our development time in half, and more than doubled our client
satisfaction. PHP has enabled us to provide database-driven dynamic
solutions which perform at phenomenal speeds."

PHP 3.0 is available for free download in source form and binaries for
several platforms at http://www.php.net/.

The PHP Development Team is an international group of programmers who
lead the open development of PHP and related projects.

For more information, the PHP Development Team can be contacted at
core@php.net.
```

After the release of PHP 3.0, usage really started to take off. Version 4.0 was prompted by a number of developers who were interested in making some fundamental changes to the architecture of PHP. These changes included abstracting the layer between the language and the web server, adding a thread-safety mechanism, and adding a more advanced, two-stage parse/execute tag-parsing system. This new parser, primarily written by Zeev and Andi, was named the Zend engine. After a lot of work by a lot of developers, PHP 4.0 was released on May 22, 2000.

As this book goes to press, PHP version 7.3 has been released for some time. There have already been a few minor “dot” releases, and the stability of this current version is quite high. As you will see in this book, there have been some major advances made in this version of PHP, primarily in code processing on the server side. Many other minor changes, function additions, and feature enhancements have also been incorporated.

## The Widespread Use of PHP

Figure 1-1 shows the usage of PHP as compiled by W3Techs as of March 2019. The most interesting piece of data here is that 79% of all the surveyed websites use it, and yet version 5.0 is still the most widely used. If you look at the methodology used in the W3Techs surveys, you will see that they select the top 10 million sites (based on traffic; website popularity) in the world. As is evident, PHP has a very broad adoption indeed!

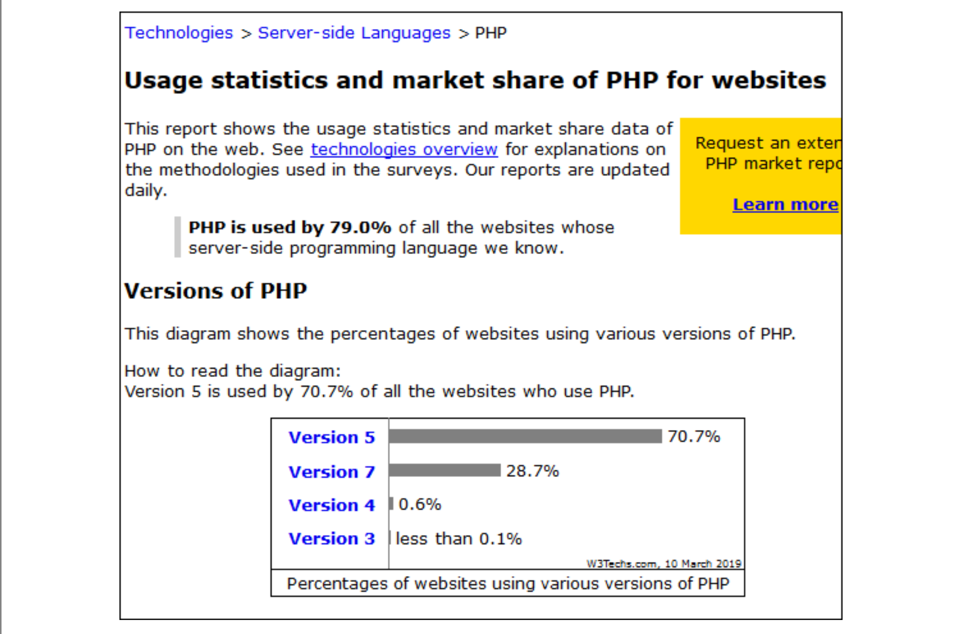



*Figure 1-1. PHP usage as of March 2019*

# Installing PHP

As mentioned, PHP is available for many operating systems and platforms. Therefore, you are encouraged to consult the PHP documentation to find the environment that most closely fits the one you will be using and follow the appropriate setup instructions.

From time to time, you may also want to change the way PHP is configured. To do that, you will have to change the PHP configuration file and restart your web (Apache) server. Each time you make a change to PHP's environment, you will have to restart the web (Apache) server in order for those changes to take effect.

PHP's configuration settings are usually maintained in a file called *php.ini*. The settings in this file control the behavior of PHP features, such as session handling and

form processing. Later chapters refer to some of the *php.ini* options, but in general the code in this book does not require a customized configuration. See the PHP documentation for more information on configuring *php.ini*.

## A Walk Through PHP

PHP pages are generally HTML pages with PHP commands embedded in them. This is in contrast to many other dynamic web page solutions, which are scripts that generate HTML. The web server processes the PHP commands and sends their output (and any HTML from the file) to the browser. Example 1-1 shows a complete PHP page.

*Example 1-1. hello_world.php*

```
<html>
 <head>
 <title>Look Out World</title>
 </head>

 <body>
 <?php echo "Hello, world!"; ?>
 </body>
</html>
```

Save the contents of Example 1-1 to a file, *hello_world.php*, and point your browser to it. The results appear in Figure 1-2.

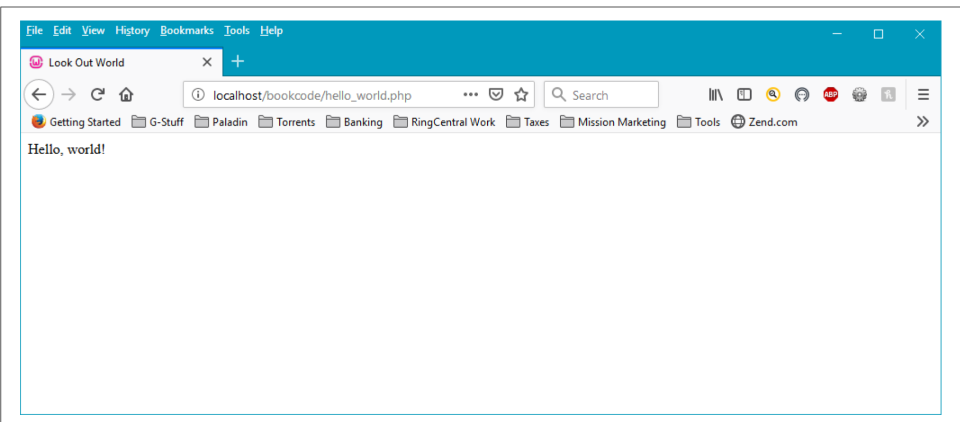



*Figure 1-2. Output of hello_world.php*

The PHP `echo` command produces output (the string "Hello, world!" in this case) inserted into the HTML file. In this example, the PHP code is placed between the

<?php and ?> tags. There are other ways to tag your PHP code—see Chapter 2 for a full description.

## Configuration Page

The PHP function `phpinfo()` creates an HTML page full of information on how PHP was installed and is currently configured. You can use it to see whether you have particular extensions installed, or whether the *php.ini* file has been customized. Example 1-2 is a complete page that displays the `phpinfo()` page.

*Example 1-2. Using phpinfo()*

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

Figure 1-3 shows the first part of the output of Example 1-2.

**PHP Version 7.4.0**

php

| | |
|---|---|
| System | Windows NT TOWERCASE 10.0 build 18362 (Windows 10) AMD64 |
| Build Date | Nov 27 2019 10:07:05 |
| Compiler | Visual C++ 2017 |
| Architecture | x64 |
| Configure Command | cscript /nologo 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 | C:\WINDOWS |
| Loaded Configuration File | D:\wamp64\bin\apache\apache2.4.41\bin\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 |
| Zend Multibyte Support | provided by mbstring |
| IPv6 Support | enabled |
| DTrace Support | disabled |
| Registered PHP Streams | php, file, glob, data, http, ftp, zip, compress.zlib, compress.bzip2, https, ftps, phar |
| Registered Stream Socket Transports | tcp, udp, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2, tlsv1.3 |
| Registered Stream Filters | convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk, zlib.*, bzip2.* |

This program makes use of the Zend Scripting Language Engine:
Zend Engine v3.4.0, Copyright (c) Zend Technologies
with Zend OPcache v7.4.0, Copyright (c), by Zend Technologies
with Xdebug v2.8.0, Copyright (c) 2002-2019, by Derick Rethans

zend engine

*Figure 1-3. Partial output of phpinfo()*

## Forms

Example 1-3 creates and processes a form. When the user submits the form, the information typed into the name field is sent back to this page via the `$_SERVER['PHP_SELF']` form action. The PHP code tests for a name field and displays a greeting if it finds one.

*Example 1-3. Processing a form (form.php)*

```
<html>
 <head>
 <title>Personalized Greeting Form</title>
 </head>

 <body>
 <?php if(!empty($_POST['name'])) {
 echo "Greetings, {$_POST['name']}, and welcome.";
 } ?>

 <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
 Enter your name: <input type="text" name="name" />
 <input type="submit" />
 </form>
 </body>
</html>
```

The form and the message are shown in Figure 1-4.

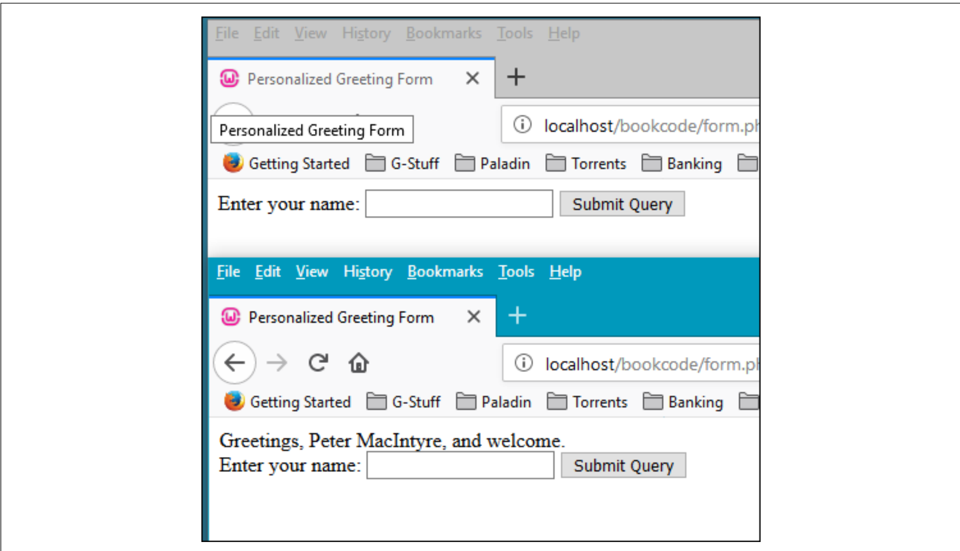



*Figure 1-4. Form and greeting page*

PHP programs access form values primarily through the `$_POST` and `$_GET` array variables. Chapter 8 discusses forms and form processing in more detail.

## Databases

PHP supports all the popular database systems, including MySQL, PostgreSQL, Oracle, Sybase, SQLite, and ODBC-compliant databases. Figure 1-5 shows part of a MySQL database query run through a PHP script, displaying the results of a book search on a book review site. It lists the book title, the year the book was published, and the book's ISBN.

| These Books are currently available | | |
|---|---|---|
| Title | Year Published | ISBN |
| Executive Orders | 1996 | 0-425-15863-2 |
| Forward the Foundation | 1993 | 0-553-56507-9 |
| Foundation | 1951 | 0-553-80371-9 |
| Foundation and Empire | 1952 | 0-553-29337-0 |
| Foundation's Edge | 1982 | 0-553-29338-9 |
| I, Robot | 1950 | 0-553-29438-5 |
| Isaac Asimov: Gold | 1995 | 0-06-055652-8 |
| Rainbow Six | 1998 | 0-425-17034-9 |
| Roots | 1974 | 0-440-17464-3 |
| Second Foundation | 1953 | 0-553-29336-2 |
| Teeth of the Tiger | 2003 | 0-399-15079-X |
| The Best of Isaac Asimov | 1973 | 0-449-20829-X |
| The Hobbit | 1937 | 0-261-10221-4 |
| The Return of The King | 1955 | 0-261-10237-0 |
| The Sum of All Fears | 1991 | 0-425-13354-0 |
| The Two Towers | 1954 | 0-261-10236-2 |

*Figure 1-5. A MySQL book list query run through a PHP script*

The code in Example 1-4 connects to the database, issues a query to retrieve all available books (with the `WHERE` clause), and produces a table as output for all returned results through a `while` loop.

The SQL code for this sample database is in the provided file *library.sql*. You can drop this code into MySQL after you create the library database and have the sample database at your disposal for testing out the following code sample as well as the related samples in Chapter 9.

*Example 1-4. Querying the books database (booklist.php)*

```
<?php

$db = new mysqli("localhost", "petermac", "password", "library");

// make sure the above credentials are correct for your environment
if ($db->connect_error) {
 die("Connect Error ({$db->connect_errno}) {$db->connect_error}");
}

$sql = "SELECT * FROM books WHERE available = 1 ORDER BY title";
$result = $db->query($sql);

?>
<html>
<body>

<table cellSpacing="2" cellPadding="6" align="center" border="1">
 <tr>
 <td colspan="4">
 <h3 align="center">These Books are currently available</h3>
 </td>
 </tr>

 <tr>
 <td align="center">Title</td>
 <td align="center">Year Published</td>
 <td align="center">ISBN</td>
 </tr>
 <?php while ($row = $result->fetch_assoc()) { ?>
 <tr>
 <td><?php echo stripslashes($row['title']); ?></td>
 <td align="center"><?php echo $row['pub_year']; ?></td>
 <td><?php echo $row['ISBN']; ?></td>
 </tr>
 <?php } ?>
</table>

</body>
</html>
```

Database-provided dynamic content drives the news, blog, and ecommerce sites at the heart of the web. More details on accessing databases from PHP are given in Chapter 9.

## Graphics

With PHP, you can easily create and manipulate images using the GD extension. Example 1-5 provides a text entry field that lets the user specify the text for a button. It takes an empty button image file, and centers the text passed as the `GET` parameter `'message'` on it. The result is then sent back to the browser as a PNG image.

*Example 1-5. Dynamic buttons (graphic_example.php)*

```
<?php
if (isset($_GET['message'])) {
 // load font and image, calculate width of text
 $font = dirname(__FILE__) . '/fonts/blazed.ttf';
 $size = 12;
 $image = imagecreatefrompng("button.png");
 $tsize = imagettfbbox($size, 0, $font, $_GET['message']);

 // center
 $dx = abs($tsize[2] - $tsize[0]);
 $dy = abs($tsize[5] - $tsize[3]);
 $x = (imagesx($image) - $dx) / 2;
 $y = (imagesy($image) - $dy) / 2 + $dy;

 // draw text
 $black = imagecolorallocate($im,0,0,0);
 imagettftext($image, $size, 0, $x, $y, $black, $font, $_GET['message']);

 // return image
 header("Content-type: image/png");
 imagepng($image);

 exit;
} ?>
<html>
 <head>
 <title>Button Form</title>
 </head>

 <body>
 <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="GET">
 Enter message to appear on button:
 <input type="text" name="message" /><br />
 <input type="submit" value="Create Button" />
 </form>
 </body>
</html>
```

The form generated by Example 1-5 is shown in Figure 1-6. The button created is shown in Figure 1-7.

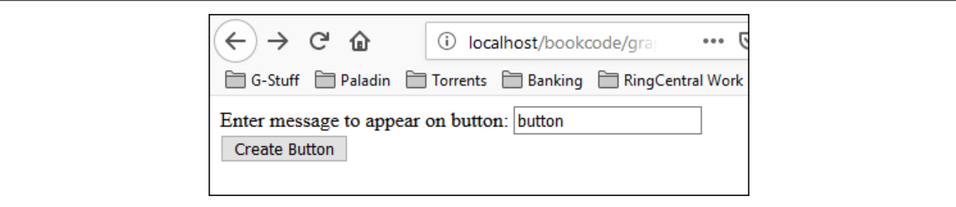



*Figure 1-6. Button creation form*

*Figure 1-7. Button created*

You can use GD to dynamically resize images, produce graphs, and much more. PHP also has several extensions to generate documents in Adobe's popular PDF format. Chapter 10 covers dynamic image generation in depth, while Chapter 11 provides instruction on how to create Adobe PDF files.

## What's Next

Now that you've had a taste of what is possible with PHP, you are ready to learn how to program in the language. We start with its basic structure, with special focus given to user-defined functions, string manipulation, and object-oriented programming. Then we move to specific application areas, such as the web, databases, graphics, XML, and security. We finish with quick references to the built-in functions and extensions. Master these chapters, and you will have mastered PHP!

CHAPTER 2

# Language Basics

This chapter provides a whirlwind tour of the core PHP language, covering such basic topics as data types, variables, operators, and flow-control statements. PHP is strongly influenced by other programming languages, such as Perl and C, so if you've had experience with those languages, PHP should be easy to pick up. If PHP is one of your first programming languages, don't panic. We start with the basic units of a PHP program and build up your knowledge from there.

## Lexical Structure

The lexical structure of a programming language is the set of basic rules that governs how you write programs in that language. It is the lowest-level syntax of the language and specifies such things as what variable names look like, what characters are used for comments, and how program statements are separated from each other.

### Case Sensitivity

The names of user-defined classes and functions, as well as built-in constructs and keywords (such as `echo`, `while`, `class`, etc.), are case-insensitive. Thus, these three lines are equivalent:

```
echo("hello, world");
ECHO("hello, world");
EcHo("hello, world");
```

Variables, on the other hand, are case-sensitive. That is, `$name`, `$NAME`, and `$NaME` are three different variables.

## Statements and Semicolons

A statement is a collection of PHP code that does something. It can be as simple as a variable assignment or as complicated as a loop with multiple exit points. Here is a small sample of PHP statements, including function calls, some variable data assignments, and an `if` statement:

```
echo "Hello, world";
myFunction(42, "O'Reilly");
$a = 1;
$name = "Elphaba";
$b = $a / 25.0;
if ($a == $b) {
 echo "Rhyme? And Reason?";
}
```

PHP uses semicolons to separate simple statements. A compound statement that uses curly braces to mark a block of code, such as a conditional test or loop, does not need a semicolon after a closing brace. Unlike in other languages, in PHP the semicolon before the closing brace is not optional:

```
if ($needed) {
 echo "We must have it!"; // semicolon required here
} // no semicolon required here after the brace
```

The semicolon, however, is optional before a closing PHP tag:

```
<?php
if ($a == $b) {
 echo "Rhyme? And Reason?";
}
echo "Hello, world" // no semicolon required before closing tag
?>
```

It's good programming practice to include optional semicolons, as they make it easier to add code later.

## Whitespace and Line Breaks

In general, whitespace doesn't matter in a PHP program. You can spread a statement across any number of lines, or lump a bunch of statements together on a single line. For example, this statement:

```
raisePrices($inventory, $inflation, $costOfLiving, $greed);
```

could just as well be written with more whitespace:

```
raisePrices (
 $inventory ,
 $inflation ,
 $costOfLiving ,
```

```
    $greed
   ) ;
```

or with less whitespace:

```
raisePrices($inventory,$inflation,$costOfLiving,$greed);
```

You can take advantage of this flexible formatting to make your code more readable (by lining up assignments, indenting, etc.). Some lazy programmers take advantage of this freeform formatting and create completely unreadable code—this is not recommended.

## Comments

Comments give information to people who read your code, but they are ignored by PHP at execution time. Even if you think you're the only person who will ever read your code, it's a good idea to include comments in your code—in retrospect, code you wrote months ago could easily look as though a stranger wrote it.

A good practice is to make your comments sparse enough not to get in the way of the code itself but plentiful enough that you can use the comments to tell what's happening. Don't comment obvious things, lest you bury the comments that describe tricky things. For example, this is worthless:

```
$x = 17; // store 17 into the variable $x
```

whereas the comments on this complex regular expression will help whoever maintains your code:

```
// convert &#nnn; entities into characters

$text = preg_replace('/&#([0-9])+;/', "chr('\\1')", $text);
```

PHP provides several ways to include comments within your code, all of which are borrowed from existing languages such as C, C++, and the Unix shell. In general, use C-style comments to comment *out* code, and C++-style comments to comment *on* code.

### Shell-style comments

When PHP encounters a hash mark character (#) within the code, everything from the hash mark to the end of the line or the end of the section of PHP code (whichever comes first) is considered a comment. This method of commenting is found in Unix shell scripting languages and is useful for annotating single lines of code or making short notes.

Because the hash mark is visible on the page, shell-style comments are sometimes used to mark off blocks of code:

```
#######################
## Cookie functions
#######################
```

Sometimes they're used before a line of code to identify what that code does, in which case they're usually indented to the same level as the code for which the comment is intended:

```
if ($doubleCheck) {
 # create an HTML form requesting that the user confirm the action
 echo confirmationForm();
}
```

Short comments on a single line of code are often put on the same line as the code:

```
$value = $p * exp($r * $t); # calculate compounded interest
```

When you're tightly mixing HTML and PHP code, it can be useful to have the closing PHP tag terminate the comment:

```
<?php $d = 4; # Set $d to 4. ?> Then another <?php echo $d; ?>
Then another 4
```

### C++ comments

When PHP encounters two slashes (//) within the code, everything from the slashes to the end of the line or the end of the section of code, whichever comes first, is considered a comment. This method of commenting is derived from C++. The result is the same as the shell comment style.

Here are the shell-style comment examples, rewritten to use C++ comments:

```
///////////////////////
// Cookie functions
///////////////////////

if ($doubleCheck) {
 // create an HTML form requesting that the user confirm the action
 echo confirmationForm();
}

$value = $p * exp($r * $t); // calculate compounded interest

<?php $d = 4; // Set $d to 4. ?> Then another <?php echo $d; ?>
Then another 4
```

### C comments

While shell-style and C++-style comments are useful for annotating code or making short notes, longer comments require a different style. Therefore, PHP supports

block comments whose syntax comes from the C programming language. When PHP encounters a slash followed by an asterisk (`/*`), everything after that, until it encounters an asterisk followed by a slash (`*/`), is considered a comment. This kind of comment, unlike those shown earlier, can span multiple lines.

Here's an example of a C-style multiline comment:

```
/* In this section, we take a bunch of variables and
 assign numbers to them. There is no real reason to
 do this, we're just having fun.
*/
$a = 1;
$b = 2;
$c = 3;
$d = 4;
```

Because C-style comments have specific start and end markers, you can tightly integrate them with code. This tends to make your code harder to read and is discouraged:

```
/* These comments can be mixed with code too,
see? */ $e = 5; /* This works just fine. */
```

C-style comments, unlike the other types, can continue past the end PHP tag markers. For example:

```
<?php
$l = 12;
$m = 13;
/* A comment begins here
?>
<p>Some stuff you want to be HTML.</p>
<?= $n = 14; ?>
*/
echo("l=$l m=$m n=$n\n");
?><p>Now <b>this</b> is regular HTML...</p>
l=12 m=13 n=
<p>Now <b>this</b> is regular HTML...</p>
```

You can indent comments as you like:

```
/* There are no
 special indenting or spacing
 rules that have to be followed, either.


 */
```

C-style comments can be useful for disabling sections of code. In the following example, we've disabled the second and third statements, as well as the inline comment, by including them in a block comment. To enable the code, all we have to do is remove the comment markers:

```
$f = 6;
/*
$g = 7; # This is a different style of comment
$h = 8;
*/
```

However, you have to be careful not to attempt to nest block comments:

```
$i = 9;
/*
$j = 10; /* This is a comment */
$k = 11;
Here is some comment text.
*/
```

In this case, PHP tries (and fails) to execute the (non)statement `Here is some comment text` and returns an error.

## Literals

A literal is a data value that appears directly in a program. The following are all literals in PHP:

```
2001
0xFE
1.4142
"Hello World"
'Hi'
true
null
```

## Identifiers

An identifier is simply a name. In PHP, identifiers are used to name variables, functions, constants, and classes. The first character of an identifier must be an ASCII letter (uppercase or lowercase), the underscore character (_), or any of the characters between ASCII 0x7F and ASCII 0xFF. After the initial character, these characters and the digits 0–9 are valid.

### Variable names

Variable names always begin with a dollar sign ($) and are case-sensitive. Here are some valid variable names:

```
$bill
$head_count
$MaximumForce
$I_HEART_PHP
$_underscore
$_int
```

Here are some illegal variable names:

```
$not valid
$|
$3wa
```

These variables are all different due to case sensitivity:

```
$hot_stuff $Hot_stuff $hot_Stuff $HOT_STUFF
```

### Function names

Function names are not case-sensitive (functions are discussed in more detail in Chapter 3). Here are some valid function names:

```
tally
list_all_users
deleteTclFiles
LOWERCASE_IS_FOR_WIMPS
_hide
```

These function names all refer to the same function:

```
howdy HoWdY HOWDY HOWdy howdy
```

### Class names

Class names follow the standard rules for PHP identifiers and are also not case-sensitive. Here are some valid class names:

```
Person
account
```

The class name `stdClass` is a reserved class name.

### Constants

A constant is an identifier for a value that will not be changed; scalar values (Boolean, integer, double, and string) and arrays can be constants. Once set, the value of a constant cannot change. Constants are referred to by their identifiers and are set using the `define()` function:

```
define('PUBLISHER', "O'Reilly Media");
echo PUBLISHER;
```

## Keywords

A keyword (or reserved word) is a word set aside by the language for its core functionality—you cannot give a function, class, or constant the same name as a keyword. Table 2-1 lists the keywords in PHP, which are case-insensitive.

*Table 2-1. PHP core language keywords*

| | | |
|---|---|---|
| `__CLASS__` | `echo` | `insteadof` |
| `__DIR__` | `else` | `interface` |
| `__FILE__` | `elseif` | `isset()` |
| `__FUNCTION__` | `empty()` | `list()` |
| `__LINE__` | `enddeclare` | `namespace` |
| `__METHOD__` | `endfor` | `new` |
| `__NAMESPACE__` | `endforeach` | `or` |
| `__TRAIT__` | `endif` | `print` |
| `__halt_compiler()` | `endswitch` | `private` |
| `abstract` | `endwhile` | `protected` |
| `and` | `eval()` | `public` |
| `array()` | `exit()` | `require` |
| `as` | `extends` | `require_once` |
| `break` | `final` | `return` |
| `callable` | `finally` | `static` |
| `case` | `for` | `switch` |
| `catch` | `foreach` | `throw` |
| `class` | `function` | `trait` |
| `clone` | `global` | `try` |
| `const` | `goto` | `unset()` |
| `continue` | `if` | `use` |
| `declare` | `implements` | `var` |
| `default` | `include` | `while` |
| `die()` | `include_once` | `xor` |
| `do` | `instanceof` | `yield` |
| | | `yield from` |

In addition, you cannot use an identifier that is the same as a built-in PHP function. For a complete list of these, see the Appendix.

# Data Types

PHP provides eight types of values, or data types. Four are scalar (single-value) types: integers, floating-point numbers, strings, and Booleans. Two are compound (collection) types: arrays and objects. The remaining two are special types: resource and NULL. Numbers, Booleans, resources, and NULL are discussed in full here, while strings, arrays, and objects are big enough topics that they get their own chapters (Chapters 4, 5, and 6, respectively).

## Integers

Integers are whole numbers, such as 1, 12, and 256. The range of acceptable values varies according to the details of your platform but typically extends from −2,147,483,648 to +2,147,483,647. Specifically, the range is equivalent to the range of

the long data type of your C compiler. Unfortunately, the C standard doesn't specify what range that long type should have, so on some systems you might see a different integer range.

Integer literals can be written in decimal, octal, binary, or hexadecimal. Decimal values are represented by a sequence of digits, without leading zeros. The sequence may begin with a plus (+) or minus (–) sign. If there is no sign, positive is assumed. Examples of decimal integers include the following:

```
1998
-641
+33
```

Octal numbers consist of a leading `0` and a sequence of digits from 0 to 7. Like decimal numbers, octal numbers can be prefixed with a plus or minus. Here are some example octal values and their equivalent decimal values:

```
0755 // decimal 493
+010 // decimal 8
```

Hexadecimal values begin with `0x`, followed by a sequence of digits (0–9) or letters (A–F). The letters can be upper- or lowercase but are usually written in capitals. As with decimal and octal values, you can include a sign in hexadecimal numbers:

```
0xFF // decimal 255
0x10 // decimal 16
-0xDAD1 // decimal -56017
```

Binary numbers begin with `0b`, followed by a sequence of digits (0 and 1). As with other values, you can include a sign in binary numbers:

```
0b01100000 // decimal 96
0b00000010 // decimal 2
-0b10 // decimal -2
```

If you try to store a variable that is too large to be stored as an integer or is not a whole number, it will automatically be turned into a floating-point number.

Use the `is_int()` function (or its `is_integer()` alias) to test whether a value is an integer:

```
if (is_int($x)) {
 // $x is an integer
}
```

## Floating-Point Numbers

Floating-point numbers (often referred to as "real" numbers) represent numeric values with decimal digits. Like integers, their limits depend on your machine's details. PHP floating-point numbers are equivalent to the range of the double data type of your C compiler. Usually, this allows numbers between 1.7E–308 and 1.7E+308 with

15 digits of accuracy. If you need more accuracy or a wider range of integer values, you can use the BC or GMP extensions.

PHP recognizes floating-point numbers written in two different formats. There's the one we all use every day:

```
3.14
0.017
-7.1
```

But PHP also recognizes numbers in scientific notation:

```
0.314E1 // 0.314*10^1, or 3.14
17.0E-3 // 17.0*10^(-3), or 0.017
```

Floating-point values are only approximate representations of numbers. For example, on many systems 3.5 is actually represented as 3.4999999999. This means you must take care to avoid writing code that assumes floating-point numbers are represented completely accurately, such as directly comparing two floating-point values using ==. The normal approach is to compare to several decimal places:

```
if (intval($a * 1000) == intval($b * 1000)) {
 // numbers equal to three decimal places
}
```

Use the `is_float()` function (or its `is_real()` alias) to test whether a value is a floating-point number:

```
if (is_float($x)) {
 // $x is a floating-point number
}
```

## Strings

Because strings are so common in web applications, PHP includes core-level support for creating and manipulating strings. A string is a sequence of characters of arbitrary length. String literals are delimited by either single or double quotes:

```
'big dog'
"fat hog"
```

Variables are expanded (interpolated) within double quotes, while within single quotes they are not:

```
$name = "Guido";
echo "Hi, $name <br/>";
echo 'Hi, $name';
Hi, Guido
Hi, $name
```

Double quotes also support a variety of string escapes, as listed in Table 2-2.

*Table 2-2. Escape sequences in double-quoted strings*

| Escape sequence | Character represented |
|---|---|
| `\"` | Double quotes |
| `\n` | Newline |
| `\r` | Carriage return |
| `\t` | Tab |
| `\\` | Backslash |
| `\$` | Dollar sign |
| `\{` | Left brace |
| `\}` | Right brace |
| `\[` | Left bracket |
| `\]` | Right bracket |
| `\0` through `\777` | ASCII character represented by octal value |
| `\x0` through `\xFF` | ASCII character represented by hex value |

A single-quoted string recognizes `\\` to get a literal backslash and `\'` to get a literal single quote:

```
$dosPath = 'C:\\WINDOWS\\SYSTEM';
$publisher = 'Tim O\'Reilly';
echo "$dosPath $publisher";
C:\WINDOWS\SYSTEM Tim O'Reilly
```

To test whether two strings are equal, use the `==` (double equals) comparison operator:

```
if ($a == $b) {
 echo "a and b are equal";
}
```

Use the `is_string()` function to test whether a value is a string:

```
if (is_string($x)) {
 // $x is a string
}
```

PHP provides operators and functions to compare, disassemble, assemble, search, replace, and trim strings, as well as a host of specialized string functions for working with HTTP, HTML, and SQL encodings. Because there are so many string-manipulation functions, we've devoted a whole chapter (Chapter 4) to covering all the details.

## Booleans

A Boolean value represents a *truth value*—it says whether something is true or not. Like most programming languages, PHP defines some values as true and others as false. Truthfulness and falseness determine the outcome of conditional code such as:

```
if ($alive) { ... }
```

In PHP, the following values all evaluate to `false`:

- The keyword `false`
- The integer `0`
- The floating-point value `0.0`
- The empty string (`""`) and the string `"0"`
- An array with zero elements
- The `NULL` value

A value that is not false is true, including all resource values (which are described later in the section "Resources").

PHP provides `true` and `false` keywords for clarity:

```
$x = 5; // $x has a true value
$x = true; // clearer way to write it
$y = ""; // $y has a false value
$y = false; // clearer way to write it
```

Use the `is_bool()` function to test whether a value is a Boolean:

```
if (is_bool($x)) {
 // $x is a Boolean
}
```

## Arrays

An array holds a group of values, which you can identify by position (a number, with zero being the first position) or some identifying name (a string), called an *associative index*:

```
$person[0] = "Edison";
$person[1] = "Wankel";
$person[2] = "Crapper";

$creator['Light bulb'] = "Edison";
$creator['Rotary Engine'] = "Wankel";
$creator['Toilet'] = "Crapper";
```

The `array()` construct creates an array. Here are two examples:

```
$person = array("Edison", "Wankel", "Crapper");
$creator = array('Light bulb' => "Edison",
 'Rotary Engine' => "Wankel",
 'Toilet' => "Crapper");
```

There are several ways to loop through arrays, but the most common is a `foreach` loop:

```
foreach ($person as $name) {
 echo "Hello, {$name}<br/>";
}

foreach ($creator as $invention => $inventor) {
 echo "{$inventor} invented the {$invention}<br/>";
}
Hello, Edison
Hello, Wankel
Hello, Crapper
Edison created the Light bulb
Wankel created the Rotary Engine
Crapper created the Toilet
```

You can sort the elements of an array with the various sort functions:

```
 sort($person);
// $person is now array("Crapper", "Edison", "Wankel")

 asort($creator);
// $creator is now array('Toilet' => "Crapper",
// 'Light bulb' => "Edison",
// 'Rotary Engine' => "Wankel");
```

Use the `is_array()` function to test whether a value is an array:

```
if (is_array($x)) {
 // $x is an array
}
```

There are functions for returning the number of items in the array, fetching every value in the array, and much more. Arrays are covered in depth in Chapter 5.

## Objects

PHP also supports *object-oriented programming* (OOP). OOP promotes clean, modular design; simplifies debugging and maintenance; and assists with code reuse. Classes are the building blocks of object-oriented design. A class is a definition of a structure that contains properties (variables) and methods (functions). Classes are defined with the `class` keyword:

```
class Person
{
 public $name = '';

 function name ($newname = NULL)
 {
 if (!is_null($newname)) {
 $this->name = $newname;
```

```
  }

  return $this->name;
  }
}
```

Once a class is defined, any number of objects can be made from it with the `new` keyword, and the object's properties and methods can be accessed with the `->` construct:

```
$ed = new Person;
$ed->name('Edison');
echo "Hello, {$ed->name} <br/>";
$tc = new Person;
$tc->name('Crapper');
echo "Look out below {$tc->name} <br/>";
Hello, Edison
Look out below Crapper
```

Use the `is_object()` function to test whether a value is an object:

```
if (is_object($x)) {
 // $x is an object
}
```

Chapter 6 describes classes and objects in much more detail, including inheritance, encapsulation, and introspection.

## Resources

Many modules provide several functions for dealing with the outside world. For example, every database extension has at least a function to connect to the database, a function to query the database, and a function to close the connection to the database. Because you can have multiple database connections open at once, the connect function gives you something by which to identify that unique connection when you call the query and close functions: a resource (or a *handle*).

Each active resource has a unique identifier. Each identifier is a numerical index into an internal PHP lookup table that holds information about all the active resources. PHP maintains information about each resource in this table, including the number of references to (or uses of) the resource throughout the code. When the last reference to a resource value goes away, the extension that created the resource is called to perform tasks such as freeing any memory or closing any connection for that resource:

```
$res = database_connect(); // fictitious database connect function
database_query($res);

$res = "boo";
// database connection automatically closed because $res is redefined
```

The benefit of this automatic cleanup is best seen within functions, when the resource is assigned to a local variable. When the function ends, the variable's value is reclaimed by PHP:

```
function search() {
 $res = database_connect();
 database_query($res);
}
```

When there are no more references to the resource, it's automatically shut down.

That said, most extensions provide a specific shutdown or close function, and it's considered good style to call that function explicitly when needed rather than to rely on variable scoping to trigger resource cleanup.

Use the `is_resource()` function to test whether a value is a resource:

```
if (is_resource($x)) {
 // $x is a resource
}
```

## Callbacks

Callbacks are functions or object methods used by some functions, such as `call_user_func()`. Callbacks can also be created by the `create_function()` method and through closures (described in Chapter 3):

```
$callback = function()
{
 echo "callback achieved";
};

call_user_func($callback);
```

## NULL

There's only one value of the NULL data type. That value is available through the case-insensitive keyword `NULL`. The `NULL` value represents a variable that has no value (similar to Perl's `undef` or Python's `None`):

```
$aleph = "beta";
$aleph = null; // variable's value is gone
$aleph = Null; // same
$aleph = NULL; // same
```

Use the `is_null()` function to test whether a value is `NULL`—for instance, to see whether a variable has a value:

```
if (is_null($x)) {
 // $x is NULL
}
```

## Variables

Variables in PHP are identifiers prefixed with a dollar sign (`$`). For example:

```
$name
$Age
$_debugging
$MAXIMUM_IMPACT
```

A variable may hold a value of any type. There is no compile-time or runtime type checking on variables. You can replace a variable's value with another of a different type:

```
$what = "Fred";
$what = 35;
$what = array("Fred", 35, "Wilma");
```

There is no explicit syntax for declaring variables in PHP. The first time the value of a variable is set, the variable is created in memory. In other words, setting a value to a variable also functions as a declaration. For example, this is a valid complete PHP program:

```
$day = 60 * 60 * 24;
echo "There are {$day} seconds in a day.";
There are 86400 seconds in a day.
```

A variable whose value has not been set behaves like the `NULL` value:

```
if ($uninitializedVariable === NULL) {
 echo "Yes!";
}
Yes!
```

### Variable Variables

You can reference the value of a variable whose name is stored in another variable by prefacing the variable reference with an additional dollar sign (`$`). For example:

```
$foo = "bar";
$$foo = "baz";
```

After the second statement executes, the variable `$bar` has the value `"baz"`.

### Variable References

In PHP, references are how you create variable aliases or pointers. To make `$black` an alias for the variable `$white`, use:

```
$black =& $white;
```

The old value of `$black`, if any, is lost. Instead, `$black` is now another name for the value that is stored in `$white`:

```
$bigLongVariableName = "PHP";
$short =& $bigLongVariableName;
$bigLongVariableName .= " rocks!";
print "\$short is $short <br/>";
print "Long is $bigLongVariableName";
$short is PHP rocks!
Long is PHP rocks!

$short = "Programming $short";
print "\$short is $short <br/>";
print "Long is $bigLongVariableName";
$short is Programming PHP rocks!
Long is Programming PHP rocks!
```

After the assignment, the two variables are alternate names for the same value. Unsetting a variable that is aliased does not affect other names for that variable's value, however:

```
$white = "snow";
$black =& $white;
unset($white);
print $black;
snow
```

Functions can return values by reference (for example, to avoid copying large strings or arrays, as discussed in Chapter 3):

```
function &retRef() // note the &
{
 $var = "PHP";

 return $var;
}

$v =& retRef(); // note the &
```

## Variable Scope

The *scope* of a variable, which is controlled by the location of the variable's declaration, determines those parts of the program that can access it. There are four types of variable scope in PHP: local, global, static, and function parameters.

### Local scope

A variable declared in a function is local to that function. That is, it is visible only to code in that function (excepting nested function definitions); it is not accessible outside the function. In addition, by default, variables defined outside a function (called *global* variables) are not accessible inside the function. For example, here's a function that updates a local variable instead of a global variable:

```
function updateCounter()
{
 $counter++;
}

$counter = 10;
updateCounter();

echo $counter;
10
```

The `$counter` inside the function is local to that function because we haven't said otherwise. The function increments its private `$counter` variable, which is destroyed when the subroutine ends. The global `$counter` remains set at 10.

Only functions can provide local scope. Unlike in other languages, in PHP you can't create a variable whose scope is a loop, conditional branch, or other type of block.

### Global scope

Variables declared outside a function are global. That is, they can be accessed from any part of the program. However, by default, they are not available inside functions. To allow a function to access a global variable, you can use the `global` keyword inside the function to declare the variable within the function. Here's how we can rewrite the `updateCounter()` function to allow it to access the global `$counter` variable:

```
function updateCounter()
{
 global $counter;
 $counter++;
}

$counter = 10;
updateCounter();
echo $counter;
11
```

A more cumbersome way to update the global variable is to use PHP's `$GLOBALS` array instead of accessing the variable directly:

```
function updateCounter()
{
 $GLOBALS['counter']++;
}

$counter = 10;
updateCounter();
echo $counter;
11
```

### Static variables

A static variable retains its value between calls to a function but is visible only within that function. You declare a variable static with the `static` keyword. For example:

```
function updateCounter()
{
 static $counter = 0;
 $counter++;

 echo "Static counter is now {$counter}<br/>";
}

$counter = 10;
updateCounter();
updateCounter();

echo "Global counter is {$counter}";
Static counter is now 1
Static counter is now 2
Global counter is 10
```

### Function parameters

As we'll discuss in more detail in Chapter 3, a function definition can have named parameters:

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

greet("Janet");
Hello, Janet
```

Function parameters are local, meaning that they are available only inside their functions. In this case, `$name` is inaccessible from outside `greet()`.

## Garbage Collection

PHP uses *reference counting* and *copy-on-write* to manage memory. Copy-on-write ensures that memory isn't wasted when you copy values between variables, and reference counting ensures that memory is returned to the operating system when it is no longer needed.

To understand memory management in PHP, you must first understand the idea of a *symbol table*. There are two parts to a variable—its name (e.g., `$name`), and its value (e.g., `"Fred"`). A symbol table is an array that maps variable names to the positions of their values in memory.

When you copy a value from one variable to another, PHP doesn't get more memory for a copy of the value. Instead, it updates the symbol table to indicate that "both of these variables are names for the same chunk of memory." So the following code doesn't actually create a new array:

```
$worker = array("Fred", 35, "Wilma");
$other = $worker; // array isn't duplicated in memory
```

If you subsequently modify either copy, PHP allocates the required memory and makes the copy:

```
$worker[1] = 36; // array is copied in memory, value changed
```

By delaying the allocation and copying, PHP saves time and memory in a lot of situations. This is copy-on-write.

Each value pointed to by a symbol table has a *reference count*, a number that represents the number of ways there are to get to that piece of memory. After the initial assignment of the array to `$worker` and `$worker` to `$other`, the array pointed to by the symbol table entries for `$worker` and `$other` has a reference count of 2.[1] In other words, that memory can be reached two ways: through `$worker` or `$other`. But after `$worker[1]` is changed, PHP creates a new array for `$worker`, and the reference count of each array is only 1.

When a variable goes out of scope at the end of a function, such as function parameters and local variables, the reference count of its value is decreased by one. When a variable is assigned a value in a different area of memory, the reference count of the old value is decreased by one. When the reference count of a value reaches 0, its memory is released. This is reference counting.

Reference counting is the preferred way to manage memory. Keep variables local to functions, pass in values that the functions need to work on, and let reference counting take care of the memory management. If you do insist on trying to get a little more information or control over freeing a variable's value, use the `isset()` and `unset()` functions.

To see if a variable has been set to something—even the empty string—use `isset()`:

```
$s1 = isset($name); // $s1 is false
$name = "Fred";
$s2 = isset($name); // $s2 is true
```

Use `unset()` to remove a variable's value:

1 It is actually 3 if you are looking at the reference count from the C API, but for the purposes of this explanation and from a user-space perspective, it is easier to think of it as 2.

```
$name = "Fred";
unset($name); // $name is NULL
```

## Expressions and Operators

An *expression* is a bit of PHP code that can be evaluated to produce a value. The simplest expressions are literal values and variables. A literal value evaluates to itself, while a variable evaluates to the value stored in the variable. More complex expressions can be formed using simple expressions and operators.

An *operator* takes some values (the operands) and does something (e.g., adds them together). Operators are sometimes written as punctuation symbols—for instance, the + and - familiar to us from math. Some operators modify their operands, while most do not.

Table 2-3 summarizes the operators in PHP, many of which were borrowed from C and Perl. The column labeled "P" gives the operator's precedence; the operators are listed in precedence order, from highest to lowest. The column labeled "A" gives the operator's associativity, which can be L (left-to-right), R (right-to-left), or N (nonassociative).

*Table 2-3. PHP operators*

| P | A | Operator | Operation |
|---|---|---|---|
| 24 | N | `clone`, `new` | Create new object |
| 23 | L | `[` | Array subscript |
| 22 | R | `**` | Exponentiation |
| 21 | R | `~` | Bitwise NOT |
| | R | `++` | Increment |
| | R | `--` | Decrement |
| | R | `(int)`, `(bool)`, `(float)`, `(string)`, `(array)`, `(object)`, `(unset)` | Cast |
| | R | `@` | Inhibit errors |
| 20 | N | `instanceof` | Type testing |
| 19 | R | `!` | Logical NOT |
| 18 | L | `*` | Multiplication |
| | L | `/` | Division |
| | L | `%` | Modulus |
| 17 | L | `+` | Addition |
| | L | `-` | Subtraction |
| | L | `.` | String concatenation |
| 16 | L | `<<` | Bitwise shift left |
| | L | `>>` | Bitwise shift right |

<table>
<tr><th>P</th><th>A</th><th>Operator</th><th>Operation</th></tr>
<tr><td>15</td><td>N</td><td><, <=</td><td>Less than, less than or equal</td></tr>
<tr><td></td><td>N</td><td>>, >=</td><td>Greater than, greater than or equal</td></tr>
<tr><td>14</td><td>N</td><td>==</td><td>Value equality</td></tr>
<tr><td></td><td>N</td><td>!=, <></td><td>Inequality</td></tr>
<tr><td></td><td>N</td><td>===</td><td>Type and value equality</td></tr>
<tr><td></td><td>N</td><td>!==</td><td>Type and value inequality</td></tr>
<tr><td></td><td>N</td><td><=></td><td>Returns an integer based on a comparison of two operands: 0 when left and right are equal, -1 when left is less than right, and 1 when left is greater than right.</td></tr>
<tr><td>13</td><td>L</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>12</td><td>L</td><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>11</td><td>L</td><td>|</td><td>Bitwise OR</td></tr>
<tr><td>10</td><td>L</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>9</td><td>L</td><td>||</td><td>Logical OR</td></tr>
<tr><td>8</td><td>R</td><td>??</td><td>Comparison</td></tr>
<tr><td>7</td><td>L</td><td>?:</td><td>Conditional operator</td></tr>
<tr><td>6</td><td>R</td><td>=</td><td>Assignment</td></tr>
<tr><td></td><td>R</td><td>+=, -=, *=, /=, .=, %=, &=, |=, ^=, ~=, <<=, >>=</td><td>Assignment with operation</td></tr>
<tr><td>5</td><td></td><td>yield from</td><td>Yield from</td></tr>
<tr><td>4</td><td></td><td>yield</td><td>Yield</td></tr>
<tr><td>3</td><td>L</td><td>and</td><td>Logical AND</td></tr>
<tr><td>2</td><td>L</td><td>xor</td><td>Logical XOR</td></tr>
<tr><td>1</td><td>L</td><td>or</td><td>Logical OR</td></tr>
</table>

## Number of Operands

Most operators in PHP are binary operators; they combine two operands (or expressions) into a single, more complex expression. PHP also supports a number of unary operators, which convert a single expression into a more complex expression. Finally, PHP supports a few ternary operators that combine numerous expressions into a single expression.

## Operator Precedence

The order in which operators in an expression are evaluated depends on their relative precedence. For example, you might write:

```
2 + 4 * 3
```

As you can see in Table 2-3, the addition and multiplication operators have different precedence, with multiplication higher than addition. So the multiplication happens

before the addition, giving `2 + 12`, or `14`, as the answer. If the precedence of addition and multiplication were reversed, `6 * 3`, or `18`, would be the answer.

To force a particular order, you can group operands with the appropriate operator in parentheses. In our previous example, to get the value `18`, you can use this expression:

```
(2 + 4) * 3
```

It is possible to write all complex expressions (expressions containing more than a single operator) simply by putting the operands and operators in the appropriate order so that their relative precedence yields the answer you want. Most programmers, however, write the operators in the order that they feel makes the most sense to them, and add parentheses to ensure it makes sense to PHP as well. Getting precedence wrong leads to code like:

```
$x + 2 / $y >= 4 ? $z : $x << $z
```

This code is hard to read and is almost definitely not doing what the programmer expected it to do.

One way many programmers deal with the complex precedence rules in programming languages is to reduce precedence down to two rules:

- Multiplication and division have higher precedence than addition and subtraction.
- Use parentheses for anything else.

## Operator Associativity

Associativity defines the order in which operators with the same order of precedence are evaluated. For example, look at:

```
2 / 2 * 2
```

The division and multiplication operators have the same precedence, but the result of the expression depends on which operation we do first:

```
2 / (2 * 2) // 0.5
(2 / 2) * 2 // 2
```

The division and multiplication operators are *left-associative*; this means that in cases of ambiguity, the operators are evaluated from left to right. In this example, the correct result is 2.

## Implicit Casting

Many operators have expectations of their operands—for instance, binary math operators typically require both operands to be of the same type. PHP's variables can store

integers, floating-point numbers, strings, and more, and to keep as much of the type details away from the programmer as possible, PHP converts values from one type to another as necessary.

The conversion of a value from one type to another is called *casting*. This kind of implicit casting is called *type juggling* in PHP. The rules for the type juggling done by arithmetic operators are shown in Table 2-4.

*Table 2-4. Implicit casting rules for binary arithmetic operations*

| Type of first operand | Type of second operand | Conversion performed |
|---|---|---|
| Integer | Floating point | The integer is converted to a floating-point number. |
| Integer | String | The string is converted to a number; if the value after conversion is a floating-point number, the integer is converted to a floating-point number. |
| Floating point | String | The string is converted to a floating-point number. |

Some other operators have different expectations of their operands, and thus have different rules. For example, the string concatenation operator converts both operands to strings before concatenating them:

```
3 . 2.74 // gives the string 32.74
```

You can use a string anywhere PHP expects a number. The string is presumed to start with an integer or floating-point number. If no number is found at the start of the string, the numeric value of that string is 0. If the string contains a period (`.`) or upper- or lowercase `e`, evaluating it numerically produces a floating-point number. For example:

```
"9 Lives" - 1; // 8 (int)
"3.14 Pies" * 2; // 6.28 (float)
"9. Lives" - 1; // 8 (float / double)
"1E3 Points of Light" + 1; // 1001 (float)
```

## Arithmetic Operators

The arithmetic operators are operators you'll recognize from everyday use. Most of the arithmetic operators are binary; however, the arithmetic negation and arithmetic assertion operators are unary. These operators require numeric values, and nonnumeric values are converted into numeric values by the rules described in the section "Casting Operators". The arithmetic operators are:

*Addition (+)*
: The result of the addition operator is the sum of the two operands.

*Subtraction (–)*
: The result of the subtraction operator is the difference between the two operands—that is, the value of the second operand subtracted from the first.

*Multiplication (*)*
: The result of the multiplication operator is the product of the two operands. For example, `3 * 4` is `12`.

*Division (/)*
: The result of the division operator is the quotient of the two operands. Dividing two integers can give an integer (e.g., `4 / 2`) or a floating-point result (e.g., `1 / 2`).

*Modulus (%)*
: The modulus operator converts both operands to integers and returns the remainder of the division of the first operand by the second operand. For example, `10 % 6` gives a remainder of `4`.

*Arithmetic negation (–)*
: The arithmetic negation operator returns the operand multiplied by –1, effectively changing its sign. For example, `-(3 - 4)` evaluates to `1`. Arithmetic negation is different from the subtraction operator, even though they both are written as a minus sign. Arithmetic negation is always unary and before the operand. Subtraction is binary and between its operands.

*Arithmetic assertion (+)*
: The arithmetic assertion operator returns the operand multiplied by +1, which has no effect. It is used only as a visual cue to indicate the sign of a value. For example, `+(3 - 4)` evaluates to `-1`, just as `(3 - 4)` does.

*Exponentiation (**)*
: The exponentiation operator returns the result of raising `$var1` to the power of `$var2`.

```
$var1 = 5;
$var2 = 3;
echo $var1 ** $var2; // outputs 125
```

## String Concatenation Operator

Manipulating strings is such a core part of PHP applications that PHP has a separate string concatenation operator (`.`). The concatenation operator appends the righthand operand to the lefthand operand and returns the resulting string. Operands are first converted to strings, if necessary. For example:

```
$n = 5;
$s = 'There were ' . $n . ' ducks.';
// $s is 'There were 5 ducks'
```

The concatenation operator is highly efficient because so much of PHP boils down to string concatenation.

## Auto-Increment and Auto-Decrement Operators

In programming, one of the most common operations is to increase or decrease the value of a variable by one. The unary auto-increment (++) and auto-decrement (−−) operators provide shortcuts for these common operations. These operators are unique in that they work only on variables; the operators change their operands' values and return a value.

There are two ways to use auto-increment or auto-decrement in expressions. If you put the operator in front of the operand, it returns the new value of the operand (incremented or decremented). If you put the operator after the operand, it returns the original value of the operand (before the increment or decrement). Table 2-5 lists the different operations.

*Table 2-5. Auto-increment and auto-decrement operations*

| Operator | Name | Value returned | Effect on `$var` |
|---|---|---|---|
| `$var++` | Post-increment | `$var` | Incremented |
| `++$var` | Pre-increment | `$var + 1` | Incremented |
| `$var--` | Post-decrement | `$var` | Decremented |
| `--$var` | Pre-decrement | `$var - 1` | Decremented |

These operators can be applied to strings as well as numbers. Incrementing an alphabetic character turns it into the next letter in the alphabet. As illustrated in Table 2-6, incrementing `"z"` or `"Z"` wraps it back to `"a"` or `"A"` and increments the previous character by one (or inserts a new `"a"` or `"A"` if at the first character of the string), as though the characters were in a base-26 number system.

*Table 2-6. Auto-increment with letters*

| Incrementing this | Gives this |
|---|---|
| `"a"` | `"b"` |
| `"z"` | `"aa"` |
| `"spaz"` | `"spba"` |
| `"K9"` | `"L0"` |
| `"42"` | `"43"` |

## Comparison Operators

As their name suggests, comparison operators compare operands. The result is always either `true`, if the comparison is truthful, and `false` otherwise.

Operands to the comparison operators can be both numeric, both string, or one numeric and one string. The operators check for truthfulness in slightly different ways based on the types and values of the operands, either using strictly numeric comparisons or using lexicographic (textual) comparisons. Table 2-7 outlines when each type of check is used.

*Table 2-7. Type of comparison performed by the comparison operators*

| First operand | Second operand | Comparison |
| --- | --- | --- |
| Number | Number | Numeric |
| String that is entirely numeric | String that is entirely numeric | Numeric |
| String that is entirely numeric | Number | Numeric |
| String that is entirely numeric | String that is not entirely numeric | Lexicographic |
| String that is not entirely numeric | Number | Numeric |
| String that is not entirely numeric | String that is not entirely numeric | Lexicographic |

One important thing to note is that two numeric strings are compared as if they were numbers. If you have two strings that consist entirely of numeric characters and you need to compare them lexicographically, use the `strcmp()` function.

The comparison operators are:

*Equality (==)*
: If both operands are equal, this operator returns `true`; otherwise, it returns `false`.

*Identity (===)*
: If both operands are equal and are of the same type, this operator returns `true`; otherwise, it returns `false`. Note that this operator does *not* do implicit type casting. This operator is useful when you don't know if the values you're comparing are of the same type. Simple comparison may involve value conversion. For instance, the strings `"0.0"` and `"0"` are not equal. The == operator says they are, but === says they are not.

*Inequality (`!=` or <>)*
: If the operands are not equal, this operator returns `true`; otherwise, it returns `false`.

*Not identical (`!==`)*
: If the operands are not equal, or they are not of the same type, this operator returns `true`; otherwise, it returns `false`.

*Greater than (`>`)*
: If the lefthand operand is greater than the righthand operand, this operator returns `true`; otherwise, it returns `false`.

*Greater than or equal to (`>=`)*
: If the lefthand operand is greater than or equal to the righthand operand, this operator returns `true`; otherwise, it returns `false`.

*Less than (`<`)*
: If the lefthand operand is less than the righthand operand, this operator returns `true`; otherwise, it returns `false`.

*Less than or equal to (`<=`)*
: If the lefthand operand is less than or equal to the righthand operand, this operator returns `true`; otherwise, it returns `false`.

*Spaceship (`<=>`), aka "Darth Vader's TIE Fighter"*
: When the lefthand and righthand operands are equal, this operator returns `0`; when the lefthand operand is less than the righthand, it returns `-1`; and when the lefthand operand is greater than the righthand, it returns `1`.

```
$var1 = 5;
$var2 = 65;

echo $var1 <=> $var2 ; // outputs -1
echo $var2 <=> $var1 ; // outputs 1
```

*Null coalescing operator (`??`)*
: This operator evaluates to the righthand operand if the lefthand operand is `NULL`; otherwise, it evaluates to the lefthand operand.

```
$var1 = null;
$var2 = 31;

echo $var1 ?? $var2 ; //outputs 31
```

## Bitwise Operators

The bitwise operators act on the binary representation of their operands. Each operand is first turned into a binary representation of the value, as described in the bitwise negation operator entry in the following list. All the bitwise operators work on numbers as well as strings, but they vary in their treatment of string operands of different lengths. The bitwise operators are:

*Bitwise negation (~)*

The bitwise negation operator changes 1s to 0s and 0s to 1s in the binary representations of the operands. Floating-point values are converted to integers before the operation takes place. If the operand is a string, the resulting value is a string the same length as the original, with each character in the string negated.

*Bitwise AND (`&`)*

The bitwise AND operator compares each corresponding bit in the binary representations of the operands. If both bits are 1, the corresponding bit in the result is 1; otherwise, the corresponding bit is 0. For example, `0755 & 0671` is `0651`. This is a little easier to understand if we look at the binary representation. Octal 0755 is binary 111101101, and octal 0671 is binary 110111001. We can then easily see which bits are in both numbers and visually come up with the answer:

```
  111101101
& 110111001
  ---------
  110101001
```

The binary number 110101001 is octal 0651.[2] You can use the PHP functions `bindec()`, `decbin()`, `octdec()`, and `decoct()` to convert numbers back and forth when you are trying to understand binary arithmetic.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise AND operation between the two corresponding characters in the operands. The resulting string is the length of the shorter of the two operands; trailing extra characters in the longer string are ignored. For example, `"wolf" & "cat"` is `"cad"`.

*Bitwise OR (`|`)*

The bitwise OR operator compares each corresponding bit in the binary representations of the operands. If both bits are 0, the resulting bit is 0; otherwise, the resulting bit is 1. For example, `0755 | 020` is `0775`.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise OR operation between the two corresponding characters in the operands. The resulting string is the length of the longer of the two operands, and the shorter string is padded at the end with binary 0s. For example, `"pussy" | "cat"` is `"suwsy"`.

2 Here's a tip: split the binary number into three groups—6 is binary 110, 5 is binary 101, and 1 is binary 001; thus, 0651 is 110101001.

*Bitwise XOR (`^`)*

The bitwise XOR operator compares each corresponding bit in the binary representation of the operands. If either of the bits in the pair, but not both, is 1, the resulting bit is 1; otherwise, the resulting bit is 0. For example, `0755 ^ 023` is `776`. If both operands are strings, this operator returns a string in which each character is the result of a bitwise XOR operation between the two corresponding characters in the operands. If the two strings are different lengths, the resulting string is the length of the shorter operand, and extra trailing characters in the longer string are ignored. For example, `"big drink" ^ "AA"` is `"#("`.

*Left shift (`<<`)*

The left-shift operator shifts the bits in the binary representation of the lefthand operand left by the number of places given in the righthand operand. Both operands will be converted to integers if they aren't already. Shifting a binary number to the left inserts a 0 as the rightmost bit of the number and moves all other bits to the left one place. For example, `3 << 1` (or binary 11 shifted one place left) results in `6` (binary 110).

Note that each place to the left that a number is shifted results in a doubling of the number. The result of left shifting is multiplying the lefthand operand by 2 to the power of the righthand operand.

*Right shift (`>>`)*

The right-shift operator shifts the bits in the binary representation of the lefthand operand right by the number of places given in the righthand operand. Both operands will be converted to integers if they aren't already. Shifting a positive binary number to the right inserts a 0 as the leftmost bit of the number and moves all other bits to the right one place. Shifting a negative binary number to the right inserts a 1 as the leftmost bit of the number and moves all other bits to the right one place. The rightmost bit is discarded. For example, `13 >> 1` (or binary 1101) shifted one bit to the right results in `6` (binary 110).

## Logical Operators

Logical operators provide ways for you to build complex logical expressions. Logical operators treat their operands as Boolean values and return a Boolean value. There are both punctuation and English versions of the operators (`||` and `or` are the same operator). The logical operators are:

*Logical AND (`&&`, `and`)*

The result of the logical AND operation is `true` if and only if both operands are `true`; otherwise, it is `false`. If the value of the first operand is `false`, the logical AND operator knows that the resulting value must also be `false`, so the righthand operand is never evaluated. This process is called *short-circuiting*, and a

common PHP idiom uses it to ensure that a piece of code is evaluated only if something is true. For example, you might connect to a database only if some flag is not `false`:

```
$result = $flag and mysql_connect();
```

The `&&` and `and` operators differ only in their precedence: `&&` comes before `and`.

*Logical OR (`||`, `or`)*
The result of the logical OR operation is `true` if either operand is `true`; otherwise, the result is `false`. Like the logical AND operator, the logical OR operator is short-circuited. If the lefthand operator is `true`, the result of the operator must be `true`, so the righthand operator is never evaluated. A common PHP idiom uses this to trigger an error condition if something goes wrong. For example:

```
$result = fopen($filename) or exit();
```

The `||` and `or` operators differ only in their precedence.

*Logical XOR (`xor`)*
The result of the logical XOR operation is `true` if either operand, but not both, is `true`; otherwise, it is `false`.

*Logical negation (`!`)*
The logical negation operator returns the Boolean value `true` if the operand evaluates to `false`, and `false` if the operand evaluates to `true`.

## Casting Operators

Although PHP is a weakly typed language, there are occasions when it's useful to consider a value as a specific type. The casting operators, `(int)`, `(float)`, `(string)`, `(bool)`, `(array)`, `(object)`, and `(unset)`, allow you to force a value into a particular type. To use a casting operator, put the operator to the left of the operand. Table 2-8 lists the casting operators, synonymous operators, and the type to which the operator changes the value.

*Table 2-8. PHP casting operators*

| Operator | Synonymous operators | Changes type to |
|---|---|---|
| `(int)` | `(integer)` | Integer |
| `(bool)` | `(boolean)` | Boolean |
| `(float)` | `(double)`, `(real)` | Floating point |
| `(string)` | | String |
| `(array)` | | Array |
| `(object)` | | Object |
| `(unset)` | | NULL |

Casting affects the way other operators interpret a value rather than changing the value in a variable. For example, the code:

```
$a = "5";
$b = (int) $a;
```

assigns `$b` the integer value of `$a`; `$a` remains the string `"5"`. To cast the value of the variable itself, you must assign the result of a cast back to the variable:

```
$a = "5";
$a = (int) $a; // now $a holds an integer
```

Not every cast is useful. Casting an array to a numeric type gives `1` (if the array is empty, it gives `0`), and casting an array to a string gives `"Array"` (seeing this in your output is a sure sign that you've printed a variable that contains an array).

Casting an object to an array builds an array of the properties, thus mapping property names to values:

```
class Person
{
 var $name = "Fred";
 var $age = 35;
}

$o = new Person;
$a = (array) $o;

print_r($a);
Array ( [name] => Fred [age] => 35)
```

You can cast an array to an object to build an object whose properties correspond to the array's keys and values. For example:

```
$a = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");
$o = (object) $a;
echo $o->name;
Fred
```

Keys that are not valid identifiers are invalid property names and are inaccessible when an array is cast to an object, but are restored when the object is cast back to an array.

## Assignment Operators

Assignment operators store or update values in variables. The auto-increment and auto-decrement operators we saw earlier are highly specialized assignment operators —here we see the more general forms. The basic assignment operator is `=`, but we'll also see combinations of assignment and binary operations, such as `+=` and `&=`.

### Assignment

The basic assignment operator (=) assigns a value to a variable. The lefthand operand is always a variable. The righthand operand can be any expression—any simple literal, variable, or complex expression. The righthand operand's value is stored in the variable named by the lefthand operand.

Because all operators are required to return a value, the assignment operator returns the value assigned to the variable. For example, the expression `$a = 5` not only assigns `5` to `$a`, but also behaves as the value `5` if used in a larger expression. Consider the following expressions:

```
$a = 5;
$b = 10;
$c = ($a = $b);
```

The expression `$a = $b` is evaluated first, because of the parentheses. Now, both `$a` and `$b` have the same value, `10`. Finally, `$c` is assigned the result of the expression `$a = $b`, which is the value assigned to the lefthand operand (in this case, `$a`). When the full expression is done evaluating, all three variables contain the same value: `10`.

### Assignment with operation

In addition to the basic assignment operator, there are several assignment operators that are convenient shorthand. These operators consist of a binary operator followed directly by an equals sign, and their effect is the same as performing the operation with the full operands, then assigning the resulting value to the lefthand operand. These assignment operators are:

*Plus-equals (+=)*
: Adds the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand. `$a += 5` is the same as `$a = $a + 5`.

*Minus-equals (−=)*
: Subtracts the righthand operand from the value of the lefthand operand, then assigns the result to the lefthand operand.

*Divide-equals (/=)*
: Divides the value of the lefthand operand by the righthand operand, then assigns the result to the lefthand operand.

*Multiply-equals (*=)*
: Multiplies the righthand operand by the value of the lefthand operand, then assigns the result to the lefthand operand.

*Modulus-equals (%=)*
: Performs the modulus operation on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

*Bitwise-XOR-equals (^=)*
: Performs a bitwise XOR on the lefthand and righthand operands, then assigns the result to the lefthand operand.

*Bitwise-AND-equals (&=)*
: Performs a bitwise AND on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

*Bitwise-OR-equals (|=)*
: Performs a bitwise OR on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

*Concatenate-equals (.=)*
: Concatenates the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand.

## Miscellaneous Operators

The remaining PHP operators are for error suppression, executing an external command, and selecting values:

*Error suppression (@)*
: Some operators or functions can generate error messages. The error suppression operator, discussed in full in Chapter 17, is used to prevent these messages from being created.

*Execution (`...`)*
: The backtick operator executes the string contained between the backticks as a shell command and returns the output. For example:

```
$listing = `ls -ls /tmp`;
echo $listing;
```

*Conditional (? :)*
: The conditional operator is, depending on the code you look at, either the most overused or most underused operator. It is the only ternary (three-operand) operator and is therefore sometimes just called the ternary operator.

The conditional operator evaluates the expression before the ?. If the expression is true, the operator returns the value of the expression between the ? and :; otherwise, the operator returns the value of the expression after the :. For instance:

```
<a href="<? echo $url; ?>"><? echo $linktext ? $linktext : $url; ?></a>
```

If text for the link `$url` is present in the variable `$linktext`, it is used as the text for the link; otherwise, the URL itself is displayed.

*Type* (`instanceof`)
: The `instanceof` operator tests whether a variable is an instantiated object of a given class or implements an interface (see Chapter 6 for more information on objects and interfaces):

```
$a = new Foo;
$isAFoo = $a instanceof Foo; // true
$isABar = $a instanceof Bar; // false
```

# Flow-Control Statements

PHP supports a number of traditional programming constructs for controlling the flow of execution of a program.

Conditional statements, such as `if`/`else` and `switch`, allow a program to execute different pieces of code, or none at all, depending on some condition. Loops, such as `while` and `for`, support the repeated execution of particular segments of code.

## if

The `if` statement checks the truthfulness of an expression and, if the expression is true, evaluates a statement. An `if` statement looks like:

```
if (expression)statement
```

To specify an alternative statement to execute when the expression is false, use the `else` keyword:

```
if (expression)
 statement
else statement
```

For example:

```
if ($user_validated)
 echo "Welcome!";
else
 echo "Access Forbidden!";
```

To include more than one statement within an `if` statement, use a *block*—a set of statements enclosed by curly braces:

```
if ($user_validated) {
 echo "Welcome!";
 $greeted = 1;
}
else {
 echo "Access Forbidden!";
```

```
 exit;
}
```

PHP provides another syntax for blocks in tests and loops. Instead of enclosing the block of statements in curly braces, end the `if` line with a colon (`:`) and use a specific keyword to end the block (`endif`, in this case). For example:

```
if ($user_validated):
 echo "Welcome!";
 $greeted = 1;
else:
 echo "Access Forbidden!";
 exit;
endif;
```

Other statements described in this chapter also have similar alternate syntax styles (and ending keywords); they can be useful if you have large blocks of HTML inside your statements. For example:

```
<?php if ($user_validated) : ?>
 <table>
 <tr>
 <td>First Name:</td><td>Sophia</td>
 </tr>
 <tr>
 <td>Last Name:</td><td>Lee</td>
 </tr>
 </table>
<?php else: ?>
 Please log in.
<?php endif ?>
```

Because `if` is a statement, you can chain (embed) more than one. This is also a good example of how the blocks can be used to help keep things organized:

```
if ($good) {
 print("Dandy!");
}
else {
 if ($error) {
 print("Oh, no!");
 }
 else {
 print("I'm ambivalent...");
 }
}
```

Such chains of `if` statements are common enough that PHP provides an easier syntax: the `elseif` statement. For example, the previous code can be rewritten as:

```
if ($good) {
 print("Dandy!");
}
```

```
elseif ($error) {
 print("Oh, no!");
}
else {
 print("I'm ambivalent...");
}
```

The ternary conditional operator (`? :`) can be used to shorten simple true/false tests. Take a common situation, such as checking to see if a given variable is true and printing something if it is. With a normal `if/else` statement, it looks like this:

```
<td><?php if($active) { echo "yes"; } else { echo "no"; } ?></td>
```

With the ternary conditional operator, it looks like this:

```
<td><?php echo $active ? "yes" : "no"; ?></td>
```

Compare the syntax of the two:

```
if (expression) { true_statement } else { false_statement }
 (expression) ? true_expression : false_expression
```

The main difference here is that the conditional operator is not a statement at all. This means that it is used on expressions, and the result of a complete ternary expression is itself an expression. In the previous example, the `echo` statement is inside the `if` condition, while when used with the ternary operator, it precedes the expression.

## switch

The value of a single variable may determine one of a number of different choices (e.g., the variable holds the username and you want to do something different for each user). The `switch` statement is designed for just this situation.

A `switch` statement is given an expression and compares its value to all cases in the switch; all statements in a matching case are executed, up to the first `break` keyword it finds. If none match, and a `default` is given, all statements following the `default` keyword are executed, up to the first `break` keyword encountered.

For example, suppose you have the following:

```
if ($name == 'ktatroe') {
 // do something
}
else if ($name == 'dawn') {
 // do something
}
else if ($name == 'petermac') {
 // do something
}
else if ($name == 'bobk') {
 // do something
}
```

You can replace that statement with the following `switch` statement:

```
switch($name) {
 case 'ktatroe':
 // do something
 break;
 case 'dawn':
 // do something
 break;
 case 'petermac':
 // do something
 break;
 case 'bobk':
 // do something
 break;
}
```

The alternative syntax for this is:

```
switch($name):
 case 'ktatroe':
 // do something
 break;
 case 'dawn':
 // do something
 break;
 case 'petermac':
 // do something
 break;
 case 'bobk':
 // do something
 break;
endswitch;
```

Because statements are executed from the matching case label to the next `break` keyword, you can combine several cases in a *fall-through*. In the following example, "yes" is printed when `$name` is equal to `sylvie` or `bruno`:

```
switch ($name) {
 case 'sylvie': // fall-through
 case 'bruno':
 print("yes");
 break;
 default:
 print("no");
 break;
}
```

Commenting the fact that you are using a fall-through case in a `switch` is a good idea, so someone doesn't come along at some point and add a `break` thinking you had forgotten it.

You can specify an optional number of levels for the `break` keyword to break out of. In this way, a `break` statement can break out of several levels of nested `switch` statements. An example of using `break` in this manner is shown in the next section.

## while

The simplest form of loop is the `while` statement:

```
while (expression)statement
```

If the *expression* evaluates to `true`, the *statement* is executed and then the *expression* is re-evaluated (if it is still `true`, the body of the loop is executed again, and so on). The loop exits when the *expression* is no longer true (i.e., evaluates to `false`).

As an example, here's some code that adds the whole numbers from 1 to 10:

```
$total = 0;
$i = 1;

while ($i <= 10) {
 $total += $i;
 $i++;
}
```

The alternative syntax for `while` has this structure:

```
while (expr):
 statement;
 more statements ;
endwhile;
```

For example:

```
$total = 0;
$i = 1;

while ($i <= 10):
 $total += $i;
 $i++;
endwhile;
```

You can prematurely exit a loop with the `break` keyword. In the following code, `$i` never reaches a value of 6, because the loop is stopped once it reaches 5:

```
$total = 0;
$i = 1;

while ($i <= 10) {
 if ($i == 5) {
 break; // breaks out of the loop
 }

 $total += $i;
```

```
 $i++;
}
```

Optionally, you can put a number after the break keyword indicating how many levels of loop structures to break out of. In this way, a statement buried deep in nested loops can break out of the outermost loop. For example:

```
$i = 0;
$j = 0;

while ($i < 10) {
 while ($j < 10) {
 if ($j == 5) {
 break 2; // breaks out of two while loops
 }

 $j++;
 }

 $i++;
}

echo "{$i}, {$j}";
0, 5
```

The continue statement skips ahead to the next test of the loop condition. As with the break keyword, you can continue through an optional number of levels of loop structure:

```
while ($i < 10) {
 $i++;

 while ($j < 10) {
 if ($j == 5) {
 continue 2; // continues through two levels
 }

 $j++;
 }
}
```

In this code, $j never has a value above 5, but $i goes through all values from 0 to 9.

PHP also supports a do/while loop, which takes the following form:

```
do
 statement
while (expression)
```

Use a do/while loop to ensure that the loop body is executed at least once (the first time):

```
$total = 0;
$i = 1;

do {
 $total += $i++;
} while ($i <= 10);
```

You can use `break` and `continue` statements in a `do`/`while` statement just as in a normal `while` statement.

The `do`/`while` statement is sometimes used to break out of a block of code when an error condition occurs. For example:

```
do {
 // do some stuff

 if ($errorCondition) {
 break;
 }

 // do some other stuff
} while (false);
```

Because the condition for the loop is `false`, the loop is executed only once, regardless of what happens inside the loop. However, if an error occurs, the code after the `break` is not evaluated.

## for

The `for` statement is similar to the `while` statement, except it adds counter initialization and counter manipulation expressions, and is often shorter and easier to read than the equivalent `while` loop.

Here's a `while` loop that counts from 0 to 9, printing each number:

```
$counter = 0;

while ($counter < 10) {
 echo "Counter is {$counter} <br/>";
 $counter++;
}
```

Here's the corresponding, more concise `for` loop:

```
for ($counter = 0; $counter < 10; $counter++) {
 echo "Counter is $counter <br/>";
}
```

The structure of a `for` statement is:

```
for (start; condition; increment) { statement(s); }
```

The expression *start* is evaluated once, at the beginning of the `for` statement. Each time through the loop, the expression *condition* is tested. If it is `true`, the body of the loop is executed; if it is `false`, the loop ends. The expression *increment* is evaluated after the loop body runs.

The alternative syntax of a `for` statement is:

```
for (expr1; expr2; expr3):
 statement;
 ...;
endfor;
```

This program adds the numbers from 1 to 10 using a `for` loop:

```
$total = 0;

for ($i= 1; $i <= 10; $i++) {
 $total += $i;
}
```

Here's the same loop using the alternate syntax:

```
$total = 0;

for ($i = 1; $i <= 10; $i++):
 $total += $i;
endfor;
```

You can specify multiple expressions for any of the expressions in a `for` statement by separating the expressions with commas. For example:

```
$total = 0;

for ($i = 0, $j = 1; $i <= 10; $i++, $j *= 2) {
 $total += $j;
}
```

You can also leave an expression empty, signaling that nothing should be done for that phase. In the most degenerate form, the `for` statement becomes an infinite loop. You probably don't want to run this example, as it never stops printing:

```
for (;;) {
 echo "Can't stop me!<br />";
}
```

In `for` loops, as in `while` loops, you can use the `break` and `continue` keywords to end the loop or the current iteration.

## foreach

The `foreach` statement allows you to iterate over elements in an array. The two forms of the `foreach` statement are further discussed in Chapter 5, where we talk in more depth about arrays. To loop over an array, accessing the value at each key, use:

```
foreach ($array as $current) {
 // ...
}
```

The alternate syntax is:

```
foreach ($array as $current):
 // ...
endforeach;
```

To loop over an array, accessing both key and value, use:

```
foreach ($array as $key => $value) {
 // ...
}
```

The alternate syntax is:

```
foreach ($array as $key => $value):
 // ...
endforeach;
```

## try...catch

The `try...catch` construct is not so much a flow-control structure as it is a more graceful way to handle system errors. For example, if you want to ensure that your web application has a valid connection to a database before continuing, you could write code like this:

```
try {
 $dbhandle = new PDO('mysql:host=localhost; dbname=library', $username, $pwd);
 doDB_Work($dbhandle); // call function on gaining a connection
 $dbhandle = null; // release handle when done
}
catch (PDOException $error) {
 print "Error!: " . $error->getMessage() . "<br/>";
 die();
}
```

Here the connection is attempted with the `try` portion of the construct and if there are any errors with it, the flow of the code automatically falls into the `catch` portion, where the `PDOException` class is instantiated into the `$error` variable. It can then be displayed on the screen and the code can "gracefully" fail, rather than making an abrupt end. You can even redirect to try connecting to an alternate database, or respond to the error any other way you wish within the `catch` portion.

See Chapter 9 for more examples of `try...catch` in relation to PDO (PHP Data Objects) and transaction processing.

## declare

The `declare` statement allows you to specify execution directives for a block of code. The structure of a `declare` statement is:

```
declare (directive)statement
```

Currently, there are only three `declare` forms: the `ticks`, `encoding`, and `strict_types` directives. You can use the `ticks` directive to specify how frequently (measured roughly in number of code statements) a tick function is registered when `register_tick_function()` is called. For example:

```
register_tick_function("someFunction");

declare(ticks = 3) {
 for($i = 0; $i < 10; $i++) {
 // do something
 }
}
```

In this code, `someFunction()` is called after every third statement within the block is executed.

You can use the `encoding` directive to specify a PHP script's output encoding. For example:

```
declare(encoding = "UTF-8");
```

This form of the `declare` statement is ignored unless you compile PHP with the `--enable-zend-multibyte` option.

Finally, you can use the `strict_types` directive to enforce the use of strict data types when defining and using variables.

## exit and return

As soon as it is reached, the `exit` statement ends the script's execution. The `return` statement returns from a function or, at the top level of the program, from the script.

The `exit` statement takes an optional value. If this is a number, it is the exit status of the process. If it is a string, the value is printed before the process terminates. The function `die()` is an alias for this form of the `exit` statement:

```
$db = mysql_connect("localhost", $USERNAME, $PASSWORD);
```

```
if (!$db) {
 die("Could not connect to database");
}
```

This is more commonly written as:

```
$db = mysql_connect("localhost", $USERNAME, $PASSWORD)
 or die("Could not connect to database");
```

See Chapter 3 for more information on using the `return` statement in functions.

## goto

The `goto` statement allows execution to "jump" to another place in the program. You specify execution points by adding a label, which is an identifier followed by a colon (`:`). You then jump to the label from another location in the script via the `goto` statement:

```
for ($i = 0; $i < $count; $i++) {
 // oops, found an error
 if ($error) {
 goto cleanup;
 }
}

cleanup:
// do some cleanup
```

You can only `goto` a label within the same scope as the `goto` statement itself, and you can't jump into a loop or switch. Generally, anywhere you might use a `goto` (or multilevel `break` statement, for that matter), you can rewrite the code to be cleaner without it.

# Including Code

PHP provides two constructs to load code and HTML from another module: `require` and `include`. Both load a file as the PHP script runs, work in conditionals and loops, and complain if the file being loaded cannot be found. Files are located by an included file path as part of the directive in the use of the function, or based on the setting of `include_path` in the *php.ini* file. The `include_path` can be overridden by the `set_include_path()` function. If all these avenues fail, PHP's last attempt is to try to find the file in the same directory as the calling script. The main difference is that attempting to `require` a nonexistent file is a fatal error, while attempting to `include` such a file produces a warning but does not stop script execution.

A common use of `include` is to separate page-specific content from general site design. Common elements such as headers and footers go in separate HTML files, and each page then looks like:

```
<?php include "header.html"; ?>
content
<?php include "footer.html"; ?>
```

We use `include` because it allows PHP to continue to process the page even if there's an error in the site design file(s). The `require` construct is less forgiving and is more suited to loading code libraries, where the page cannot be displayed if the libraries do not load. For example:

```
require "codelib.php";
mysub(); // defined in codelib.php
```

A marginally more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized site elements:

```
<?php require "design.php";
header(); ?>
content
<?php footer();
```

If PHP cannot parse some part of a file added by `include` or `require`, a warning is printed and execution continues. You can silence the warning by prepending the call with the silence operator (`@`)—for example, `@include`.

If the `allow_url_fopen` option is enabled through PHP's configuration file, *php.ini*, you can include files from a remote site by providing a URL instead of a simple local path:

```
include "http://www.example.com/codelib.php";
```

If the filename begins with *http://*, *https://*, or *ftp://*, the file is retrieved from a remote site and loaded.

Files included with `include` and `require` can be arbitrarily named. Common extensions are *.php*, *.php5*, and *.html*.

Note that remotely fetching a file that ends in *.php* from a web server that has PHP enabled fetches the *output* of that PHP script—it executes the PHP code in that file.

If a program uses `include` or `require` to include the same file twice (mistakenly done in a loop, for example), the file is loaded and the code is run, or the HTML is printed twice. This can result in errors about the redefinition of functions, or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the `include_once` and `require_once` constructs. They behave the same as `include` and `require` the first time a file is loaded, but quietly ignore subsequent attempts to load the same file. For example, many page elements, each stored in separate files, need to

know the current user's preferences. The element libraries should load the user preferences library with `require_once`. The page designer can then include a page element without worrying about whether the user preference code has already been loaded.

Code in an included file is imported at the scope that is in effect where the `include` statement is found, so the included code can see and alter your code's variables. This can be useful—for instance, a user-tracking library might store the current user's name in the global `$user` variable:

```
// main page
include "userprefs.php";
echo "Hello, {$user}.";
```

The ability of libraries to see and change your variables can also be a problem. You have to know every global variable used by a library to ensure that you don't accidentally try to use one of them for your own purposes, thereby overwriting the library's value and disrupting how it works.

If the `include` or `require` construct is in a function, the variables in the included file become function-scope variables for that function.

Because `include` and `require` are keywords, not real statements, you must always enclose them in curly braces in conditional and loop statements:

```
for ($i = 0; $i < 10; $i++) {
 include "repeated_element.html";
}
```

Use the `get_included_files()` function to learn which files your script has included or required. It returns an array containing the full system path filenames of each included or required file. Files that did not parse are not included in this array.

# Embedding PHP in Web Pages

Although it is possible to write and run standalone PHP programs, most PHP code is embedded in HTML or XML files. This is, after all, why it was created in the first place. Processing such documents involves replacing each chunk of PHP source code with the output it produces when executed.

Because a single file usually contains PHP and non-PHP source code, we need a way to identify the regions of PHP code to be executed. PHP provides four different ways to do this.

As you'll see, the first, and preferred, method looks like XML. The second method looks like SGML. The third method is based on ASP tags. The fourth method uses the standard HTML `<script>` tag; this makes it easy to edit pages with enabled PHP using a regular HTML editor.

## Standard (XML) Style

Because of the advent of the eXtensible Markup Language (XML) and the migration of HTML to an XML language (XHTML), the currently preferred technique for embedding PHP uses XML-compliant tags to denote PHP instructions.

Coming up with tags to demark PHP commands in XML was easy, because XML allows the definition of new tags. To use this style, surround your PHP code with `<?php` and `?>`. Everything between these markers is interpreted as PHP, and anything outside the markers is not. Although it is not necessary to include spaces between the markers and the enclosed text, doing so improves readability. For example, to get PHP to print "Hello, world," you can insert the following line in a web page:

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

The trailing semicolon on the statement is optional, because the end of the block also forces the end of the expression. Embedded in a complete HTML file, this looks like:

```
<!doctype html>
<html>
<head>
 <title>This is my first PHP program!</title>
</head>

<body>
<p>
 Look, ma! It's my first PHP program:<br />
 <?php echo "Hello, world"; ?><br />
 How cool is that?
</p>
</body>

</html>
```

Of course, this isn't very exciting—we could have done it without PHP. The real value of PHP comes when we put dynamic information from sources such as databases and form values into the web page. That's for a later chapter, though. Let's get back to our "Hello, world" example. When a user visits this page and views its source, it looks like this:

```
<!doctype html>
<html>
<head>
 <title>This is my first PHP program!</title>
</head>

<body>
<p>
 Look, ma! It's my first PHP program:<br />
 Hello, world!<br />
 How cool is that?
```

```
</p>
</body>

</html>
```

Notice that there's no trace of the PHP source code from the original file. The user sees only its output.

Also notice that we switched between PHP and non-PHP, all in the space of a single line. PHP instructions can be put anywhere in a file, even within valid HTML tags. For example:

```
<input type="text" name="first_name" value="<?php echo "Peter"; ?>" />
```

When PHP is done with this text, it will read:

```
<input type="text" name="first_name" value="Peter" />
```

The PHP code within the opening and closing markers does not have to be on the same line. If the closing marker of a PHP instruction is the last thing on a line, the line break following the closing tag is removed as well. Thus, we can replace the PHP instructions in the "Hello, world" example with:

```
<?php
echo "Hello, world"; ?>
<br />
```

with no change in the resulting HTML.

## SGML Style

Another style of embedding PHP comes from SGML instruction processing tags. To use this method, simply enclose the PHP in `<?` and `?>`. Here's the "Hello, world" example again:

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

This style, known as *short tags*, is off by default. You can turn on support for short tags by building PHP with the `--enable-short-tags` option, or enable `short_open_tag` in the PHP configuration file. This is discouraged as it depends on the state of this setting; if you export your code to another platform, it may or may not work.

The short echo tag, `<?= ... ?>`, is available regardless of the availability of short tags.

## Echoing Content Directly

Perhaps the single most common operation within a PHP application is displaying data to the user. In the context of a web application, this means inserting into the HTML document information that will become HTML when viewed by the user.

To simplify this operation, PHP provides a special version of the SGML tags that automatically take the value inside the tag and insert it into the HTML page. To use this feature, add an equals sign (=) to the opening tag. With this technique, we can rewrite our form example as:

```
<input type="text" name="first_name" value="<?= "Dawn"; ?>">
```

## What's Next

Now that you have the basics of the language under your belt—a foundational understanding of what variables are and how to name them, what data types are, and how code flow control works—we will move on to some finer details of the PHP language. Next we'll cover three topics that are so important to PHP that they each have their own dedicated chapters: how to define functions (Chapter 3), manipulate strings (Chapter 4), and manage arrays (Chapter 5).

# CHAPTER 3
# Functions

A *function* is a named block of code that performs a specific task, possibly acting upon a set of values given to it, aka *parameters*, and possibly returning a single value or set of values in an array. Functions save on compile time—no matter how many times you call them, functions are compiled only once for the page. They also improve reliability by allowing you to fix any bugs in one place rather than everywhere you perform a task, and they improve readability by isolating code that performs specific tasks.

This chapter introduces the syntax of function calls and function definitions and discusses how to manage variables in functions and pass values to functions (including pass-by-value and pass-by-reference). It also covers variable functions and anonymous functions.

## Calling a Function

Functions in a PHP program can be built in (or, by being in an extension, effectively built in) or user-defined. Regardless of their source, all functions are evaluated in the same way:

```
$someValue = function_name( [ parameter, ... ] );
```

The number of parameters a function requires differs from function to function (and, as we'll see later, may even vary for the same function). The parameters supplied to the function may be any valid expression and must be in the specific order expected by the function. If the parameters are given out of order, the function may still run by a fluke, but it's basically a case of "garbage in = garbage out." A function's documentation will tell you what parameters the function expects and what value(s) you can expect to be returned.

Here are some examples of functions:

```
// strlen() is a PHP built-in function that returns the length of a string
$length = strlen("PHP"); // $length is now 3
// sin() and asin() are the sine and arcsine math functions
$result = sin(asin(1)); // $result is the sine of arcsin(1), or 1.0

// unlink() deletes a file
$result = unlink("functions.txt");
// $result = true or false depending on success or failure
```

In the first example, we give an argument, `"PHP"`, to the function `strlen()`, which gives us the number of characters in the provided string. In this case, it returns `3`, which is assigned to the variable `$length`. This is the simplest and most common way to use a function.

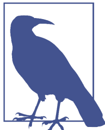
The second example passes the result of `asin(1)` to the `sin()` function. Since the sine and arcsine functions are inverses, taking the sine of the arcsine of any value will always return that same value. Here we see that a function can be called within another function. The returned value of the inner call is subsequently sent to the outer function before the overall result is returned and stored in the `$result` variable.

In the final example, we give a filename to the `unlink()` function, which attempts to delete the file. Like many functions, it returns `false` when it fails. This allows you to use another built-in function, `die()`, and the short-circuiting property of the logic operators. Thus, this example might be rewritten as:

```
$result = unlink("functions.txt") or die("Operation failed!");
```

The `unlink()` function, unlike the other two examples, affects something outside of the parameters given to it. In this case, it deletes a file from the filesystem. All such side effects of a function should be carefully documented and carefully considered.

PHP has a huge array of functions already defined for you to use in your programs. These extensions perform tasks such as accessing databases, creating graphics, reading and writing XML files, grabbing files from remote systems, and more. PHP's built-in functions are described in detail in the Appendix.



Not all functions return a value. They can perform an action like sending an email and then just return controlling behavior to the calling code; having completed their task, they have nothing to "say."

## Defining a Function

To define a function, use the following syntax:

```
function [&] function_name([parameter[, ...]])
{
 statement list
}
```

The statement list can include HTML. You can declare a PHP function that doesn't contain any PHP code. For instance, the `column()` function simply gives a convenient short name to HTML code that may be needed many times throughout the page:

```
<?php function column()
{ ?>
 </td><td>
<?php }
```

The function name can be any string that starts with a letter or underscore followed by zero or more letters, underscores, and digits. Function names are case-insensitive; that is, you can call the `sin()` function as `sin(1)`, `SIN(1)`, `SiN(1)`, and so on, because all these names refer to the same function. By convention, built-in PHP functions are called with all lowercase.

Typically, functions return some value. To return a value from a function, use the `return` statement: put `return` *expr* inside your function. When a `return` statement is encountered during execution, control reverts to the calling statement, and the evaluated results of *expr* will be returned as the value of the function. You can include any number of `return` statements in a function (for example, if you have a `switch` statement to determine which of several values to return).

Let's take a look at a simple function. Example 3-1 takes two strings, concatenates them, and then returns the result (in this case, we've created a slightly slower equivalent to the concatenation operator, but bear with us for the sake of the example).

*Example 3-1. String concatenation*

```
function strcat($left, $right)
{
 $combinedString = $left . $right;

 return $combinedString;
}
```

The function takes two arguments, `$left` and `$right`. Using the concatenation operator, the function creates a combined string in the variable `$combinedString`. Finally, in order to cause the function to have a value when it's evaluated with our arguments, we return the value `$combinedString`.

Because the `return` statement can accept any expression, even complex ones, we can simplify the program as shown here:

```
function strcat($left, $right)
{
 return $left . $right;
}
```

If we put this function on a PHP page, we can call it from anywhere within the page. Take a look at Example 3-2.

*Example 3-2. Using our concatenation function*

```
<?php
function strcat($left, $right)
{
 return $left . $right;
}
$first = "This is a ";
$second = " complete sentence!";

echo strcat($first, $second);
```

When this page is displayed, the full sentence is shown.

In this next example a function takes in an integer, doubles it by bit-shifting the original value, and returns the result:

```
function doubler($value)
{
 return $value << 1;
}
```

Once the function is defined, you can use it anywhere on the page. For example:

```
<?php echo "A pair of 13s is " . doubler(13); ?>
```

You can nest function declarations, but with limited effect. Nested declarations do not limit the visibility of the inner-defined function, which may be called from anywhere in your program. The inner function does not automatically get the outer function's arguments. And, finally, the inner function cannot be called until the outer function has been called, and also cannot be called from code parsed after the outer function:

```
function outer ($a)
{
 function inner ($b)
 {
 echo "there $b";
 }

 echo "$a, hello ";
}
```

```
// outputs "well, hello there reader"
outer("well");
inner("reader");
```

# Variable Scope

If you don't use functions, any variable you create can be used anywhere in a page. With functions, this is not always true. Functions keep their own sets of variables that are distinct from those of the page and of other functions.

The variables defined in a function, including its parameters, are not accessible outside the function, and, by default, variables defined outside a function are not accessible inside the function. The following example illustrates this:

```
$a = 3;

function foo()
{
 $a += 2;
}

foo();
echo $a;
```

The variable `$a` inside the function `foo()` is a different variable than the variable `$a` outside the function; even though `foo()` uses the add-and-assign operator, the value of the outer `$a` remains `3` throughout the life of the page. Inside the function, `$a` has the value `2`.

As we discussed in Chapter 2, the extent to which a variable can be seen in a program is called the *scope* of the variable. Variables created within a function are inside the scope of the function (i.e., have *function-level scope*). Variables created outside of functions and objects have *global scope* and exist anywhere outside of those functions and objects. A few variables provided by PHP have both function-level and global scope (often referred to as *super-global variables*).

At first glance, even an experienced programmer may think that in the previous example `$a` will be `5` by the time the `echo` statement is reached, so keep that in mind when choosing names for your variables.

## Global Variables

If you want a variable in the global scope to be accessible from within a function, you can use the `global` keyword. Its syntax is:

```
global var1, var2, ... ;
```

Changing the previous example to include a `global` keyword, we get:

```
$a = 3;

function foo()
{
 global $a;

 $a += 2;
}

foo();
echo $a;
```

Instead of creating a new variable called `$a` with function-level scope, PHP uses the global `$a` within the function. Now, when the value of `$a` is displayed, it will be `5`.

You must include the `global` keyword in a function before any uses of the global variable or variables you want to access. Because they are declared before the body of the function, function parameters can never be global variables.

Using `global` is equivalent to creating a reference to the variable in the `$GLOBALS` variable. That is, both of the following declarations create a variable in the function's scope that is a reference to the same value as the variable `$var` in the global scope:

```
global $var;
$var = & $GLOBALS['var'];
```

## Static Variables

Like C, PHP supports declaring function variables as *static*. A static variable retains its value between all calls to the function and is initialized during a script's execution only the first time the function is called. Use the `static` keyword at the first use of a function variable to declare it as static. Typically, the first use of a static variable assigns an initial value:

```
static var [= value][, ... ];
```

In Example 3-3, the variable `$count` is incremented by one each time the function is called.

*Example 3-3. Static variable counter*

```
<?php
function counter()
{
 static $count = 0;

 return $count++;
}
```

```
for ($i = 1; $i <= 5; $i++) {
 print counter();
}
```

When the function is called for the first time, the static variable `$count` is assigned a value of `0`. The value is returned and `$count` is incremented. When the function ends, `$count` is not destroyed like a nonstatic variable, and its value remains the same until the next time `counter()` is called. The `for` loop displays the numbers from 0 to 4.

# Function Parameters

Functions can expect an arbitrary number of arguments, declared by the function definition. There are two different ways to pass parameters to a function. The first, and more common, is by value. The second is by reference.

## Passing Parameters by Value

In most cases, you pass parameters by value. The argument is any valid expression. That expression is evaluated, and the resulting value is assigned to the appropriate variable in the function. In all of the examples so far, we've been passing arguments by value.

## Passing Parameters by Reference

Passing by reference allows you to override the normal scoping rules and give a function direct access to a variable. To be passed by reference, the argument must be a variable; you indicate that a particular argument of a function will be passed by reference by preceding the variable name in the parameter list with an ampersand (`&`). Example 3-4 revisits our `doubler()` function with a slight change.

*Example 3-4. doubler() redux*

```
<?php
function doubler(&$value)
{
 $value = $value << 1;
}

$a = 3;
doubler($a);

echo $a;
```

Because the function's `$value` parameter is passed by reference, the actual value of `$a`, rather than a copy of that value, is modified by the function. Before, we had to `return` the doubled value, but now we change the caller's variable to be the doubled value.

This is another place where a function has side effects: since we passed the variable `$a` into `doubler()` by reference, the value of `$a` is at the mercy of the function. In this case, `doubler()` assigns a new value to it.

Only variables—and not constants—can be supplied to parameters declared as passing by reference. Thus, if we included the statement `<?php echo doubler(7); ?>` in the previous example, it would issue an error. However, you may assign a default value to parameters passed by reference (in the same manner as you provide default values for parameters passed by value).

Even in cases where your function does not affect the given value, you may want a parameter to be passed by reference. When passing by value, PHP must copy the value. Particularly for large strings and objects, this can be an expensive operation. Passing by reference removes the need to copy the value.

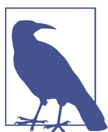
## Default Parameters

Sometimes a function may need to accept a particular parameter. For example, when you call a function to get the preferences for a site, the function may take in a parameter with the name of the preference to retrieve. Rather than using some special keyword to designate that you want to retrieve all of the preferences, you can simply not supply any argument. This behavior works by using default arguments.

To specify a default parameter, assign the parameter value in the function declaration. The value assigned to a parameter as a default value cannot be a complex expression, only a scalar value:

```
function getPreferences($whichPreference = 'all')
{
 // if $whichPreference is "all", return all prefs;
 // otherwise, get the specific preference requested...
}
```

When you call `getPreferences()`, you can choose to supply an argument. If you do, it returns the preference matching the string you give it; if not, it returns all preferences.



A function may have any number of parameters with default values. However, these defaulted parameters must be listed after all parameters that do not have default values.

## Variable Parameters

A function may require a variable number of arguments. For example, the getPreferences() example in the previous section might return the preferences for any number of names, rather than for just one. To declare a function with a variable number of arguments, leave out the parameter block entirely:

```
function getPreferences()
{
 // some code
}
```

PHP provides three functions you can use in the function to retrieve the parameters passed to it. func_get_args() returns an array of all parameters provided to the function; func_num_args() returns the number of parameters provided to the function; and func_get_arg() returns a specific argument from the parameters. For example:

```
$array = func_get_args();
$count = func_num_args();
$value = func_get_arg(argument_number);
```

In Example 3-5, the count_list() function takes in any number of arguments. It loops over those arguments and returns the total of all the values. If no parameters are given, it returns false.

*Example 3-5. Argument counter*

```
<?php
function countList()
{
 if (func_num_args() == 0) {
 return false;
 }
 else {
 $count = 0;

 for ($i = 0; $i < func_num_args(); $i++) {
 $count += func_get_arg($i);
 }

 return $count;
 }
}

echo countList(1, 5, 9); // outputs "15"
```

The result of any of these functions cannot directly be used as a parameter to another function. Instead, you must first set a variable to the result of the function, and then use that in the function call. The following expression will not work:

```
foo(func_num_args());
```

Instead, use:

```
$count = func_num_args();
foo($count);
```

## Missing Parameters

PHP lets you be as lazy as you want—when you call a function, you can pass any number of arguments to the function. Any parameters the function expects that are not passed to it remain unset, and a warning is issued for each of them:

```
function takesTwo($a, $b)
{
 if (isset($a)) {
 echo " a is set\n";
 }

 if (isset($b)) {
 echo " b is set\n";
 }
}

 echo "With two arguments:\n";
takesTwo(1, 2);

echo "With one argument:\n";
takesTwo(1);
With two arguments:
 a is set
 b is set
With one argument:
Warning: Missing argument 2 for takes_two()
 in /path/to/script.php on line 6
a is set
```

## Type Hinting

When defining a function, you can add type hinting—that is, you can require that a parameter be an instance of a particular class (including instances of classes that extend that class), an instance of a class that implements a particular interface, an array, or a callable. To add type hinting to a parameter, include the class name, `array`, or `callable` *before* the variable name in the function's parameter list. For example:

```
class Entertainment {}
```

```
class Clown extends Entertainment {}

class Job {}

function handleEntertainment(Entertainment $a, callable $callback = NULL)
{
 echo "Handling " . get_class($a) . " fun\n";

 if ($callback !== NULL) {
 $callback();
 }
}

$callback = function()
{
 // do something
};

handleEntertainment(new Clown); // works
handleEntertainment(new Job, $callback); // runtime error
```

A type-hinted parameter must be `NULL`, an instance of the given class or a subclass of the class, an array, or callable as a specified parameter. Otherwise, a runtime error occurs.

You can define a data type for a property in a class.

# Return Values

PHP functions can return only a single value with the `return` keyword:

```
function returnOne()
{
 return 42;
}
```

To return multiple values, return an array:

```
function returnTwo()
{
 return array("Fred", 35);
}
```

If no return value is provided by a function, the function returns `NULL` instead. You can set a return data type by declaring it in the function definition. For example, the following code will return an integer of 50 when it is executed:

```
function someMath($var1, $var2): int
{
 return $var1 * $var2;
}
```

```
echo someMath(10, 5);
```

By default, values are copied out of the function. To return a value by reference, prepend the function name with `&` both when declaring it and when assigning the returned value to a variable:

```
$names = array("Fred", "Barney", "Wilma", "Betty");

function &findOne($n) {
 global $names;

 return $names[$n];
}

$person =& findOne(1); // Barney
$person = "Barnetta"; // changes $names[1]
```

In this code, the `findOne()` function returns an alias for `$names[1]` instead of a copy of its value. Because we assign by reference, `$person` is an alias for `$names[1]`, and the second assignment changes the value in `$names[1]`.

This technique is sometimes used to return large string or array values efficiently from a function. However, PHP implements copy-on-write for variable values, meaning that returning a reference from a function is typically unnecessary. Returning a reference to a value is slower than returning the value itself.

## Variable Functions

As with variable variables where the expression refers to the value of the variable whose name is the value held by the apparent variable (the `$$` construct), you can add parentheses after a variable to call the function whose name is the value held by the apparent variable—for example, `$variable()`. Consider this situation, where a variable is used to determine which of three functions to call:

```
switch ($which) {
 case 'first':
 first();
 break;

 case 'second':
 second();
 break;

 case 'third':
 third();
 break;
}
```

In this case, we could use a variable function call to call the appropriate function. To make a variable function call, include the parameters for a function in parentheses after the variable. To rewrite the previous example:

```
$which(); // if $which is "first", the function first() is called, etc...
```

If no function exists for the variable, a runtime error occurs when the code is evaluated. To prevent this, before calling the function you can use the built-in `function_exists()` function to determine whether a function exists for the value of the variable:

```
$yesOrNo = function_exists(function_name);
```

For example:

```
if (function_exists($which)) {
 $which(); // if $which is "first", the function first() is called, etc...
}
```

Language constructs such as `echo()` and `isset()` cannot be called through variable functions:

```
$which = "echo";
$which("hello, world"); // does not work
```

# Anonymous Functions

Some PHP functions do part of their work by using a function you provide to them. For example, the `usort()` function uses a function you create and pass to it as a parameter to determine the sort order of the items in an array.

Although you can define a function for such purposes, as shown previously, these functions tend to be localized and temporary. To reflect the transient nature of the callback, create and use an *anonymous function* (also known as a *closure*).

You can create an anonymous function using the normal function definition syntax, but assign it to a variable or pass it directly.

Example 3-6 shows an example using `usort()`.

*Example 3-6. Anonymous functions*

```
$array = array("really long string here, boy", "this", "middling length", "larger");

usort($array, function($a, $b) {
 return strlen($a) - strlen($b);
});

print_r($array);
```

The array is sorted by `usort()` using the anonymous function, in order of string length.

Anonymous functions can use the variables defined in their enclosing scope using the `use` syntax. For example:

```
$array = array("really long string here, boy", "this", "middling length",
"larger");
$sortOption = 'random';

usort($array, function($a, $b) use ($sortOption)
{
 if ($sortOption == 'random') {
 // sort randomly by returning (-1, 0, 1) at random
 return rand(0, 2) - 1;
 }
 else {
 return strlen($a) - strlen($b);
 }
});

print_r($array);
```

Note that incorporating variables from the enclosing scope is not the same as using global variables—global variables are always in the global scope, while incorporating variables allows a closure to use the variables defined in the enclosing scope. Also note that this is not necessarily the same as the scope in which the closure is called. For example:

```
$array = array("really long string here, boy", "this", "middling length",
"larger");
$sortOption = "random";

function sortNonrandom($array)
{
 $sortOption = false;

 usort($array, function($a, $b) use ($sortOption)
 {
 if ($sortOption == "random") {
 // sort randomly by returning (-1, 0, 1) at random
 return rand(0, 2) - 1;
 }
 else {
 return strlen($a) - strlen($b);
 }
 });

 print_r($array);
}

print_r(sortNonrandom($array));
```

In this example, `$array` is sorted normally, rather than randomly—the value of `$sortOption` inside the closure is the value of `$sortOption` in the scope of `sortNonrandom()`, not the value of `$sortOption` in the global scope.

## What's Next

User-defined functions can be confusing to write and complex to debug, so be sure to test them well and to try to limit them to performing one task each. In the next chapter we'll be looking at strings and everything that they entail, which is another complex and potentially confusing topic. Don't get discouraged: remember that we are building strong foundations for writing good, solid, succinct PHP code. Once you have a firm grasp of the key concepts of functions, strings, arrays, and objects, you'll be well on your way to becoming a good PHP developer.

CHAPTER 4

# Strings

Most data you encounter as you program will be sequences of characters, or *strings*. Strings can hold people's names, passwords, addresses, credit card numbers, links to photographs, purchase histories, and more. For that reason, PHP has an extensive selection of functions for working with strings.

This chapter shows the many ways to create strings in your programs, including the sometimes tricky subject of *interpolation* (placing a variable's value into a string), then covers functions for changing, quoting, manipulating, and searching strings. By the end of this chapter, you'll be a string-handling expert.

## Quoting String Constants

There are four ways to write a string literal in your PHP code: using single quotes, double quotes, the *here document* (*heredoc*) format derived from the Unix shell, and its "cousin" *now document* (*nowdoc*). These methods differ in whether they recognize special *escape sequences* that let you encode other characters or interpolate variables.

### Variable Interpolation

When you define a string literal using double quotes or a heredoc, the string is subject to *variable interpolation*. Interpolation is the process of replacing variable names in the string with their contained values. There are two ways to interpolate variables into strings.

The simpler of the two ways is to put the variable name in a double-quoted string or in a heredoc:

```
$who = 'Kilroy';
$where = 'here';
```

```
echo "$who was $where";
Kilroy was here
```

The other way is to surround the variable being interpolated with curly braces. Using this syntax ensures the correct variable is interpolated. The classic use of curly braces is to disambiguate the variable name from any surrounding text:

```
$n = 12;
echo "You are the {$n}th person";
You are the 12th person
```

Without the curly braces, PHP would try to print the value of the `$nth` variable.

Unlike in some shell environments, in PHP, strings are not repeatedly processed for interpolation. Instead, any interpolations in a double-quoted string are processed first and the result is used as the value of the string:

```
$bar = 'this is not printed';
$foo = '$bar'; // single quotes
print("$foo");
$bar
```

## Single-Quoted Strings

Single-quoted strings and nowdocs do not interpolate variables. Thus, the variable name in the following string is not expanded because the string literal in which it occurs is single-quoted:

```
$name = 'Fred';
$str = 'Hello, $name'; // single-quoted
echo $str;
Hello, $name
```

The only escape sequences that work in single-quoted strings are `\'`, which puts a single quote in a single-quoted string, and `\\`, which puts a backslash in a single-quoted string. Any other occurrence of a backslash is interpreted simply as a backslash:

```
$name = 'Tim O\'Reilly';// escaped single quote
echo $name;
$path = 'C:\\WINDOWS'; // escaped backslash
echo $path;
$nope = '\n'; // not an escape
echo $nope;
Tim O'Reilly
C:\WINDOWS
\n
```

## Double-Quoted Strings

Double-quoted strings interpolate variables and expand the many PHP escape sequences. Table 4-1 lists the escape sequences recognized by PHP in double-quoted strings.

*Table 4-1. Escape sequences in double-quoted strings*

| Escape sequence | Character represented |
|---|---|
| `\"` | Double quotes |
| `\n` | Newline |
| `\r` | Carriage return |
| `\t` | Tab |
| `\\` | Backslash |
| `\$` | Dollar sign |
| `\{` | Left curly brace |
| `\}` | Right curly brace |
| `\[` | Left square bracket |
| `\]` | Right square bracket |
| `\0` through `\777` | ASCII character represented by octal value |
| `\x0` through `\xFF` | ASCII character represented by hex value |
| `\u` | UTF-8 encoding |

If an unknown escape sequence (i.e., a backslash followed by a character that is not one of those in Table 4-1) is found in a double-quoted string literal, it is ignored (if you have the warning level `E_NOTICE` set, a warning is generated for such unknown escape sequences):

```
$str = "What is \c this?";// unknown escape sequence
echo $str;
What is \c this?
```

## Here Documents

You can easily put multiline strings into your program with a heredoc, as follows:

```
$clerihew = <<< EndOfQuote
Sir Humphrey Davy
Abominated gravy.
He lived in the odium
Of having discovered sodium.

EndOfQuote;
echo $clerihew;
Sir Humphrey Davy
Abominated gravy.
```

```
He lived in the odium
Of having discovered sodium.
```

The `<<<` identifier token tells the PHP parser that you're writing a heredoc. You get to pick the identifier (`EndOfQuote` in this case), and you can put it in double quotes if you wish (e.g., `"EndOfQuote"`). The next line starts the text being quoted by the heredoc, which continues until it reaches a line containing only the identifier. To ensure the quoted text is displayed in the output area exactly as you've laid it out, turn on plain-text mode by adding this command at the top of your code file:

```
header('Content-Type: text/plain;');
```

Alternately, if you have control of your server settings, you could set `default_mimetype` to `plain` in the *php.ini* file:

```
default_mimetype = "text/plain"
```

This is not recommended, however, as it puts *all* output from the server in plain-text mode, which would affect the layout of most of your web code.

If you do not set plain-text mode for your heredoc, the default is typically HTML mode, which simply displays the output all on one line.

When using a heredoc for a simple expression, you can put a semicolon after the terminating identifier to end the statement (as shown in the first example). If you are using a heredoc in a more complex expression, however, you'll need to continue the expression on the next line, as shown here:

```
printf(<<< Template
%s is %d years old.
Template
, "Fred", 35);
```

Single and double quotes in a heredoc are preserved:

```
$dialogue = <<< NoMore
"It's not going to happen!" she fumed.
He raised an eyebrow. "Want to bet?"
NoMore;
echo $dialogue;
"It's not going to happen!" she fumed.
He raised an eyebrow. "Want to bet?"
```

As is whitespace:

```
$ws = <<< Enough
 boo
 hoo
Enough;
// $ws = " boo\n hoo";
```

New to PHP 7.3 is the indentation of the heredoc terminator. This allows for more legible formatting in the case of embedded code, as in the following function:

```
function sayIt() {
 $ws = <<< "StufftoSay"
 The quick brown fox
 Jumps over the lazy dog.
 StufftoSay;
return $ws;
}

echo sayIt() ;

    The quick brown fox
    Jumps over the lazy dog.
```

The newline before the trailing terminator is removed, so these two assignments are identical:

```
$s = 'Foo';
// same as
$s = <<< EndOfPointlessHeredoc
Foo
EndOfPointlessHeredoc;
```

If you want a newline to end your heredoc-quoted string, you'll need to add one yourself:

```
$s = <<< End
Foo

End;
```

# Printing Strings

There are four ways to send output to the browser. The `echo` construct lets you print many values at once, while `print()` prints only one value. The `printf()` function builds a formatted string by inserting values into a template. The `print_r()` function is useful for debugging; it prints the contents of arrays, objects, and other things in a more or less human-readable form.

## echo

To put a string into the HTML of a PHP-generated page, use `echo`. While it looks—and for the most part behaves—like a function, `echo` is a language construct. This means that you can omit the parentheses, so the following expressions are equivalent:

```
echo "Printy";
echo("Printy"); // also valid
```

You can specify multiple items to print by separating them with commas:

```
echo "First", "second", "third";
Firstsecondthird
```

It is a parse error to use parentheses when trying to echo multiple values:

```
// this is a parse error
echo("Hello", "world");
```

Because echo is not a true function, you can't use it as part of a larger expression:

```
// parse error
if (echo("test")) {
 echo("It worked!");
}
```

You can easily remedy such errors by using the print() or printf() functions.

## print()

The print() function sends one value (its argument) to the browser:

```
if (print("test\n")) {
 print("It worked!");
}
test
It worked!
```

## printf()

The printf() function outputs a string built by substituting values into a template (the *format string*). It is derived from the function of the same name in the standard C library. The first argument to printf() is the format string. The remaining arguments are the values to be substituted. A % character in the format string indicates a substitution.

### Format modifiers

Each substitution marker in the template consists of a percent sign (%), possibly followed by modifiers from the following list, and ends with a type specifier. (Use %% to get a single percent character in the output.) The modifiers must appear in the order in which they are listed here:

1. A padding specifier denoting the character to use to pad the results to the appropriate string size. Specify 0, a space, or any character prefixed with a single quote. Padding with spaces is the default.
2. A sign. This has a different effect on strings than on numbers. For strings, a minus (-) here forces the string to be left-justified (the default is right-justified). For numbers, a plus (+) here forces positive numbers to be printed with a leading plus sign (e.g., 35 will be printed as +35).

3. The minimum number of characters that this element should contain. If the result would be less than this number of characters, the sign and padding specifier govern how to pad to this length.
4. For floating-point numbers, a precision specifier consisting of a period and a number; this dictates how many decimal digits will be displayed. For types other than double, this specifier is ignored.

#### Type specifiers

The type specifier tells `printf()` what type of data is being substituted. This determines the interpretation of the previously listed modifiers. There are eight types, as listed in Table 4-2.

*Table 4-2. printf() type specifiers*

| Specifier | Meaning |
|---|---|
| `%` | Displays the percent sign. |
| `b` | The argument is an integer and is displayed as a binary number. |
| `c` | The argument is an integer and is displayed as the character with that value. |
| `d` | The argument is an integer and is displayed as a decimal number. |
| `e` | The argument is a double and is displayed in scientific notation. |
| `E` | The argument is a double and is displayed in scientific notation using uppercase letters. |
| `f` | The argument is a floating-point number and is displayed as such in the current locale's format. |
| `F` | The argument is a floating-point number and is displayed as such. |
| `g` | The argument is a double and is displayed either in scientific notation (as with the `%e` type specifier) or as a floating-point number (as with the `%f` type specifier), whichever is shorter. |
| `G` | The argument is a double and is displayed either in scientific notation (as with the `%E` type specifier) or as a floating-point number (as with the `%f` type specifier), whichever is shorter. |
| `o` | The argument is an integer and is displayed as an octal (base-8) number. |
| `s` | The argument is a string and is displayed as such. |
| `u` | The argument is an unsigned integer and is displayed as a decimal number. |
| `x` | The argument is an integer and is displayed as a hexadecimal (base-16) number; lowercase letters are used. |
| `X` | The argument is an integer and is displayed as a hexadecimal (base-16) number; uppercase letters are used. |

The `printf()` function looks outrageously complex to people who aren't C programmers. Once you get used to it, though, you'll find it a powerful formatting tool. Here are some examples:

- A floating-point number to two decimal places:

  ```
  printf('%.2f', 27.452);
  27.45
  ```

- Decimal and hexadecimal output:

```
printf('The hex value of %d is %x', 214, 214);
The hex value of 214 is d6
```

- Padding an integer to three decimal places:

```
printf('Bond. James Bond. %03d.', 7);
Bond. James Bond. 007.
```

- Formatting a date:

```
printf('%02d/%02d/%04d', $month, $day, $year);
02/15/2005
```

- A percentage:

```
printf('%.2f%% Complete', 2.1);
2.10% Complete
```

- Padding a floating-point number:

```
printf('You\'ve spent $%5.2f so far', 4.1);
You've spent $ 4.10 so far
```

The `sprintf()` function takes the same arguments as `printf()` but returns the built-up string instead of printing it. This lets you save the string in a variable for later use:

```
$date = sprintf("%02d/%02d/%04d", $month, $day, $year);
// now we can interpolate $date wherever we need a date
```

## print_r() and var_dump()

The `print_r()` function intelligently displays what is passed to it, rather than casting everything to a string, as `echo` and `print()` do. Strings and numbers are simply printed. Arrays appear as parenthesized lists of keys and values, prefaced by `Array`:

```
$a = array('name' => 'Fred', 'age' => 35, 'wife' => 'Wilma');
print_r($a);
Array
(
 [name] => Fred
 [age] => 35
 [wife] => Wilma)
```

Using `print_r()` on an array moves the internal iterator to the position of the last element in the array. See Chapter 5 for more on iterators and arrays.

When you `print_r()` an object, you see the word `Object`, followed by the initialized properties of the object displayed as an array:

```
class P {
 var $name = 'nat';
 // ...
}
```

```
$p = new P;
print_r($p);
Object
(
 [name] => nat)
```

Boolean values and NULL are not meaningfully displayed by print_r():

```
print_r(true); // prints "1";
1
print_r(false); // prints "";

print_r(null); // prints "";
```

For this reason, var_dump() is preferred over print_r() for debugging. The var_dump() function displays any PHP value in a human-readable format:

```
var_dump(true);
var_dump(false);
var_dump(null);
var_dump(array('name' => "Fred", 'age' => 35));
class P {
 var $name = 'Nat';
 // ...
}
$p = new P;
var_dump($p);
bool(true)
bool(false)
bool(null)
array(2) {
 ["name"]=>
 string(4) "Fred"
 ["age"]=>
 int(35)
}
object(p)(1) {
 ["name"]=>
 string(3) "Nat"
}
```

Beware of using print_r() or var_dump() on a recursive structure such as $GLOBALS (which has an entry for GLOBALS that points back to itself). The print_r() function loops infinitely, while var_dump() cuts off after visiting the same element three times.

## Accessing Individual Characters

The strlen() function returns the number of characters in a string:

```
$string = 'Hello, world';
$length = strlen($string); // $length is 12
```

You can use the string offset syntax on a string to address individual characters:

```
$string = 'Hello';
for ($i=0; $i < strlen($string); $i++) {
 printf("The %dth character is %s\n", $i, $string{$i});
}
The 0th character is H
The 1th character is e
The 2th character is l
The 3th character is l
The 4th character is o
```

# Cleaning Strings

Often, the strings we get from files or users need to be cleaned up before we can use them. Two common problems with raw data are the presence of extraneous whitespace and incorrect capitalization (uppercase versus lowercase).

## Removing Whitespace

You can remove leading or trailing whitespace with the `trim()`, `ltrim()`, and `rtrim()` functions:

```
$trimmed = trim(string [, charlist ]);
$trimmed = ltrim(string [, charlist ]);
$trimmed = rtrim(string [, charlist ]);
```

`trim()` returns a copy of *string* with whitespace removed from the beginning and the end. `ltrim()` (the *l* is for *left*) does the same, but removes whitespace only from the start of the string. `rtrim()` (the *r* is for *right*) removes whitespace only from the end of the string. The optional *charlist* argument is a string that specifies all the characters to strip. The default characters to strip are given in Table 4-3.

*Table 4-3. Default characters removed by trim(), ltrim(), and rtrim()*

| Character | ASCII value | Meaning |
|---|---|---|
| `" "` | 0x20 | Space |
| `"\t"` | 0x09 | Tab |
| `"\n"` | 0x0A | Newline (line feed) |
| `"\r"` | 0x0D | Carriage return |
| `"\0"` | 0x00 | NUL-byte |
| `"\x0B"` | 0x0B | Vertical tab |

For example:

```
$title = " Programming PHP \n";
$str1 = ltrim($title); // $str1 is "Programming PHP \n"
```

```
$str2 = rtrim($title); // $str2 is " Programming PHP"
$str3 = trim($title); // $str3 is "Programming PHP"
```

Given a line of tab-separated data, use the *charlist* argument to remove leading or trailing whitespace without deleting the tabs:

```
$record = " Fred\tFlintstone\t35\tWilma\t \n";
$record = trim($record, " \r\n\0\x0B");
// $record is "Fred\tFlintstone\t35\tWilma"
```

## Changing Case

PHP has several functions for changing the case of strings: `strtolower()` and `strtoupper()` operate on entire strings, `ucfirst()` operates only on the first character of the string, and `ucwords()` operates on the first character of each word in the string. Each function takes a string to operate on as an argument and returns a copy of that string, appropriately changed. For example:

```
$string1 = "FRED flintstone";
$string2 = "barney rubble";
print(strtolower($string1));
print(strtoupper($string1));
print(ucfirst($string2));
print(ucwords($string2));
fred flintstone
FRED FLINTSTONE
Barney rubble
Barney Rubble
```

If you've got a mixed-case string that you want to convert to "title case," where the first letter of each word is in uppercase and the rest of the letters are in lowercase (and you're not sure what case the string is in to begin with), use a combination of `strtolower()` and `ucwords()`:

```
print(ucwords(strtolower($string1)));
Fred Flintstone
```

# Encoding and Escaping

Because PHP programs often interact with HTML pages, web addresses (URLs), and databases, there are functions to help you work with those types of data. HTML, web addresses, and database commands are all strings, but they each require different characters to be escaped in different ways. For instance, a space in a web address must be written as `%20`, while a literal less-than sign (<) in an HTML document must be written as `<`. PHP has a number of built-in functions to convert to and from these encodings.

## HTML

Special characters in HTML are represented by *entities* such as `&` (&) and `<` (<). There are two PHP functions that turn special characters in a string into their entities: one for removing HTML tags, and one for extracting only meta tags.

### Entity-quoting all special characters

The `htmlentities()` function changes all characters with HTML entity equivalents into those equivalents (with the exception of the space character). This includes the less-than sign (<), the greater-than sign (>), the ampersand (&), and accented characters.

For example:

```
$string = htmlentities("Einstürzende Neubauten");
echo $string;
Einstürzende Neubauten
```

The entity-escaped version, `ü` (seen by viewing the source), correctly displays as ü in the rendered web page. As you can see, the space has not been turned into ` `.

The `htmlentities()` function actually takes up to three arguments:

```
$output = htmlentities(input, flags, encoding);
```

The *encoding* parameter, if given, identifies the character set. The default is "UTF-8." The *flags* parameter controls whether single and double quotes are turned into their entity forms. `ENT_COMPAT` (the default) converts only double quotes, `ENT_QUOTES` converts both types of quotes, and `ENT_NOQUOTES` converts neither. There is no option to convert only single quotes. For example:

```
$input = <<< End
"Stop pulling my hair!" Jane's eyes flashed.<p>
End;

$double = htmlentities($input);
// &quot;Stop pulling my hair!&quot; Jane's eyes flashed.&lt;p&gt;

$both = htmlentities($input, ENT_QUOTES);
// &quot;Stop pulling my hair!&quot; Jane&#039;s eyes flashed.&lt;p&gt;

$neither = htmlentities($input, ENT_NOQUOTES);
// "Stop pulling my hair!" Jane's eyes flashed.&lt;p&gt;
```

### Entity-quoting only HTML syntax characters

The `htmlspecialchars()` function converts the smallest set of entities possible to generate valid HTML. The following entities are converted:

- Ampersands (&) are converted to &
- Double quotes (") are converted to "
- Single quotes (') are converted to ' (if ENT_QUOTES is on, as described for htmlentities())
- Less-than signs (<) are converted to <
- Greater-than signs (>) are converted to >

If you have an application that displays data that a user has entered in a form, you need to run that data through htmlspecialchars() before displaying or saving it. If you don't, and the user enters a string like "angle < 30" or "sturm & drang", the browser will think the special characters are HTML, resulting in a garbled page.

Like htmlentities(), htmlspecialchars() can take up to three arguments:

```
$output = htmlspecialchars(input, [flags, [encoding]]);
```

The *flags* and *encoding* arguments have the same meaning that they do for htmlentities().

There are no functions specifically for converting back from the entities to the original text, because this is rarely needed. There is a relatively simple way to do this, though. Use the get_html_translation_table() function to fetch the translation table used by either of these functions in a given quote style. For example, to get the translation table that htmlentities() uses, do this:

```
$table = get_html_translation_table(HTML_ENTITIES);
```

To get the table for htmlspecialchars() in ENT_NOQUOTES mode, use:

```
$table = get_html_translation_table(HTML_SPECIALCHARS, ENT_NOQUOTES);
```

A nice trick is to use this translation table, flip it using array_flip(), and feed it to strtr() to apply it to a string, thereby effectively doing the reverse of htmlentities():

```
$str = htmlentities("Einstürzende Neubauten"); // now it is encoded

$table = get_html_translation_table(HTML_ENTITIES);
$revTrans = array_flip($table);

echo strtr($str, $revTrans); // back to normal
Einstürzende Neubauten
```

You can, of course, also fetch the translation table, add whatever other translations you want to it, and then do the strtr(). For example, if you wanted htmlentities() to also encode each space to  , you would do:

```
$table = get_html_translation_table(HTML_ENTITIES);
$table[' '] = ' ';
$encoded = strtr($original, $table);
```

### Removing HTML tags

The `strip_tags()` function removes HTML tags from a string:

```
$input = '<p>Howdy, &quot;Cowboy&quot;</p>';
$output = strip_tags($input);
// $output is 'Howdy, &quot;Cowboy&quot;'
```

The function may take a second argument that specifies a string of tags to leave in the string. List only the opening forms of the tags. The closing forms of tags listed in the second parameter are also preserved:

```
$input = 'The <b>bold</b> tags will <i>stay</i><p>';
$output = strip_tags($input, '<b>');
// $output is 'The <b>bold</b> tags will stay'
```

Attributes in preserved tags are not changed by `strip_tags()`. Because attributes such as `style` and `onmouseover` can affect the look and behavior of web pages, preserving some tags with `strip_tags()` won't necessarily remove the potential for abuse.

### Extracting meta tags

The `get_meta_tags()` function returns an array of the meta tags for an HTML page, specified as a local filename or URL. The name of the meta tag (`keywords`, `author`, `description`, etc.) becomes the key in the array, and the content of the meta tag becomes the corresponding value:

```
$metaTags = get_meta_tags('http://www.example.com/');
echo "Web page made by {$metaTags['author']}";
Web page made by John Doe
```

The general form of the function is:

```
$array = get_meta_tags(filename [, use_include_path]);
```

Pass a `true` value for *use_include_path* to let PHP attempt to open the file using the standard include path.

## URLs

PHP provides functions to convert to and from URL encoding, which allows you to build and decode URLs. There are actually two types of URL encoding, which differ in how they treat spaces. The first (specified by RFC 3986) treats a space as just another illegal character in a URL and encodes it as `%20`. The second (implementing

the application/x-www-form-urlencoded system) encodes a space as a + and is used in building query strings.

Note that you don't want to use these functions on a complete URL, such as *http://www.example.com/hello*, as they will escape the colons and slashes to produce:

```
http%3A%2F%2Fwww.example.com%2Fhello
```

Encode only partial URLs (the bit after *http://www.example.com/*) and add the protocol and domain name later.

### RFC 3986 encoding and decoding

To encode a string according to the URL conventions, use rawurlencode():

```
$output = rawurlencode(input);
```

This function takes a string and returns a copy with illegal URL characters encoded in the %dd convention.

If you are dynamically generating hypertext references for links in a page, you need to convert them with rawurlencode():

```
$name = "Programming PHP";
$output = rawurlencode($name);
echo "http://localhost/{$output}";
http://localhost/Programming%20PHP
```

The rawurldecode() function decodes URL-encoded strings:

```
$encoded = 'Programming%20PHP';
echo rawurldecode($encoded);
Programming PHP
```

### Query-string encoding

The urlencode() and urldecode() functions differ from their raw counterparts only in that they encode spaces as plus signs (+) instead of as the sequence %20. This is the format for building query strings and cookie values. These functions can be useful in supplying form-like URLs in the HTML. PHP automatically decodes query strings and cookie values, so you don't need to use these functions to process those values. The functions are useful for generating query strings:

```
$baseUrl = 'http://www.google.com/q=';
$query = 'PHP sessions -cookies';
$url = $baseUrl . urlencode($query);
echo $url;

http://www.google.com/q=PHP+sessions+-cookies
```

## SQL

Most database systems require that string literals in your SQL queries be escaped. SQL's encoding scheme is pretty simple—single quotes, double quotes, NUL-bytes, and backslashes need to be preceded by a backslash. The `addslashes()` function adds these slashes, and the `stripslashes()` function removes them:

```
$string = <<< EOF
"It's never going to work," she cried,
as she hit the backslash (\) key.
EOF;
$string = addslashes($string);
echo $string;
echo stripslashes($string);
\"It\'s never going to work,\" she cried,
as she hit the backslash (\\) key.
"It's never going to work," she cried,
as she hit the backslash (\) key.
```

## C-String Encoding

The `addcslashes()` function escapes arbitrary characters by placing backslashes before them. With the exception of the characters in Table 4-4, characters with ASCII values less than 32 or above 126 are encoded with their octal values (e.g., `"\002"`). The `addcslashes()` and `stripcslashes()` functions are used with nonstandard database systems that have their own ideas of which characters need to be escaped.

*Table 4-4. Single-character escapes recognized by addcslashes() and stripcslashes()*

| ASCII value | Encoding |
|---|---|
| 7 | \a |
| 8 | \b |
| 9 | \t |
| 10 | \n |
| 11 | \v |
| 12 | \f |
| 13 | \r |

Call `addcslashes()` with two arguments—the string to encode and the characters to escape:

```
$escaped = addcslashes(string, charset);
```

Specify a range of characters to escape with the `".."` construct:

```
echo addcslashes("hello\tworld\n", "\x00..\x1fz..\xff");
hello\tworld\n
```

Beware of specifying '0', 'a', 'b', 'f', 'n', 'r', 't', or 'v' in the character set, as they will be turned into '\0', '\a', and so on. These escapes are recognized by C and PHP and may cause confusion.

`stripcslashes()` takes a string and returns a copy with the escapes expanded:

```
$string = stripcslashes(escaped);
```

For example:

```
$string = stripcslashes('hello\tworld\n');
// $string is "hello\tworld\n"
```

# Comparing Strings

PHP has two operators and six functions for comparing strings to each other.

## Exact Comparisons

You can compare two strings for equality with the == and === operators. These operators differ in how they deal with nonstring operands. The == operator casts string operands to numbers, so it reports that `3` and `"3"` are equal. Due to the rules for casting strings to numbers, it would also report that `3` and `"3b"` are equal, as only the portion of the string up to a non-number character is used in casting. The === operator does not cast, and returns `false` if the data types of the arguments differ:

```
$o1 = 3;
$o2 = "3";

if ($o1 == $o2) {
 echo("== returns true<br>");
}
if ($o1 === $o2) {
 echo("=== returns true<br>");
}
== returns true
```

The comparison operators (<, <=, >, >=) also work on strings:

```
$him = "Fred";
$her = "Wilma";

if ($him < $her) {
 print "{$him} comes before {$her} in the alphabet.\n";
}
Fred comes before Wilma in the alphabet
```

However, the comparison operators give unexpected results when comparing strings and numbers:

```
$string = "PHP Rocks";
$number = 5;

if ($string < $number) {
 echo("{$string} < {$number}");
}
PHP Rocks < 5
```

When one argument to a comparison operator is a number, the other argument is cast to a number. This means that `"PHP Rocks"` is cast to a number, giving `0` (since the string does not start with a number). Because 0 is less than 5, PHP prints `"PHP Rocks < 5"`.

To explicitly compare two strings as strings, casting numbers to strings if necessary, use the `strcmp()` function:

```
$relationship = strcmp(string_1, string_2);
```

The function returns a number less than 0 if *string_1* sorts before *string_2*, greater than 0 if *string_2* sorts before *string_1*, or 0 if they are the same:

```
$n = strcmp("PHP Rocks", 5);
echo($n);
1
```

A variation on `strcmp()` is `strcasecmp()`, which converts strings to lowercase before comparing them. Its arguments and return values are the same as those for `strcmp()`:

```
$n = strcasecmp("Fred", "frED"); // $n is 0
```

Another variation on string comparison is to compare only the first few characters of the string. The `strncmp()` and `strncasecmp()` functions take an additional argument, the initial number of characters to use for the comparisons:

```
$relationship = strncmp(string_1, string_2, len);
$relationship = strncasecmp(string_1, string_2, len);
```

The final variation on these functions is *natural-order* comparison with `strnatcmp()` and `strnatcasecmp()`, which take the same arguments as `strcmp()` and return the same kinds of values. Natural-order comparison identifies numeric portions of the strings being compared and sorts the string parts separately from the numeric parts.

Table 4-5 shows strings in natural order and ASCII order.

*Table 4-5. Natural order versus ASCII order*

| Natural order | ASCII order |
| --- | --- |
| pic1.jpg | pic1.jpg |
| pic5.jpg | pic10.jpg |
| pic10.jpg | pic5.jpg |
| pic50.jpg | pic50.jpg |

## Approximate Equality

PHP provides several functions that let you test whether two strings are approximately equal—`soundex()`, `metaphone()`, `similar_text()`, and `levenshtein()`:

```
$soundexCode = soundex($string);
$metaphoneCode = metaphone($string);
$inCommon = similar_text($string_1, $string_2 [, $percentage ]);
$similarity = levenshtein($string_1, $string_2);
$similarity = levenshtein($string_1, $string_2 [, $cost_ins, $cost_rep,
$cost_del ]);
```

The Soundex and Metaphone algorithms each yield a string that represents roughly how a word is pronounced in English. To see whether two strings are approximately equal with these algorithms, compare their pronunciations. You can compare Soundex values only to Soundex values and Metaphone values only to Metaphone values. The Metaphone algorithm is generally more accurate, as the following example demonstrates:

```
$known = "Fred";
$query = "Phred";

if (soundex($known) == soundex($query)) {
 print "soundex: {$known} sounds like {$query}<br>";
}
else {
 print "soundex: {$known} doesn't sound like {$query}<br>";
}

if (metaphone($known) == metaphone($query)) {
 print "metaphone: {$known} sounds like {$query}<br>";
}
else {
 print "metaphone: {$known} doesn't sound like {$query}<br>";
}
soundex: Fred doesn't sound like Phred
metaphone: Fred sounds like Phred
```

The `similar_text()` function returns the number of characters that its two string arguments have in common. The third argument, if present, is a variable in which to store the commonality as a percentage:

```
$string1 = "Rasmus Lerdorf";
$string2 = "Razmus Lehrdorf";
$common = similar_text($string1, $string2, $percent);
printf("They have %d chars in common (%.2f%%).", $common, $percent);
They have 13 chars in common (89.66%).
```

The Levenshtein algorithm calculates the similarity of two strings based on how many characters you must add, substitute, or remove to make them the same. For

instance, "cat" and "cot" have a Levenshtein distance of 1, because you need to change only one character (the "a" to an "o") to make them the same:

```
$similarity = levenshtein("cat", "cot"); // $similarity is 1
```

This measure of similarity is generally quicker to calculate than that used by the `similar_text()` function. Optionally, you can pass three values to the `levenshtein()` function to individually weight insertions, deletions, and replacements—for instance, to compare a word against a contraction.

This example excessively weights insertions when comparing a string against its possible contraction, because contractions should never insert characters:

```
echo levenshtein('would not', 'wouldn\'t', 500, 1, 1);
```

# Manipulating and Searching Strings

PHP has many functions to work with strings. The most commonly used functions for searching and modifying strings are those that use regular expressions to describe the string in question. The functions described in this section do not use regular expressions—they are faster than regular expressions, but they work only when you're looking for a fixed string (for instance, if you're looking for "12/11/01" rather than "any numbers separated by slashes").

## Substrings

If you know where the data that you are interested in lies in a larger string, you can copy it out with the `substr()` function:

```
$piece = substr(string, start [, length ]);
```

The *start* argument is the position in *string* at which to begin copying, with `0` meaning the start of the string. The *length* argument is the number of characters to copy (the default is to copy until the end of the string). For example:

```
$name = "Fred Flintstone";
$fluff = substr($name, 6, 4); // $fluff is "lint"
$sound = substr($name, 11); // $sound is "tone"
```

To learn how many times a smaller string occurs within a larger one, use `substr_count()`:

```
$number = substr_count(big_string, small_string);
```

For example:

```
$sketch = <<< EndOfSketch
Well, there's egg and bacon; egg sausage and bacon; egg and spam;
egg bacon and spam; egg bacon sausage and spam; spam bacon sausage
and spam; spam egg spam spam bacon and spam; spam sausage spam spam
```

```
bacon spam tomato and spam;
EndOfSketch;
$count = substr_count($sketch, "spam");
print("The word spam occurs {$count} times.");
The word spam occurs 14 times.
```

The `substr_replace()` function permits many kinds of string modifications:

```
$string = substr_replace(original, new, start [, length ]);
```

The function replaces the part of *original* indicated by the *start* (`0` means the start of the string) and *length* values with the string *new*. If no fourth argument is given, `substr_replace()` removes the text from *start* to the end of the string.

For instance:

```
$greeting = "good morning citizen";
$farewell = substr_replace($greeting, "bye", 5, 7);
// $farewell is "good bye citizen"
```

Use a *length* of `0` to insert without deleting:

```
$farewell = substr_replace($farewell, "kind ", 9, 0);
// $farewell is "good bye kind citizen"
```

Use a replacement of `""` to delete without inserting:

```
$farewell = substr_replace($farewell, "", 8);
// $farewell is "good bye"
```

Here's how you can insert at the beginning of the string:

```
$farewell = substr_replace($farewell, "now it's time to say ", 0, 0);
// $farewell is "now it's time to say good bye"'
```

A negative value for *start* indicates the number of characters from the end of the string from which to start the replacement:

```
$farewell = substr_replace($farewell, "riddance", -3);
// $farewell is "now it's time to say good riddance"
```

A negative *length* indicates the number of characters from the end of the string at which to stop deleting:

```
$farewell = substr_replace($farewell, "", -8, -5);
// $farewell is "now it's time to say good dance"
```

## Miscellaneous String Functions

The `strrev()` function takes a string and returns a reversed copy of it:

```
$string = strrev(string);
```

For example:

```
echo strrev("There is no cabal");
labac on si erehT
```

The `str_repeat()` function takes a string and a count and returns a new string consisting of the argument *string* repeated *count* times:

```
$repeated = str_repeat(string, count);
```

For example, to build a crude wavy horizontal rule:

```
echo str_repeat('_.-.', 40);
```

The `str_pad()` function pads one string with another. Optionally, you can say what string to pad with, and whether to pad on the left, right, or both:

```
$padded = str_pad(to_pad, length [, with [, pad_type ]]);
```

The default is to pad on the right with spaces:

```
$string = str_pad('Fred Flintstone', 30);
echo "{$string}:35:Wilma";
Fred Flintstone :35:Wilma
```

The optional third argument is the string to pad with:

```
$string = str_pad('Fred Flintstone', 30, '. ');
echo "{$string}35";
Fred Flintstone. . . . . . . .35
```

The optional fourth argument can be `STR_PAD_RIGHT` (the default), `STR_PAD_LEFT`, or `STR_PAD_BOTH` (to center). For example:

```
echo '[' . str_pad('Fred Flintstone', 30, ' ', STR_PAD_LEFT) . "]\n";
echo '[' . str_pad('Fred Flintstone', 30, ' ', STR_PAD_BOTH) . "]\n";
[ Fred Flintstone]
[ Fred Flintstone ]
```

## Decomposing a String

PHP provides several functions to let you break a string into smaller components. In increasing order of complexity, they are `explode()`, `strtok()`, and `sscanf()`.

### Exploding and imploding

Data often arrives as strings, which must be broken down into an array of values. For instance, you might want to split up the comma-separated fields from a string such as "`Fred,25,Wilma`." In these situations, use the `explode()` function:

```
$array = explode(separator, string [, limit]);
```

The first argument, *separator*, is a string containing the field separator. The second argument, *string*, is the string to split. The optional third argument, *limit*, is the maxi-

mum number of values to return in the array. If the limit is reached, the last element of the array contains the remainder of the string:

```
$input = 'Fred,25,Wilma';
$fields = explode(',', $input);
// $fields is array('Fred', '25', 'Wilma')
$fields = explode(',', $input, 2);
// $fields is array('Fred', '25,Wilma')
```

The `implode()` function does the exact opposite of `explode()`—it creates a large string from an array of smaller strings:

```
$string = implode(separator, array);
```

The first argument, *separator*, is the string to put between the elements of the second argument, *array*. To reconstruct the simple comma-separated value string, simply say:

```
$fields = array('Fred', '25', 'Wilma');
$string = implode(',', $fields); // $string is 'Fred,25,Wilma'
```

The `join()` function is an alias for `implode()`.

### Tokenizing

The `strtok()` function lets you iterate through a string, getting a new chunk (token) each time. The first time you call it, you need to pass two arguments: the string to iterate over and the token separator. For example:

```
$firstChunk = strtok(string, separator);
```

To retrieve the rest of the tokens, repeatedly call `strtok()` with only the separator:

```
$nextChunk = strtok(separator);
```

For instance, consider this invocation:

```
$string = "Fred,Flintstone,35,Wilma";
$token = strtok($string, ",");

while ($token !== false) {
 echo("{$token}<br />");
 $token = strtok(",");
}
Fred
Flintstone
35
Wilma
```

The `strtok()` function returns `false` when there are no more tokens to be returned.

Call `strtok()` with two arguments to reinitialize the iterator. This restarts the tokenizer from the start of the string.

### sscanf()

The `sscanf()` function decomposes a string according to a `printf()`-like template:

```
$array = sscanf(string, template);
$count = sscanf(string, template, var1, ... );
```

If used without the optional variables, `sscanf()` returns an array of fields:

```
$string = "Fred\tFlintstone (35)";
$a = sscanf($string, "%s\t%s (%d)");
print_r($a);
Array
(
 [0] => Fred
 [1] => Flintstone
 [2] => 35)
```

Pass references to variables to have the fields stored in those variables. The number of fields assigned is returned:

```
$string = "Fred\tFlintstone (35)";
$n = sscanf($string, "%s\t%s (%d)", $first, $last, $age);
echo "Matched {$n} fields: {$first} {$last} is {$age} years old";
Matched 3 fields: Fred Flintstone is 35 years old
```

## String-Searching Functions

Several functions find a string or character within a larger string. They come in three families: `strpos()` and `strrpos()`, which return a position; `strstr()`, `strchr()`, and friends, which return the string they find; and `strspn()` and `strcspn()`, which return how much of the start of the string matches a mask.

In all cases, if you specify a number as the "string" to search for, PHP treats that number as the ordinal value of the character to search for. Thus, these function calls are identical because 44 is the ASCII value of the comma:

```
$pos = strpos($large, ","); // find first comma
$pos = strpos($large, 44); // also find first comma
```

All the string-searching functions return `false` if they can't find the substring you specified. If the substring occurs at the beginning of the string, the functions return `0`. Because `false` casts to the number `0`, always compare the return value with `===` when testing for failure:

```
if ($pos === false) {
 // wasn't found
}
else {
 // was found, $pos is offset into string
}
```

### Searches returning position

The `strpos()` function finds the first occurrence of a small string in a larger string:

```
$position = strpos(large_string, small_string);
```

If the small string isn't found, `strpos()` returns `false`.

The `strrpos()` function finds the last occurrence of a character in a string. It takes the same arguments and returns the same type of value as `strpos()`.

For instance:

```
$record = "Fred,Flintstone,35,Wilma";
$pos = strrpos($record, ","); // find last comma
echo("The last comma in the record is at position {$pos}");
The last comma in the record is at position 18
```

### Searches returning rest of string

The `strstr()` function finds the first occurrence of a small string in a larger string and returns from that small string on. For instance:

```
$record = "Fred,Flintstone,35,Wilma";
$rest = strstr($record, ","); // $rest is ",Flintstone,35,Wilma"
```

The variations on `strstr()` are:

`stristr()`
: Case-insensitive `strstr()`

`strchr()`
: Alias for `strstr()`

`strrchr()`
: Finds last occurrence of a character in a string

As with `strrpos()`, `strrchr()` searches backward in the string, but only for a single character, not for an entire string.

### Searches using masks

If you thought `strrchr()` was esoteric, you haven't seen anything yet. The `strspn()` and `strcspn()` functions tell you how many characters at the beginning of a string are composed of certain characters:

```
$length = strspn(string, charset);
```

For example, this function tests whether a string holds an octal number:

```
function isOctal($str)
{
```

```
 return strspn($str, '01234567') == strlen($str);
}
```

The *c* in `strcspn()` stands for *complement*—it tells you how much of the start of the string is not composed of the characters in the character set. Use it when the number of interesting characters is greater than the number of uninteresting characters. For example, this function tests whether a string has any NUL-bytes, tabs, or carriage returns:

```
function hasBadChars($str)
{
 return strcspn($str, "\n\t\0") != strlen($str);
}
```

### Decomposing URLs

The `parse_url()` function returns an array of components of a URL:

```
$array = parse_url(url);
```

For example:

```
$bits = parse_url("http://me:secret@example.com/cgi-bin/board?user=fred");
print_r($bits);

Array
(
 [scheme] => http
 [host] => example.com
 [user] => me
 [pass] => secret
 [path] => /cgi-bin/board
 [query] => user=fred)
```

The possible keys of the hash are `scheme`, `host`, `port`, `user`, `pass`, `path`, `query`, and `fragment`.

## Regular Expressions

If you need more complex searching functionality than the previous methods provide, you can use a regular expression—a string that represents a *pattern*. The regular expression functions compare that pattern to another string and see if any of the string matches the pattern. Some functions tell you whether there was a match, while others make changes to the string.

There are three uses for regular expressions: matching, which can also be used to extract information from a string; substituting new text for matching text; and splitting a string into an array of smaller chunks. PHP has functions for all. For instance, `preg_match()` does a regular expression match.

Perl has long been considered the benchmark for powerful regular expressions. PHP uses a C library called *pcre* to provide almost complete support for Perl's arsenal of regular expression features. Perl regular expressions act on arbitrary binary data, so you can safely match with patterns or strings that contain the NUL-byte (`\x00`).

## The Basics

Most characters in a regular expression are literal characters, meaning that they match only themselves. For instance, if you search for the regular expression `"/cow/"` in the string `"Dave was a cowhand"`, you get a match because `"cow"` occurs in that string.

Some characters have special meanings in regular expressions. For instance, a caret (`^`) at the beginning of a regular expression indicates that it must match the beginning of the string (or, more precisely, *anchors* the regular expression to the beginning of the string):

```
preg_match("/^cow/", "Dave was a cowhand"); // returns false
preg_match("/^cow/", "cowabunga!"); // returns true
```

Similarly, a dollar sign (`$`) at the end of a regular expression means that it must match the end of the string (i.e., anchors the regular expression to the end of the string):

```
preg_match("/cow$/", "Dave was a cowhand"); // returns false
preg_match("/cow$/", "Don't have a cow"); // returns true
```

A period (`.`) in a regular expression matches any single character:

```
preg_match("/c.t/", "cat"); // returns true
preg_match("/c.t/", "cut"); // returns true
preg_match("/c.t/", "c t"); // returns true
preg_match("/c.t/", "bat"); // returns false
preg_match("/c.t/", "ct"); // returns false
```

If you want to match one of these special characters (called a *metacharacter*), you have to escape it with a backslash:

```
preg_match("/\$5.00/", "Your bill is $5.00 exactly"); // returns true
preg_match("/$5.00/", "Your bill is $5.00 exactly"); // returns false
```

Regular expressions are case-sensitive by default, so the regular expression `"/cow/"` doesn't match the string `"COW"`. If you want to perform a case-insensitive match, you specify a flag to indicate that (as you'll see later in this chapter).

So far, we haven't done anything we couldn't have done with the string functions we've already seen, like `strstr()`. The real power of regular expressions comes from their ability to specify abstract patterns that can match many different character sequences. You can specify three basic types of abstract patterns in a regular expression:

- A set of acceptable characters that can appear in the string (e.g., alphabetic characters, numeric characters, specific punctuation characters)
- A set of alternatives for the string (e.g., "com", "edu", "net", or "org")
- A repeating sequence in the string (e.g., at least one but not more than five numeric characters)

These three kinds of patterns can be combined in countless ways to create regular expressions that match such things as valid phone numbers and URLs.

## Character Classes

To specify a set of acceptable characters in your pattern, you can either build a character class yourself or use a predefined one. You can build your own character class by enclosing the acceptable characters in square brackets:

```
preg_match("/c[aeiou]t/", "I cut my hand"); // returns true
preg_match("/c[aeiou]t/", "This crusty cat"); // returns true
preg_match("/c[aeiou]t/", "What cart?"); // returns false
preg_match("/c[aeiou]t/", "14ct gold"); // returns false
```

The regular expression engine finds a "c", then checks that the next character is one of "a", "e", "i", "o", or "u". If it isn't a vowel, the match fails and the engine goes back to looking for another "c". If a vowel is found, the engine checks that the next character is a "t". If it is, the engine is at the end of the match and returns true. If the next character isn't a "t", the engine goes back to looking for another "c".

You can negate a character class with a caret (^) at the start:

```
preg_match("/c[^aeiou]t/", "I cut my hand"); // returns false
preg_match("/c[^aeiou]t/", "Reboot chthon"); // returns true
preg_match("/c[^aeiou]t/", "14ct gold"); // returns false
```

In this case, the regular expression engine is looking for a "c" followed by a character that isn't a vowel, followed by a "t".

You can define a range of characters with a hyphen (-). This simplifies character classes like "all letters" and "all digits":

```
preg_match("/[0-9]%/", "we are 25% complete"); // returns true
preg_match("/[0123456789]%/", "we are 25% complete"); // returns true
preg_match("/[a-z]t/", "11th"); // returns false
preg_match("/[a-z]t/", "cat"); // returns true
preg_match("/[a-z]t/", "PIT"); // returns false
preg_match("/[a-zA-Z]!/", "11!"); // returns false
preg_match("/[a-zA-Z]!/", "stop!"); // returns true
```

When you are specifying a character class, some special characters lose their meaning, while others take on new meanings. In particular, the $ anchor and the period

lose their meaning in a character class, while the ^ character is no longer an anchor but negates the character class if it is the first character after the open bracket. For instance, [^\]] matches any nonclosing bracket character, while [$.^] matches any dollar sign, period, or caret.

The various regular expression libraries define shortcuts for character classes, including digits, alphabetic characters, and whitespace.

## Alternatives

You can use the vertical pipe (|) character to specify alternatives in a regular expression:

```
preg_match("/cat|dog/", "the cat rubbed my legs"); // returns true
preg_match("/cat|dog/", "the dog rubbed my legs"); // returns true
preg_match("/cat|dog/", "the rabbit rubbed my legs"); // returns false
```

The precedence of alternation can be a surprise: "/^cat|dog$/" selects from "^cat" and "dog$", meaning that it matches a line that either starts with "cat" or ends with "dog". If you want a line that contains just "cat" or "dog", you need to use the regular expression "/^(cat|dog)$/".

You can combine character classes and alternation to, for example, check for strings that don't start with a capital letter:

```
preg_match("/^([a-z]|[0-9])/", "The quick brown fox"); // returns false
preg_match("/^([a-z]|[0-9])/", "jumped over"); // returns true
preg_match("/^([a-z]|[0-9])/", "10 lazy dogs"); // returns true
```

## Repeating Sequences

To specify a repeating pattern, you use a *quantifier*. The quantifier goes after the pattern that's repeated and says how many times to repeat that pattern. Table 4-6 shows the quantifiers that are supported by PHP's regular expressions.

*Table 4-6. Regular expression quantifiers*

| Quantifier | Meaning |
|---|---|
| ? | 0 or 1 |
| * | 0 or more |
| + | 1 or more |
| { *n* } | Exactly *n* times |
| { *n* , *m* } | At least *n*, no more than *m* times |
| { *n* ,} | At least *n* times |

To repeat a single character, simply put the quantifier after the character:

```
preg_match("/ca+t/", "caaaaaaat"); // returns true
preg_match("/ca+t/", "ct"); // returns false
preg_match("/ca?t/", "caaaaaaat"); // returns false
preg_match("/ca*t/", "ct"); // returns true
```

With quantifiers and character classes, we can actually do something useful, like matching valid US telephone numbers:

```
preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "303-555-1212"); // returns true
preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "64-9-555-1234"); // returns false
```

## Subpatterns

You can use parentheses to group bits of a regular expression together to be treated as a single unit called a *subpattern*:

```
preg_match("/a (very )+big dog/", "it was a very very big dog"); // returns true
preg_match("/^(cat|dog)$/", "cat"); // returns true
preg_match("/^(cat|dog)$/", "dog"); // returns true
```

The parentheses also cause the substring that matches the subpattern to be captured. If you pass an array as the third argument to a match function, the array is populated with any captured substrings:

```
preg_match("/([0-9]+)/", "You have 42 magic beans", $captured);
// returns true and populates $captured
```

The zeroth element of the array is set to the entire string being matched against. The first element is the substring that matched the first subpattern (if there is one), the second element is the substring that matched the second subpattern, and so on.

## Delimiters

Perl-style regular expressions emulate the Perl syntax for patterns, which means that each pattern must be enclosed in a pair of delimiters. Traditionally, the forward slash (`/`) character is used; for example, `/`*pattern*`/`. However, any nonalphanumeric character other than the backslash character (`\`) can be used to delimit a Perl-style pattern. This is useful for matching strings containing slashes, such as filenames. For example, the following are equivalent:

```
preg_match("/\/usr\/local\//", "/usr/local/bin/perl"); // returns true
preg_match("#/usr/local/#", "/usr/local/bin/perl"); // returns true
```

Parentheses (`()`), curly braces (`{}`), square brackets (`[]`), and angle brackets (`<>`) can be used as pattern delimiters:

```
preg_match("{/usr/local/}", "/usr/local/bin/perl"); // returns true
```

The section "Trailing Options" on page 114 discusses the single-character modifiers you can put after the closing delimiter to modify the behavior of the regular expression engine. A very useful one is x, which makes the regular expression engine strip whitespace and #-marked comments from the regular expression before matching. These two patterns are the same, but one is much easier to read:

```
'/([[:alpha:]]+)\s+\1/'
'/( # start capture
[[:alpha:]]+ # a word
\s+ # whitespace
\1 # the same word again
 ) # end capture
/x'
```

## Match Behavior

The period (.) matches any character except for a newline (\n). The dollar sign ($) matches at the end of the string or, if the string ends with a newline, just before that newline:

```
preg_match("/is (.*)$/", "the key is in my pants", $captured);
// $captured[1] is 'in my pants'
```

## Character Classes

As shown in Table 4-7, Perl-compatible regular expressions define a number of named sets of characters that you can use in character classes. The expansions in Table 4-7 are for English. The actual letters vary from locale to locale.

aEach [: *something* :] class can be used in place of a character in a character class. For instance, to find any character that's a digit, an uppercase letter, or an "at" sign (@), use the following regular expression:

```
[@[:digit:][:upper:]]
```

However, you can't use a character class as the endpoint of a range:

```
preg_match("/[A-[:lower:]]/", "string");// invalid regular expression
```

Some locales consider certain character sequences as if they were a single character—these are called *collating sequences*. To match one of these multicharacter sequences in a character class, enclose it with [. and .]. For example, if your locale has the collating sequence ch, you can match s, t, or ch with this character class:

```
[st[.ch.]]
```

The final extension to character classes is the *equivalence class*, which you specify by enclosing the character within [= and =]. Equivalence classes match characters that have the same collating order, as defined in the current locale. For example, a locale

may define a, á, and ä as having the same sorting precedence. To match any one of them, the equivalence class is [=a=].

*Table 4-7. Character classes*

| Class | Description | Expansion |
|---|---|---|
| [:alnum:] | Alphanumeric characters | [0-9a-zA-Z] |
| [:alpha:] | Alphabetic characters (letters) | [a-zA-Z] |
| [:ascii:] | 7-bit ASCII | [\x01-\x7F] |
| [:blank:] | Horizontal whitespace (space, tab) | [ \t] |
| [:cntrl:] | Control characters | [\x01-\x1F] |
| [:digit:] | Digits | [0-9] |
| [:graph:] | Characters that use ink to print (nonspace, noncontrol) | [^\x01-\x20] |
| [:lower:] | Lowercase letter | [a-z] |
| [:print:] | Printable character (graph class plus space and tab) | [\t\x20-\xFF] |
| [:punct:] | Any punctuation character, such as the period (.) and the semicolon (;) | [-!"#$%&'()*+,./:;<=>?@[\\\]^_'{\|}~] |
| [:space:] | Whitespace (newline, carriage return, tab, space, vertical tab) | [\n\r\t \x0B] |
| [:upper:] | Uppercase letter | [A-Z] |
| [:xdigit:] | Hexadecimal digit | [0-9a-fA-F] |
| \s | Whitespace | [\r\n \t] |
| \S | Nonwhitespace | [^\r\n \t] |
| \w | Word (identifier) character | [0-9A-Za-z_] |
| \W | Nonword (identifier) character | [^0-9A-Za-z_] |
| \d | Digit | [0-9] |
| \D | Nondigit | [^0-9] |

## Anchors

An anchor limits a match to a particular location in the string (anchors do not match actual characters in the target string). Table 4-8 lists the anchors supported by regular expressions.

*Table 4-8. Anchors*

| Anchor | Matches |
|---|---|
| ^ | Start of string |
| $ | End of string |
| [[:<:]] | Start of word |
| [[:>:]] | End of word |
| \b | Word boundary (between \w and \W or at start or end of string) |
| \B | Nonword boundary (between \w and \w, or \W and \W) |

| Anchor | Matches |
| --- | --- |
| \A | Beginning of string |
| \Z | End of string or before \n at end |
| \z | End of string |
| ^ | Start of line (or after \n if /m flag is enabled) |
| $ | End of line (or before \n if /m flag is enabled) |

A word boundary is defined as the point between a whitespace character and an identifier (alphanumeric or underscore) character:

```
preg_match("/[[:<:]]gun[[:>:]]/", "the Burgundy exploded"); // returns false
preg_match("/gun/", "the Burgundy exploded"); // returns true
```

Note that the beginning and end of a string also qualify as word boundaries.

## Quantifiers and Greed

Regular expression quantifiers are typically *greedy*. That is, when faced with a quantifier, the engine matches as much as it can while still satisfying the rest of the pattern. For instance:

```
preg_match("/(<.*>)/", "do <b>not</b> press the button", $match);
// $match[1] is '<b>not</b>'
```

The regular expression matches from the first less-than sign to the last greater-than sign. In effect, the `.*` matches everything after the first less-than sign, and the engine backtracks to make it match less and less until finally there's a greater-than sign to be matched.

This greediness can be a problem. Sometimes you need *minimal (nongreedy) matching*—that is, quantifiers that match as few times as possible to satisfy the rest of the pattern. Perl provides a parallel set of quantifiers that match minimally. They're easy to remember, because they're the same as the greedy quantifiers, but with a question mark (`?`) appended. Table 4-9 shows the corresponding greedy and nongreedy quantifiers supported by Perl-style regular expressions.

*Table 4-9. Greedy and nongreedy quantifiers in Perl-compatible regular expressions*

| Greedy quantifier | Nongreedy quantifier |
| --- | --- |
| ? | ?? |
| * | *? |
| + | +? |
| {m} | {m}? |
| {m,} | {m,}? |
| {m,n} | {m,n}? |

Here's how to match a tag using a nongreedy quantifier:

```
preg_match("/(<.*?>)/", "do <b>not</b> press the button", $match);
// $match[1] is "<b>"
```

Another, faster way is to use a character class to match every non-greater-than character up to the next greater-than sign:

```
preg_match("/(<[^>]*>)/", "do <b>not</b> press the button", $match);
// $match[1] is '<b>'
```

## Noncapturing Groups

If you enclose a part of a pattern in parentheses, the text that matches that subpattern is captured and can be accessed later. Sometimes, though, you want to create a subpattern without capturing the matching text. In Perl-compatible regular expressions, you can do this using the `(?:` *subpattern* `)` construct:

```
preg_match("/(?:ello)(.*)/", "jello biafra", $match);
// $match[1] is " biafra"
```

## Backreferences

You can refer to text captured earlier in a pattern with a *backreference*: `\1` refers to the contents of the first subpattern, `\2` refers to the second, and so on. If you nest subpatterns, the first begins with the first opening parenthesis, the second begins with the second opening parenthesis, and so on.

For instance, this identifies doubled words:

```
preg_match("/([[:alpha:]]+)\s+\1/", "Paris in the the spring", $m);
// returns true and $m[1] is "the"
```

The `preg_match()` function captures at most 99 subpatterns; subpatterns after the 99th are ignored.

## Trailing Options

Perl-style regular expressions let you put single-letter options (flags) after the regular expression pattern to modify the interpretation, or behavior, of the match. For instance, to match case-insensitively, simply use the `i` flag:

```
preg_match("/cat/i", "Stop, Catherine!"); // returns true
```

Table 4-10 shows which Perl modifiers are supported in Perl-compatible regular expressions.

*Table 4-10. Perl flags*

| Modifier | Meaning |
|---|---|
| */regexp*/i | Match case-insensitively |
| */regexp*/s | Make period (.) match any character, *including* newline (\n) |
| */regexp*/x | Remove whitespace and comments from the pattern |
| */regexp*/m | Make caret (^) match after, and dollar sign ($) match before, internal newlines (\n) |
| */regexp*/e | If the replacement string is PHP code, eval() it to get the actual replacement string |

PHP's Perl-compatible regular expression functions also support other modifiers that aren't supported by Perl, as listed in Table 4-11.

*Table 4-11. Additional PHP flags*

| Modifier | Meaning |
|---|---|
| */regexp*/U | Reverses the greediness of the subpattern; * and + now match as little as possible, instead of as much as possible |
| */regexp*/u | Causes pattern strings to be treated as UTF-8 |
| */regexp*/X | Causes a backslash followed by a character with no special meaning to emit an error |
| */regexp*/A | Causes the beginning of the string to be anchored as if the first character of the pattern were ^ |
| */regexp*/D | Causes the $ character to match only at the end of a line |
| */regexp*/S | Causes the expression parser to more carefully examine the structure of the pattern, so it may run slightly faster the next time (such as in a loop) |

It's possible to use more than one option in a single pattern, as demonstrated in the following example:

```
$message = <<< END
To: you@youcorp
From: me@mecorp
Subject: pay up

Pay me or else!
END;

preg_match("/^subject: (.*)/im", $message, $match);

print_r($match);

// output: Array ( [0] => Subject: pay up [1] => pay up )
```

## Inline Options

In addition to specifying pattern-wide options after the closing pattern delimiter, you can specify options within a pattern to have them apply only to part of the pattern. The syntax for this is:

```
(?flags:subpattern)
```

For example, only the word "PHP" is case-insensitive in this example:

```
echo preg_match('/I like (?i:PHP)/', 'I like pHp', $match);
print_r($match) ;
// returns true (echo: 1)
// $match[0] is 'I like pHp'
```

The `i`, `m`, `s`, `U`, `x`, and `X` options can be applied internally in this fashion. You can use multiple options at once:

```
preg_match('/eat (?ix:foo d)/', 'eat FoOD'); // returns true
```

Prefix an option with a hyphen (`-`) to turn it off:

```
echo preg_match('/I like (?-i:PHP)/', 'I like pHp', $match);
print_r($matche) ;
// returns false (echo: 0)
// $match[0] is ''
```

An alternative form enables or disables the flags until the end of the enclosing subpattern or pattern:

```
preg_match('/I like (?i)PHP/', 'I like pHp'); // returns true
preg_match('/I (like (?i)PHP) a lot/', 'I like pHp a lot', $match);
// $match[1] is 'like pHp'
```

Inline flags do not enable capturing. You need an additional set of capturing parentheses to do that.

## Lookahead and Lookbehind

In patterns it's sometimes useful to be able to say "match here if this is next." This is particularly common when you are splitting a string. The regular expression describes the separator, which is not returned. You can use *lookahead* to make sure (without matching it, thus preventing it from being returned) that there's more data after the separator. Similarly, *lookbehind* checks the preceding text.

Lookahead and lookbehind come in two forms: *positive* and *negative*. A positive lookahead or lookbehind says "the next/preceding text must be like this." A negative lookahead or lookbehind indicates "the next/preceding text must not be like this." Table 4-12 shows the four constructs you can use in Perl-compatible patterns. None of these constructs captures text.

*Table 4-12. Lookahead and lookbehind assertions*

| Construct | Meaning |
|---|---|
| `(?=`*subpattern*`)` | Positive lookahead |
| `(?!`*subpattern*`)` | Negative lookahead |
| `(?<=`*subpattern*`)` | Positive lookbehind |
| `(?<!`*subpattern*`)` | Negative lookbehind |

A simple use of positive lookahead is splitting a Unix mbox mail file into individual messages. The word `"From"` starting a line by itself indicates the start of a new message, so you can split the mailbox into messages by specifying the separator as the point where the next text is `"From"` at the start of a line:

```
$messages = preg_split('/(?=^From )/m', $mailbox);
```

A simple use of negative lookbehind is to extract quoted strings that contain quoted delimiters. For instance, here's how to extract a single-quoted string (note that the regular expression is commented using the `x` modifier):

```
$input = <<< END
name = 'Tim O\'Reilly';
END;

$pattern = <<< END
' # opening quote
( # begin capturing
 .*? # the string
 (?<! \\\\ ) # skip escaped quotes
) # end capturing
' # closing quote
END;
preg_match( "($pattern)x", $input, $match);
echo $match[1];
Tim O\'Reilly
```

The only tricky part is that to get a pattern that looks behind to see if the last character was a backslash, we need to escape the backslash to prevent the regular expression engine from seeing `\)`, which would mean a literal close parenthesis. In other words, we have to backslash that backslash: `\\)`. But PHP's string-quoting rules say that `\\` produces a literal single backslash, so we end up requiring *four* backslashes to get one through the regular expression! This is why regular expressions have a reputation for being hard to read.

Perl limits lookbehind to constant-width expressions. That is, the expressions cannot contain quantifiers, and if you use alternation, all the choices must be the same length. The Perl-compatible regular expression engine also forbids quantifiers in lookbehind, but does permit alternatives of different lengths.

## Cut

The rarely used once-only subpattern, or *cut*, prevents worst-case behavior by the regular expression engine on some kinds of patterns. The subpattern is never backed out of once matched.

The common use for the once-only subpattern is when you have a repeated expression that may itself be repeated:

```
/(a+|b+)*\.+/
```

This code snippet takes several seconds to report failure:

```
$p = '/(a+|b+)*\.+$/';
$s = 'abababababbabbbabbaaaaaabbbbabbababababababbba..!';

if (preg_match($p, $s)) {
 echo "Y";
}
else {
 echo "N";
}
```

This is because the regular expression engine tries all the different places to start the match, but has to backtrack out of each one, which takes time. If you know that once something is matched it should never be backed out of, you should mark it with `(?>subpattern)`:

```
$p = '/(?>a+|b+)*\.+$/';
```

The cut never changes the outcome of the match; it simply makes it fail faster.

## Conditional Expressions

A conditional expression is like an `if` statement in a regular expression. The general form is:

```
(?(condition)yespattern)
(?(condition)yespattern|nopattern)
```

If the assertion succeeds, the regular expression engine matches the *yespattern*. With the second form, if the assertion doesn't succeed, the regular expression engine skips the *yespattern* and tries to match the *nopattern*.

The assertion can be one of two types: either a backreference, or a lookahead or lookbehind match. To reference a previously matched substring, the assertion is a number from 1 to 99 (the most backreferences available). The condition uses the pattern in the assertion only if the backreference was matched. If the assertion is not a backreference, it must be a positive or negative lookahead or lookbehind assertion.

## Functions

There are five classes of functions that work with Perl-compatible regular expressions: matching, replacing, splitting, filtering, and a utility function for quoting text.

### Matching

The `preg_match()` function performs Perl-style pattern matching on a string. It's the equivalent of the `m//` operator in Perl. The `preg_match_all()` function takes the same arguments and gives the same return value as the `preg_match()` function, except that it takes a Perl-style pattern instead of a standard pattern:

```
$found = preg_match(pattern, string [, captured ]);
```

For example:

```
preg_match('/y.*e$/', 'Sylvie'); // returns true
preg_match('/y(.*)e$/', 'Sylvie', $m); // $m is array('ylvie', 'lvi')
```

While there's a `preg_match()` function to match case-insensitively, there's no `preg_matchi()` function. Instead, use the `i` flag on the pattern:

```
preg_match('y.*e$/i', 'SyLvIe'); // returns true
```

The `preg_match_all()` function repeatedly matches from where the last match ended, until no more matches can be made:

```
$found = preg_match_all(pattern, string, matches [, order ]);
```

The *order* value, either `PREG_PATTERN_ORDER` or `PREG_SET_ORDER`, determines the layout of *matches*. We'll look at both, using this code as a guide:

```
$string = <<< END
13 dogs
12 rabbits
8 cows
1 goat
END;
preg_match_all('/(\d+) (\S+)/', $string, $m1, PREG_PATTERN_ORDER);
preg_match_all('/(\d+) (\S+)/', $string, $m2, PREG_SET_ORDER);
```

With `PREG_PATTERN_ORDER` (the default), each element of the array corresponds to a particular capturing subpattern. So `$m1[0]` is an array of all the substrings that matched the pattern, `$m1[1]` is an array of all the substrings that matched the first subpattern (the numbers), and `$m1[2]` is an array of all the substrings that matched the second subpattern (the words). The array `$m1` has one more element than it has subpatterns.

With `PREG_SET_ORDER`, each element of the array corresponds to the next attempt to match the whole pattern. So `$m2[0]` is an array of the first set of matches (`'13 dogs'`, `'13'`, `'dogs'`), `$m2[1]` is an array of the second set of matches (`'12 rabbits'`, `'12'`,

'rabbits'), and so on. The array $m2 has as many elements as there were successful matches of the entire pattern.

Example 4-1 fetches the HTML at a particular web address into a string and extracts the URLs from that HTML. For each URL, it generates a link back to the program that will display the URLs at that address.

*Example 4-1. Extracting URLs from an HTML page*

```
<?php
if (getenv('REQUEST_METHOD') == 'POST') {
 $url = $_POST['url'];
}
else {
 $url = $_GET['url'];
}
?>

<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <p>URL: <input type="text" name="url" value="<?php echo $url ?>" /><br />
 <input type="submit">
</form>

<?php
if ($url) {
 $remote = fopen($url, 'r'); {
 $html = fread($remote, 1048576); // read up to 1 MB of HTML
 }
 fclose($remote);

 $urls = '(http|telnet|gopher|file|wais|ftp)';
 $ltrs = '\w';
 $gunk = '/#~:.?+=&%@!\-';
 $punc = '.:?\-';
 $any = "{$ltrs}{$gunk}{$punc}";

 preg_match_all("{
 \b # start at word boundary
 {$urls}: # need resource and a colon
 [{$any}] +? # followed by one or more of any valid
 # characters—but be conservative
 # and take only what you need
 (?= # the match ends at
 [{$punc}]* # punctuation
 [^{$any}] # followed by a non-URL character
 | # or
 \$ # the end of the string
 )
 }x", $html, $matches);

 printf("I found %d URLs<P>\n", sizeof($matches[0]));
```

```
 foreach ($matches[0] as $u) {
 $link = $_SERVER['PHP_SELF'] . '?url=' . urlencode($u);
 echo "<a href=\"{$link}\">{$u}</a><br />\n";
 }
}
```

### Replacing

The `preg_replace()` function behaves like the search-and-replace operation in your text editor. It finds all occurrences of a pattern in a string and changes those occurrences to something else:

```
$new = preg_replace(pattern, replacement, subject [, limit ]);
```

The most common usage has all the argument strings except for the integer *limit*. The limit is the maximum number of occurrences of the pattern to replace (the default, and the behavior when a limit of `-1` is passed, is all occurrences):

```
$better = preg_replace('/<.*?>/', '!', 'do <b>not</b> press the button');
// $better is 'do !not! press the button'
```

Pass an array of strings as *subject* to make the substitution on all of them. The new strings are returned from `preg_replace()`:

```
$names = array('Fred Flintstone',
 'Barney Rubble',
 'Wilma Flintstone',
 'Betty Rubble');
$tidy = preg_replace('/(\w)\w* (\w+)/', '\1 \2', $names);
// $tidy is array ('F Flintstone', 'B Rubble', 'W Flintstone', 'B Rubble')
```

To perform multiple substitutions on the same string or array of strings with one call to `preg_replace()`, pass arrays of patterns and replacements:

```
$contractions = array("/don't/i", "/won't/i", "/can't/i");
$expansions = array('do not', 'will not', 'can not');
$string = "Please don't yell - I can't jump while you won't speak";
$longer = preg_replace($contractions, $expansions, $string);
// $longer is 'Please do not yell - I can not jump while you will not speak';
```

If you give fewer replacements than patterns, text matching the extra patterns is deleted. This is a handy way to delete a lot of things at once:

```
$htmlGunk = array('/<.*?>/', '/&.*?;/');
$html = '&eacute; : <b>very</b> cute';
$stripped = preg_replace($htmlGunk, array(), $html);
// $stripped is ' : very cute'
```

If you give an array of patterns but a single string replacement, the same replacement is used for every pattern:

```
$stripped = preg_replace($htmlGunk, '', $html);
```

The replacement can use backreferences. Unlike backreferences in patterns, though, the preferred syntax for backreferences in replacements is `$1`, `$2`, `$3`, and so on. For example:

```
echo preg_replace('/(\w)\w+\s+(\w+)/', '$2, $1.', 'Fred Flintstone')
Flintstone, F.
```

The `/e` modifier makes `preg_replace()` treat the replacement string as PHP code that returns the actual string to use in the replacement. For example, this converts every Celsius temperature to Fahrenheit:

```
$string = 'It was 5C outside, 20C inside';
echo preg_replace('/(\d+)C\b/e', '$1*9/5+32', $string);
It was 41 outside, 68 inside
```

This more complex example expands variables in a string:

```
$name = 'Fred';
$age = 35;
$string = '$name is $age';
preg_replace('/\$(\w+)/e', '$$1', $string);
```

Each match isolates the name of a variable (`$name`, `$age`). The `$1` in the replacement refers to those names, so the PHP code actually executed is `$name` and `$age`. That code evaluates to the value of the variable, which is what's used as the replacement. Whew!

A variation on `preg_replace()` is `preg_replace_callback()`. This calls a function to get the replacement string. The function is passed an array of matches (the zeroth element is all the text that matched the pattern, the first is the contents of the first captured subpattern, and so on). For example:

```
function titlecase($s)
{
 return ucfirst(strtolower($s[0]));
}

$string = 'goodbye cruel world';
$new = preg_replace_callback('/\w+/', 'titlecase', $string);
echo $new;

Goodbye Cruel World
```

## Splitting

Whereas you use `preg_match_all()` to extract chunks of a string when you know what those chunks are, use `preg_split()` to extract chunks when you know what *separates* the chunks from each other:

```
$chunks = preg_split(pattern, string [, limit [, flags ]]);
```

The *pattern* matches a separator between two chunks. By default, the separators are not returned. The optional *limit* specifies the maximum number of chunks to return (`-1` is the default, which means all chunks). The *flags* argument is a bitwise OR combination of the flags `PREG_SPLIT_NO_EMPTY` (empty chunks are not returned) and `PREG_SPLIT_DELIM_CAPTURE` (parts of the string captured in the pattern are returned).

For example, to extract just the operands from a simple numeric expression, use:

```
$ops = preg_split('{[+*/-]}', '3+5*9/2');
// $ops is array('3', '5', '9', '2')
```

To extract the operands and the operators, use:

```
$ops = preg_split('{([+*/-])}', '3+5*9/2', -1, PREG_SPLIT_DELIM_CAPTURE);
// $ops is array('3', '+', '5', '*', '9', '/', '2')
```

An empty pattern matches at every boundary between characters in the string, and at the start and end of the string. This lets you split a string into an array of characters:

```
$array = preg_split('//', $string);
```

### Filtering an array with a regular expression

The `preg_grep()` function returns those elements of an array that match a given pattern:

```
$matching = preg_grep(pattern, array);
```

For instance, to get only the filenames that end in *.txt*, use:

```
$textfiles = preg_grep('/\.txt$/', $filenames);
```

### Quoting for regular expressions

The `preg_quote()` function creates a regular expression that matches only a given string:

```
$re = preg_quote(string [, delimiter ]);
```

Every character in *string* that has special meaning inside a regular expression (e.g., `*` or `$`) is prefaced with a backslash:

```
echo preg_quote('$5.00 (five bucks)');
\$5\.00 \(five bucks\)
```

The optional second argument is an extra character to be quoted. Usually, you pass your regular expression delimiter here:

```
$toFind = '/usr/local/etc/rsync.conf';
$re = preg_quote($toFind, '/');

if (preg_match("/{$re}/", $filename)) {
 // found it!
}
```

## Differences from Perl Regular Expressions

Although very similar, PHP's implementation of Perl-style regular expressions has a few minor differences from actual Perl regular expressions:

- The NULL character (ASCII 0) is not allowed as a literal character within a pattern string. You can reference it in other ways, however (`\000`, `\x00`, etc.).
- The `\E`, `\G`, `\L`, `\l`, `\Q`, `\u`, and `\U` options are not supported.
- The `(?{` *some perl code* `})` construct is not supported.
- The `/D`, `/G`, `/U`, `/u`, `/A`, and `/X` modifiers are supported.
- The vertical tab `\v` counts as a whitespace character.
- Lookahead and lookbehind assertions cannot be repeated using `*`, `+`, or `?`.
- Parenthesized submatches within negative assertions are not remembered.
- Alternation branches within a lookbehind assertion can be of different lengths.

# What's Next

Now that you know everything there is to know about strings and working with them, the next major part of PHP we'll focus on is arrays. These compound data types will challenge you, but you need to get well acquainted with them, as PHP works with them in many areas. Learning how to add array elements, sort arrays, and deal with multidimensional forms of arrays is essential to being a good PHP developer.

# CHAPTER 5
# Arrays

As we discussed in Chapter 2, PHP supports both scalar and compound data types. In this chapter, we'll discuss one of the compound types: arrays. An *array* is a collection of data values organized as an ordered collection of key-value pairs. It may help to think of an array, in loose terms, like an egg carton. Each compartment of an egg carton can hold an egg, but it travels around as one overall container. And, just as an egg carton doesn't have to contain only eggs (you can put anything in there, like rocks, snowballs, four-leaf clovers, or nuts and bolts), so too an array is not limited to one type of data. It can hold strings, integers, Booleans, and so on. Plus, array compartments can also contain other arrays—but more on that later.

This chapter talks about creating an array, adding and removing elements from an array, and looping over the contents of an array. Because arrays are very common and useful, there are many built-in functions that work with them in PHP. For example, if you want to send email to more than one email address, you'll store the email addresses in an array and then loop through the array, sending the message to the current email address. Also, if you have a form that permits multiple selections, the items the user selected are returned in an array.

## Indexed Versus Associative Arrays

There are two kinds of arrays in PHP: indexed and associative. The keys of an *indexed* array are integers, beginning at 0. Indexed arrays are used when you identify things by their position. *Associative* arrays have strings as keys and behave more like two-column tables. The first column is the key, which is used to access the value.

PHP internally stores all arrays as associative arrays; the only difference between associative and indexed arrays is what the keys happen to be. Some array features are provided mainly for use with indexed arrays because they assume that you have or

want keys that are consecutive integers beginning at 0. In both cases, the keys are unique. In other words, you can't have two elements with the same key, regardless of whether the key is a string or an integer.

PHP arrays have an internal order to their elements that is independent of the keys and values, and there are functions that you can use to traverse the arrays based on this internal order. The order is normally that in which values were inserted into the array, but the sorting functions described later in this chapter let you change the order to one based on keys, values, or anything else you choose.

# Identifying Elements of an Array

Before we look at creating an array, let's look at the structure of an existing array. You can access specific values from an existing array using the array variable's name, followed by the element's key, or *index*, within square brackets:

```
$age['fred']
$shows[2]
```

The key can be either a string or an integer. String values that are equivalent to integer numbers (without leading zeros) are treated as integers. Thus, `$array[3]` and `$array['3']` reference the same element, but `$array['03']` references a different element. Negative numbers are valid keys, but they don't specify positions from the end of the array as they do in Perl.

You don't have to quote single-word strings. For instance, `$age['fred']` is the same as `$age[fred]`. However, it's considered good PHP style to always use quotes, because quoteless keys are indistinguishable from constants. When you use a constant as an unquoted index, PHP uses the value of the constant as the index and emits a warning. This will throw an error in future versions of PHP:

```
$person = array("name" => 'Peter');
print "Hello, {$person[name]}";
// output: Hello, Peter
// this 'works' but emits this warning as well:
Warning: Use of undefined constant name - assumed 'name' (this will throw an
Error in a future version of PHP)
```

You must use quotes if you're using interpolation to build the array index:

```
$person = array("name" => 'Peter');
print "Hello, {$person["name"]}";// output: Hello, Peter (with no warning)
```

Although it's technically optional, you should also quote the key if you're interpolating an array lookup to ensure that you get the value you expect. Consider this example:

```
define('NAME', 'bob');
$person = array("name" => 'Peter');
```

```
echo "Hello, {$person['name']}";
echo "<br/>" ;
echo "Hello, NAME";
echo "<br/>" ;
echo NAME ;
// output:
Hello, Peter
Hello, NAME
bob
```

## Storing Data in Arrays

Storing a value in an array will create the array if it doesn't already exist, but trying to retrieve a value from an array that hasn't been defined won't create the array. For example:

```
// $addresses not defined before this point
echo $addresses[0]; // prints nothing
echo $addresses; // prints nothing

$addresses[0] = "spam@cyberpromo.net";
echo $addresses; // prints "Array"
```

Using simple assignment to initialize an array in your program can lead to code like this:

```
$addresses[0] = "spam@cyberpromo.net";
$addresses[1] = "abuse@example.com";
$addresses[2] = "root@example.com";
```

That's an indexed array, with integer indices beginning at 0. Here's an associative array:

```
$price['gasket'] = 15.29;
$price['wheel'] = 75.25;
$price['tire'] = 50.00;
```

An easier way to initialize an array is to use the `array()` construct, which builds an array from its arguments. This builds an indexed array, and the index values (starting at 0) are created automatically:

```
$addresses = array("spam@cyberpromo.net", "abuse@example.com",
"root@example.com");
```

To create an associative array with `array()`, use the `=>` symbol to separate indices (keys) from values:

```
$price = array(
 'gasket' => 15.29,
 'wheel' => 75.25,
 'tire' => 50.00
);
```

Notice the use of whitespace and alignment. We could have bunched up the code, but it wouldn't have been as easy to read (this is equivalent to the previous code sample), or as easy to add or remove values:

```
$price = array('gasket' => 15.29, 'wheel' => 75.25, 'tire' => 50.00);
```

You can also specify an array using a shorter, alternate syntax:

```
$price = ['gasket' => 15.29, 'wheel' => 75.25, 'tire' => 50.0];
```

To construct an empty array, pass no arguments to `array()`:

```
$addresses = array();
```

You can specify an initial key with => and then a list of values. The values are inserted into the array starting with that key, with subsequent values having sequential keys:

```
$days = array(1 => "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
// 2 is Tue, 3 is Wed, etc.
```

If the initial index is a non-numeric string, subsequent indices are integers beginning at 0. Thus, the following code is probably a mistake:

```
$whoops = array('Fri' => "Black", "Brown", "Green");

// same as
$whoops = array('Fri' => "Black", 0 => "Brown", 1 => "Green");
```

## Appending Values to an Array

To add more values to the end of an existing indexed array, use the `[]` syntax:

```
$family = array("Fred", "Wilma");
$family[] = "Pebbles"; // $family[2] is "Pebbles"
```

This construct assumes the array's indices are numbers and assigns elements into the next available numeric index, starting from 0. Attempting to append to an associative array without appropriate keys is almost always a programmer mistake, but PHP will give the new elements numeric indices without issuing a warning:

```
$person = array('name' => "Fred");
$person[] = "Wilma"; // $person[0] is now "Wilma"
```

## Assigning a Range of Values

The `range()` function creates an array of consecutive integer or character values between and including the two values you pass to it as arguments. For example:

```
$numbers = range(2, 5); // $numbers = array(2, 3, 4, 5);
$letters = range('a', 'z'); // $letters holds the alphabet
$reversedNumbers = range(5, 2); // $reversedNumbers = array(5, 4, 3, 2);
```

Only the first letter of a string argument is used to build the range:

```
range("aaa", "zzz"); // same as range('a','z')
```

## Getting the Size of an Array

The `count()` and `sizeof()` functions are identical in use and effect. They return the number of elements in the array. There is no stylistic preference about which function you use. Here's an example:

```
$family = array("Fred", "Wilma", "Pebbles");
$size = count($family); // $size is 3
```

This function counts only array values that are actually set:

```
$confusion = array( 10 => "ten", 11 => "eleven", 12 => "twelve");
$size = count($confusion); // $size is 3
```

## Padding an Array

To create an array with values initialized to the same content, use `array_pad()`. The first argument to `array_pad()` is the array, the second argument is the minimum number of elements you want the array to have, and the third argument is the value to give any elements that are created. The `array_pad()` function returns a new padded array, leaving its argument (source) array alone.

Here's `array_pad()` in action:

```
$scores = array(5, 10);
$padded = array_pad($scores, 5, 0); // $padded is now array(5, 10, 0, 0, 0)
```

Notice how the new values are appended to the array. If you want the new values added to the start of the array, use a negative second argument:

```
$padded = array_pad($scores, −5, 0); // $padded is now array(0, 0, 0, 5, 10);
```

If you pad an associative array, existing keys will be preserved. New elements will have numeric keys starting at 0.

# Multidimensional Arrays

The values in an array can themselves be arrays. This lets you easily create multidimensional arrays:

```
$row0 = array(1, 2, 3);
$row1 = array(4, 5, 6);
$row2 = array(7, 8, 9);
$multi = array($row0, $row1, $row2);
```

You can refer to elements of multidimensional arrays by appending more square brackets, `[]`:

```
$value = $multi[2][0]; // row 2, column 0. $value = 7
```

To interpolate a lookup of a multidimensional array, you must enclose the entire array lookup in curly braces:

```
echo("The value at row 2, column 0 is {$multi[2][0]}\n");
```

Failing to use the curly braces results in output like this:

```
The value at row 2, column 0 is Array[0]
```

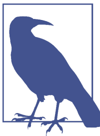
## Extracting Multiple Values

To copy all of an array's values into variables, use the `list()` construct:

```
list ($variable, ...) = $array;
```

The array's values are copied into the listed variables in the array's internal order. By default that's the order in which they were inserted, but the sort functions described later let you change that. Here's an example:

```
$person = array("Fred", 35, "Betty");
list($name, $age, $wife) = $person;
// $name is "Fred", $age is 35, $wife is "Betty"
```



The use of the `list()` function is a common practice for picking up values from a database selection where only one row is returned. This automatically loads the data from the simple query into a series of local variables. Here is an example of selecting two opposing teams from a sports scheduling database:

```
$sql = "SELECT HomeTeam, AwayTeam FROM schedule WHERE
Ident = 7";
$result = mysql_query($sql);
list($hometeam, $awayteam) = mysql_fetch_assoc($result);
```

There is more coverage on databases in Chapter 9.

If you have more values in the array than in the `list()`, the extra values are ignored:

```
$person = array("Fred", 35, "Betty");
list($name, $age) = $person; // $name is "Fred", $age is 35
```

If you have more values in the `list()` than in the array, the extra values are set to `NULL`:

```
$values = array("hello", "world");
list($a, $b, $c) = $values; // $a is "hello", $b is "world", $c is NULL
```

Two or more consecutive commas in the `list()` skip values in the array:

```
$values = range('a', 'e'); // use range to populate the array
list($m, , $n, , $o) = $values; // $m is "a", $n is "c", $o is "e"
```

## Slicing an Array

To extract only a subset of the array, use the `array_slice()` function:

```
$subset = array_slice(array, offset, length);
```

The `array_slice()` function returns a new array consisting of a consecutive series of values from the original array. The *offset* parameter identifies the initial element to copy (`0` represents the first element in the array), and the *length* parameter identifies the number of values to copy. The new array has consecutive numeric keys starting at 0. For example:

```
$people = array("Tom", "Dick", "Harriet", "Brenda", "Jo");
$middle = array_slice($people, 2, 2); // $middle is array("Harriet", "Brenda")
```

It is generally only meaningful to use `array_slice()` on indexed arrays (i.e., those with consecutive integer indices starting at 0):

```
// this use of array_slice() makes no sense
$person = array('name' => "Fred", 'age' => 35, 'wife' => "Betty");
$subset = array_slice($person, 1, 2); // $subset is array(0 => 35, 1 => "Betty")
```

Combine `array_slice()` with `list()` to extract only some values to variables:

```
$order = array("Tom", "Dick", "Harriet", "Brenda", "Jo");
list($second, $third) = array_slice($order, 1, 2);
// $second is "Dick", $third is "Harriet"
```

## Splitting an Array into Chunks

To divide an array into smaller, evenly sized arrays, use the `array_chunk()` function:

```
$chunks = array_chunk(array, size [, preserve_keys]);
```

The function returns an array of the smaller arrays. The third argument, *preserve_keys*, is a Boolean value that determines whether the elements of the new arrays have the same keys as in the original (useful for associative arrays) or new numeric keys starting from 0 (useful for indexed arrays). The default is to assign new keys, as shown here:

```
$nums = range(1, 7);
$rows = array_chunk($nums, 3);
print_r($rows);

Array (
 [0] => Array (
 [0] => 1
 [1] => 2
 [2] => 3
 )
 [1] => Array (
 [0] => 4
```

```
 [1] => 5
 [2] => 6
 )
 [2] => Array (
 [0] => 7
 )
)
```

## Keys and Values

The `array_keys()` function returns an array consisting of only the keys in the array in internal order:

```
$arrayOfKeys = array_keys(array);
```

Here's an example:

```
$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");
$keys = array_keys($person); // $keys is array("name", "age", "wife")
```

PHP also provides a (generally less useful) function to retrieve an array of just the values in an array, `array_values()`:

```
$arrayOfValues = array_values(array);
```

As with `array_keys()`, the values are returned in the array's internal order:

```
$values = array_values($person); // $values is array("Fred", 35, "Wilma");
```

## Checking Whether an Element Exists

To see if an element exists in the array, use the `array_key_exists()` function:

```
if (array_key_exists(key, array)) { ... }
```

The function returns a Boolean value that indicates whether the first argument is a valid key in the array given as the second argument.

It's not sufficient to simply say:

```
if ($person['name']) { ... } // this can be misleading
```

Even if there is an element in the array with the key `name`, its corresponding value might be false (i.e., `0`, `NULL`, or the empty string). Instead, use `array_key_exists()`, as follows:

```
$person['age'] = 0; // unborn?

if ($person['age']) {
 echo "true!\n";
}

if (array_key_exists('age', $person)) {
 echo "exists!\n";
```

```
}

exists!
```

Many people use the `isset()` function instead, which returns `true` if the element exists and is not `NULL`:

```
$a = array(0, NULL, '');

function tf($v)
{
 return $v ? 'T' : 'F';
}

for ($i=0; $i < 4; $i++) {
 printf("%d: %s %s\n", $i, tf(isset($a[$i])), tf(array_key_exists($i, $a)));
}
0: T T
1: F T
2: T T
3: F F
```

## Removing and Inserting Elements in an Array

The `array_splice()` function can remove or insert elements in an array and optionally create another array from the removed elements:

```
$removed = array_splice(array, start [, length [, replacement ] ]);
```

We'll look at `array_splice()` using this array:

```
$subjects = array("physics", "chem", "math", "bio", "cs", "drama", "classics");
```

We can remove the `"math"`, `"bio"`, and `"cs"` elements by telling `array_splice()` to start at position 2 and remove 3 elements:

```
$removed = array_splice($subjects, 2, 3);
// $removed is array("math", "bio", "cs")
// $subjects is array("physics", "chem", "drama", "classics")
```

If you omit the length, `array_splice()` removes to the end of the array:

```
$removed = array_splice($subjects, 2);
// $removed is array("math", "bio", "cs", "drama", "classics")
// $subjects is array("physics", "chem")
```

If you simply want to delete elements from the source array and you don't care about retaining their values, you don't need to store the results of `array_splice()`:

```
array_splice($subjects, 2);
// $subjects is array("physics", "chem");
```

To insert elements where others were removed, use the fourth argument:

```
$new = array("law", "business", "IS");
array_splice($subjects, 4, 3, $new);
// $subjects is array("physics", "chem", "math", "bio", "law", "business", "IS")
```

The size of the replacement array doesn't have to be the same as the number of elements you delete. The array grows or shrinks as needed:

```
$new = array("law", "business", "IS");
array_splice($subjects, 3, 4, $new);
// $subjects is array("physics", "chem", "math", "law", "business", "IS")
```

To insert new elements into the array while pushing existing elements to the right, delete zero elements:

```
$subjects = array("physics", "chem", "math');
$new = array("law", "business");
array_splice($subjects, 2, 0, $new);
// $subjects is array("physics", "chem", "law", "business", "math")
```

Although the examples so far have used an indexed array, `array_splice()` also works on associative arrays:

```
$capitals = array(
 'USA' => "Washington",
 'Great Britain' => "London",
 'New Zealand' => "Wellington",
 'Australia' => "Canberra",
 'Italy' => "Rome",
 'Canada' => "Ottawa"
);

$downUnder = array_splice($capitals, 2, 2); // remove New Zealand and Australia
$france = array('France' => "Paris");

array_splice($capitals, 1, 0, $france); // insert France between USA and GB
```

# Converting Between Arrays and Variables

PHP provides two functions, `extract()` and `compact()`, that convert between arrays and variables. The names of the variables correspond to keys in the array, and the values of the variables become the values in the array. For instance, this array

```
$person = array('name' => "Fred", 'age' => 35, 'wife' => "Betty");
```

can be converted to, or built from, these variables:

```
$name = "Fred";
$age = 35;
$wife = "Betty";
```

## Creating Variables from an Array

The extract() function automatically creates local variables from an array. The indices of the array elements become the variable names:

```
extract($person); // $name, $age, and $wife are now set
```

If a variable created by the extraction has the same name as an existing one, the existing variable's value is overwritten with the one from the array.

You can modify extract()'s behavior by passing a second argument. The Appendix describes the possible values for this second argument. The most useful value is EXTR_PREFIX_ALL, which indicates that the third argument to extract() is a prefix for the variable names that are created. This helps ensure that you create unique variable names when you use extract(). It is good PHP style to always use EXTR_PREFIX_ALL, as shown here:

```
$shape = "round";
$array = array('cover' => "bird", 'shape' => "rectangular");

extract($array, EXTR_PREFIX_ALL, "book");
echo "Cover: {$book_cover}, Book Shape: {$book_shape}, Shape: {$shape}";

Cover: bird, Book Shape: rectangular, Shape: round
```

## Creating an Array from Variables

The compact() function is the reverse of extract(); you pass it the variable names to compact either as separate parameters or in an array. The compact() function creates an associative array whose keys are the variable names and whose values are the variable's values. Any names in the array that do not correspond to actual variables are skipped. Here's an example of compact() in action:

```
$color = "indigo";
$shape = "curvy";
$floppy = "none";

$a = compact("color", "shape", "floppy");
// or
$names = array("color", "shape", "floppy");
$a = compact($names);
```

# Traversing Arrays

The most common task with arrays is to do something with every element—for instance, sending mail to each element of an array of addresses, updating each file in an array of filenames, or adding up each element of an array of prices. There are

several ways to traverse arrays in PHP, and the one you choose will depend on your data and the task you're performing.

## The foreach Construct

The most common way to loop over elements of an array is to use the `foreach` construct:

```
$addresses = array("spam@cyberpromo.net", "abuse@example.com");

foreach ($addresses as $value) {
 echo "Processing {$value}\n";
}
Processing spam@cyberpromo.net
Processing abuse@example.com
```

PHP executes the body of the loop (the `echo` statement) once for each element of `$addresses` in turn, with `$value` set to the current element. Elements are processed by their internal order.

An alternative form of `foreach` gives you access to the current key:

```
$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

foreach ($person as $key => $value) {
 echo "Fred's {$key} is {$value}\n";
}
Fred's name is Fred
Fred's age is 35
Fred's wife is Wilma
```

In this case, the key for each element is placed in `$key` and the corresponding value is placed in `$value`.

The `foreach` construct does not operate on the array itself, but rather on a copy of it. You can insert or delete elements in the body of a `foreach` loop, safe in the knowledge that the loop won't attempt to process the deleted or inserted elements.

## The Iterator Functions

Every PHP array keeps track of the current element you're working with; the pointer to the current element is known as the *iterator*. PHP has functions to set, move, and reset this iterator. The iterator functions are:

`current()`
: Returns the element currently pointed at by the iterator.

`reset()`
: Moves the iterator to the first element in the array and returns it.

`next()`
: Moves the iterator to the next element in the array and returns it.

`prev()`
: Moves the iterator to the previous element in the array and returns it.

`end()`
: Moves the iterator to the last element in the array and returns it.

`each()`
: Returns the key and value of the current element as an array and moves the iterator to the next element in the array.

`key()`
: Returns the key of the current element.

The `each()` function is used to loop over the elements of an array. It processes elements according to their internal order:

```
reset($addresses);

while (list($key, $value) = each($addresses)) {
 echo "{$key} is {$value}<br />\n";
}
0 is spam@cyberpromo.net
1 is abuse@example.com
```

This approach does not make a copy of the array, as `foreach` does. This is useful for very large arrays when you want to conserve memory.

The iterator functions are useful when you need to consider some parts of the array separately from others. Example 5-1 shows code that builds a table, treating the first index and value in an associative array as table column headings.

*Example 5-1. Building a table with the iterator functions*

```
$ages = array(
 'Person' => "Age",
 'Fred' => 35,
 'Barney' => 30,
 'Tigger' => 8,
 'Pooh' => 40
);

// start table and print heading
reset($ages);

list($c1, $c2) = each($ages);

echo("<table>\n<tr><th>{$c1}</th><th>{$c2}</th></tr>\n");
```

```
// print the rest of the values
while (list($c1, $c2) = each($ages)) {
 echo("<tr><td>{$c1}</td><td>{$c2}</td></tr>\n");
}

// end the table
echo("</table>");
```

## Using a for Loop

If you know that you are dealing with an indexed array, where the keys are consecutive integers beginning at 0, you can use a `for` loop to count through the indices. The `for` loop operates on the array itself, not on a copy of the array, and processes elements in key order regardless of their internal order.

Here's how to print an array using `for`:

```
$addresses = array("spam@cyberpromo.net", "abuse@example.com");
$addressCount = count($addresses);

for ($i = 0; $i < $addressCount; $i++) {
 $value = $addresses[$i];
 echo "{$value}\n";
}
spam@cyberpromo.net
abuse@example.com
```

## Calling a Function for Each Array Element

PHP provides a mechanism, `array_walk()`, for calling a user-defined function once per element in an array:

```
array_walk(array, callable);
```

The function you define takes in two or, optionally, three arguments: the first is the element's value, the second is the element's key, and the third is a value supplied to `array_walk()` when it is called. For instance, here's another way to print table columns made of the values from an array:

```
$printRow = function ($value, $key)
{
 print("<tr><td>{$key}</td><td>{$value}</td></tr>\n");
};

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

echo "<table border=1>";

array_walk($person, $printRow);
```

```
echo "</table>";
```

A variation of this example specifies a background color using the optional third argument to `array_walk()`. This parameter gives us the flexibility we need to print many tables, with many background colors:

```
function printRow($value, $key, $color)
{
 echo "<tr>\n<td bgcolor=\"{$color}\">{$value}</td>";
 echo "<td bgcolor=\"{$color}\">{$key}</td>\n</tr>\n";
}

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

echo "<table border=\"1\">";

array_walk($person, "printRow", "lightblue");
echo "</table>";
```

If you have multiple options you want to pass into the called function, simply pass an array in as a third parameter:

```
$extraData = array('border' => 2, 'color' => "red");
$baseArray = array("Ford", "Chrysler", "Volkswagen", "Honda", "Toyota");

array_walk($baseArray, "walkFunction", $extraData);

function walkFunction($item, $index, $data)
{
 echo "{$item} <- item, then border: {$data['border']}";
 echo " color->{$data['color']}<br />" ;
}
Ford <- item, then border: 2 color->red
Crysler <- item, then border: 2 color->red
VW <- item, then border: 2 color->red
Honda <- item, then border: 2 color->red
Toyota <- item, then border: 2 color->red
```

The `array_walk()` function processes elements in their internal order.

## Reducing an Array

A cousin of `array_walk()`, `array_reduce()` applies a function to each element of the array in turn, to build a single value:

```
$result = array_reduce(array, callable [, default ]);
```

The function takes two arguments: the running total, and the current value being processed. It should return the new running total. For instance, to add up the squares of the values of an array, use:

```
$addItUp = function ($runningTotal, $currentValue)
{
 $runningTotal += $currentValue * $currentValue;

 return $runningTotal;
};

$numbers = array(2, 3, 5, 7);
$total = array_reduce($numbers, $addItUp);

echo $total;

87
```

The `array_reduce()` line makes these function calls:

```
addItUp(0, 2);
addItUp(4, 3);
addItUp(13, 5);
addItUp(38, 7);
```

The *default* argument, if provided, is a seed value. For instance, if we change the call to `array_reduce()` in the previous example to:

```
$total = array_reduce($numbers, "addItUp", 11);
```

The resulting function calls are:

```
addItUp(11, 2);
addItUp(15, 3);
addItUp(24, 5);
addItUp(49, 7);
```

If the array is empty, `array_reduce()` returns the *default* value. If no default value is given and the array is empty, `array_reduce()` returns `NULL`.

## Searching for Values

The `in_array()` function returns `true` or `false`, depending on whether the first argument is an element in the array given as the second argument:

```
if (in_array(to_find, array [, strict])) { ... }
```

If the optional third argument is `true`, the types of *to_find* and the value in the array must match. The default is to not check the data types.

Here's a simple example:

```
$addresses = array("spam@cyberpromo.net", "abuse@example.com",
"root@example.com");
$gotSpam = in_array("spam@cyberpromo.net", $addresses); // $gotSpam is true
$gotMilk = in_array("milk@tucows.com", $addresses); // $gotMilk is false
```

PHP automatically indexes the values in arrays, so `in_array()` is generally much faster than a loop checking every value in the array to find the one you want.

Example 5-2 checks whether the user has entered information in all the required fields in a form.

*Example 5-2. Searching an array*

```
<?php
function hasRequired($array, $requiredFields) {
 $array =

 $keys = array_keys ( $array );
 foreach ( $requiredFields as $fieldName ) {
 if (! in_array ( $fieldName, $keys )) {
 return false;
 }
 }
 return true;
}
if ($_POST ['submitted']) {
 $testArray = array_filter($_POST);
 echo "<p>You ";
 echo hasRequired ( $testArray, array (
 'name',
 'email_address'
 ) ) ? "did" : "did not";
 echo " have all the required fields.</p>";
}
?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <p>
 Name: <input type="text" name="name" /><br /> Email address: <input
 type="text" name="email_address" /><br /> Age (optional): <input
 type="text" name="age" />
 </p>
 <p align="center">
 <input type="submit" value="submit" name="submitted" />
 </p>
</form>
```

A variation on `in_array()` is the `array_search()` function. While `in_array()` returns `true` if the value is found, `array_search()` returns the key of the element, if found:

```
$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");
$k = array_search("Wilma", $person);

echo("Fred's {$k} is Wilma\n");

Fred's wife is Wilma
```

The `array_search()` function also takes the optional third *strict* argument, which requires that the types of the value being searched for and the value in the array match.

# Sorting

Sorting changes the internal order of elements in an array and optionally rewrites the keys to reflect this new order. For example, you might use sorting to arrange a list of scores from biggest to smallest, to alphabetize a list of names, or to order a set of users based on how many messages they posted.

PHP provides three ways to sort arrays—sorting by keys, sorting by values without changing the keys, or sorting by values and then changing the keys. Each kind of sort can be done in ascending order, descending order, or an order determined by a user-defined function.

## Sorting One Array at a Time

The functions provided by PHP to sort an array are shown in Table 5-1.

*Table 5-1. PHP functions for sorting an array*

| Effect | Ascending | Descending | User-defined order |
|---|---|---|---|
| Sort array by values, then reassign indices starting with 0 | `sort()` | `rsort()` | `usort()` |
| Sort array by values | `asort()` | `arsort()` | `uasort()` |
| Sort array by keys | `ksort()` | `krsort()` | `uksort()` |

The `sort()`, `rsort()`, and `usort()` functions are designed to work on indexed arrays because they assign new numeric keys to represent the ordering. They're useful when you need to answer questions such as "What are the top 10 scores?" and "Who's the third person in alphabetical order?" The other sort functions can be used on indexed arrays, but you'll only be able to access the sorted ordering by using traversal constructs such as `foreach` and `next()`.

To sort names into ascending alphabetical order, do something like this:

```
$names = array("Cath", "Angela", "Brad", "Mira");
sort($names); // $names is now "Angela", "Brad", "Cath", "Mira"
```

To get them in reverse alphabetical order, simply call `rsort()` instead of `sort()`.

If you have an associative array that maps usernames to minutes of login time, you can use `arsort()` to display a table of the top three, as shown here:

```
$logins = array(
 'njt' => 415,
 'kt' => 492,
```

```
    'rl' => 652,
    'jht' => 441,
    'jj' => 441,
    'wt' => 402,
    'hut' => 309,
   );

   arsort($logins);

   $numPrinted = 0;

   echo "<table>\n";

   foreach ($logins as $user => $time) {
    echo("<tr><td>{$user}</td><td>{$time}</td></tr>\n");

    if (++$numPrinted == 3) {
    break; // stop after three
    }
   }

   echo "</table>";
```

If you want that table displayed in ascending order by username, use `ksort()` instead.

User-defined ordering requires that you provide a function that takes two values and returns a value that specifies the order of the two values in the sorted array. The function should return `1` if the first value is greater than the second, `-1` if the first value is less than the second, and `0` if the values are the same for the purposes of your custom sort order.

The program in Example 5-3 applies the various sorting functions to the same data.

*Example 5-3. Sorting arrays*

```
<?php
function userSort($a, $b)
{
 // smarts is all-important, so sort it first
 if ($b == "smarts") {
 return 1;
 }
 else if ($a == "smarts") {
 return -1;
 }

 return ($a == $b) ? 0 : (($a < $b) ? -1 : 1);
}

$values = array(
 'name' => "Buzz Lightyear",
```

```
 'email_address' => "buzz@starcommand.gal",
 'age' => 32,
 'smarts' => "some"
);

if ($_POST['submitted']) {
 $sortType = $_POST['sort_type'];

 if ($sortType == "usort" || $sortType == "uksort" || $sortType == "uasort") {
 $sortType($values, "userSort");
 }
 else {
 $sortType($values);
 }
} ?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?> " method="post">
 <p>
 <input type="radio" name="sort_type"
 value="sort" checked="checked" /> Standard<br />
 <input type="radio" name="sort_type" value="rsort" /> Reverse<br />
 <input type="radio" name="sort_type" value="usort" /> User-defined<br />
 <input type="radio" name="sort_type" value="ksort" /> Key<br />
 <input type="radio" name="sort_type" value="krsort" /> Reverse key<br />
 <input type="radio" name="sort_type"
 value="uksort" /> User-defined key<br />
 <input type="radio" name="sort_type" value="asort" /> Value<br />
 <input type="radio" name="sort_type"
 value="arsort" /> Reverse value<br />
 <input type="radio" name="sort_type"
 value="uasort" /> User-defined value<br />
 </p>

 <p align="center"><input type="submit" value="Sort" name="submitted" /></p>

 <p>Values <?php echo $_POST['submitted'] ? "sorted by {$sortType}" : "unsorted";
 ?>:</p>

 <ul>
 <?php foreach ($values as $key => $value) {
 echo "<li><b>{$key}</b>: {$value}</li>";
 } ?>
 </ul>
</form>
```

## Natural-Order Sorting

PHP's built-in sort functions correctly sort strings and numbers, but they don't correctly sort strings that contain numbers. For example, if you have the filenames *ex10.php*, *ex5.php*, and *ex1.php*, the normal sort functions will rearrange them in this order: *ex1.php*, *ex10.php*, *ex5.php*. To correctly sort strings that contain numbers, use the `natsort()` and `natcasesort()` functions:

```
$output = natsort(input);
$output = natcasesort(input);
```

## Sorting Multiple Arrays at Once

The `array_multisort()` function sorts multiple indexed arrays at once:

```
array_multisort(array1 [, array2, ... ]);
```

Pass it a series of arrays and sorting orders (identified by the `SORT_ASC` or `SORT_DESC` constants), and it reorders the elements of all the arrays, assigning new indices. It is similar to a join operation on a relational database.

Imagine that you have a lot of people, and several pieces of data on each person:

```
$names = array("Tom", "Dick", "Harriet", "Brenda", "Joe");
$ages = array(25, 35, 29, 35, 35);
$zips = array(80522, '02140', 90210, 64141, 80522);
```

The first element of each array represents a single record—all the information known about Tom. Similarly, the second element constitutes another record—all the information known about Dick. The `array_multisort()` function reorders the elements of the arrays, preserving the records. That is, if `"Dick"` ends up first in the `$names` array after the sort, the rest of Dick's information will be first in the other arrays too. (Note that we needed to quote Dick's zip code to prevent it from being interpreted as an octal constant.)

Here's how to sort the records first ascending by age, then descending by zip code:

```
array_multisort($ages, SORT_ASC, $zips, SORT_DESC, $names, SORT_ASC);
```

We need to include `$names` in the function call to ensure that Dick's name stays with his age and zip code. Printing out the data shows the result of the sort:

```
for ($i = 0; $i < count($names); $i++) {
 echo "{$names[$i]}, {$ages[$i]}, {$zips[$i]}\n";
}
Tom, 25, 80522
Harriet, 29, 90210
Joe, 35, 80522
Brenda, 35, 64141
Dick, 35, 02140
```

## Reversing Arrays

The `array_reverse()` function reverses the internal order of elements in an array:

```
$reversed = array_reverse(array);
```

Numeric keys are renumbered starting at 0, while string indices are unaffected. In general, it's better to use the reverse-order sorting functions instead of sorting and then reversing the order of an array.

The `array_flip()` function returns an array that reverses the order of each original element's key-value pair:

```
$flipped = array_flip(array);
```

That is, for each element of the array whose value is a valid key, the element's value becomes its key and the element's key becomes its value. For example, if you have an array that maps usernames to home directories, you can use `array_flip()` to create an array that maps home directories to usernames:

```
$u2h = array(
 'gnat' => "/home/staff/nathan",
 'frank' => "/home/action/frank",
 'petermac' => "/home/staff/petermac",
 'ktatroe' => "/home/staff/kevin"
);
$h2u = array_flip($u2h);

$user = $h2u["/home/staff/kevin"]; // $user is now 'ktatroe'
```

Elements whose original values are neither strings nor integers are left alone in the resulting array. The new array lets you discover the key in the original array given its value, but this technique works effectively only when the original array has unique values.

## Randomizing Order

To traverse the elements in an array in random order, use the `shuffle()` function. It replaces all existing keys—string or numeric—with consecutive integers starting at 0.

Here's how to randomize the order of the days of the week:

```
$weekdays = array("Monday", "Tuesday", "Wednesday", "Thursday", "Friday");
shuffle($weekdays);

print_r($weekdays);

Array(
 [0] => Tuesday
 [1] => Thursday
 [2] => Monday
 [3] => Friday
 [4] => Wednesday
)
```

Obviously, the order after you `shuffle()` may not be the same as the sample output here due to the random nature of the function. Unless you are interested in getting multiple random elements from an array without repeating any specific item, using the `rand()` function to pick an index is more efficient.

# Acting on Entire Arrays

PHP has several useful built-in functions for modifying or applying an operation to all elements of an array. You can calculate the sum of an array, merge multiple arrays, find the difference between two arrays, and more.

## Calculating the Sum of an Array

The `array_sum()` function adds up the values in an indexed or associative array:

```
$sum = array_sum(array);
```

For example:

```
$scores = array(98, 76, 56, 80);
$total = array_sum($scores); // $total = 310
```

## Merging Two Arrays

The `array_merge()` function intelligently merges two or more arrays:

```
$merged = array_merge(array1, array2 [, array ... ])
```

If a numeric key from an earlier array is repeated, the value from the later array is assigned a new numeric key:

```
$first = array("hello", "world"); // 0 => "hello", 1 => "world"
$second = array("exit", "here"); // 0 => "exit", 1 => "here"

$merged = array_merge($first, $second);
// $merged = array("hello", "world", "exit", "here")
```

If a string key from an earlier array is repeated, the earlier value is replaced by the later value:

```
$first = array('bill' => "clinton", 'tony' => "danza");
$second = array('bill' => "gates", 'adam' => "west");

$merged = array_merge($first, $second);
// $merged = array('bill' => "gates", 'tony' => "danza", 'adam' => "west")
```

## Calculating the Difference Between Two Arrays

The `array_diff()` function calculates the difference between two or more arrays, returning an array with values from the first array that are not present in the others:

```
$diff = array_diff(array1, array2 [, array ... ]);
```

For example:

```
$a1 = array("bill", "claire", "ella", "simon", "judy");
$a2 = array("jack", "claire", "toni");
$a3 = array("ella", "simon", "garfunkel");
```

```
// find values of $a1 not in $a2 or $a3
$difference = array_diff($a1, $a2, $a3);
print_r($difference);

Array(
 [0] => "bill",
 [4] => "judy"
);
```

Values are compared using the strict comparison operator `===`, so `1` and `"1"` are considered different. The keys of the first array are preserved, so in `$diff` the key of `"bill"` is `0` and the key of `"judy"` is `4`.

In another example, the following code returns the difference of two arrays:

```
$first = array(1, "two", 3);
$second = array("two", "three", "four");

$difference = array_diff($first, $second);
print_r($difference);

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

## Filtering Elements from an Array

To identify a subset of an array based on its values, use the `array_filter()` function:

```
$filtered = array_filter(array, callback);
```

Each value of *array* is passed to the function named in *callback*. The returned array contains only those elements of the original array for which the function returns a `true` value. For example:

```
function isOdd ($element) {
 return $element % 2;
}

$numbers = array(9, 23, 24, 27);
$odds = array_filter($numbers, "isOdd");

// $odds is array(0 => 9, 1 => 23, 3 => 27)
```

As you can see, the keys are preserved. This function is most useful with associative arrays.

# Using Arrays to Implement Data Types

Arrays crop up in almost every PHP program. In addition to their obvious purpose of storing collections of values, they're also used to implement various abstract data types. In this section, we show how to use arrays to implement sets and stacks.

## Sets

Arrays enable you to implement the basic operations of set theory: union, intersection, and difference. Each set is represented by an array, and various PHP functions implement the set operations. The values in the set are the values in the array—the keys are not used, but they are generally preserved by the operations.

The *union* of two sets is all the elements from both sets with duplicates removed. The `array_merge()` and `array_unique()` functions let you calculate the union. Here's how to find the union of two arrays:

```
function arrayUnion($a, $b)
{
 $union = array_merge($a, $b); // duplicates may still exist
 $union = array_unique($union);

 return $union;
}

$first = array(1, "two", 3);
$second = array("two", "three", "four");

$union = arrayUnion($first, $second);
print_r($union);

Array(
 [0] => 1
 [1] => two
 [2] => 3
 [4] => three
 [5] => four
)
```

The *intersection* of two sets is the set of elements they have in common. PHP's built-in `array_intersect()` function takes any number of arrays as arguments and returns an array of those values that exist in each. If multiple keys have the same value, the first key with that value is preserved.

## Stacks

Although not as common in PHP programs as in other programs, one fairly common data type is the last-in first-out (LIFO) stack. We can create stacks using a pair of PHP functions, `array_push()` and `array_pop()`. The `array_push()` function is identical

to an assignment to `$array[]`. We use `array_push()` because it accentuates the fact that we're working with stacks, and the parallelism with `array_pop()` makes our code easier to read. There are also `array_shift()` and `array_unshift()` functions for treating an array like a queue.

Stacks are particularly useful for maintaining state. Example 5-4 provides a simple state debugger that allows you to print out a list of which functions have been called up to this point (i.e., the *stack trace*).

*Example 5-4. State debugger*

```
$callTrace = array();

function enterFunction($name)
{
 global $callTrace;
 $callTrace[] = $name;

 echo "Entering {$name} (stack is now: " . join(' -> ', $callTrace) . ")<br />";
}

function exitFunction()
{
 echo "Exiting<br />";

 global $callTrace;
 array_pop($callTrace);
}

function first()
{
 enterFunction("first");
 exitFunction();
}

function second()
{
 enterFunction("second");
 first();
 exitFunction();
}

function third()
{
 enterFunction("third");
 second();
 first();
 exitFunction();
}
```

```
first();
third();
```

Here's the output from Example 5-4:

```
Entering first (stack is now: first)
Exiting
Entering third (stack is now: third)
Entering second (stack is now: third -> second)
Entering first (stack is now: third -> second -> first)
Exiting
Exiting
Entering first (stack is now: third -> first)
Exiting
Exiting
```

# Implementing the Iterator Interface

Using the `foreach` construct, you can iterate not only over arrays, but also over instances of classes that implement the `Iterator` interface (see Chapter 6 for more information on objects and interfaces). To implement the `Iterator` interface, you must implement five methods on your class:

`current()`
: Returns the element currently pointed at by the iterator.

`key()`
: Returns the key for the element currently pointed at by the iterator.

`next()`
: Moves the iterator to the next element in the object and returns it.

`rewind()`
: Moves the iterator to the first element in the array.

`valid()`
: Returns `true` if the iterator currently points at a valid element, and `false` otherwise.

Example 5-5 reimplements a simple iterator class containing a static array of data.

*Example 5-5. Iterator interface*

```
class BasicArray implements Iterator
{
 private $position = 0;
 private $array = ["first", "second", "third"];

 public function __construct()
```

```
    {
    $this->position = 0;
    }

    public function rewind()
    {
    $this->position = 0;
    }

    public function current()
    {
    return $this->array[$this->position];
    }

    public function key()
    {
    return $this->position;
    }

    public function next()
    {
    $this->position += 1;
    }

    public function valid()
    {
    return isset($this->array[$this->position]);
    }
}

$basicArray = new BasicArray;

foreach ($basicArray as $value) {
    echo "{$value}\n";
}

foreach ($basicArray as $key => $value) {
    echo "{$key} => {$value}\n";
}

first
second
third

0 => first
1 => second
2 => third
```

When you implement the `Iterator` interface on a class, it allows you only to traverse elements in instances of that class using the `foreach` construct; it does not allow you to treat those instances as arrays or parameters to other methods. This, for example,

rewinds the `Iterator` pointing at `$trie`'s properties using the built-in `rewind()` function instead of calling the `rewind()` method on `$trie`:

```
class Trie implements Iterator
{
 const POSITION_LEFT = "left";
 const POSITION_THIS = "this";
 const POSITION_RIGHT = "right";

 var $leftNode;
 var $rightNode;

 var $position;

 // implement Iterator methods here...
}

$trie = new Trie();

rewind($trie);
```

The optional SPL library provides a wide variety of useful iterators, including filesystem directory, tree, and regex matching iterators.

## What's Next

The last three chapters—on functions, strings, and arrays—have covered a lot of foundational ground. The next chapter builds on this foundation and takes you into the newish world of objects and object-oriented programming (OOP). Some argue that OOP is the better way to program, as it is more encapsulated and reusable than procedural programming. That debate continues, but once you get into the object-oriented approach to programming and understand its benefits, you can make an informed decision about how you'll program in the future. That said, the overall trend in the programming world is to use OOP as much as possible.

One word of caution before you continue: there are many situations where a novice OOP programmer can get lost, so be sure you're really comfortable with OOP before you do anything major or mission-critical with it.

# CHAPTER 6
# Objects

In this chapter you'll learn how to define, create, and use objects in PHP. Object-oriented programming (OOP) opens the door to cleaner designs, easier maintenance, and greater code reuse. OOP has proven so valuable that few today would dare to introduce a language that wasn't object-oriented. PHP supports many useful features of OOP, and this chapter shows you how to use them, covering basic OOP concepts as well as advanced topics such as introspection and serialization.

## Objects

Object-oriented programming acknowledges the fundamental connection between data and the code that works on it, and lets you design and implement programs around that connection. For example, a bulletin-board system usually keeps track of many users. In a procedural programming language, each user is represented by a data structure, and there would probably be a set of functions that work with those data structures (to create the new users, get their information, etc.). In an OOP language, each user is represented by an *object*—a data structure with attached code. The data and the code are still there, but they're treated as an inseparable unit. The object, as a union of code and data, is the modular unit for application development and code reuse.

In this hypothetical bulletin-board design, objects can represent not just users but also messages and threads. A user object has a username and password for that user, and code to identify all the messages by that author. A message object knows which thread it belongs to and has code to post a new message, reply to an existing message, and display messages. A thread object is a collection of message objects, and it has code to display a thread index. This is only one way of dividing the necessary functionality into objects, though. For instance, in an alternate design, the code to post a new message lives in the user object, not the message object.

Designing object-oriented systems is a complex topic, and many books have been written on it. The good news is that however you design your system, you can implement it in PHP. Let's begin by introducing some of the key terms and concepts you'll need to know before diving into this programming approach.

# Terminology

Every object-oriented language seems to have a different set of terms for the same old concepts. This section describes the terms that PHP uses, but be warned that in other languages these terms may have other meanings.

Let's return to the example of the users of a bulletin board. You need to keep track of the same information for each user, and the same functions can be called on each user's data structure. When you design the program, you decide the fields for each user and come up with the functions. In OOP terms, you're designing the user *class*. A class is a template for building objects.

An *object* is an instance (or occurrence) of a class. In this case, it's an actual user data structure with attached code. Objects and classes are a bit like values and data types. There's only one integer data type, but there are many possible integers. Similarly, your program defines only one user class but can create many different (or identical) users from it.

The data associated with an object are called its *properties*. The functions associated with an object are called its *methods*. When you define a class, you define the names of its properties and give the code for its methods.

Debugging and maintenance of programs is much easier if you use *encapsulation*. This is the idea that a class provides certain methods (the *interface*) to the code that uses its objects, so the outside code does not directly access the data structures of those objects. Debugging is thus easier because you know where to look for bugs—the only code that changes an object's data structures is within the class—and maintenance is easier because you can swap out implementations of a class without changing the code that uses the class, as long as you maintain the same interface.

Any nontrivial object-oriented design probably involves *inheritance*. This is a way of defining a new class by saying that it's like an existing class, but with certain new or changed properties and methods. The original class is called the *superclass* (or parent or base class), and the new class is called the *subclass* (or derived class). Inheritance is a form of code reuse—the superclass code is reused instead of being copied and pasted into the subclass. Any improvements or modifications to the superclass are automatically passed on to the subclass.

## Creating an Object

It's much easier to create (or *instantiate*) objects and use them than it is to define object classes, so before we discuss how to define classes, let's look at creating objects. To create an object of a given class, use the `new` keyword:

```
$object = new Class;
```

Assuming that a `Person` class has been defined, here's how to create a `Person` object:

```
$moana = new Person;
```

Do not quote the class name, or you'll get a compilation error:

```
$moana = new "Person"; // does not work
```

Some classes permit you to pass arguments to the `new` call. The class's documentation should say whether it accepts arguments. If it does, you'll create objects like this:

```
$object = new Person("Sina", 35);
```

The class name does not have to be hardcoded into your program. You can supply the class name through a variable:

```
$class = "Person";
$object = new $class;
// is equivalent to
$object = new Person;
```

Specifying a class that doesn't exist causes a runtime error.

Variables containing object references are just normal variables—they can be used in the same ways as other variables. Note that variable variables work with objects, as shown here:

```
$account = new Account;
$object = "account";
${$object}->init(50000, 1.10); // same as $account->init
```

## Accessing Properties and Methods

Once you have an object, you can use the `->` notation to access methods and properties of the object:

```
$object->propertyname $object->methodname([arg, ... ])
```

For example:

```
echo "Moana is {$moana->age} years old.\n"; // property access
$moana->birthday(); // method call
$moana->setAge(21); // method call with arguments
```

Methods act the same as functions (only specifically to the object in question), so they can take arguments and return a value:

```
$clan = $moana->family("extended");
```

Within a class's definition, you can specify which methods and properties are publicly accessible and which are accessible only from within the class itself using the public and private access modifiers. You can use these to provide encapsulation.

You can use variable variables with property names:

```
$prop = 'age';
echo $moana->$prop;
```

A static method is one that is called on a class, not on an object. Such methods cannot access properties. The name of a static method is the class name followed by two colons and the function name. For instance, this calls the `p()` static method in the HTML class:

```
HTML::p("Hello, world");
```

When declaring a class, you define which properties and methods are static using the static access property.

Once created, objects are passed by reference—that is, instead of copying around the entire object itself (a time- and memory-consuming endeavor), a reference to the object is passed around instead. For example:

```
$f = new Person("Pua", 75);

$b = $f; // $b and $f point at same object
$b->setName("Hei Hei");

printf("%s and %s are best friends.\n", $b->getName(), $f->getName());
Hei Hei and Hei Hei are best friends.
```

If you want to create a true copy of an object, you use the clone operator:

```
$f = new Person("Pua", 35);

$b = clone $f; // make a copy
$b->setName("Hei Hei");// change the copy

printf("%s and %s are best friends.\n", $b->getName(), $f->getName());
Pua and Hei Hei are best friends.
```

When you use the clone operator to create a copy of an object and that class declares the `__clone()` method, that method is called on the new object immediately after it's cloned. You might use this in cases where an object holds external resources (such as file handles) to create new resources, rather than copying the existing ones.

# Declaring a Class

To design your program or code library in an object-oriented fashion, you'll need to define your own classes, using the `class` keyword. A class definition includes the class name and the properties and methods of the class. Class names are case-insensitive and must conform to the rules for PHP identifiers. Among others, the class name `stdClass` is reserved. Here's the syntax for a class definition:

```
class classname [ extends baseclass ] [ implements interfacename ,
 [interfacename, ... ] ] {
 [ use traitname, [ traitname, ... ]; ]

 [ visibility $property [ = value ]; ... ]

 [ function functionname (args) [: type ] {
 // code
 }
 ...
 ]
}
```

## Declaring Methods

A method is a function defined inside a class. Although PHP imposes no special restrictions, most methods act only on data within the object in which the method resides. Method names beginning with two underscores (__) may be used in the future by PHP (and are currently used for the object serialization methods `__sleep()` and `__wakeup()`, described later in this chapter, among others), so it's recommended that you do not begin your method names with this sequence.

Within a method, the `$this` variable contains a reference to the object on which the method was called. For instance, if you call `$moana->birthday()`, inside the `birthday()` method, `$this` holds the same value as `$moana`. Methods use the `$this` variable to access the properties of the current object and to call other methods on that object.

Here's a simple class definition of the `Person` class that shows the `$this` variable in action:

```
class Person {
 public $name = '';

 function getName() {
 return $this->name;
 }

 function setName($newName) {
 $this->name = $newName;
 }
}
```

As you can see, the `getName()` and `setName()` methods use `$this` to access and set the `$name` property of the current object.

To declare a method as a static method, use the `static` keyword. Inside of static methods the variable `$this` is not defined. For example:

```
class HTMLStuff {
 static function startTable() {
 echo "<table border=\"1\">\n";
 }

 static function endTable() {
 echo "</table>\n";
 }
}

HTMLStuff::startTable();
 // print HTML table rows and columns
HTMLStuff::endTable();
```

If you declare a method using the `final` keyword, subclasses cannot override that method. For example:

```
class Person {
 public $name;

 final function getName() {
 return $this->name;
 }
}

class Child extends Person {
 // syntax error
 function getName() {
 // do something
 }
}
```

Using access modifiers, you can change the visibility of methods. Methods that are accessible outside methods on the object should be declared `public`; methods on an instance that can be called only by methods within the same class should be declared `private`. Finally, methods declared as `protected` can be called only from within the object's class methods and the class methods of classes inheriting from the class. Defining the visibility of class methods is optional; if a visibility is not specified, a method is public. For example, you might define:

```
class Person {
 public $age;

 public function __construct() {
 $this->age = 0;
```

```
 }

 public function incrementAge() {
 $this->age += 1;
 $this->ageChanged();
 }

 protected function decrementAge() {
 $this->age -= 1;
 $this->ageChanged();
 }

 private function ageChanged() {
 echo "Age changed to {$this->age}";
 }
}

class SupernaturalPerson extends Person {
 public function incrementAge() {
 // ages in reverse
 $this->decrementAge();
 }
}

$person = new Person;
$person->incrementAge();
$person->decrementAge(); // not allowed
$person->ageChanged(); // also not allowed

$person = new SupernaturalPerson;
$person->incrementAge(); // calls decrementAge under the hood
```

You can use type hinting (described in Chapter 3) when declaring a method on an object:

```
class Person {
 function takeJob(Job $job) {
 echo "Now employed as a {$job->title}\n";
 }
}
```

When a method returns a value, you can use type hinting to declare the method's return value type:

```
class Person {
 function bestJob(): Job {
 $job = Job("PHP developer");

 return $job;
 }
}
```

## Declaring Properties

In the previous definition of the Person class, we explicitly declared the $name property. Property declarations are optional and are simply a courtesy to whomever maintains your program. It's good PHP style to declare your properties, but you can add new properties at any time.

Here's a version of the Person class that has an undeclared $name property:

```
class Person {
 function getName() {
 return $this->name;
 }

 function setName($newName) {
 $this->name = $newName;
 }
}
```

You can assign default values to properties, but those default values must be simple constants:

```
public $name = "J Doe"; // works
public $age = 0; // works
public $day = 60 * 60 * hoursInDay(); // doesn't work
```

Using access modifiers, you can change the visibility of properties. Properties that are accessible outside the object's scope should be declared public; properties on an instance that can be accessed only by methods within the same class should be declared private. Finally, properties declared as protected can be accessed only by the object's class methods and the class methods of classes inheriting from the class. For example, you might declare a user class:

```
class Person {
 protected $rowId = 0;

 public $username = 'Anyone can see me';

 private $hidden = true;
}
```

In addition to properties on instances of objects, PHP allows you to define static properties, which are variables on an object class, and can be accessed by referencing the property with the class name. For example:

```
class Person {
 static $global = 23;
}

$localCopy = Person::$global;
```

Inside an instance of the object class, you can also refer to the static property using the `self` keyword, like `echo self::$global;`.

If a property is accessed on an object that doesn't exist, and if the `__get()` or `__set()` method is defined for the object's class, that method is given an opportunity to either retrieve a value or set the value for that property.

For example, you might declare a class that represents data pulled from a database, but you might not want to pull in large data values—such as Binary Large Objects (BLOBs)—unless specifically requested. One way to implement that, of course, would be to create access methods for the property that read and write the data whenever requested. Another method might be to use these overloading methods:

```
class Person {
 public function __get($property) {
 if ($property === 'biography') {
 $biography = "long text here..."; // would retrieve from database

 return $biography;
 }
 }

 public function __set($property, $value) {
 if ($property === 'biography') {
 // set the value in the database
 }
 }
}
```

## Declaring Constants

As with global constants, assigned through the `define()` function, PHP provides a way to assign constants within a class. Like static properties, constants can be accessed directly through the class or within object methods using the `self` notation. Once a constant is defined, its value cannot be changed:

```
class PaymentMethod {
 public const TYPE_CREDITCARD = 0;
 public const TYPE_CASH = 1;
}

echo PaymentMethod::TYPE_CREDITCARD;
0
```

As with global constants, it is common practice to define class constants with uppercase identifiers.

Using access modifiers, you can change the visibility of class constants. Class constants that are accessible outside methods on the object should be declared `public`; class constants on an instance that can be accessed only by methods within the same

class should be declared `private`. Finally, constants declared as `protected` can be accessed only from within the object's class methods and the class methods of classes inheriting from the class. Defining the visibility of class constants is optional; if a visibility is not specified, a method is `public`. For example, you might define:

```
class Person {
 protected const PROTECTED_CONST = false;
 public const DEFAULT_USERNAME = "<unknown>";
 private INTERNAL_KEY = "ABC1234";
}
```

## Inheritance

To inherit the properties and methods from another class, use the `extends` keyword in the class definition, followed by the name of the base class:

```
class Person {
 public $name, $address, $age;
}

class Employee extends Person {
 public $position, $salary;
}
```

The `Employee` class contains the `$position` and `$salary` properties, as well as the `$name`, `$address`, and `$age` properties inherited from the `Person` class.

If a derived class has a property or method with the same name as one in its parent class, the property or method in the derived class takes precedence over the property or method in the parent class. Referencing the property returns the value of the property on the child, while referencing the method calls the method on the child.

Use the `parent::`*`method`*`()` notation to access an overridden method on an object's parent class:

```
parent::birthday(); // call parent class's birthday() method
```

A common mistake is to hardcode the name of the parent class into calls to overridden methods:

```
Creature::birthday(); // when Creature is the parent class
```

This is a mistake because it distributes knowledge of the parent class's name throughout the derived class. Using `parent::` centralizes the knowledge of the parent class in the `extends` clause.

If a method might be subclassed and you want to ensure that you're calling it on the current class, use the `self::`*`method`*`()` notation:

```
self::birthday(); // call this class's birthday() method
```

To check if an object is an instance of a particular class or if it implements a particular interface (see the section "Interfaces"), you can use the `instanceof` operator:

```
if ($object instanceof Animal) {
 // do something
}
```

## Interfaces

Interfaces provide a way for defining contracts to which a class adheres; the interface provides method prototypes and constants, and any class that implements the interface must provide implementations for all methods in the interface. Here's the syntax for an interface definition:

```
interface interfacename {
 [ function functionname();
 ...
 ]
}
```

To declare that a class implements an interface, include the `implements` keyword and any number of interfaces, separated by commas:

```
interface Printable {
 function printOutput();
}

class ImageComponent implements Printable {
 function printOutput() {
 echo "Printing an image...";
 }
}
```

An interface may inherit from other interfaces (including multiple interfaces) as long as none of the interfaces it inherits from declare methods with the same name as those declared in the child interface.

## Traits

Traits provide a mechanism for reusing code outside of a class hierarchy. Traits allow you to share functionality across different classes that don't (and shouldn't) share a common ancestor in a class hierarchy. Here's the syntax for a trait definition:

```
trait traitname [ extends baseclass ] {
 [ use traitname, [ traitname, ... ]; ]

 [ visibility $property [ = value ]; ... ]

 [ function functionname (args) {
 // code
 }
```

```
  ...
  ]
}
```

To declare that a class should include a trait's methods, include the `use` keyword and any number of traits, separated by commas:

```
trait Logger {
 public function log($logString) {
 $className = __CLASS__;
 echo date("Y-m-d h:i:s", time()) . ": [{$className}] {$logString}";
 }
}

class User {
 use Logger;

 public $name;

 function __construct($name = '') {
 $this->name = $name;
 $this->log("Created user '{$this->name}'");
 }

 function __toString() {
 return $this->name;
 }
}

class UserGroup {
 use Logger;

 public $users = array();

 public function addUser(User $user) {
 if (!in_array($this->users, $user)) {
 $this->users[] = $user;
 $this->log("Added user '{$user}' to group");
 }
 }
}

$group = new UserGroup;
$group->addUser(new User("Franklin"));
2012-03-09 07:12:58: [User] Created user 'Franklin'2012-03-09 07:12:58:
[UserGroup] Added user 'Franklin' to group
```

The methods defined by the `Logger` trait are available to instances of the `UserGroup` class as if they were defined in that class.

To declare that a trait should be composed of other traits, include the `use` statement in the trait's declaration, followed by one or more trait names separated by commas, as shown here:

```
trait First {
 public function doFirst( {
 echo "first\n";
 }
}

trait Second {
 public function doSecond() {
 echo "second\n";
 }
}

trait Third {
 use First, Second;

 public function doAll() {
 $this->doFirst();
 $this->doSecond();
 }
}

class Combined {
 use Third;
}

$object = new Combined;
$object->doAll();
firstsecond
```

Traits can declare abstract methods.

If a class uses multiple traits defining the same method, PHP gives a fatal error. However, you can override this behavior by telling the compiler specifically which implementation of a given method you want to use. When defining which traits a class includes, use the `insteadof` keyword for each conflict:

```
trait Command {
 function run() {
 echo "Executing a command\n";
 }
}

trait Marathon {
 function run() {
 echo "Running a marathon\n";
 }
}
```

```
class Person {
 use Command, Marathon {
 Marathon::run insteadof Command;
 }
}

$person = new Person;
$person->run();
Running a marathon
```

Instead of picking just one method to include, you can use the as keyword to alias a trait's method within the class including it to a different name. You must still explicitly resolve any conflicts in the included traits. For example:

```
trait Command {
 function run() {
 echo "Executing a command";
 }
}

trait Marathon {
 function run() {
 echo "Running a marathon";
 }
}

class Person {
 use Command, Marathon {
 Command::run as runCommand;
 Marathon::run insteadof Command;
 }
}

$person = new Person;
$person->run();
$person->runCommand();
Running a marathonExecuting a command
```

## Abstract Methods

PHP also provides a mechanism for declaring that certain methods on the class must be implemented by subclasses—the implementation of those methods is not defined in the parent class. In these cases, you provide an abstract method; in addition, if a class contains any methods defined as abstract, you must also declare the class as an abstract class:

```
abstract class Component {
 abstract function printOutput();
}

class ImageComponent extends Component {
```

```
 function printOutput() {
 echo "Pretty picture";
 }
}
```

Abstract classes cannot be instantiated. Also note that, unlike some languages, PHP does not allow you to provide a default implementation for abstract methods.

Traits can also declare abstract methods. Classes that include a trait that defines an abstract method must implement that method:

```
trait Sortable {
 abstract function uniqueId();

 function compareById($object) {
 return ($object->uniqueId() < $this->uniqueId()) ? -1 : 1;
 }
}

class Bird {
 use Sortable;

 function uniqueId() {
 return __CLASS__ . ":{$this->id}";
 }
}

// this will not compile
class Car {
 use Sortable;
}

$bird = new Bird;
$car = new Car;
$comparison = $bird->compareById($car);
```

When you implement an abstract method in a child class, the method signatures must match—that is, they must take in the same number of required parameters, and if any of the parameters have type hints, those type hints must match. In addition, the method must have the same or less restricted visibility.

## Constructors

You may also provide a list of arguments following the class name when instantiating an object:

```
$person = new Person("Fred", 35);
```

These arguments are passed to the class's *constructor*, a special function that initializes the properties of the class.

A constructor is a function in the class called `__construct()`. Here's a constructor for the `Person` class:

```
class Person {
 function __construct($name, $age) {
 $this->name = $name;
 $this->age = $age;
 }
}
```

PHP does not provide for an automatic chain of constructors; that is, if you instantiate an object of a derived class, only the constructor in the derived class is automatically called. For the constructor of the parent class to be called, the constructor in the derived class must explicitly call the constructor. In this example, the `Employee` class constructor calls the `Person` constructor:

```
class Person {
 public $name, $address, $age;

 function __construct($name, $address, $age) {
 $this->name = $name;
 $this->address = $address;
 $this->age = $age;
 }
}

class Employee extends Person {
 public $position, $salary;

 function __construct($name, $address, $age, $position, $salary) {
 parent::__construct($name, $address, $age);

 $this->position = $position;
 $this->salary = $salary;
 }
}
```

## Destructors

When an object is destroyed, such as when the last reference to an object is removed or the end of the script is reached, its *destructor* is called. Because PHP automatically cleans up all resources when they fall out of scope and at the end of a script's execution, their application is limited. The destructor is a method called `__destruct()`:

```
class Building {
 function __destruct() {
 echo "A Building is being destroyed!";
 }
}
```

## Anonymous Classes

While creating mock objects for testing, it's useful to create anonymous classes. An anonymous class behaves the same as any other class, except that you do not provide a name (which means it cannot be directly instantiated):

```
class Person {
 public $name = ‘';

 function getName() {
 return $this->name;
 }
}

// return an anonymous implementation of Person
$anonymous = new class() extends Person {
 public function getName() {
 // return static value for testing purposes
 return "Moana";
 }
}; // note: requires closing semicolon, unlike nonanonymous class definitions
```

Unlike instances of named classes, instances of anonymous classes cannot be serialized. Attempting to serialize an instance of an anonymous class results in an error.

## Introspection

*Introspection* is the ability of a program to examine an object's characteristics, such as its name, parent class (if any), properties, and methods. With introspection, you can write code that operates on any class or object. You don't need to know which methods or properties are defined when you write your code; instead, you can discover that information at runtime, which makes it possible for you to write generic debuggers, serializers, profilers, and the like. In this section, we look at the introspective functions provided by PHP.

### Examining Classes

To determine whether a class exists, use the `class_exists()` function, which takes in a string and returns a Boolean value. Alternately, you can use the `get_declared_classes()` function, which returns an array of defined classes and checks if the class name is in the returned array:

```
$doesClassExist = class_exists(classname);

$classes = get_declared_classes();
$doesClassExist = in_array(classname, $classes);
```

You can get the methods and properties that exist in a class (including those that are inherited from superclasses) using the `get_class_methods()` and `get_class_vars()` functions. These functions take a class name and return an array:

```
$methods = get_class_methods(classname);
$properties = get_class_vars(classname);
```

The class name can be either a variable containing the class name, a bare word, or a quoted string:

```
$class = "Person";
$methods = get_class_methods($class);
$methods = get_class_methods(Person); // same
$methods = get_class_methods("Person"); // same
```

The array returned by `get_class_methods()` is a simple list of method names. The associative array returned by `get_class_vars()` maps property names to values and also includes inherited properties.

One quirk of `get_class_vars()` is that it returns only properties that have default values and are visible in the current scope; there's no way to discover uninitialized properties.

Use `get_parent_class()` to find a class's parent class:

```
$superclass = get_parent_class(classname);
```

Example 6-1 lists the `displayClasses()` function, which displays all currently declared classes and the methods and properties for each.

*Example 6-1. Displaying all declared classes*

```
function displayClasses() {
 $classes = get_declared_classes();

 foreach ($classes as $class) {
 echo "Showing information about {$class}<br />";
 $reflection = new ReflectionClass($class);

 $isAnonymous = $reflection->isAnonymous() ? "yes" : "no";
 echo "Is Anonymous: {$isAnonymous}<br />";

 echo "Class methods:<br />";
 $methods = $reflection->getMethods(ReflectionMethod::IS_STATIC);

 if (!count($methods)) {
 echo "<i>None</i><br />";
 }
 else {
 foreach ($methods as $method) {
 echo "<b>{$method}</b>()<br />";
```

```
 }
 }

 echo "Class properties:<br />";

 $properties = $reflection->getProperties();

 if (!count($properties)) {
 echo "<i>None</i><br />";
 }
 else {
 foreach(array_keys($properties) as $property) {
 echo "<b>\${$property}</b><br />";
 }
 }

 echo "<hr />";
 }
}
```

## Examining an Object

To get the class to which an object belongs, first make sure it is an object using the `is_object()` function, and then get the class with the `get_class()` function:

```
$isObject = is_object(var);
$classname = get_class(object);
```

Before calling a method on an object, you can ensure that it exists using the `method_exists()` function:

```
$methodExists = method_exists(object, method);
```

Calling an undefined method triggers a runtime exception.

Just as `get_class_vars()` returns an array of properties for a class, `get_object_vars()` returns an array of properties set in an object:

```
$array = get_object_vars(object);
```

And just as `get_class_vars()` returns only those properties with default values, `get_object_vars()` returns only those properties that are set:

```
class Person {
 public $name;
 public $age;
}

$fred = new Person;
$fred->name = "Fred";
$props = get_object_vars($fred); // array('name' => "Fred", 'age' => NULL);
```

The `get_parent_class()` function accepts either an object or a class name. It returns the name of the parent class, or `FALSE` if there is no parent class:

```
class A {}
class B extends A {}

$obj = new B;
echo get_parent_class($obj);
echo get_parent_class(B);
AA
```

## Sample Introspection Program

Example 6-2 shows a collection of functions that display a reference page of information about an object's properties, methods, and inheritance tree.

*Example 6-2. Object introspection functions*

```
// return an array of callable methods (include inherited methods)
function getCallableMethods($object): Array {
 $reflection = new ReflectionClass($object);
 $methods = $reflection->getMethods();

 return $methods;
}

// return an array of superclasses
function getLineage($object): Array {
 $reflection = new ReflectionClass($object);

 if ($reflection->getParentClass()) {
 $parent = $reflection->getParentClass();

 $lineage = getLineage($parent);
 $lineage[] = $reflection->getName();
 }
 else {
 $lineage = array($reflection->getName());
 }

 return $lineage;
}

// return an array of subclasses
function getChildClasses($object): Array {
 $reflection = new ReflectionClass($object);

 $classes = get_declared_classes();

 $children = array();
```

```
  foreach ($classes as $class) {
  $checkedReflection = new ReflectionClass($class);

  if ($checkedReflection->isSubclassOf($reflection->getName())) {
  $children[] = $checkedReflection->getName();
  }
  }

  return $children;
}

// return an array of properties
function getProperties($object): Array {
  $reflection = new ReflectionClass($object);

  return $reflection->getProperties();
}

// display information on an object
function printObjectInfo($object) {
  $reflection = new ReflectionClass($object);
  echo "<h2>Class</h2>";
  echo "<p>{$reflection->getName()}</p>";

  echo "<h2>Inheritance</h2>";

  echo "<h3>Parents</h3>";
  $lineage = getLineage($object);
  array_pop($lineage);

  if (count($lineage) > 0) {
  echo "<p>" . join(" -&gt; ", $lineage) . "</p>";
  }
  else {
  echo "<i>None</i>";
  }

  echo "<h3>Children</h3>";
  $children = getChildClasses($object);
  echo "<p>";

  if (count($children) > 0) {
  echo join(', ', $children);
  }
  else {
  echo "<i>None</i>";
  }

  echo "</p>";

  echo "<h2>Methods</h2>";
  $methods = getCallableMethods($object);
```

```
 if (!count($methods)) {
 echo "<i>None</i><br />";
 }
 else {
 foreach($methods as $method) {
 echo "<b>{$method}</b>();<br />";
 }
 }

 echo "<h2>Properties</h2>";
 $properties = getProperties($object);

 if (!count($properties)) {
 echo "<i>None</i><br />";
 }
 else {
 foreach(array_keys($properties) as $property) {
 echo "<b>\${$property}</b> = " . $object->$property . "<br />";
 }
 }

 echo "<hr />";
}
```

Here are some sample classes and objects that exercise the introspection functions from Example 6-2:

```
class A {
 public $foo = "foo";
 public $bar = "bar";
 public $baz = 17.0;

 function firstFunction() { }

 function secondFunction() { }
}

class B extends A {
 public $quux = false;

 function thirdFunction() { }
}

class C extends B { }

$a = new A();
$a->foo = "sylvie";
$a->bar = 23;

$b = new B();
$b->foo = "bruno";
```

```
$b->quux = true;

$c = new C();

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

# Serialization

*Serializing* an object means converting it to a bytestream representation that can be stored in a file. This is useful for persistent data; for example, PHP sessions automatically save and restore objects. Serialization in PHP is mostly automatic—it requires little extra work from you, beyond calling the `serialize()` and `unserialize()` functions:

```
$encoded = serialize(something);
$something = unserialize(encoded);
```

Serialization is most commonly used with PHP's sessions, which handle the serialization for you. All you need to do is tell PHP which variables to keep track of, and they're automatically preserved between visits to pages on your site. However, sessions are not the only use of serialization—if you want to implement your own form of persistent objects, `serialize()` and `unserialize()` are a natural choice.

An object's class must be defined before unserialization can occur. Attempting to unserialize an object whose class is not yet defined puts the object into `stdClass`, which renders it almost useless. One practical consequence of this is that if you use PHP sessions to automatically serialize and unserialize objects, you must include the file containing the object's class definition in every page on your site. For example, your pages might start like this:

```
include "object_definitions.php"; // load object definitions
session_start(); // load persistent variables
?>
<html>...
```

PHP has two hooks for objects during the serialization and unserialization process: `__sleep()` and `__wakeup()`. These methods are used to notify objects that they're being serialized or unserialized. Objects can be serialized if they do not have these methods; however, they won't be notified about the process.

The `__sleep()` method is called on an object just before serialization; it can perform any cleanup necessary to preserve the object's state, such as closing database connections, writing out unsaved persistent data, and so on. It should return an array containing the names of the data members that need to be written into the bytestream. If you return an empty array, no data is written.

Conversely, the `__wakeup()` method is called on an object immediately after an object is created from a bytestream. The method can take any action it requires, such as reopening database connections and other initialization tasks.

Example 6-3 is an object class, `Log`, that provides two useful methods: `write()` to append a message to the logfile, and `read()` to fetch the current contents of the logfile. It uses `__wakeup()` to reopen the logfile and `__sleep()` to close the logfile.

*Example 6-3. The Log.php file*

```
class Log {
 private $filename;
 private $fh;

 function __construct($filename) {
 $this->filename = $filename;
 $this->open();
 }

 function open() {
 $this->fh = fopen($this->filename, 'a') or die("Can't open {$this->filename}");
 }

 function write($note) {
 fwrite($this->fh, "{$note}\n");
 }

 function read() {
 return join('', file($this->filename));
 }

 function __wakeup(array $data): void {
 $this->filename = $data["filename"];
 $this->open();
 }

 function __sleep() {
 // write information to the account file
 fclose($this->fh);

 return ["filename" => $this->filename];
 }
}
```

Store the `Log` class definition in a file called *Log.php*. The HTML front page in Example 6-4 uses the `Log` class and PHP sessions to create a persistent log variable, `$logger`.

*Example 6-4. front.php*

```
<?php
include_once "Log.php";
session_start();
?>

<html><head><title>Front Page</title></head>
<body>

<?php
$now = strftime("%c");

if (!isset($_SESSION['logger'])) {
 $logger = new Log("/tmp/persistent_log");
 $_SESSION['logger'] = $logger;
 $logger->write("Created $now");

 echo("<p>Created session and persistent log object.</p>");
}
else {
 $logger = $_SESSION['logger'];
}

$logger->write("Viewed first page {$now}");

echo "<p>The log contains:</p>";
echo nl2br($logger->read());
?>

<a href="next.php">Move to the next page</a>

</body></html>
```

Example 6-5 shows the file *next.php*, an HTML page. Following the link from the front page to this page triggers the loading of the persistent object `$logger`. The `__wakeup()` call reopens the logfile so the object is ready to be used.

*Example 6-5. next.php*

```
<?php
include_once "Log.php";
session_start();
?>

<html><head><title>Next Page</title></head>
<body>

<?php
$now = strftime("%c");
$logger = $_SESSION['logger'];
```

```
$logger->write("Viewed page 2 at {$now}");

echo "<p>The log contains:";
echo nl2br($logger->read());
echo "</p>";
?>

</body></html>
```

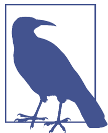
## What's Next

Learning how to use objects in your own scripts is an enormous task. In the next chapter, we transition from language semantics to practice and show you one of PHP's most commonly used set of object-oriented classes—the date and time classes.

# CHAPTER 7
# Dates and Times

The typical PHP developer likely needs to be aware of the available date and time functions, such as when adding a date stamp to a database record entry or calculating the difference between two dates. PHP provides a `DateTime` class that can handle both date and time information simultaneously, as well as a `DateTimeZone` class that works hand in hand with it.

Time zone management has become more prominent in recent years with the onset of web portals and social web communities like Facebook and Twitter. To be able to post information to a website and have it recognize where you are in the world in relation to others on the same site is definitely a requirement these days. However, keep in mind that a function like `date()` takes the default information from the server on which the script is running, so unless the human clients tell you where they are in the world, it can be quite difficult to determine time zone location automatically. Once you know the information, though, it's easy to manipulate that data (more on time zones later in this chapter).



The original date (and related) functions contain a timing flaw on Windows and some Unix installations. They cannot process dates prior to December 13, 1901, or beyond January 19, 2038, due to the nature of the underlying 32-bit signed integer used to manage the date and time data. Therefore, it is recommended to use the newer `DateTime` class family for better accuracy going forward.

There are four interrelated classes for handling dates and times. The `DateTime` class handles dates themselves; the `DateTimeZone` class handles time zones; the `DateInterval` class handles spans of time between two `DateTime` instances; and finally, the `DatePeriod` class handles traversal over regular intervals of dates and times. There are two other rarely used supporting classes called `DateTimeImmutable`

and DateTimeInterface that are part of the whole DateTime "family," but we won't cover those in this chapter.

The constructor of the DateTime class is naturally where it all starts. This method takes two parameters, the timestamp and the time zone. For example:

```
$dt = new DateTime("2019-06-27 16:42:33", new DateTimeZone("America/Halifax"));
```

We create the $dt object, assign it a date and time string with the first parameter, and set the time zone with the second parameter. Here, we're instantiating the DateTimeZone instance inline, but you could alternately instantiate the DateTimeZone object into its own variable and then use that in the constructor, like so:

```
$dtz = new DateTimeZone("America/Halifax");
$dt = new DateTime("2019-06-27 16:42:33", $dtz);
```

Now obviously we are assigning hardcoded values to these classes, and this type of information may not always be available to your code or it may not be what you want. Alternatively, we can pick up the value of the time zone from the server and use that inside the DateTimeZone class. To pick up the current server value, use code similar to the following:

```
$tz = ini_get('date.timezone');
$dtz = new DateTimeZone($tz);
$dt = new DateTime("2019-06-27 16:42:33", $dtz);
```

These code examples establish a set of values for two classes, DateTime and DateTimeZone. Eventually, you will be using that information in some way elsewhere in your script. One of the methods of the DateTime class is called format(), and it uses the same formatting output codes as the date_format() function does. Those date format codes are all listed in the Appendix, in the section for the date_format() function. Here is a sample of the format() method being sent to the browser as output:

```
echo "date: " . $dt->format("Y-m-d h:i:s");
date: 2019-06-27 04:42:33
```

So far we have provided the date and time to the constructor, but sometimes you will also want to pick up the date and time values from the server. To do that, simply provide the string "now" as the first parameter.

The following code does the same as the other examples, except here we are getting the date and time class values from the server. In fact, since we are getting the information from the server, the class properties are much more fully populated (note that some instances of PHP will not have this parameter set and thus will return an error, and the server's time zone may not match your own):

```
$tz = ini_get('date.timezone');
$dtz = new DateTimeZone($tz);
$dt = new DateTime("now", $dtz);
```

```
echo "date: " . $dt->format("Y-m-d h:i:s");
date: 2019-06-27 04:02:54
```

The `diff()` method of `DateTime` does what you might expect—it returns the difference between two dates. The return value of the method is an instance of the `DateInterval` class.

To get the difference between two `DateTime` instances, use:

```
$tz = ini_get('date.timezone');
$dtz = new DateTimeZone($tz);

$past = new DateTime("2019-02-12 16:42:33", $dtz);
$current = new DateTime("now", $dtz);

// creates a new instance of DateInterval
$diff = $past->diff($current);

$pastString = $past->format("Y-m-d");
$currentString = $current->format("Y-m-d");
$diffString = $diff->format("%yy %mm, %dd");

echo "Difference between {$pastString} and {$currentString} is {$diffString}";
Difference between 2019-02-12 and 2019-06-27 is 0y 4m, 14d
```

The `diff()` method is called on one of the `DateTime` objects with the other `DateTime` object passed in as a parameter. Then we prepare the browser output with the `format()` method calls.

Notice that the `DateInterval` class has a `format()` method as well. Since it deals with the difference between two dates, the format character codes are slightly different from that of the `DateTime` class. Precede each character code with a percent sign, `%`. The available character codes are provided in Table 7-1.

*Table 7-1. DateInterval formatting control characters*

| Character | Formatting Effect |
|---|---|
| `a` | Number of days (e.g., 23) |
| `d` | Number of days not already included in the number of months |
| `D` | Number of days, including a leading zero if under 10 days (e.g., 02 and 125) |
| `f` | Numeric microseconds (e.g., 6602 or 41569) |
| `F` | Numeric microseconds with leading zero, at least six digits in length (e.g., 006602 or 041569) |
| `h` | Number of hours |
| `H` | Number of hours, including a leading zero if under 10 hours (e.g., 12 and 04) |
| `i` | Number of minutes |
| `I` | Number of minutes, including a leading zero if under 10 minutes (e.g., 05 and 33) |
| `m` | Number of months |
| `M` | Number of months, including a leading zero if under 10 months (e.g., 05 and 1533) |

| Character | Formatting Effect |
| --- | --- |
| r | – if the difference is negative; empty if the difference is positive |
| R | – if the difference is negative; + if the difference is positive |
| s | Number of seconds |
| S | Number of seconds, including a leading zero if under 10 seconds (e.g., 05 and 15) |
| y | Number of years |
| Y | Number of years, including a leading zero if under 10 years (e.g., 00 and 12) |
| % | A literal % |

Let's look a little more closely at the `DateTimeZone` class now. The time zone setting can be lifted out of the *php.ini* file with `get_ini()`. You can get more information from the time zone object using the `getLocation()` method. It provides the country of origin of the time zone, the longitude and the latitude, plus some comments. With these few lines of code, you can have the beginnings of a web-based GPS system:

```
$tz = ini_get('date.timezone');
$dtz = new DateTimeZone($tz);

echo "Server's Time Zone: {$tz}<br/>";

foreach ($dtz->getLocation() as $key => $value) {
 echo "{$key} {$value}<br/>";
}
Server's Time Zone: America/Halifax
country_code CA
latitude 44.65
longitude -63.6
comments Atlantic - NS (most areas); PE
```

If you want to set a time zone other than the server's, you must pass that value to the constructor of the `DateTimeZone` object. This example sets the time zone for Rome, Italy, and displays the information with the `getLocation()` method:

```
$dtz = new DateTimeZone("Europe/Rome");

echo "Time Zone: " . $dtz->getName() . "<br/>";

foreach ($dtz->getLocation() as $key => $value) {
 echo "{$key} {$value}<br/>";
}

Time Zone: Europe/Rome
country_code IT
latitude 41.9
longitude 12.48333
comments
```

A list of valid time zone names by global regions can be found in the PHP online manual.

Using this same technique, you can make a website "local" to a visitor by providing a list of supported time zones for the visitor to choose from and then temporarily adjusting your *php.ini* setting with the `ini_set()` function for the duration of the visit.

While there's a fair amount of date and time processing power provided by the classes that we discussed in this chapter, it's only the proverbial tip of the iceberg. Be sure to read more about these classes and what they can do on the PHP website.

## What's Next

There's so much more than date management to understand when you're designing websites within PHP, and as a result there are many issues that can cause you stress and increase the PITA (pain in the ass) factor. The next chapter provides multiple tips and tricks, as well as some "gotchas" to watch out for, to help reduce these pain points. Techniques for working with variables, managing form data, and using SSL (Secure Sockets Layer) web data security are among the topics covered. Buckle up!

CHAPTER 8

# Web Techniques

PHP was designed as a web-scripting language and, although it is possible to use it in purely command-line and GUI scripts, the web accounts for the vast majority of PHP uses. A dynamic website may have forms, sessions, and sometimes redirection, and this chapter explains how to implement those elements in PHP. You'll learn how PHP provides access to form parameters and uploaded files, how to send cookies and redirect the browser, how to use PHP sessions, and more.

## HTTP Basics

The web runs on HTTP, or Hypertext Transfer Protocol. This protocol governs how web browsers request files from web servers and how the servers send the files back. To understand the various techniques we'll show you in this chapter, you need to have a basic understanding of HTTP. For a more thorough discussion of HTTP, see the *HTTP Pocket Reference* (O'Reilly) by Clinton Wong.

When a web browser requests a web page, it sends an HTTP request message to a web server. The request message always includes some header information, and it sometimes also includes a body. The web server responds with a reply message, which always includes header information and usually contains a body. The first line of an HTTP request looks like this:

```
GET /index.html HTTP/1.1
```

This line specifies an HTTP command, called a *method*, followed by the address of a document and the version of the HTTP protocol being used. In this case, the request is using the `GET` method to ask for the *index.html* document using HTTP 1.1. After this initial line, the request can contain optional header information that gives the server additional data about the request.

For example:

```
User-Agent: Mozilla/5.0 (Windows 2000; U) Opera 6.0 [en]
Accept: image/gif, image/jpeg, text/*, */*
```

The `User-Agent` header provides information about the web browser, while the `Accept` header specifies the MIME types that the browser accepts. After any headers, the request contains a blank line to indicate the end of the header section. The request can also contain additional data, if that is appropriate for the method being used (e.g., with the `POST` method, as we'll discuss shortly). If the request doesn't contain any data, it ends with a blank line.

The web server receives the request, processes it, and sends a response. The first line of an HTTP response looks like this:

```
HTTP/1.1 200 OK
```

This line specifies the protocol version, a status code, and a description of that code. In this case, the status code is `200`, meaning that the request was successful (hence the description `OK`). After the status line, the response contains headers that give the client additional information about the response. For example:

```
Date: Sat, 29 June 2019 14:07:50 GMT
Server: Apache/2.2.14 (Ubuntu)
Content-Type: text/html
Content-Length: 1845
```

The `Server` header provides information about the web server software, while the `Content-Type` header specifies the MIME type of the data included in the response. After the headers, the response contains a blank line, followed by the requested data if the request was successful.

The two most common HTTP methods are `GET` and `POST`. The `GET` method is designed for retrieving information, such as a document, an image, or the results of a database query, from the server. The `POST` method is meant for posting information, such as a credit card number or information to be stored in a database, to the server. The `GET` method is what a web browser uses when the user types in a URL or clicks on a link. When the user submits a form, either the `GET` or `POST` method can be used, as specified by the `method` attribute of the `form` tag. We'll discuss the `GET` and `POST` methods in more detail in the section "Processing Forms".

# Variables

Server configuration and request information—including form parameters and cookies—are accessible in three different ways from your PHP scripts. Collectively, this information is referred to as *EGPCS* (short for *environment*, `GET`, `POST`, *cookies*, and *server*).

PHP creates six global arrays that contain the EGPCS information:

`$_ENV`
: Contains the values of any environment variables, where the keys of the array are the names of the environment variables.

`$_GET`
: Contains any parameters that are part of a `GET` request, where the keys of the array are the names of the form parameters.

`$_COOKIE`
: Contains any cookie values passed as part of the request, where the keys of the array are the names of the cookies.

`$_POST`
: Contains any parameters that are part of a `POST` request, where the keys of the array are the names of the form parameters.

`$_SERVER`
: Contains useful information about the web server, as described in the next section.

`$_FILES`
: Contains information about any uploaded files.

These variables are not only global, but also visible from within function definitions. The `$_REQUEST` array is created by PHP automatically and contains the elements of the `$_GET`, `$_POST`, and `$_COOKIE` arrays all in one array variable.

## Server Information

The `$_SERVER` array contains a lot of useful information from the web server, much of which comes from the environment variables required in the Common Gateway Interface (CGI) specification). Here is a complete list of the `$_SERVER` entries that come from CGI, including some example values:

`PHP_SELF`
: The name of the current script, relative to the document root (e.g., `/store/cart.php`). You have already seen this used in some of the sample code in earlier chapters. This variable is useful when creating self-referencing scripts, as we'll see later.

`SERVER_SOFTWARE`
: A string that identifies the server (e.g., `"Apache/1.3.33 (Unix) mod_perl/1.26 PHP/5.0.4"`).

SERVER_NAME
: The hostname, DNS alias, or IP address for self-referencing URLs (e.g., `www.example.com`).

GATEWAY_INTERFACE
: The version of the CGI standard being followed (e.g., `CGI/1.1`).

SERVER_PROTOCOL
: The name and revision of the request protocol (e.g., `HTTP/1.1`).

SERVER_PORT
: The server port number to which the request was sent (e.g., `80`).

REQUEST_METHOD
: The method the client used to fetch the document (e.g., `GET`).

PATH_INFO
: Extra path elements given by the client (e.g., `/list/users`).

PATH_TRANSLATED
: The value of `PATH_INFO`, translated by the server into a filename (e.g., `/home/httpd/htdocs/list/users`).

SCRIPT_NAME
: The URL path to the current page, which is useful for self-referencing scripts (e.g., `/~me/menu.php`).

QUERY_STRING
: Everything after the `?` in the URL (e.g., `name=Fred+age=35`).

REMOTE_HOST
: The hostname of the machine that requested this page (e.g., `http://dialup-192-168-0-1.example.com`). If there's no DNS for the machine, this is blank and `REMOTE_ADDR` is the only information given.

REMOTE_ADDR
: A string containing the IP address of the machine that requested this page (e.g., `"192.168.0.250"`).

AUTH_TYPE
: The authentication method used to protect the page, if the page is password-protected (e.g., `basic`).

REMOTE_USER
: The username with which the client authenticated, if the page is password-protected (e.g., fred). Note that there's no way to find out what password was used.

The Apache server also creates entries in the $_SERVER array for each HTTP header in the request. For each key, the header name is converted to uppercase, hyphens (-) are turned into underscores (_), and the string "HTTP_" is prepended. For example, the entry for the User-Agent header has the key "HTTP_USER_AGENT". The two most common and useful headers are:

HTTP_USER_AGENT
: The string the browser used to identify itself (e.g., "Mozilla/5.0 (Windows 2000; U) Opera 6.0 [en]").

HTTP_REFERER
: The page the browser said it came from to get to the current page (e.g., http://www.example.com/last_page.html).

# Processing Forms

It's easy to process forms with PHP, as the form parameters are available in the $_GET and $_POST arrays. This section describes some tricks and techniques that will make it even easier.

## Methods

As we already discussed, there are two HTTP methods that a client can use to pass form data to the server: GET and POST. The method that a particular form uses is specified with the method attribute to the form tag. In theory, methods are case-insensitive in HTML, but in practice some broken browsers require the method name to be in all uppercase.

A GET request encodes the form parameters in the URL in a *query string*, which is indicated by the text that follows the ?:

```
/path/to/chunkify.php?word=despicable&length=3
```

A POST request passes the form parameters in the body of the HTTP request, leaving the URL untouched.

The most visible difference between GET and POST is the URL line. Because all of a form's parameters are encoded in the URL with a GET request, users can bookmark GET queries. They cannot do this with POST requests, however.

The biggest difference between GET and POST requests, however, is far subtler. The HTTP specification says that GET requests are *idempotent*—that is, one GET request for a particular URL, including form parameters, is the same as two or more requests for that URL. Thus, web browsers can cache the response pages for GET requests, because the response page doesn't change regardless of how many times the page is loaded. Because of idempotence, GET requests should be used only for queries such as splitting a word into smaller chunks or multiplying numbers, where the response page is never going to change.

POST requests are not idempotent. This means that they cannot be cached, and the server is contacted every time the page is displayed. You've probably seen your web browser prompt you with "Repost form data?" before displaying or reloading certain pages. This makes POST requests the appropriate choice for queries whose response pages may change over time—for example, displaying the contents of a shopping cart or the current messages in a bulletin board.

That said, idempotence is often ignored in the real world. Browser caches are generally so poorly implemented, and the Reload button so easy to hit, that programmers tend to use GET and POST simply based on whether they want the query parameters shown in the URL or not. What you need to remember is that GET requests should not be used for any actions that cause a change in the server, such as placing an order or updating a database.

The type of method that was used to request a PHP page is available through $_SERVER['REQUEST_METHOD']. For example:

```
if ($_SERVER['REQUEST_METHOD'] == 'GET') {
 // handle a GET request
}
else {
 die("You may only GET this page.");
}
```

## Parameters

Use the $_POST, $_GET, and $_FILES arrays to access form parameters from your PHP code. The keys are the parameter names, and the values are the values of those parameters. Because periods are legal in HTML field names but not in PHP variable names, periods in field names are converted to underscores (_) in the array.

Example 8-1 shows an HTML form that chunkifies a string supplied by the user. The form contains two fields: one for the string (parameter name word) and one for the size of chunks to produce (parameter name number).

*Example 8-1. The chunkify form (chunkify.html)*

```
<html>
 <head><title>Chunkify Form</title></head>

 <body>
 <form action="chunkify.php" method="POST">
 Enter a word: <input type="text" name="word" /><br />

 How long should the chunks be?
 <input type="text" name="number" /><br />
 <input type="submit" value="Chunkify!">
 </form>
</body>

</html>
```

Example 8-2 lists the PHP script, *chunkify.php*, to which the form in Example 8-1 submits. The script copies the parameter values into variables and uses them.

*Example 8-2. The chunkify script (chunkify.php)*

```
<?php
$word = $_POST['word'];
$number = $_POST['number'];

$chunks = ceil(strlen($word) / $number);

echo "The {$number}-letter chunks of '{$word}' are:<br />\n";

for ($i = 0; $i < $chunks; $i++) {
 $chunk = substr($word, $i * $number, $number);
 printf("%d: %s<br />\n", $i + 1, $chunk);
}
?>
```

Figure 8-1 shows both the chunkify form and the resulting output.

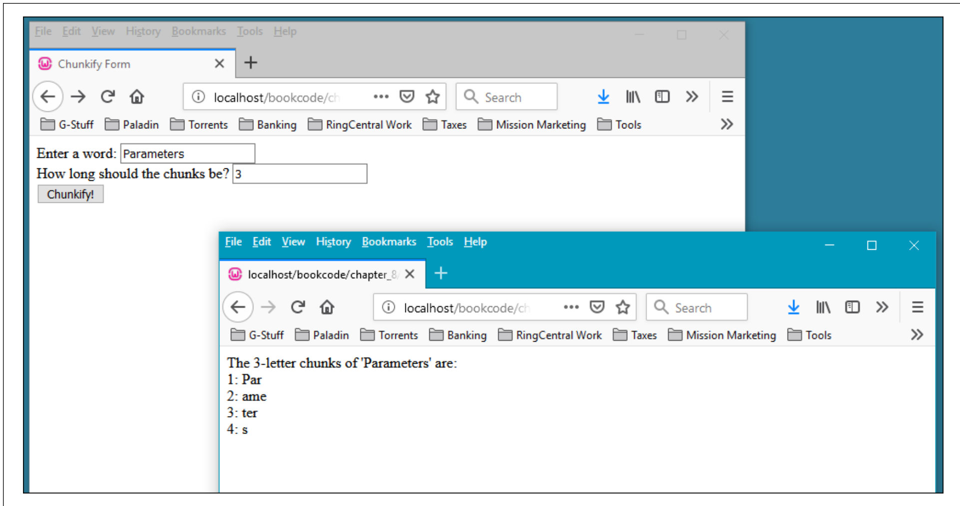



*Figure 8-1. The chunkify form and its output*

## Self-Processing Pages

One PHP page can be used to both generate a form and subsequently process it. If the page shown in Example 8-3 is requested with the `GET` method, it prints a form that accepts a Fahrenheit temperature. If called with the `POST` method, however, the page calculates and displays the corresponding Celsius temperature.

*Example 8-3. A self-processing temperature conversion page (temp.php)*

```
<html>
<head><title>Temperature Conversion</title></head>
<body>

<?php if ($_SERVER['REQUEST_METHOD'] == 'GET') { ?>
 <form action="<?php echo $_SERVER['PHP_SELF'] ?>" method="POST">
 Fahrenheit temperature:
 <input type="text" name="fahrenheit" /><br />
 <input type="submit" value="Convert to Celsius!" />
 </form>

<?php }
else if ($_SERVER['REQUEST_METHOD'] == 'POST') {
 $fahrenheit = $_POST['fahrenheit'];
 $celsius = ($fahrenheit - 32) * 5 / 9;

 printf("%.2fF is %.2fC", $fahrenheit, $celsius);
}
else {
 die("This script only works with GET and POST requests.");
```

```
} ?>

</body>
</html>
```

Figure 8-2 shows the temperature-conversion page and the resulting output.

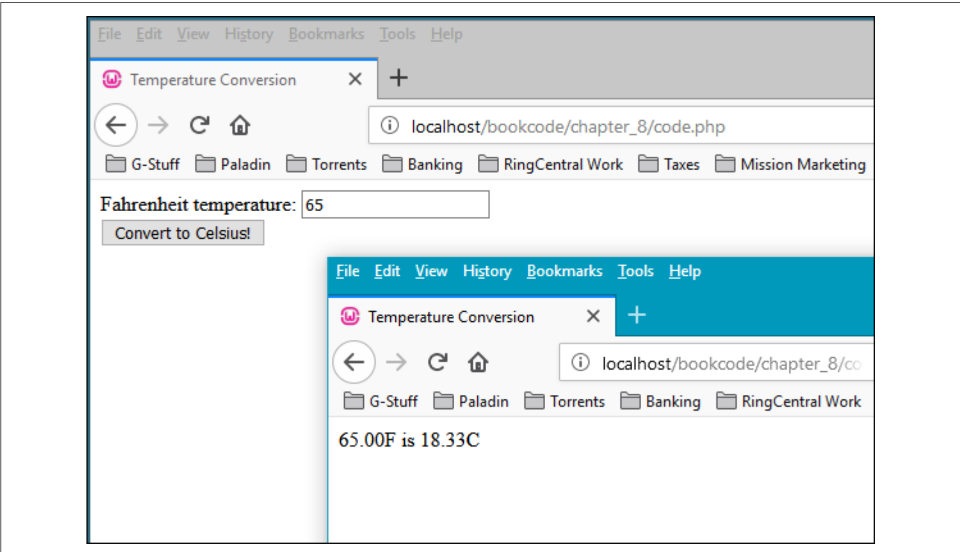



*Figure 8-2. The temperature-conversion page and its output*

Another way for a script to decide whether to display a form or process it is to see whether or not one of the parameters has been supplied. This lets you write a self-processing page that uses the GET method to submit values. Example 8-4 shows a new version of the temperature-conversion page that submits parameters using a GET request. This page uses the presence or absence of parameters to determine what to do.

*Example 8-4. Temperature conversion using the GET method (temp2.php)*

```
<html>
<head>
<title>Temperature Conversion</title>
</head>
<body>
<?php
if (isset ( $_GET ['fahrenheit'] )) {
 $fahrenheit = $_GET ['fahrenheit'];
} else {
 $fahrenheit = null;
}
```

```
if (is_null ( $fahrenheit )) {
 ?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="GET">
 Fahrenheit temperature: <input type="text" name="fahrenheit" /><br />
 <input type="submit" value="Convert to Celsius!" />
 </form>
<?php
} else {
 $celsius = ($fahrenheit - 32) * 5 / 9;
 printf ( "%.2fF is %.2fC", $fahrenheit, $celsius );
}
?>
</body>
</html>
```

In Example 8-4, we copy the form parameter value into `$fahrenheit`. If we weren't given that parameter, `$fahrenheit` contains `NULL`, so we could use `is_null()` to test whether we should display the form or process the form data.

## Sticky Forms

Many websites use a technique known as *sticky forms*, in which the results of a query are accompanied by a search form whose default values are those of the previous query. For instance, if you search Google for "Programming PHP," the top of the results page contains another search box, which already contains "Programming PHP." To refine your search to "Programming PHP from O'Reilly," you can simply add the extra keywords.

This sticky behavior is easy to implement. Example 8-5 shows our temperature-conversion script from Example 8-4, with the form made sticky. The basic technique is to use the submitted form value as the default value when creating the HTML field.

*Example 8-5. Temperature conversion with a sticky form (sticky_form.php)*

```
<html>
<head><title>Temperature Conversion</title></head>
<body>
<?php $fahrenheit = $_GET['fahrenheit']; ?>

<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="GET">
 Fahrenheit temperature:
 <input type="text" name="fahrenheit" value="<?php echo $fahrenheit; ?>" /><br />
 <input type="submit" value="Convert to Celsius!" />
</form>

<?php if (!is_null($fahrenheit)) {
 $celsius = ($fahrenheit - 32) * 5 / 9;
 printf("%.2fF is %.2fC", $fahrenheit, $celsius);
} ?>
```

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

## Multivalued Parameters

HTML selection lists, created with the `select` tag, can allow multiple selections. To ensure that PHP recognizes the multiple values that the browser passes to a form-processing script, you need to use square brackets, `[]`, after the name of the field in the HTML form. For example:

```
<select name="languages[]">

 <option name="c">C</option>
 <option name="c++">C++</option>
 <option name="php">PHP</option>
 <option name="perl">Perl</option>
</select>
```

Now, when the user submits the form, `$_GET['languages']` contains an array instead of a simple string. This array contains the values that were selected by the user.

Example 8-6 illustrates multiple selections of values within an HTML selection list. The form provides the user with a set of personality attributes. When the user submits the form, it returns a (not very interesting) description of the user's personality.

*Example 8-6. Multiple selection values with a select box (select_array.php)*

```
<html>
<head><title>Personality</title></head>
<body>

<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="GET">
 Select your personality attributes:<br />
 <select name="attributes[]" multiple>
 <option value="perky">Perky</option>
 <option value="morose">Morose</option>
 <option value="thinking">Thinking</option>
 <option value="feeling">Feeling</option>
 <option value="thrifty">Spend-thrift</option>
 <option value="shopper">Shopper</option>
 </select><br />
 <input type="submit" name="s" value="Record my personality!" />
</form>
<?php if (array_key_exists('s', $_GET)) {
 $description = join(' ', $_GET['attributes']);
 echo "You have a {$description} personality.";
} ?>
```

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

In Example 8-6, the submit button has a name, "s". We check for the presence of this parameter value to see whether we have to produce a personality description. Figure 8-3 shows the multiple-selection page and the resulting output.

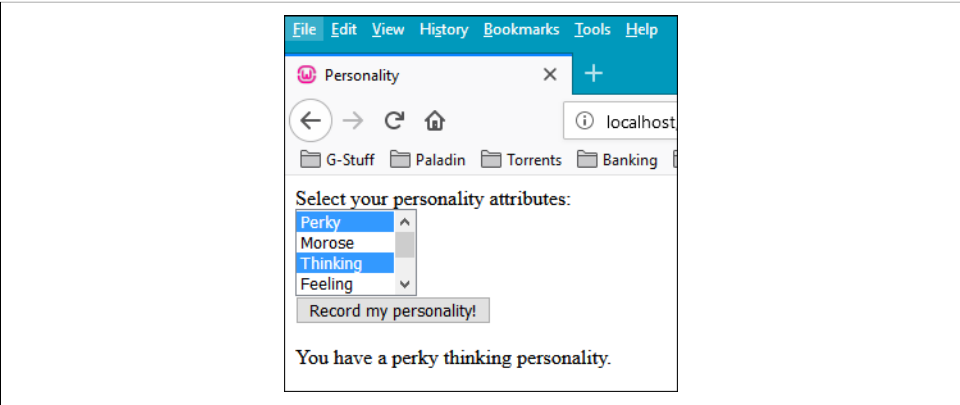



*Figure 8-3. Multiple-selection page and its output*

The same technique applies for any form field where multiple values can be returned. Example 8-7 shows a revised version of our personality form that is rewritten to use checkboxes instead of a select box. Notice that only the HTML has changed—the code to process the form doesn't need to know whether the multiple values came from checkboxes or a select box.

*Example 8-7. Multiple selection values in checkboxes (checkbox_array.php)*

```
<html>
<head><title>Personality</title></head>
<body>

<form action="<?php $_SERVER['PHP_SELF']; ?>" method="GET">
 Select your personality attributes:<br />
 <input type="checkbox" name="attributes[]" value="perky" /> Perky<br />
 <input type="checkbox" name="attributes[]" value="morose" /> Morose<br />
 <input type="checkbox" name="attributes[]" value="thinking" /> Thinking<br />
 <input type="checkbox" name="attributes[]" value="feeling" /> Feeling<br />
 <input type="checkbox" name="attributes[]" value="thrifty" />Spend-thrift<br />
 <input type="checkbox" name="attributes[]" value="shopper" /> Shopper<br />
 <br />
 <input type="submit" name="s" value="Record my personality!" />
</form>
<?php if (array_key_exists('s', $_GET)) {
 $description = join (' ', $_GET['attributes']);
```

```
 echo "You have a {$description} personality.";
} ?>

</body>
</html>
```

## Sticky Multivalued Parameters

So now you're probably wondering, *Can I make multiple-selection form elements sticky?* You can, but it isn't easy. You'll need to check whether each possible value in the form was one of the submitted values. For example:

```
Perky: <input type="checkbox" name="attributes[]" value="perky"
<?php
if (is_array($_GET['attributes']) && in_array('perky', $_GET['attributes'])) {
 echo "checked";
} ?> /><br />
```

You could use this technique for each checkbox, but that's repetitive and error-prone. At this point, it's easier to write a function to generate the HTML for the possible values and work from a copy of the submitted parameters. Example 8-8 shows a new version of the multiple-selection checkboxes, with the form made sticky. Although this form looks just like the one in Example 8-7, behind the scenes there are substantial changes to the way the form is generated.

*Example 8-8. Sticky multivalued checkboxes (checkbox_array2.php)*

```
<html>
<head><title>Personality</title></head>
<body>
<?php // fetch form values, if any
$attrs = $_GET['attributes'];

if (!is_array($attrs)) {
 $attrs = array();
}

// create HTML for identically named checkboxes

function makeCheckboxes($name, $query, $options)
{
 foreach ($options as $value => $label) {
 $checked = in_array($value, $query) ? "checked" : '';

 echo "<input type=\"checkbox\" name=\"{$name}\"
 value=\"{$value}\" {$checked} />";
 echo "{$label}<br />\n";
 }
}
```

```
// the list of values and labels for the checkboxes
$personalityAttributes = array(
 'perky' => "Perky",
 'morose' => "Morose",
 'thinking' => "Thinking",
 'feeling' => "Feeling",
 'thrifty' => "Spend-thrift",
 'prodigal' => "Shopper"
); ?>

<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="GET">
 Select your personality attributes:<br />
 <?php makeCheckboxes('attributes[]', $attrs, $personalityAttributes); ?><br />

 <input type="submit" name="s" value="Record my personality!" />
</form>

<?php if (array_key_exists('s', $_GET)) {
 $description = join (' ', $_GET['attributes']);
 echo "You have a {$description} personality.";
} ?>

</body>
</html>
```

The heart of this code is the `makeCheckboxes()` function. It takes three arguments: the name for the group of checkboxes, the array of on-by-default values, and the array that maps values to descriptions. The list of options for the checkboxes is in the `$personalityAttributes` array.

## File Uploads

To handle file uploads (supported in most modern browsers), use the `$_FILES` array. Using the various authentication and file upload functions, you can control who is allowed to upload files and what to do with those files once they're on your system. Security concerns to take note of are described in Chapter 14.

The following code displays a form that allows file uploads to the same page:

```
<form enctype="multipart/form-data"
 action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <input type="hidden" name="MAX_FILE_SIZE" value="10240">
 File name: <input name="toProcess" type="file" />
 <input type="submit" value="Upload" />
</form>
```

The biggest problem with file uploads is the risk of getting a file that is too large to process. PHP has two ways of preventing this: a hard limit and a soft limit. The `upload_max_filesize` option in *php.ini* gives a hard upper limit on the size of uploaded files (it is set to 2 MB by default). If your form submits a parameter called

MAX_FILE_SIZE before any file field parameters, PHP uses that value as the soft upper limit. For instance, in the previous example, the upper limit is set to 10 KB. PHP ignores attempts to set MAX_FILE_SIZE to a value larger than upload_max_filesize.

Also, notice that the form tag takes an enctype attribute with the value "multipart/form-data".

Each element in $_FILES is itself an array, giving information about the uploaded file. The keys are:

name
: The name of the uploaded file as supplied by the browser. It's difficult to make meaningful use of this, as the client machine may have different filename conventions than the web server (e.g., a file path of *D:\PHOTOS\ME.JPG* from a client machine running Windows would be meaningless to a web server running Unix).

type
: The MIME type of the uploaded file as guessed at by the client.

size
: The size of the uploaded file (in bytes). If the user attempted to upload a file that was too large, the size would be reported as 0.

tmp_name
: The name of the temporary file on the server that holds the uploaded file. If the user attempted to upload a file that was too large, the name is given as "none".

The correct way to test whether a file was successfully uploaded is to use the function is_uploaded_file(), as follows:

```
if (is_uploaded_file($_FILES['toProcess']['tmp_name'])) {
 // successfully uploaded
}
```

Files are stored in the server's default temporary files directory, which is specified in *php.ini* with the upload_tmp_dir option. To move a file, use the move_uploaded_file() function:

```
move_uploaded_file($_FILES['toProcess']['tmp_name'], "path/to/put/file/{$file}");
```

The call to move_uploaded_file() automatically checks whether it was an uploaded file. When a script finishes, any files uploaded to that script are deleted from the temporary directory.

## Form Validation

When you allow users to input data, you typically need to validate that data before using it or storing it for later use. There are several strategies available for validating data. The first is JavaScript on the client side. However, since the user can choose to turn JavaScript off, or may even be using a browser that doesn't support it, this cannot be the only validation you do.

A more secure choice is to use PHP to do the validation. Example 8-9 shows a self-processing page with a form. The page allows the user to input a media item; three of the form elements—the name, media type, and filename—are required. If the user neglects to give a value to any of them, the page is presented anew with a message detailing what's wrong. Any form fields the user already filled out are set to the values originally entered. Finally, as an additional clue to the user, the text of the submit button changes from "Create" to "Continue" when the user is correcting the form.

*Example 8-9. Form validation (data_validation.php)*

```
<?php
$name = $_POST['name'];
$mediaType = $_POST['media_type'];
$filename = $_POST['filename'];
$caption = $_POST['caption'];
$status = $_POST['status'];

$tried = ($_POST['tried'] == 'yes');

if ($tried) {
 $validated = (!empty($name) && !empty($mediaType) && !empty($filename));

 if (!$validated) { ?>
 <p>The name, media type, and filename are required fields. Please fill
 them out to continue.</p>
 <?php }
}

if ($tried && $validated) {
 echo "<p>The item has been created.</p>";
}

// was this type of media selected? print "selected" if so
function mediaSelected($type)
{
 global $mediaType;

 if ($mediaType == $type) {
 echo "selected"; }
} ?>
```

```
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 Name: <input type="text" name="name" value="<?php echo $name; ?>" /><br />

 Status: <input type="checkbox" name="status" value="active"
 <?php if ($status == "active") { echo "checked"; } ?> /> Active<br />

 Media: <select name="media_type">
 <option value="">Choose one</option>
 <option value="picture" <?php mediaSelected("picture"); ?> />Picture</option>
 <option value="audio" <?php mediaSelected("audio"); ?> />Audio</option>
 <option value="movie" <?php mediaSelected("movie"); ?> />Movie</option>
 </select><br />

 File: <input type="text" name="filename" value="<?php echo $filename; ?>" /><br />

 Caption: <textarea name="caption"><?php echo $caption; ?></textarea><br />

 <input type="hidden" name="tried" value="yes" />
 <input type="submit" value="<?php echo $tried ? "Continue" : "Create"; ?>" />
</form>
```

In this case, the validation is simply a check that a value was supplied. We set `$validated` to be `true` only if `$name`, `$type`, and `$filename` are all nonempty. Other possible validations include checking that an email address is valid or checking that the supplied filename is local and exists.

For example, to validate an age field to ensure that it contains a non-negative integer, use this code:

```
$age = $_POST['age'];
$validAge = strspn($age, "1234567890") == strlen($age);
```

The call to `strspn()` finds the number of digits at the start of the string. In a non-negative integer, the whole string should be composed of digits, so it's a valid age if the entire string is made of digits. We could also have done this check with a regular expression:

```
$validAge = preg_match('/^\d+$/', $age);
```

Validating email addresses is a nigh-impossible task. There's no way to take a string and see whether it corresponds to a valid email address. However, you can catch typos by requiring the user to enter the email address twice (into two different fields). You can also prevent people from entering email addresses like *me* or *me@aol* by requiring an at sign (@) and a period somewhere after it, and for bonus points you can check for domains to which you don't want to send mail (e.g., *whitehouse.gov*, or a competitor site). For example:

```
$email1 = strtolower($_POST['email1']);
$email2 = strtolower($_POST['email2']);

if ($email1 !== $email2) {
```

```
 die("The email addresses didn't match");
}

if (!preg_match('/@.+\..+$/', $email1)) {
 die("The email address is malformed");
}

if (strpos($email1, "whitehouse.gov")) {
 die("I will not send mail to the White House");
}
```

Field validation is basically string manipulation. In this example, we've used regular expressions and string functions to ensure that the string provided by the user is the type of string we expect.

# Setting Response Headers

As we've already discussed, the HTTP response that a server sends back to a client contains headers that identify the type of content in the body of the response, the server that sent the response, how many bytes are in the body, when the response was sent, and so on. PHP and Apache normally take care of the headers for you (identifying the document as HTML, calculating the length of the HTML page, etc.). Most web applications never need to set headers themselves. However, if you want to send back something that's not HTML, set the expiration time for a page, redirect the client's browser, or generate a specific HTTP error, you'll need to use the `header()` function.

The only catch to setting headers is that you must do so before any of the body is generated. This means that all calls to `header()` (or `setcookie()`, if you're setting cookies) must happen at the very top of your file, even before the `<html>` tag. For example:

```
<?php header("Content-Type: text/plain"); ?>
Date: today
From: fred
To: barney
Subject: hands off!

My lunchbox is mine and mine alone. Get your own,
you filthy scrounger!
```

Attempting to set headers after the document has started results in this warning:

```
Warning: Cannot add header information - headers already sent
```

You can instead use an output buffer; see `ob_start()`, `ob_end_flush()`, and related functions for more information on using output buffers.

## Different Content Types

The `Content-Type` header identifies the type of document being returned. Ordinarily this is `"text/html"`, indicating an HTML document, but there are other useful document types. For example, `"text/plain"` forces the browser to treat the page as plain text. This type is like an automatic "view source," and it is useful when debugging.

In Chapter 10 and Chapter 11, we'll make heavy use of the `Content-Type` header as we generate documents that are actually graphic images and Adobe PDF files.

## Redirections

To send the browser to a new URL, known as a *redirection*, you set the `Location` header. Generally, you'll also exit immediately afterward, so the script doesn't bother generating and outputting the remainder of the code listing:

```
header("Location: http://www.example.com/elsewhere.html");
exit();
```

When you provide a partial URL (e.g., `/elsewhere.html`), the web server handles this redirection internally. This is only rarely useful, as the browser generally won't learn that it isn't getting the page it requested. If there are relative URLs in the new document, the browser interprets those URLs as being relative to the requested document, rather than to the document that was ultimately sent. In general, you'll want to redirect to an absolute URL.

## Expiration

A server can explicitly inform the browser, and any proxy caches that might be between the server and browser, of a specific date and time for the document to expire. Proxy and browser caches can hold the document until that time or expire it earlier. Repeated reloads of a cached document do not contact the server. However, an attempt to fetch an expired document does contact the server.

To set the expiration time of a document, use the `Expires` header:

```
header("Expires: Tue, 02 Jul 2019 05:30:00 GMT");
```

To force a document to expire three hours from the time the page was generated, use `time()` and `gmstrftime()` to generate the expiration date string:

```
$now = time();
$then = gmstrftime("%a, %d %b %Y %H:%M:%S GMT", $now + 60 * 60 * 3);

header("Expires: {$then}");
```

To indicate that a document "never" expires, use the time a year from now:

```
$now = time();
$then = gmstrftime("%a, %d %b %Y %H:%M:%S GMT", $now + 365 * 86440);

header("Expires: {$then}");
```

To mark a document as expired, use the current time or a time in the past:

```
$then = gmstrftime("%a, %d %b %Y %H:%M:%S GMT");

header("Expires: {$then}");
```

This is the best way to prevent a browser or proxy cache from storing your document:

```
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("Cache-Control: no-store, no-cache, must-revalidate");
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache");
```

For more information on controlling the behavior of browser and web caches, see Chapter 6 of *Web Caching* (O'Reilly) by Duane Wessels .

## Authentication

HTTP authentication works through request headers and response statuses. A browser can send a username and password (the *credentials*) in the request headers. If the credentials aren't sent or aren't satisfactory, the server sends a "401 Unauthorized" response and identifies the *realm* of authentication (a string such as `"Mary's Pictures"` or `"Your Shopping Cart"`) via the `WWW-Authenticate` header. This typically pops up an "Enter username and password for . . ." dialog box on the browser, and the page is then re-requested with the updated credentials in the header.

To handle authentication in PHP, check the username and password (the `PHP_AUTH_USER` and `PHP_AUTH_PW` items of `$_SERVER`) and call `header()` to set the realm and send a "401 Unauthorized" response:

```
header('WWW-Authenticate: Basic realm="Top Secret Files"');
header("HTTP/1.0 401 Unauthorized");
```

You can do anything you want to authenticate the username and password; for example, you could consult a database, read a file of valid users, or consult a Microsoft domain server.

This example checks to make sure that the password is the username reversed (not the most secure authentication method, to be sure!):

```
$authOK = false;

$user = $_SERVER['PHP_AUTH_USER'];
$password = $_SERVER['PHP_AUTH_PW'];

if (isset($user) && isset($password) && $user === strrev($password)) {
```

```
  $authOK = true;
 }

 if (!$authOK) {
  header('WWW-Authenticate: Basic realm="Top Secret Files"');
  header('HTTP/1.0 401 Unauthorized');

  // anything else printed here is only seen if the client hits "Cancel"
  exit;
 }

 <!-- your password-protected document goes here -->
```

If you're protecting more than one page, put the preceding code into a separate file and include it at the top of every protected page.

If your host is using the CGI version of PHP rather than an Apache module, these variables cannot be set and you'll need to use some other form of authentication—for example, by gathering the username and password through an HTML form.

# Maintaining State

HTTP is a *stateless* protocol, which means that once a web server completes a client's request for a web page, the connection between the two goes away. In other words, there is no way for a server to recognize that a sequence of requests all originate from the same client.

State is useful, though. You can't build a shopping-cart application, for example, if you can't keep track of a sequence of requests from a single user. You need to know when a user adds items to the cart or removes them, and what's in the cart when the user decides to check out.

To get around the web's lack of state, programmers have come up with many tricks to track state information between requests (also known as *session tracking*). One such technique is to use hidden form fields to pass around information. PHP treats hidden form fields just like normal form fields, so the values are available in the `$_GET` and `$_POST` arrays. Using hidden form fields, you can pass around the entire contents of a shopping cart. However, it's more common to assign each user a unique identifier and pass the ID around using a single hidden form field. While hidden form fields work in all browsers, they work only for a sequence of dynamically generated forms, so they aren't as generally useful as some other techniques.

Another technique is URL rewriting, where every local URL on which the user might click is dynamically modified to include extra information. This extra information is often specified as a parameter in the URL. For example, if you assign every user a unique ID, you might include that ID in all URLs, as follows:

```
http://www.example.com/catalog.php?userid=123
```

If you make sure to dynamically modify all local links to include a user ID, you can now keep track of individual users in your application. URL rewriting works for all dynamically generated documents, not just forms, but actually performing the rewriting can be tedious.

The third and most widespread technique for maintaining state is to use cookies. A *cookie* is a bit of information that the server can give to a client. On every subsequent request the client will give that information back to the server, thus identifying itself. Cookies are useful for retaining information through repeated visits by a browser, but they're not without their own problems. The main issue is that most browsers allow users to disable cookies. So any application that uses cookies for state maintenance needs to use another technique as a fallback mechanism. We'll discuss cookies in more detail shortly.

The best way to maintain state with PHP is to use the built-in session-tracking system. This system lets you create persistent variables that are accessible from different pages of your application, as well as in different visits to the site by the same user. Behind the scenes, PHP's session-tracking mechanism uses cookies (or URLs) to elegantly solve most problems that require state, taking care of all the details for you. We'll cover PHP's session-tracking system in detail later in this chapter.

## Cookies

A cookie is basically a string that contains several fields. A server can send one or more cookies to a browser in the headers of a response. Some of the cookie's fields indicate the pages for which the browser should send the cookie as part of the request. The `value` field of the cookie is the payload—servers can store any data they like there (within limits), such as a unique code identifying the user, preferences, and the like.

Use the `setcookie()` function to send a cookie to the browser:

```
setcookie(name [, value [, expires [, path [, domain [, secure [,
httponly ]]]]]]);
```

This function creates the cookie string from the given arguments and creates a `Cookie` header with that string as its value. Because cookies are sent as headers in the response, `setcookie()` must be called before any of the body of the document is sent. The parameters of `setcookie()` are:

*name*
: A unique name for a particular cookie. You can have multiple cookies with different names and attributes. The name must not contain whitespace or semicolons.

*value*
: The arbitrary string value attached to this cookie. The original Netscape specification limited the total size of a cookie (including name, expiration date, and other information) to 4 KB, so while there's no specific limit on the size of a cookie value, it probably can't be much larger than 3.5 KB.

*expires*
: The expiration date for this cookie. If no expiration date is specified, the browser saves the cookie in memory and not on disk. When the browser exits, the cookie disappears. The expiration date is specified as the number of seconds since midnight, January 1, 1970 (GMT). For example, pass `time() + 60 * 60 * 2` to expire the cookie in two hours' time.

*path*
: The browser will return the cookie only for URLs below this path. The default is the directory in which the current page resides. For example, if */store/front/cart.php* sets a cookie and doesn't specify a path, the cookie will be sent back to the server for all pages whose URL path starts with */store/front/*.

*domain*
: The browser will return the cookie only for URLs within this domain. The default is the server hostname.

*secure*
: The browser will transmit the cookie only over *https* connections. The default is `false`, meaning that it's OK to send the cookie over insecure connections.

*httponly*
: If this parameter is set to `TRUE`, the cookie will be available only via the HTTP protocol, and thus inaccessible via other means like JavaScript. Whether this allows for a more secure cookie is still up for debate, so use this parameter cautiously and test well.

The `setcookie()` function also has an alternate syntax:

```
setcookie ($name [, $value = "" [, $options = [] ]] )
```

where `$options` is an array that holds the other parameters following the `$value` content. This saves a little on the code line length for the `setcookie()` function, but the `$options` array will have to be built prior to its use, so there is a trade-off of sorts in play.

When a browser sends a cookie back to the server, you can access that cookie through the `$_COOKIE` array. The key is the cookie name, and the value is the cookie's `value` field. For instance, the following code at the top of a page keeps track of the number of times the page has been accessed by this client:

```
$pageAccesses = $_COOKIE['accesses'];
setcookie('accesses', ++$pageAccesses);
```

When cookies are decoded, any periods (`.`) in a cookie's name are turned into underscores. For instance, a cookie named `tip.top` is accessible as `$_COOKIE['tip_top']`.

Let's take a look at cookies in action. First, Example 8-10 shows an HTML page that gives a range of options for background and foreground colors.

*Example 8-10. Preference selection (colors.php)*

```
<html>
<head><title>Set Your Preferences</title></head>
<body>
<form action="prefs.php" method="post">
 <p>Background:
 <select name="background">
 <option value="black">Black</option>
 <option value="white">White</option>
 <option value="red">Red</option>
 <option value="blue">Blue</option>
 </select><br />

 Foreground:
 <select name="foreground">
 <option value="black">Black</option>
 <option value="white">White</option>
 <option value="red">Red</option>
 <option value="blue">Blue</option>
 </select></p>

 <input type="submit" value="Change Preferences">
</form>

</body>
</html>
```

The form in Example 8-10 submits to the PHP script *prefs.php*, which is shown in Example 8-11. This script then sets cookies for the color preferences specified in the form. Note that the calls to `setcookie()` are made after the HTML page is started.

*Example 8-11. Setting preferences with cookies (prefs.php)*

```
<html>
<head><title>Preferences Set</title></head>
<body>

<?php
$colors = array(
 'black' => "#000000",
```

```
 'white' => "#ffffff",
 'red' => "#ff0000",
 'blue' => "#0000ff"
);

$backgroundName = $_POST['background'];
$foregroundName = $_POST['foreground'];

setcookie('bg', $colors[$backgroundName]);
setcookie('fg', $colors[$foregroundName]);
?>

<p>Thank you. Your preferences have been changed to:<br />
Background: <?php echo $backgroundName; ?><br />
Foreground: <?php echo $foregroundName; ?></p>

<p>Click <a href="prefs_demo.php">here</a> to see the preferences
in action.</p>

</body>
</html>
```

The page created by Example 8-11 contains a link to another page, shown in Example 8-12, that uses the color preferences by accessing the `$_COOKIE` array.

*Example 8-12. Using the color preferences with cookies (prefs_demo.php)*

```
<html>
<head><title>Front Door</title></head>
<?php
$backgroundName = $_COOKIE['bg'];
$foregroundName = $_COOKIE['fg'];
?>
<body bgcolor="<?php echo $backgroundName; ?>" text="<?php echo $foregroundName; ?>">

<h1>Welcome to the Store</h1>

<p>We have many fine products for you to view. Please feel free to browse
the aisles and stop an assistant at any time. But remember, you break it
you bought it!</p>

<p>Would you like to <a href="colors.php">change your preferences?</a></p>

</body>
</html>
```

There are plenty of caveats about the use of cookies. Not all clients (browsers) support or accept cookies, and even if the client does support cookies, the user can turn them off. Furthermore, the cookie specification says that no cookie can exceed 4 KB in size, only 20 cookies are allowed per domain, and a total of 300 cookies can be

stored on the client side. Some browsers may have higher limits, but you can't rely on that. Finally, you have no control over when browsers actually expire cookies—if a browser is at capacity and needs to add a new cookie, it may discard a cookie that has not yet expired. You should also be careful of setting cookies to expire quickly. Expiration times rely on the client's clock being as accurate as yours. Many people do not have their system clocks set accurately, so you can't rely on rapid expirations.

Despite these limitations, cookies are very useful for retaining information through repeated visits by a browser.

## Sessions

PHP has built-in support for sessions, handling all the cookie manipulation for you to provide persistent variables that are accessible from different pages and across multiple visits to the site. Sessions allow you to easily create multipage forms (such as shopping carts), save user authentication information from page to page, and store persistent user preferences on a site.

Each first-time visitor is issued a unique session ID. By default, the session ID is stored in a cookie called `PHPSESSID`. If the user's browser does not support cookies or has cookies turned off, the session ID is propagated in URLs within the website.

Every session has a data store associated with it. You can *register* variables to be loaded from the data store when each page starts and saved back to the data store when the page ends. Registered variables persist between pages, and changes to variables made on one page are visible from others. For example, an "add this to your shopping cart" link can take the user to a page that adds an item to a registered array of items in the cart. This registered array can then be used on another page to display the contents of the cart.

### Session basics

Sessions start automatically when a script begins running. A new session ID is generated if necessary, possibly creating a cookie to be sent to the browser, and loads any persistent variables from the store.

You can register a variable with the session by passing the name of the variable to the `$_SESSION[]` array. For example, here is a basic hit counter:

```
session_start();
$_SESSION['hits'] = $_SESSION['hits'] + 1;

echo "This page has been viewed {$_SESSION['hits']} times.";
```

The `session_start()` function loads registered variables into the associative array `$_SESSION`. The keys are the variables' names (e.g., `$_SESSION['hits']`). If you're curious, the `session_id()` function returns the current session ID.

To end a session, call `session_destroy()`. This removes the data store for the current session, but it doesn't remove the cookie from the browser cache. This means that, on subsequent visits to sessions-enabled pages, the user will have the same session ID as before the call to `session_destroy()`, but none of the data.

Example 8-13 shows the code from Example 8-11 rewritten to use sessions instead of manually setting cookies.

*Example 8-13. Setting preferences with sessions (prefs_session.php)*

```
<?php session_start(); ?>

<html>
<head><title>Preferences Set</title></head>
<body>

<?php
$colors = array(
 'black' => "#000000",
 'white' => "#ffffff",
 'red' => "#ff0000",
 'blue' => "#0000ff"
);

$bg = $colors[$_POST['background']];
$fg = $colors[$_POST['foreground']];

$_SESSION['bg'] = $bg;
$_SESSION['fg'] = $fg;
?>

<p>Thank you. Your preferences have been changed to:<br />
Background: <?php echo $_POST['background']; ?><br />
Foreground: <?php echo $_POST['foreground']; ?></p>

<p>Click <a href="prefs_session_demo.php">here</a> to see the preferences
in action.</p>

</body>
</html>
```

Example 8-14 shows Example 8-12 rewritten to use sessions. Once the session is started, the `$bg` and `$fg` variables are created, and all the script has to do is use them.

*Example 8-14. Using preferences from sessions (prefs_session_demo.php)*

```
<?php
session_start() ;
$backgroundName = $_SESSION['bg'] ;
```

```
$foregroundName = $_SESSION['fg'] ;
?>
<html>
<head><title>Front Door</title></head>
<body bgcolor="<?php echo $backgroundName; ?>" text="<?php echo $foregroundName; ?>">

<h1>Welcome to the Store</h1>

<p>We have many fine products for you to view. Please feel free to browse
the aisles and stop an assistant at any time. But remember, you break it
you bought it!</p>

<p>Would you like to <a href="colors.php">change your preferences?</a></p>

</body></html>
```

To see this change, simply update the action destination in the *colors.php* file. By default, PHP session ID cookies expire when the browser closes. That is, sessions don't persist after the browser ceases to exist. To change this, you'll need to set the `session.cookie_lifetime` option in *php.ini* to the lifetime of the cookie in seconds.

#### Alternatives to cookies

By default, the session ID is passed from page to page in the `PHPSESSID` cookie. However, PHP's session system supports two alternatives: form fields and URLs. Passing the session ID via hidden form fields is extremely awkward, as it forces you to make every link between pages to be a form's submit button. We will not discuss this method further here.

The URL system for passing around the session ID, however, is somewhat more elegant. PHP can rewrite your HTML files, adding the session ID to every relative link. For this to work, though, PHP must be configured with the `-enable-trans-id` option when compiled. There is a performance penalty for this, as PHP must parse and rewrite every page. Busy sites may wish to stick with cookies, as they do not incur the slowdown caused by page rewriting. In addition, this exposes your session IDs, potentially allowing for man-in-the-middle attacks.

#### Custom storage

By default, PHP stores session information in files in your server's temporary directory. Each session's variables are stored in a separate file. Every variable is serialized into the file in a proprietary format. You can change all of these values in the *php.ini* file.

You can change the location of the session files by setting the `session.save_path` value in *php.ini*. If you are on a shared server with your own installation of PHP, set

the directory to somewhere in your own directory tree, so other users on the same machine cannot access your session files.

PHP can store session information in one of two formats in the current session store —either PHP's built-in format or Web Distributed Data eXchange (WDDX). You can change the format by setting the `session.serialize_handler` value in your *php.ini* file to either `php` for the default behavior, or `wddx` for WDDX format.

## Combining Cookies and Sessions

Using a combination of cookies and your own session handler, you can preserve state across visits. Any state that should be forgotten when a user leaves the site, such as which page the user is on, can be left up to PHP's built-in sessions. Any state that should persist between user visits, such as a unique user ID, can be stored in a cookie. With the user ID, you can retrieve the user's more permanent state (display preferences, mailing address, etc.) from a permanent store, such as a database.

Example 8-15 allows the user to select text and background colors and stores those values in a cookie. Any visits to the page within the next week send the color values in the cookie.

*Example 8-15. Saving state across visits (save_state.php)*

```
<?php
if($_POST['bgcolor']) {
 setcookie('bgcolor', $_POST['bgcolor'], time() + (60 * 60 * 24 * 7));
}

if (isset($_COOKIE['bgcolor'])) {
 $backgroundName = $_COOKIE['bgcolor'];
}
else if (isset($_POST['bgcolor'])) {
 $backgroundName = $_POST['bgcolor'];
}
else {
 $backgroundName = "gray";
} ?>
<html>
<head><title>Save It</title></head>
<body bgcolor="<?php echo $backgroundName; ?>">

<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <p>Background color:
 <select name="bgcolor">
 <option value="gray">Gray</option>
 <option value="white">White</option>
 <option value="black">Black</option>
 <option value="blue">Blue</option>
 <option value="green">Green</option>
```

```
  <option value="red">Red</option>
  </select></p>

  <input type="submit" />
</form>

</body>
</html>
```

## SSL

The Secure Sockets Layer (SSL) provides a secure channel over which regular HTTP requests and responses can flow. PHP doesn't specifically concern itself with SSL, so you cannot control the encryption in any way from PHP. An *https://* URL indicates a secure connection for that document, unlike an *http://* URL.

The `HTTPS` entry in the `$_SERVER` array is set to `'on'` if the PHP page was generated in response to a request over an SSL connection. To prevent a page from being generated over a nonencrypted connection, simply use:

```
if ($_SERVER['HTTPS'] !== 'on') {
 die("Must be a secure connection.");
}
```

A common mistake is to send a form over a secure connection (e.g., *https://www.example.com/form.html*), but have the `action` of the `form` submit to an *http://* URL. Any form parameters then entered by the user are sent over an insecure connection, and a trivial packet sniffer can reveal them.

## What's Next

There are many tips, tricks, and gotchas in modern web development, and we hope that the ones this chapter has pointed out will be helpful as you build great sites. Coming in the next chapter is a discussion on saving data to data stores within PHP. We will cover most of the more commonly used approaches, like databases, SQL and NoSQL style, SQLite, and direct file information storage.

# CHAPTER 9
# Databases

PHP has support for over 20 databases, including the most popular commercial and open source varieties. Relational database systems such as MariaDB, MySQL, PostgreSQL, and Oracle are the backbone of most modern dynamic websites. In these are stored shopping-cart information, purchase histories, product reviews, user information, credit card numbers, and sometimes even web pages themselves.

This chapter covers how to access databases from PHP. We focus on the built-in *PHP Data Objects* (PDO) library, which lets you use the same functions to access any database, rather than on the myriad database-specific extensions. In this chapter, you'll learn how to fetch data from the database, store data in the database, and handle errors. We finish with a sample application that shows how to put various database techniques into action.

This book cannot go into all the details of creating web database applications with PHP. For a more in-depth look at the PHP/MySQL combination, see *Web Database Applications with PHP and MySQL*, Second Edition (O'Reilly), by Hugh Williams and David Lane.

## Using PHP to Access a Database

There are two ways to access databases from PHP. One is to use a database-specific extension; the other is to use the database-independent PDO library. There are advantages and disadvantages to each approach.

If you use a database-specific extension, your code is intimately tied to the database you're using. For example, the MySQL extension's function names, parameters, error handling, and so on are completely different from those of the other database extensions. If you want to move your database from MySQL to PostgreSQL, it will involve significant changes to your code. PDO, on the other hand, hides the database-specific

functions from you with an abstraction layer, so moving between database systems can be as simple as changing one line of your program or your *php.ini* file.

The portability of an abstraction layer like the PDO library comes at a price, however, as code that uses it is also typically a little slower than code that uses a native database-specific extension.

Keep in mind that an abstraction layer does absolutely nothing when it comes to making sure your actual SQL queries are portable. If your application uses any sort of nongeneric SQL, you'll have to do significant work to convert your queries from one database to another. We will be looking briefly at both approaches to database interfaces in this chapter and then look at alternative methods to managing dynamic content for the web.

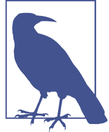
## Relational Databases and SQL

A Relational Database Management System (RDBMS) is a server that manages data for you. The data is structured into tables, where each table has a number of columns, each of which has a name and a type. For example, to keep track of science fiction books, we might have a "books" table that records the title (a string), the year of release (a number), and the author.

Tables are grouped together into databases, so a science fiction book database might have tables for time periods, authors, and villains. An RDBMS usually has its own user system, which controls access rights for databases (e.g., "user Fred can update database authors").

PHP communicates with relational databases such as MariaDB and Oracle using the Structured Query Language (SQL). You can use SQL to create, modify, and query relational databases.

The syntax for SQL is divided into two parts. The first, Data Manipulation Language (DML), is used to retrieve and modify data in an existing database. DML is remarkably compact, consisting of only four actions or verbs: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. The set of SQL commands used to create and modify the database structures that hold the data is known as Data Definition Language, or DDL. The syntax for DDL is not as standardized as that for DML, but as PHP just sends any SQL commands you give it to the database, you can use any SQL commands your database supports.



The SQL command file for creating this sample library database is available in a file called *library.sql*.

Assuming you have a table called books, this SQL statement would insert a new row:

```
INSERT INTO books VALUES (null, 4, 'I, Robot', '0-553-29438-5', 1950, 1);
```

This SQL statement inserts a new row but specifies the columns for which there are values:

```
INSERT INTO books (authorid, title, ISBN, pub_year, available)
 VALUES (4, 'I, Robot', '0-553-29438-5', 1950, 1);
```

To delete all books that were published in 1979 (if any), we could use this SQL statement:

```
DELETE FROM books WHERE pub_year = 1979;
```

To change the year for *Roots* to 1983, use this SQL statement:

```
UPDATE books SET pub_year=1983 WHERE title='Roots';
```

To fetch only the books published in the 1980s, use:

```
SELECT * FROM books WHERE pub_year > 1979 AND pub_year < 1990;
```

You can also specify the fields you want returned. For example:

```
SELECT title, pub_year FROM books WHERE pub_year > 1979 AND pub_year < 1990;
```

You can issue queries that bring together information from multiple tables. For example, this query joins together the book and author tables to let us see who wrote each book:

```
SELECT authors.name, books.title FROM books, authors
 WHERE authors.authorid = books.authorid;
```

You can even short-form (or alias) the table names like this:

```
SELECT a.name, b.title FROM books b, authors a WHERE a.authorid = b.authorid;
```

For more on SQL, see *SQL in a Nutshell*, Third Edition (O'Reilly), by Kevin Kline.

## PHP Data Objects

The PHP website has this to say about PDO:

> The PHP Data Objects (PDO) extension defines 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. Note that you cannot perform any database functions using the PDO extension by itself; you must use a database-specific PDO driver to access a database server.

Among its other unique features, PDO:

- Is a native C extension
- Takes advantage of the latest PHP 7 internals

- Uses buffered reading of data from the result set
- Provides common database features as a base
- Is still able to access database-specific functions
- Can use transaction-based techniques
- Can interact with LOBS (Large Objects) in the database
- Can use prepared and executable SQL statements with bound parameters
- Can implement scrollable cursors
- Has access to `SQLSTATE` error codes and has very flexible error handling

Since there are a number of features here, we will touch on only a few of them to illustrate just how beneficial PDO can be.

First, a little about PDO. It has drivers for almost all database engines in existence, and those drivers that PDO does not supply should be accessible through PDO's generic ODBC connection. PDO is modular in that it has to have at least two extensions enabled to be active: the PDO extension itself and the PDO extension specific to the database to which you will be interfacing. See the online documentation) to set up the connections for the database of your choice. As an example, for establishing PDO on a Windows server for MySQL interaction, simply enter the following two lines into your *php.ini* file and restart your server:

```
extension=php_pdo.dll
extension=php_pdo_mysql.dll
```

The PDO library is also an object-oriented extension (as you will see in the code examples that follow).

### Making a connection

The first requirement for PDO is to make a connection to the database in question and hold that connection in a connection handle variable, as in the following code:

```
$db = new PDO($dsn, $username, $password);
```

The `$dsn` stands for *data source name*, and the other two parameters are self-explanatory. Specifically, for a MySQL connection, you would write the following code:

```
$db = new PDO("mysql:host=localhost;dbname=library", "petermac", "abc123");
```

Of course, you could (should) maintain variable-based username and password parameters for code reuse and flexibility reasons.

### Interacting with the database

Once you have connected to your database engine and the database that you want to interact with, you can use that connection to send SQL commands to the server. A simple UPDATE statement would look like this:

```
$db->query("UPDATE books SET authorid=4 WHERE pub_year=1982");
```

This code simply updates the books table and releases the query. This allows you to send simple SQL commands (e.g., UPDATE, DELETE, INSERT) directly to the database.

### Using PDO and prepared statements

More typically, you'll use *prepared statements*, issuing PDO calls in stages or steps. Consider the following code:

```
$statement = $db->prepare("SELECT * FROM books");
$statement->execute();

// handle row results, one at a time
while($row = $statement->fetch()) {
 print_r($row);
 // ... or probably do something more meaningful with each returned row
}

$statement = null;
```

In this code, we "prepare" the SQL code and then "execute" it. Next, we cycle through the result with the while code and, finally, we release the result object by assigning null to it. This may not look all that powerful in this simple example, but there are other features that can be used with prepared statements. Now, consider this code:

```
$statement = $db->prepare("INSERT INTO books (authorid, title, ISBN, pub_year)"
 . "VALUES (:authorid, :title, :ISBN, :pub_year)");

$statement->execute(array(
 'authorid' => 4,
 'title' => "Foundation",
 'ISBN' => "0-553-80371-9",
 'pub_year' => 1951),
);
```

Here, we prepare the SQL statement with four named placeholders: *authorid*, *title*, *ISBN*, and *pub_year*. In this case, these happen to be the same names as the columns in the database, but this is done only for clarity—the placeholder names can be anything that is meaningful to you. In the execute call, we replace these placeholders with the actual data that we want to use in this particular query. One of the advantages of prepared statements is that you can execute the same SQL command and pass in different values through the array each time. You can also do this type of statement preparation with positional placeholders (not actually naming them), signified by a ?,

which is the positional item to be replaced. Look at the following variation of the previous code:

```
$statement = $db->prepare("INSERT INTO books (authorid, title, ISBN, pub_year)"
 . "VALUES (?, ?, ?, ?)");

$statement->execute(array(4, "Foundation", "0-553-80371-9", 1951));
```

This accomplishes the same thing but with less code, as the value area of the SQL statement does not name the elements to be replaced, and therefore the array in the `execute` statement needs to send in only the raw data and no names. You just have to be sure about the position of the data that you are sending into the prepared statement.

### Handling transactions

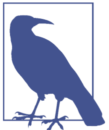
Some RDBMSs support *transactions*, in which a series of database changes can be committed (all applied at once) or rolled back (discarded, with none of the changes applied to the database). For example, when a bank handles a money transfer, the withdrawal from one account and deposit into another must happen together—neither should happen without the other, and there should be no time lag between the two actions. PDO handles transactions elegantly with `try...catch` structures like this one in Example 9-1.

*Example 9-1. The try...catch code structure*

```
try {
 // connection successful
 $db = new PDO("mysql:host=localhost;dbname=banking_sys", "petermac", "abc123");
} catch (Exception $error) {
 die("Connection failed: " . $error->getMessage());
}

try {
 $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
 $db->beginTransaction();

 $db->exec("insert into accounts (account_id, amount) values (23, '5000')" );
 $db->exec("insert into accounts (account_id, amount) values (27, '-5000')" );

 $db->commit();
} catch (Exception $error) {
 $db->rollback();
 echo "Transaction not completed: " . $error->getMessage();
}
```

If the entirety of the transaction can't be completed, none of it will be, and an exception will be thrown.

If you call `commit()` or `rollback()` on a database that doesn't support transactions, the methods return `DB_ERROR`.



Be sure to check your underlying database product to ensure that it supports transactions.

### Debugging statements

The PDO interface provides a method for showing details about a PDO statement, which can be useful for debugging if something goes wrong.

```
$statement = $db->prepare("SELECT title FROM books WHERE authorid = ?)";

$statement->bindParam(1, "12345678", PDO::PARAM_STR);
$statement->execute();

$statement->debugDumpParams();
```

Calling the `debugDumpParams()` method on the statement object prints a variety of information about the call:

```
SQL: [35] SELECT title
 FROM books
 WHERE authorID = ?
Sent SQL: [44] SELECT title
 FROM books
 WHERE authorid = "12345678"
Params: 1
Key: Position #0:
paramno=0
name[0] ""
is_param=1
param_type=2
```

The `Sent SQL` section is displayed only after the statement is executed; prior to that, only the `SQL` and `Params` sections are available.

## MySQLi Object Interface

The most popular database platform used with PHP is the MySQL database. If you look at the MySQL website, you'll discover that there are a few different versions of MySQL you can use. We will look at the freely distributable version known as the *community server*. PHP has a number of different interfaces to this database tool as well, so we will look at the object-oriented interface known as MySQLi, aka the *MySQL Improved* extension.

Recently, MariaDB has started overtaking MySQL as the database of choice for PHP developers. By design, MariaDB is client language–, connection tool–, and binary file–compatible with MySQL; this means that you can install MariaDB, uninstall MySQL, and point your PHP configuration to MariaDB instead, and likely need no other changes.

If you are not overly familiar with OOP interfaces and concepts, be sure to review Chapter 6 before you get too far into this section.

Since this object-oriented interface is built into PHP with a standard installation configuration (you simply activate the MySQLi extension in your PHP environment), all you have to do to start using it is instantiate its class, as in the following code:

```
$db = new mysqli(host, user, password, databaseName);
```

In this example, we have a database named `library`, and we will use the fictitious username of `petermac` and the password of `1q2w3e9i8u7y`. The actual code that would be used is:

```
$db = new mysqli("localhost", "petermac", "1q2w3e9i8u7y", "library");
```

This gives us access to the database engine itself within the PHP code; we will specifically access tables and other data later. Once this class is instantiated into the variable `$db`, we can use methods on that object to do our database work.

A brief example of generating some code to insert a new book into the `library` database would look something like this:

```
$db = new mysqli("localhost", "petermac", "1q2w3e9i8u7y", "library");

$sql = "INSERT INTO books (authorid, title, ISBN, pub_year, available)
 VALUES (4, 'I, Robot', '0-553-29438-5', 1950, 1)";

if ($db->query($sql)) {
 echo "Book data saved successfully.";
} else {
 echo "INSERT attempt failed, please try again later, or call tech support" ;
}

$db->close();
```

First, we instantiate the MySQLi class into the variable `$db`. Next, we build our SQL command string and save it to a variable called `$sql`. Then we call the query method of the class and at the same time test its return value to determine if it was successful (`TRUE`), and then comment to the screen accordingly. You may not want to `echo` out to the browser at this stage, as again this is only an example. Last, we call the `close()` method on the class to tidy up and destroy the class from memory.

## Retrieving Data for Display

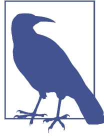
In another area of your website, you may want to draw out a listing of your books and show who their authors are. We can accomplish this by employing the same MySQLi class and working with the result set that is generated from a `SELECT` SQL command. There are many ways to display the information in the browser, and we'll look at one example of how this can be done. Notice that the result returned is a different object than the `$db` that we first instantiate. PHP instantiates the result object for you and fills it with any returned data.

```
$db = new mysqli("localhost", "petermac", "1q2w3e9i8u7y", "library");
$sql = "SELECT a.name, b.title FROM books b, authors a WHERE
a.authorid=b.authorid";
$result = $db->query($sql);

while ($row = $result->fetch_assoc()) {
 echo "{$row['name']} is the author of: {$row['title']}<br />";
}

$result->close();
$db->close();
```

Here, we are using the `query()` method call and storing the returned information into the variable called `$result`. Then we are using a method of the result object called `fetch_assoc()` to provide one row of data at a time, and we are storing that single row into the variable called `$row`. This continues as long as there are rows to process. Within that `while` loop, we are dumping content out to the browser window. Finally, we are closing both the result and the database objects.

The output would look like this:

```
J.R.R. Tolkien is the author of: The Two Towers
J.R.R. Tolkien is the author of: The Return of The King
J.R.R. Tolkien is the author of: The Hobbit
Alex Haley is the author of: Roots
Tom Clancy is the author of: Rainbow Six
Tom Clancy is the author of: Teeth of the Tiger
Tom Clancy is the author of: Executive Orders...
```



One of the most useful methods in MySQLi is `multi_query()`, which allows you to run multiple SQL commands in the same statement. If you want to do an `INSERT` and then an `UPDATE` statement based on similar data, you can do it all in one method call—one step.

We have, of course, just scratched the surface of what the MySQLi class has to offer. If you review its documentation), you'll see the extensive list of methods that are part of this class, as well as each result class documented within the appropriate subject area.

## SQLite

SQLite is a compact, highly performant (for small data sets), and—as its name suggests—lightweight database. SQLite is ready to go right out of the box when you install PHP, so if it sounds like a good fit for your database needs, be sure to read up on it.

All the database storage in SQLite is file-based, and therefore accomplished without the use of a separate database engine. This can be very advantageous if you are trying to build an application with a small database footprint and no product dependencies other than PHP. All you have to do to start using SQLite is to reference it in your code.

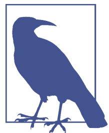
There is an OOP interface to SQLite, so you can instantiate an object with the following statement:

```
$db = new SQLiteDatabase("library.sqlite");
```

The neat thing about this statement is that if the file is not found at the specified location, SQLite creates it for you. Continuing with our `library` database example, the command to create the authors table and insert a sample row within SQLite would look something like Example 9-2.

*Example 9-2. SQLite library authors table*

```
$sql = "CREATE TABLE 'authors' ('authorid' INTEGER PRIMARY KEY, 'name' TEXT)";

if (!$database->queryExec($sql, $error)) {
 echo "Create Failure - {$error}<br />";
} else {
 echo "Table Authors was created <br />";
}

$sql = <<<SQL
INSERT INTO 'authors' ('name') VALUES ('J.R.R. Tolkien');
INSERT INTO 'authors' ('name') VALUES ('Alex Haley');
INSERT INTO 'authors' ('name') VALUES ('Tom Clancy');
INSERT INTO 'authors' ('name') VALUES ('Isaac Asimov');
SQL;

if (!$database->queryExec($sql, $error)) {
 echo "Insert Failure - {$error}<br />";
} else {
 echo "INSERT to Authors - OK<br />";
}
Table Authors was createdINSERT to Authors - OK
```



In SQLite, unlike MySQL, there is no `AUTO_INCREMENT` option. SQLite instead makes any column that is defined with `INTEGER` and `PRIMARY KEY` an automatically incrementing column. You can override this default behavior by providing a value to the column when an `INSERT` statement is executed.

Notice that the data types are quite different from what we have seen in MySQL. Remember that SQLite is a trimmed-down database tool and therefore it is "lite" on its data types; see Table 9-1 for a listing of the data types that it uses.

*Table 9-1. Data types available in SQLite*

| Data type | Explanation |
|---|---|
| Text | Stores data as `NULL`, `TEXT`, or `BLOB` content. If a number is supplied to a text field, it is converted to text before it is stored. |
| Numeric | Can store either integer or real data. If text data is supplied, SQLite attempts to convert the information to numerical format. |
| Integer | Behaves the same as the numeric data type. However, if data of the real type is supplied, it is stored as an integer. This may affect data storage accuracy. |
| Real | Behaves the same as the numeric data type, except that it forces integer values into floating-point representation. |
| None | This is a catchall data type; it does not prefer one base type to another. Data is stored exactly as supplied. |

Run the following code in Example 9-3 to create the books table and insert some data into the database file.

*Example 9-3. SQLite library books table*

```
$db = new SQLiteDatabase("library.sqlite");

$sql = "CREATE TABLE 'books' ('bookid' INTEGER PRIMARY KEY,
 'authorid' INTEGER,
 'title' TEXT,
 'ISBN' TEXT,
 'pub_year' INTEGER,
 'available' INTEGER,
)";

if ($db->queryExec($sql, $error) == FALSE) {
 echo "Create Failure - {$error}<br />";
} else {
 echo "Table Books was created<br />";
}

$sql = <<<SQL
INSERT INTO books ('authorid', 'title', 'ISBN', 'pub_year', 'available')
```

```
VALUES (1, 'The Two Towers', '0-261-10236-2', 1954, 1);

INSERT INTO books ('authorid', 'title', 'ISBN', 'pub_year', 'available')
VALUES (1, 'The Return of The King', '0-261-10237-0', 1955, 1);

INSERT INTO books ('authorid', 'title', 'ISBN', 'pub_year', 'available')
VALUES (2, 'Roots', '0-440-17464-3', 1974, 1);

INSERT INTO books ('authorid', 'title', 'ISBN', 'pub_year', 'available')
VALUES (4, 'I, Robot', '0-553-29438-5', 1950, 1);

INSERT INTO books ('authorid', 'title', 'ISBN', 'pub_year', 'available')
VALUES (4, 'Foundation', '0-553-80371-9', 1951, 1);
SQL;

if (!$db->queryExec($sql, $error)) {
 echo "Insert Failure - {$error}<br />";
} else {
 echo "INSERT to Books - OK<br />";
}
```

Notice that we can execute multiple SQL commands at the same time. We could also do this with MySQLi, but you'd have to remember to use the `multi_query()` method; with SQLite, it's available with the `queryExec()` method. After loading the database with some data, run the code in Example 9-4.

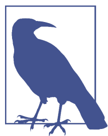
*Example 9-4. SQLite select books*

```
$db = new SQLiteDatabase("c:/copy/library.sqlite");

$sql = "SELECT a.name, b.title FROM books b, authors a WHERE a.authorid=b.authorid";
$result = $db->query($sql);

while ($row = $result->fetch()) {
 echo "{$row['a.name']} is the author of: {$row['b.title']}<br/>";
}
```

The preceding code produces the following output:

```
J.R.R. Tolkien is the author of: The Two Towers
J.R.R. Tolkien is the author of: The Return of The King
Alex Haley is the author of: Roots
Isaac Asimov is the author of: I, Robot
Isaac Asimov is the author of: Foundation
```

SQLite can do almost as much as the "bigger" database engines—the "lite" refers not to its functionality but to its demand for system resources. You should always consider SQLite when you require a database that's more portable and less demanding of resources.



If you are just getting started with the dynamic aspect of web development, you can use PDO to interface with SQLite. In this way, you can start with a lightweight database and grow into a more robust database server like MySQL when you are ready.

# Direct File-Level Manipulation

PHP has many little hidden features within its vast toolset. One of these features (which is often overlooked) is its uncanny ability to handle complex files. Sure, everyone knows that PHP can open a file, but what can it really do with that file? Consider the following example highlighting the true range of its possibilities. One of this book's authors was contacted by a prospective client who had "no money" but wanted a dynamic web survey developed. Of course, the author initially offered the client the wonders of PHP and database interaction with MySQLi. Upon hearing the monthly fees from a local ISP, however, the client asked if there was any other (cheaper) way to accomplish the work. It turns out that if you don't want to use SQLite, an alternative is to use files to manage and manipulate small amounts of text for later retrieval. The functions we'll discuss here are nothing out of the ordinary when taken individually —in fact, they're really part of the basic PHP toolset everyone is probably familiar with, as you can see in Table 9-2.

*Table 9-2. Commonly used PHP file management functions*

| Function name | Description of use |
|---|---|
| `mkdir()` | Used to make a directory on the server. |
| `file_exists()` | Used to determine if a file or directory exists at the supplied location. |
| `fopen()` | Used to open an existing file for reading or writing (see detailed options for correct usage). |
| `fread()` | Used to read in the contents of a file to a variable for PHP use. |
| `flock()` | Used to gain an exclusive lock on a file for writing. |
| `fwrite()` | Used to write the contents of a variable to a file. |
| `filesize()` | When reading in a file, this is used to determine how many bytes to read in at a time. |
| `fclose()` | Used to close the file once its usefulness has passed. |

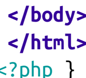
The interesting part is in tying all the functions together to accomplish your objective. For example, let's create a small web form survey that covers two pages of questions. Users can enter some opinions and return at a later date to finish the survey, picking up right where they left off. We'll scope out the logic of our little application and, hopefully, you will see that its basic premise can be expanded to a full production-type employment.

The first thing that we want to do is allow users to return to this survey at any time to provide additional input. To do this, we need to have a unique identifier to differentiate one user from another. Generally, a person's email address is unique (other people

might know it and use it, but that is a question of website security and/or controlling identity theft). For the sake of simplicity, we'll assume honesty here in the use of email addresses and not bother with a password system. So, once we have the user's email address, we need to store that information in a location that is distinct from that of other site visitors. For this purpose, we will create a directory folder for each visitor on the server (this, of course, assumes that you have access and proper rights to a location on the server that permits the reading and writing of files). Since we have the relatively unique identifier in the visitor's email address, we will simply name the new directory location with that identifier. Once we've created a directory (testing to see if the user has returned from a previous session), we will read in any file contents that are already there and display them in a `<textarea>` form control so that the visitor can see what (if anything) he or she has written previously. We then save the visitor's comments upon the submission of the form and move on to the next survey question. Example 9-5 shows the code for the first page (the `<?php` tags are included here because there are places where they are turned on and off throughout the listing).

*Example 9-5. File-level access*

```
session_start();

if (!empty($_POST['posted']) && !empty($_POST['email'])) {
 $folder = "surveys/" . strtolower($_POST['email']);

 // send path information to the session
 $_SESSION['folder'] = $folder;

 if (!file_exists($folder)) {
 // make the directory and then add the empty files
 mkdir($folder, 0777, true);
 }

 header("Location: 08_6.php");
} else { ?>
<html>
 <head>
 <title>Files & folders - On-line Survey</title>
 </head>

 <body bgcolor="white" text="black">
 <h2>Survey Form</h2>

 <p>Please enter your e-mail address to start recording your comments</p>

 <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <input type="hidden" name="posted" value="1">
 <p>Email address: <input type="text" name="email" size="45" /><br />
 <input type="submit" name="submit" value="Submit"></p>
 </form>
```



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

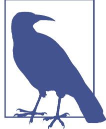
Figure 9-1 shows the web page that asks the visitor to submit an email address.

Figure 9-1. Survey login screen

As you can see, the first thing that we do is open a new session to pass the visitor's information on to subsequent pages. Then we test to confirm that the form further down in the code has indeed been submitted and that there is something entered in the email address field. If this test fails, the form is simply redisplayed. Of course, the production version of this functionality would send out an error message telling the user to enter valid text.

Once this test has passed (assuming the form has been submitted correctly) we create a `$folder` variable that contains the directory structure where we want to save the survey information and append the user's email address to it; we also save the contents of this newly created variable (`$folder`) into the session for later use. Here we simply take the email address and use it (again, if this were a secure site, we would protect the data with proper security measures).

Next, we want to see if the directory already exists. If it does not, we create it with the `mkdir()` function. This function takes the argument of the path and the name of the directory we want to create and attempts to create it.



In a Linux environment, there are other options on the `mkdir()` function that control access levels and permissions on the newly created directory, so be sure to look into those options if this applies to your environment.

After we verify that the directory exists, we simply direct the browser to the first page of the survey.

Now that we are on the first page of the survey (see Figure 9-2), the form is ready for use.

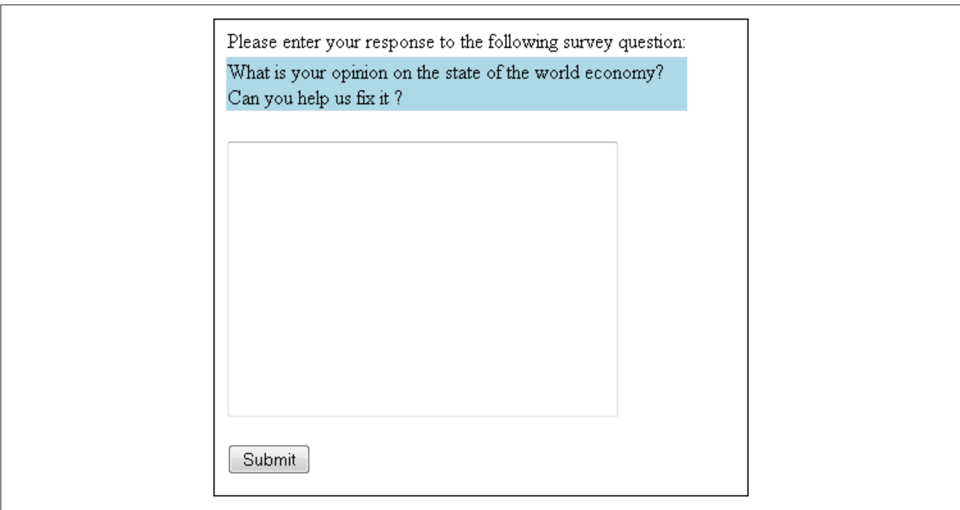



*Figure 9-2. The first page of the survey*

This, however, is a dynamically generated form, as you can see in Example 9-6.

*Example 9-6. File-level access, continued*

```
<?php
session_start();
$folder = $_SESSION['folder'];
$filename = $folder . "/question1.txt";

// open file for reading then clean it out
$file_handle = fopen($filename, "a+");

// pick up any text in the file that may already be there
$comments = file_get_contents($filename) ;
fclose($file_handle); // close this handle

if (!empty($_POST['posted'])) {
 // create file if first time and then
 //save text that is in $_POST['question1']
 $question1 = $_POST['question1'];
 $file_handle = fopen($filename, "w+");

 // open file for total overwrite
 if (flock($file_handle, LOCK_EX)) {
 // do an exclusive lock
 if (fwrite($file_handle, $question1) == FALSE) {
 echo "Cannot write to file ($filename)";
 }

 // release the lock
```

```
 flock($file_handle, LOCK_UN);
 }

 // close the file handle and redirect to next page ?
 fclose($file_handle);
 header( "Location: page2.php" );
} else { ?>
 <html>
 <head>
 <title>Files & folders - On-line Survey</title>
 </head>

 <body>
 <table border="0">
 <tr>
 <td>Please enter your response to the following survey question:</td>
 </tr>
 <tr bgcolor=lightblue>
 <td>
 What is your opinion on the state of the world economy?<br/>
 Can you help us fix it ?
 </td>
 </tr>
 <tr>
 <td>
 <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
 <input type="hidden" name="posted" value="1"><br/>
 <textarea name="question1" rows=12 cols=35><?= $comments ?></textarea>
 </td>
 </tr>

 <tr>
 <td><input type="submit" name="submit" value="Submit"></form></td>
 </tr>
 </table>
<?php } ?>
```

Let's highlight a few of the lines of code here, because this is where the file management and manipulation really takes place. After taking in the session information that we need and appending the filename to the `$filename` variable, we are ready to start working with the files. Keep in mind that the point of this process is to display any information that may already be saved in the file and allow users to enter information (or alter what they have already entered). So, near the top of the code you see this command:

```
$file_handle = fopen($filename, "a+");
```

Using the file opening function, `fopen()`, we ask PHP to provide us with a handle to that file and store it in the aptly named variable `$file_handle`. Notice that there is another parameter passed to the function here: the `a+` option. The PHP site provides

a full listing of these option letters and what they mean. The a+ option causes the file to open for reading and writing, with the file pointer placed at the end of any existing file content. If the file does not exist, PHP will attempt to create it. Looking at the next two lines of code, you'll see that the entire file is read (using the file_get_contents() function) into the $comments variable, and then it is closed:

```
$comments = file_get_contents($filename);
fclose($file_handle);
```

Next, we want to see if the form portion of this program file has been executed and, if so, we have to save any information that was entered into the text area. This time, we open the same file again, but we use the w+ option, which causes the interpreter to open the file for writing only—creating it if it doesn't exist, or emptying it if it does. The file pointer is then placed at the beginning of the file. Essentially, we want to empty out the current contents of the file and replace it with a totally new volume of text. For this purpose, we employ the fwrite() function:

```
// do an exclusive lock
if (flock($file_handle, LOCK_EX)) {
 if (fwrite($file_handle, $question1) == FALSE){
 echo "Cannot write to file ($filename)";
 }
 // release the lock
 flock($file_handle, LOCK_UN);
}
```

We have to be sure that this information is indeed saved into the designated file, so we wrap a few conditional statements around our file-writing operations to make sure everything will go smoothly. First, we attempt to gain an exclusive lock on the file in question (using the flock() function); this will ensure that no other process can access the file while we're operating on it. After the writing is complete, we release the lock on the file. This is merely a precaution, since the file management is unique to the entered email address on the first web page form and each survey has its own folder location, so usage collisions should never occur unless two people happen to be using the same email address.

As you can see, the file write function uses the $file_handle variable to add the contents of the $question1 variable to the file. Then we simply close the file when we are finished with it and move on to the next page of the survey, as shown in Figure 9-3.

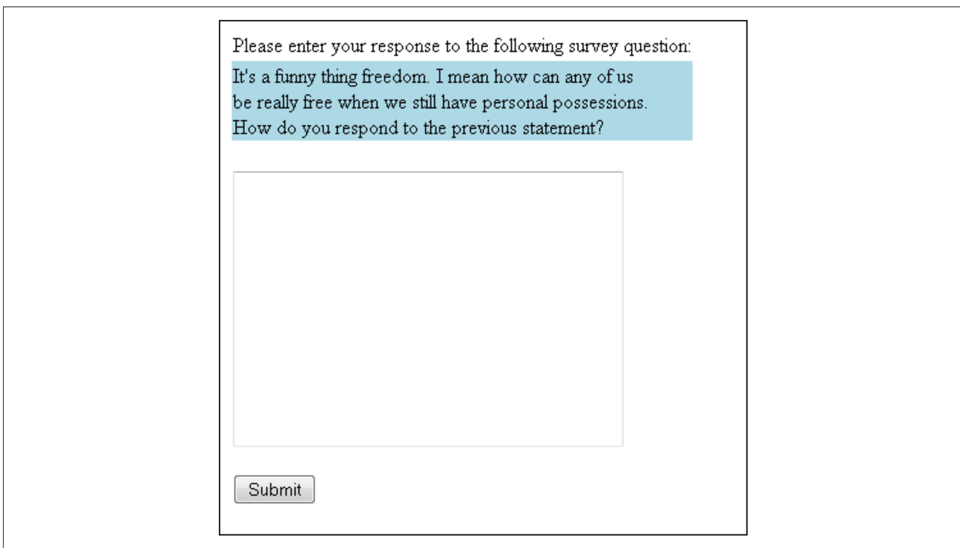



*Figure 9-3. Page 2 of the survey*

As you can see in Example 9-7, the code for processing this file (called *question2.txt*) is identical to the previous one except for its name.

*Example 9-7. File-level access, continued*

```
<?php
session_start();
$folder = $_SESSION['folder'];
$filename = $folder . "/question2.txt" ;

// open file for reading then clean it out
$file_handle = fopen($filename, "a+");

// pick up any text in the file that may already be there
$comments = fread($file_handle, filesize($filename));
fclose($file_handle); // close this handle

if ($_POST['posted']) {
 // create file if first time and then save
 //text that is in $_POST['question2']
 $question2 = $_POST['question2'];

 // open file for total overwrite
 $file_handle = fopen($filename, "w+");

 if(flock($file_handle, LOCK_EX)) { // do an exclusive lock
 if(fwrite($file_handle, $question2) == FALSE) {
 echo "Cannot write to file ($filename)";
```

```
 }

 flock($file_handle, LOCK_UN); // release the lock
 }

 // close the file handle and redirect to next page ?
 fclose($file_handle);

 header( "Location: last_page.php" );
} else { ?>
 <html>
 <head>
 <title>Files & folders - On-line Survey</title>
 </head>

 <body>
 <table border="0">
 <tr>
 <td>Please enter your comments to the following survey statement:</td>
 </tr>

 <tr bgcolor="lightblue">
 <td>It's a funny thing freedom. I mean how can any of us <br/>
 be really free when we still have personal possessions.
 How do you respond to the previous statement?</td>
 </tr>

 <tr>
 <td>
 <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method=POST>
 <input type="hidden" name="posted" value="1"><br/>
 <textarea name="question2" rows="12" cols="35"><?= $comments ?></textarea>
 </td>
 </tr>

 <tr>
 <td><input type="submit" name="submit" value="Submit"></form></td>
 </tr>
</table>
<?php } ?>
```

This kind of file processing can continue for as long as you like, and therefore your surveys can be as long as you like. To make it more interesting, you can ask multiple questions on the same page and simply give each question its own filename. The only unique item here to point out is that once this page is submitted and the text is stored, it is directed to a PHP file called *last_page.php*. This page is not included in the code samples, as it is merely a page thanking users for filling out the survey.

Of course, after a few pages, with as many as five questions per page, you may find yourself with a large volume of individual files needing management. Fortunately, PHP has other file-handling functions that you can use. The `file()` function, for

example, is an alternative to the `fread()` function that reads the entire contents of a file in an array, one element per line. If your information is formatted properly—with each line delimited by the end of line sequence, `\n`—you can store multiple pieces of information in a single file very easily. Naturally, this would also entail the use of the appropriate looping controls for handling the creation of the HTML form, as well as recording the entries into that form.

When it comes to file handling, there are still many more options that you can look at on the PHP website. If you go to "Filesystem" on page 388, you will find a list of over 70 functions—including, of course, the ones discussed here. You can check to see if a file is either readable or writable with the `is_readable()` or `is_writable()` functions, respectively. You can check on file permissions, free disk space, or total disk space, and you can delete files, copy files, and much more. When you get right down to it, if you have enough time and desire, you can even write an entire web application without ever needing or using a database system.

When the day comes, and it most likely will, that you have a client who does not want to pay big bucks for the use of a database engine, you will have an alternative approach to offer them.

## MongoDB

The last database type that we will look at is a NoSQL database. NoSQL databases are rising in popularity because they are also quite lightweight in terms of system resources, but more importantly, they work outside the typical SQL command structure. NoSQL databases are also becoming more popular with mobile devices like tablets and smartphones for the same two reasons.

One of the frontrunners in the NoSQL database world is known as MongoDB. We'll only be touching the surface of MongoDB here, just to give you a taste of what is possible with its use. For more detailed coverage of this topic, please refer to *MongoDB and PHP* (O'Reilly) by Steve Francia.

The first thing to get your head around with MongoDB is that it is not a traditional database. It has its own setup and terminology. Getting used to how to work with it will take some time for the traditional SQL database user. Table 9-3 attempts to draw some parallels with "standard" SQL terminology.

*Table 9-3. Typical MongoDB/SQL equivalents*

| Traditional SQL terms | MongoDB terms |
| --- | --- |
| Database | Database |
| Tables | Collections |
| Rows | Documents. No correlation, not like database "rows"; rather, think of arrays. |

There's not an exact equivalent of a database row within the MongoDB paradigm. One of the best ways to think of the data within a collection is like that of a multidimensional array, as you'll see shortly when we revamp our `library` database example.

If you just want to try MongoDB out on your own localhost (recommended for getting familiar with it), you can use an all-in-one tool like Zend Server CE to set up a local environment with the Mongo drivers all installed. You'll still have to download the server itself from the MongoDB website and follow the instructions for setting up the database server engine for your own local environment.

One very useful web-based tool for browsing MongoDB data and manipulating the collections and documents is Genghis. Simply download the project and drop it into its own folder in the localhost and call *genghis.php*. If the database engine is running, it will be picked up and displayed to you (see Figure 9-4).

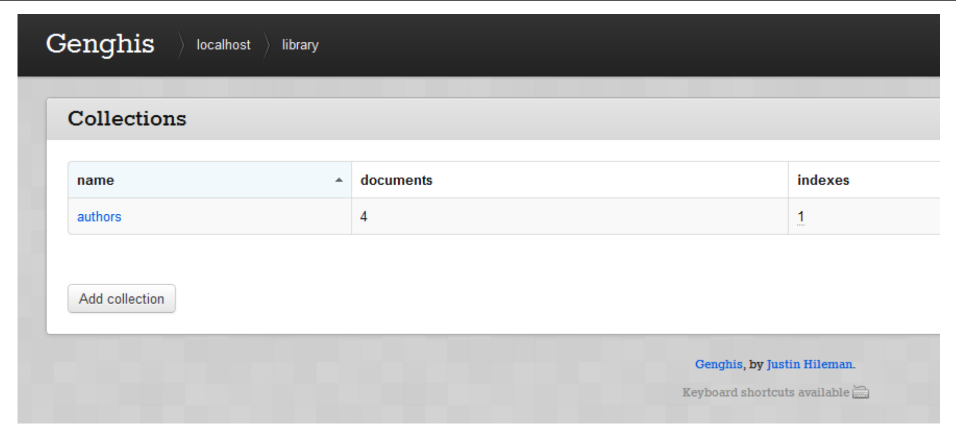



*Figure 9-4. Genghis MongoDB web interface sample*

Now let's get into some sample code. Take a look at Example 9-8 to see the beginnings of a Mongo database taking shape.

*Example 9-8. MongoDB library*

```
$mongo = new Mongo();
$db = $mongo->library;
$authors = $db->authors;

$author = array('authorid' => 1, 'name' => "J.R.R. Tolkien");
$authors->insert($author);

$author = array('authorid' => 2, 'name' => "Alex Haley");
$authors->insert($author);

$author = array('authorid' => 3, 'name' => "Tom Clancy");
$authors->save($author);

$author = array('authorid' => 4, 'name' => "Isaac Asimov");
$authors->save($author);
```

The first line creates a new connection to the MongoDB engine, and creates an object interface to it as well. The next line connects to the `library` "collection"; if this collection does not exist, Mongo creates it for you (so there is no need to precreate a collection in Mongo). We then create an object interface with the `$db` connection to the `library` database and create a collection where we will store our author data. The next four groupings of code add documents to the `authors` collection in two different ways. The first two samples use the `insert()` method, and the last two use the `save()` method. The only difference between these two methods is that `save()` will update a value if it is already in the document and has an existing `_id` key (more on `_id` shortly).

Execute this code within a browser, and you should see the sample data shown in Figure 9-5. As you can see, an entity called `_id` is created with the inserted data. This is the automatic primary key that is assigned to all created collections. If we wanted to depend on that key—and there is no reason why we shouldn't (other than its obvious complexity)—we wouldn't have had to add in our own `authorid` information in the preceding code.

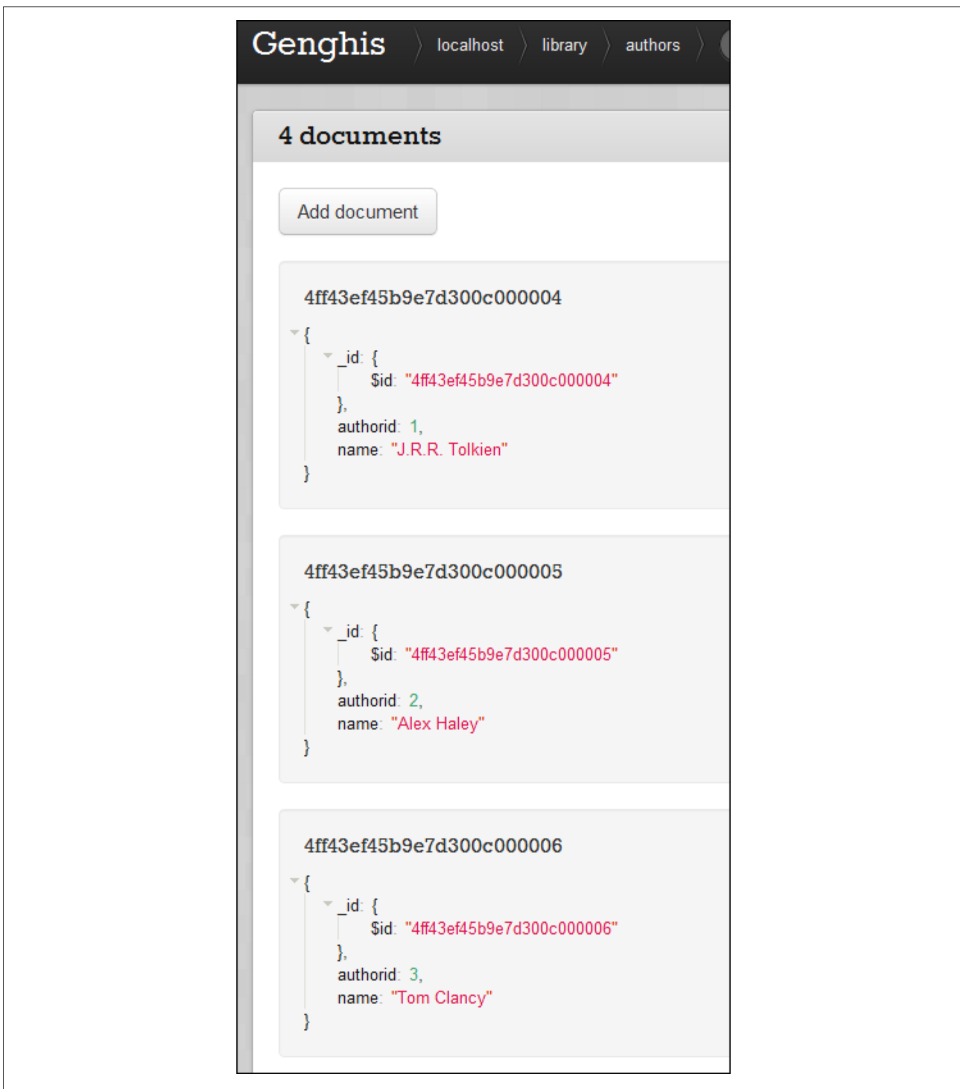



*Figure 9-5. Sample Mongo document data for authors*

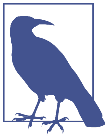
## Retrieving Data

Once the data is stored, we can now start looking at ways in which to access it. Example 9-9 shows one option.

*Example 9-9. MongoDB data selection example*

```
$mongo = new Mongo();
$db = $mongo->library;
$authors = $db->authors;

$data = $authors->findone(array('authorid' => 4));

echo "Generated Primary Key: {$data['_id']}<br />";
echo "Author name: {$data['name']}";
```

The first three lines of code are the same as before, since we still want to connect to the same database and make use of the same collection (`library`) and document (`authors`). After that, we use the `findone()` method, passing it an array containing a unique piece of data that can be used to find the information that we want—in this case, the `authorid` for Isaac Asimov, `4`. We store the returned information into an array called `$data`.



As a good oversimplification, you can think of information within a Mongo document as array-based.

Then we can use that array as we wish to display the returned data from the document. The following is the resulting output from the previous code. Notice the size of the primary key that Mongo has created.

```
Generated Primary Key: 4ff43ef45b9e7d300c000007
Author name: Isaac Asimov
```

## Inserting More Complex Data

Next we want to continue our `library` example database by adding some books to the document in relation to a particular author. Here is where the analogy of different tables within a database can collapse. Consider Example 9-10, which adds four books to the `authors` document, essentially as a multidimensional array.

*Example 9-10. MongoDB simple data update/insert*

```
$mongo = new Mongo();
$db = $mongo->library;
$authors = $db->authors;

$authors->update(
 array('name' => "Isaac Asimov"),
 array('$set' =>
 array('books' =>
 array(
 "0-425-17034-9" => "Foundation",
 "0-261-10236-2" => "I, Robot",
 "0-440-17464-3" => "Second Foundation",
 "0-425-13354-0" => "Pebble In The Sky",
 )
 )
 )
);
```

Here, after making the needed connections, we use the `update()` method and use the first element of the array (the first parameter of the `update()` method) as the unique lookup identifier, and a defined operator called `$set` as the second parameter to attach the book data to the provided key of the first parameter.

You should research and fully understand the special operators `$set` and `$push` (not covered here) before using them in a production environment. See the MongoDB documentation for more information and a full listing of these operators.

Example 9-11 provides another approach to accomplishing the same goal, except that we are preparing the array to be inserted and attached ahead of time and using the Mongo-created `_id` as the location key.

*Example 9-11. MongoDB data update/insert*

```
$mongo = new Mongo();
$db = $mongo->library;
$authors = $db->authors;

$data = $authors->findone(array('name' => "Isaac Asimov"));

$bookData = array(
 array(
 "ISBN" => "0-553-29337-0",
 "title" => "Foundation",
 "pub_year" => 1951,
 "available" => 1,
 ),
 array(
 "ISBN" => "0-553-29438-5",
 "title" => "I, Robot",
 "pub_year" => 1950,
 "available" => 1,
 ),
 array(
 "ISBN" => "0-517-546671",
 "title" => "Exploring the Earth and the Cosmos",
 "pub_year" => 1982,
 "available" => 1,
 ),
 array(
 "ISBN' => "0-553-29336-2",
 'title" => "Second Foundation",
 "pub_year" => 1953,
 "available" => 1,
 ),
);

$authors->update(
 array("_id" => $data["_id"]),
 array("$set" => array("books" => $bookData))
);
```

In both of our two previous code examples, we did not add any keys to the array of book data. We could do this, but it's just as easy to allow Mongo to manage that data as if it were a multidimensional array. Figure 9-6 shows how the data from Example 9-11 will look when it is displayed in Genghis.

```
4ff43ef45b9e7d300c000007
{
  _id: {
    $id: "4ff43ef45b9e7d300c000007"
  },
  authorid: 4,
  books: [
    {
      ISBN: "0-553-29337-0",
      title: "Foundation",
      pub_year: 1951,
      available: 1
    },
    {
      ISBN: "0-553-29438-5",
      title: "I, Robot",
      pub_year: 1950,
      available: 1
    },
    {
      ISBN: "0-517-546671",
      title: "Exploring the Earth and the Cosmos",
      pub_year: 1982,
      available: 1
    },
    {
      ISBN: "0-553-29336-2",
      title: "Second Foundation",
      pub_year: 1953,
      available: 1
    }
  ],
  name: "Isaac Asimov"
}
```

*Figure 9-6. Book data added to an author*

Example 9-12 shows a little more of what data is stored in our Mongo database. It adds just a few more lines of code to Example 9-9; here we are referencing the automatic natural keys generated in the previous code that inserted the book detail information.

*Example 9-12. MongoDB data find and display*

```
$mongo = new Mongo();
$db = $mongo->library;
$authors = $db->authors;

$data = $authors->findone(array("authorid" => 4));

echo "Generated Primary Key: {$data['_id']}<br />";
```

```
echo "Author name: {$data['name']}<br />";
echo "2nd Book info - ISBN: {$data['books'][1]['ISBN']}<br />";
echo "2nd Book info - Title: {$data['books'][1]['title']<br />";
```

The generated output of the preceding code looks like this (remember that arrays are zero-based):

```
Generated Primary Key: 4ff43ef45b9e7d300c000007
Author name: Isaac Asimov
2nd Book info - ISBN: 0-553-29438-5
2nd Book info - Title: I, Robot
```

For more information on how MongoDB can be used and manipulated within PHP, see the documentation on the PHP website.

# What's Next

In the next chapter, we'll explore various techniques for including graphics media within pages generated by PHP, as well as dynamically generating and manipulating graphics on your web server.

CHAPTER 10

# Graphics

The web is much more visual than textual; that is obvious. Images appear in the form of logos, buttons, photographs, charts, advertisements, and icons. Many of these images are static and never change, built with tools such as Photoshop. But many are dynamically created—from advertisements for Amazon's referral program that include your name to graphs of stock performance.

PHP supports graphics creation with the built-in GD extension library. In this chapter, we'll show you how to generate images dynamically within PHP.

## Embedding an Image in a Page

A common misconception is that there is a mixture of text and graphics flowing across a single HTTP request. After all, when you view a page, you see a single page containing such a mixture. It is important to understand that a standard web page containing text and graphics is created through a series of HTTP requests from the web browser; each request is answered by a response from the web server. Each response can contain one and only one type of data, and each image requires a separate HTTP request and web server response. Thus, if you see a page that contains some text and two images, you know that it has taken three HTTP requests and corresponding responses to construct this page.

Take this HTML page, for example:

```
<html>
 <head>
 <title>Example Page</title>
 </head>

 <body>
 This page contains two images.
```

```
  <img src="image1.png" alt="Image 1" />
  <img src="image2.png" alt="Image 2" />
  </body>
</html>
```

The series of requests sent by the web browser for this page looks something like this:

```
GET /page.html HTTP/1.0
GET /image1.png HTTP/1.0
GET /image2.png HTTP/1.0
```

The web server sends back a response to each of these requests. The `Content-Type` headers in these responses look like this:

```
Content-Type: text/html
Content-Type: image/png
Content-Type: image/png
```

To embed a PHP-generated image in an HTML page, pretend that the PHP script that generates the image is actually the image. Thus, if we have *image1.php* and *image2.php* scripts that create images, we can modify the previous HTML to look like this (the image names are PHP extensions now):

```
<html>
  <head>
  <title>Example Page</title>
  </head>

  <body>
  This page contains two images.
  <img src="image1.php" alt="Image 1" />
  <img src="image2.php" alt="Image 2" />
  </body>
</html>
```

Instead of referring to real images on your web server, the `<img>` tags now refer to the PHP scripts that generate and return image data.

Furthermore, you can pass variables to these scripts, so instead of having separate scripts to generate each image, you could write your `<img>` tags like this:

```
<img src="image.php?num=1" alt="Image 1" />
<img src="image.php?num=2" alt="Image 2" />
```

Then, inside the called PHP file *image.php*, you can access the request parameter `$_GET['num']` to generate the appropriate image.

# Basic Graphics Concepts

An *image* is a rectangle of pixels of various colors. Colors are identified by their position in the *palette*, an array of colors. Each entry in the palette has three separate color values—one each for red, green, and blue. Each value ranges from `0` (color not

present) to 255 (color at full intensity). This is known as its *RGB value*. There are also hexadecimal, or "hex" values—alphanumeric representations of colors that are commonly used in HTML. Some image tools, such as ColorPic, will convert RGB values to hex for you.

Image files are rarely a straightforward dump of the pixels and the palette. Instead, various *file formats* (GIF, JPEG, PNG, etc.) have been created that attempt to compress the data somewhat to make smaller files.

Different file formats handle image *transparency*, which controls whether and how the background shows through the image, in different ways. Some, such as PNG, support an *alpha channel*, an extra value for every pixel reflecting the transparency at that point. Others, such as GIF, simply designate one entry in the palette as indicating transparency. Still others, like JPEG, don't support transparency at all.

Rough and jagged edges, an effect known as *aliasing*, can make for unappealing images. *Antialiasing* involves moving or recoloring pixels at the edge of a shape to transition more gradually between the shape and its background. Some functions that draw on an image implement antialiasing.

With 256 possible values for each of red, green, and blue, there are 16,777,216 possible colors for each pixel. Some file formats limit the number of colors you can have in a palette (e.g., GIF supports no more than 256 colors); others let you have as many colors as you need. The latter are known as *true color* formats, because 24-bit color (8 bits each for red, green, and blue) gives more hues than the human eye can distinguish.

## Creating and Drawing Images

For now, let's start with the simplest possible GD example. Example 10-1 is a script that generates a black-filled square. The code works with any version of GD that supports the PNG image format.

*Example 10-1. A black square on a white background (black.php)*

```
<?php
$image = imagecreate(200, 200);

$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);
imagefilledrectangle($image, 50, 50, 150, 150, $black);

header("Content-Type: image/png");
imagepng($image);
```

Example 10-1 illustrates the basic steps in generating any image: creating the image, allocating colors, drawing the image, and then saving or sending the image. Figure 10-1 shows the output of Example 10-1.

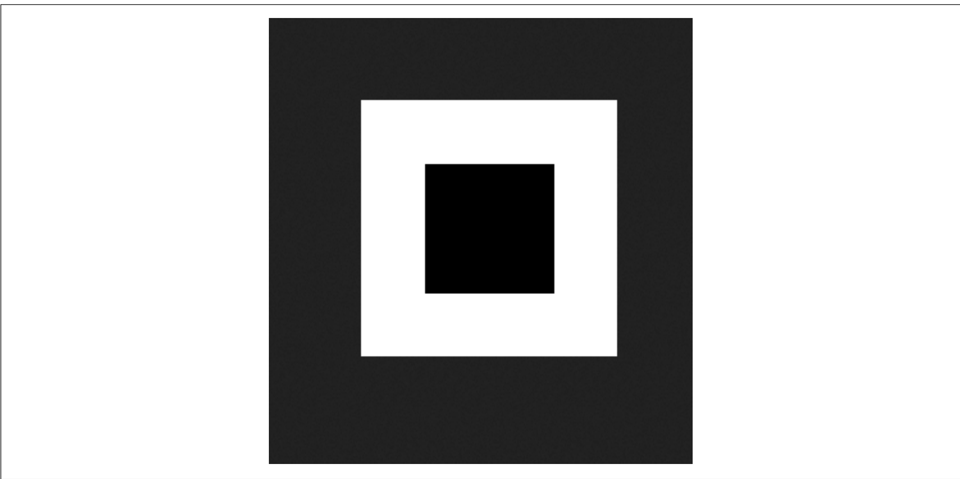


*Figure 10-1. A black square on a white background*

To see the result, simply point your browser at the *black.php* page. To embed this image in a web page, use:

```
<img src="black.php" />
```

## The Structure of a Graphics Program

Most dynamic image-generation programs follow the same basic steps outlined in Example 10-1.

You can create a 256-color image with the `imagecreate()` function, which returns an image handle:

```
$image = imagecreate(width, height);
```

All colors used in an image must be allocated with the `imagecolorallocate()` function. The first color allocated becomes the background color for the image:[1]

```
$color = imagecolorallocate(image, red, green, blue);
```

The arguments are the numeric RGB (red, green, blue) components of the color. In Example 10-1, we wrote the color values in hexadecimal to bring the function call closer to the HTML color representation `#FFFFFF` and `#000000`.

1 This is true only for images with a color palette. True color images created using `ImageCreateTrueColor()` do not obey this rule.

There are many drawing primitives in GD. Example 10-1 uses `imagefilledrectangle()`, in which you specify the dimensions of the rectangle by passing the coordinates of the top-left and bottom-right corners:

```
imagefilledrectangle(image, tlx, tly, brx, bry, color);
```

The next step is to send a `Content-Type` header to the browser with the appropriate content type for the kind of image being created. Once that is done, we call the appropriate output function. The `imagejpeg()`, `imagegif()`, `imagepng()`, and `imagewbmp()` functions create GIF, JPEG, PNG, and WBMP files from the image, respectively:

```
imagegif(image [, filename ]);
imagejpeg(image [, filename [, quality ]]);
imagepng(image [, filename ]);
imagewbmp(image [, filename ]);
```

If no *filename* is given, the image is output to the browser; otherwise, it creates (or overwrites) the image to the given file path. The *quality* argument for JPEGs is a value from `0` (worst-looking) to `100` (best-looking). The lower the quality, the smaller the JPEG file. The default setting is `75`.

In Example 10-1, we set the HTTP header immediately before calling the output-generating function `imagepng()`. If you set the `Content-Type` at the very start of the script, any errors that are generated are treated as image data and the browser displays a broken image icon. Table 10-1 lists the image formats and their `Content-Type` values.

*Table 10-1. Content-Type values for image formats*

| Format | Content-Type |
|---|---|
| GIF | image/gif |
| JPEG | image/jpeg |
| PNG | image/png |
| WBMP | image/vnd.wap.wbmp |

## Changing the Output Format

As you may have deduced, generating an image stream of a different type requires only two changes to the script: send a different `Content-Type` and use a different image-generating function. Example 10-2 shows Example 10-1 modified to generate a JPEG instead of a PNG image.

*Example 10-2. JPEG version of the black square*

```
<?php
$image = imagecreate(200, 200);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
```

```
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);

imagefilledrectangle($image, 50, 50, 150, 150, $black);

header("Content-Type: image/jpeg");
imagejpeg($image);
```

## Testing for Supported Image Formats

If you are writing code that must be portable across systems that may support different image formats, use the `imagetypes()` function to check which image types are supported. This function returns a bit field; you can use the bitwise AND operator (`&`) to check if a given bit is set. The constants `IMG_GIF`, `IMG_JPG`, `IMG_PNG`, and `IMG_WBMP` correspond to the bits for those image formats.

Example 10-3 generates PNG files if PNG is supported, JPEG files if PNG is not supported, and GIF files if neither PNG nor JPEG is supported.

*Example 10-3. Checking for image format support*

```
<?php
$image = imagecreate(200, 200);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);

imagefilledrectangle($image, 50, 50, 150, 150, $black);

if (imagetypes() & IMG_PNG) {
 header("Content-Type: image/png");
 imagepng($image);
}
else if (imagetypes() & IMG_JPG) {
 header("Content-Type: image/jpeg");
 imagejpeg($image);
}
else if (imagetypes() & IMG_GIF) {
 header("Content-Type: image/gif");
 imagegif($image);
}
```

## Reading an Existing File

If you want to start with an existing image and then modify it, use `imagecreatefromgif()`, `imagecreatefromjpeg()`, or `imagecreatefrompng()`:

```
$image = imagecreatefromgif(filename);
$image = imagecreatefromjpeg(filename);
$image = imagecreatefrompng(filename);
```

## Basic Drawing Functions

GD has functions for drawing basic points, lines, arcs, rectangles, and polygons. This section describes the base functions supported by GD 2.x.

The most basic function is `imagesetpixel()`, which sets the color of a specified pixel:

```
imagesetpixel(image, x, y, color);
```

There are two functions for drawing lines, `imageline()` and `imagedashedline()`:

```
imageline(image, start_x, start_ y, end_x, end_ y, color);
imagedashedline(image, start_x, start_ y, end_x, end_ y, color);
```

There are two functions for drawing rectangles, one that simply draws the outline and one that fills the rectangle with the specified color:

```
imagerectangle(image, tlx, tly, brx, bry, color);
imagefilledrectangle(image, tlx, tly, brx, bry, color);
```

Specify the location and size of the rectangle by passing the coordinates of the top-left and bottom-right corners.

You can draw arbitrary polygons with the `imagepolygon()` and `imagefilled polygon()` functions:

```
imagepolygon(image, points, number, color);
imagefilledpolygon(image, points, number, color);
```

Both functions take an array of points. This array has two integers (the *x* and *y* coordinates) for each vertex on the polygon. The *number* argument is the number of vertices in the array (typically `count($points)/2`).

The `imagearc()` function draws an arc (a portion of an ellipse):

```
imagearc(image, center_x, center_y, width, height, start, end, color);
```

The ellipse is defined by its center, width, and height (height and width are the same for a circle). The start and end points of the arc are given as degrees counting counterclockwise from 3 o'clock. Draw the full ellipse with a *start* of `0` and an *end* of `360`.

There are two ways to fill in already-drawn shapes. The `imagefill()` function performs a flood fill, changing the color of the pixels starting at the given location. Any change in pixel color marks the limits of the fill. The `imagefilltoborder()` function lets you pass the particular color of the limits of the fill:

```
imagefill(image, x, y, color);
imagefilltoborder(image, x, y, border_color, color);
```

Another thing that you may want to do with your images is rotate them. This could be helpful if you are trying to create a web-style brochure, for example. The `imagerotate()` function allows you to rotate an image by an arbitrary angle:

```
imagerotate(image, angle, background_color);
```

The code in Example 10-4 shows the black box image from before, rotated by 45 degrees. The *background_color* option, used to specify the color of the uncovered area after the image is rotated, has been set to `1` to show the contrast of the black and white colors. Figure 10-2 shows the result of this code.

*Example 10-4. Image rotation example*

```
<?php
$image = imagecreate(200, 200);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);
imagefilledrectangle($image, 50, 50, 150, 150, $black);

$rotated = imagerotate($image, 45, 1);

header("Content-Type: image/png");
imagepng($rotated);
```

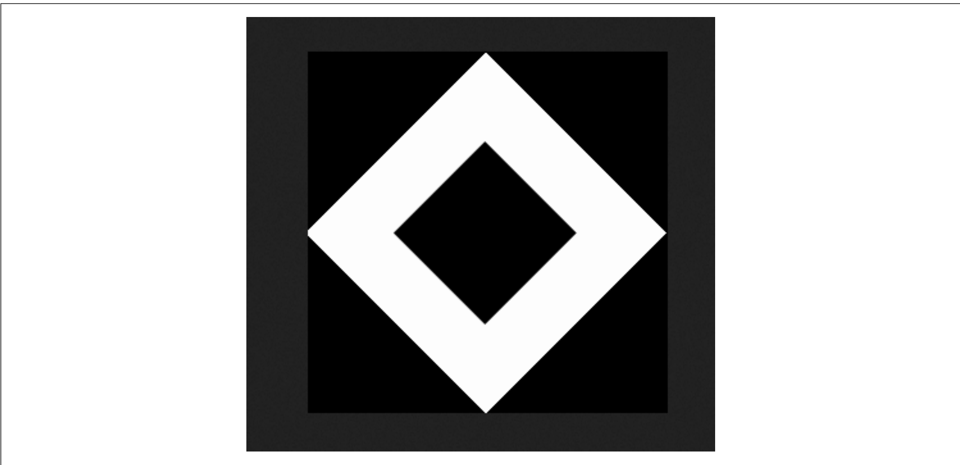


*Figure 10-2. Black box image rotated 45 degrees*

## Images with Text

Often it is necessary to add text to images. GD has built-in fonts for this purpose. Example 10-5 adds some text to our black square image.

*Example 10-5. Adding text to an image*

```
<?php
$image = imagecreate(200, 200);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
```

```
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);

imagefilledrectangle($image, 50, 50, 150, 150, $black);
imagestring($image, 5, 50, 160, "A Black Box", $black);

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-3 shows the output of Example 10-5.

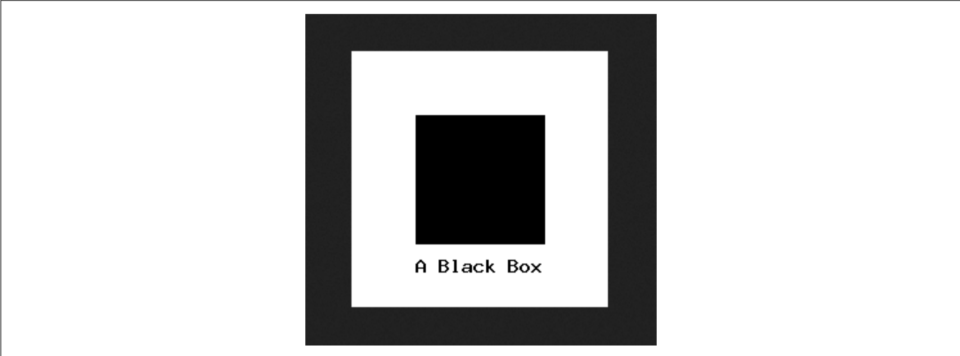



*Figure 10-3. The black box image with added text*

The `imagestring()` function adds text to an image. Specify the top-left point of the text, as well as the color and the font (by GD font identifier) to use:

```
imagestring(image, font_id, x, y, text, color);
```

## Fonts

GD identifies fonts by an ID. Five fonts are built in, and you can load additional fonts through the `imageloadfont()` function. The five built-in fonts are shown in Figure 10-4.

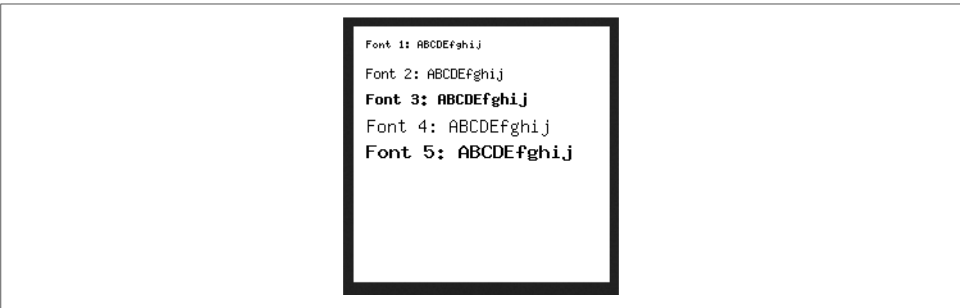



*Figure 10-4. Native GD fonts*

Here is the code used to show you these fonts:

```
<?php
$image = imagecreate(200, 200);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);

imagestring($image, 1, 10, 10, "Font 1: ABCDEfghij", $black);
imagestring($image, 2, 10, 30, "Font 2: ABCDEfghij", $black);
imagestring($image, 3, 10, 50, "Font 3: ABCDEfghij", $black);
imagestring($image, 4, 10, 70, "Font 4: ABCDEfghij", $black);
imagestring($image, 5, 10, 90, "Font 5: ABCDEfghij", $black);

header("Content-Type: image/png");
imagepng($image);
```

You can create your own bitmap fonts and load them into GD using the `imageloadfont()` function. However, these fonts are binary and architecture-dependent, making them nonportable from machine to machine. Using TrueType fonts with the TrueType functions in GD provides much more flexibility.

## TrueType Fonts

TrueType is an outline font standard; it provides more precise control over the rendering of the characters. To add text in a TrueType font to an image, use `imagettftext()`:

```
imagettftext(image, size, angle, x, y, color, font, text);
```

The *size* is measured in pixels. The *angle* is in degrees from 3 o'clock (`0` gives horizontal text, `90` gives vertical text going up the image, etc.). The *x* and *y* coordinates specify the lower-left corner of the baseline for the text. The text may include UTF-8[2] sequences of the form `ê` to print high-bit ASCII characters.

The font parameter is the location of the TrueType font to use for rendering the string. If the font does not begin with a leading `/` character, the *.ttf* extension is added and the font is looked up in */usr/share/fonts/truetype*.

By default, text in a TrueType font is antialiased. This makes most fonts much easier to read, although very slightly blurred. Antialiasing can make very small text harder to read, though—small characters have fewer pixels, so the adjustments of antialiasing are more significant.

You can turn off antialiasing by using a negative color index (e.g., `-4` means to use color index 4 without antialiasing the text).

2 UTF-8 is an 8-bit Unicode (*http://www.unicode.org*) encoding scheme.

Example 10-6 uses a TrueType font to add text to an image, searching for the font in the same location as the script, but still having to provide the full path to the location of the font file (included in the book's code examples).

*Example 10-6. Using a TrueType font*

```
<?php
$image = imagecreate(350, 70);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
$black = imagecolorallocate($image, 0x00, 0x00, 0x00);


$fontname = "c:/wamp64/www/bookcode/chapter_10/IndieFlower.ttf";

imagettftext($image, 20, 0, 10, 40, $black, $fontname, "The Quick Brown Fox");

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-5 shows the output of Example 10-6.

*Figure 10-5. Indie Flower TrueType font*

Example 10-7 uses `imagettftext()` to add vertical text to an image.

*Example 10-7. Displaying vertical TrueType text*

```
<?php
$image = imagecreate(70, 350);
$white = imagecolorallocate($image, 255, 255, 255);
$black = imagecolorallocate($image, 0, 0, 0);

$fontname = "c:/wamp64/www/bookcode/chapter_10/IndieFlower.ttf";

imagettftext($image, 20, 270, 28, 10, $black, $fontname, "The Quick Brown Fox");

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-6 shows the output of Example 10-7.

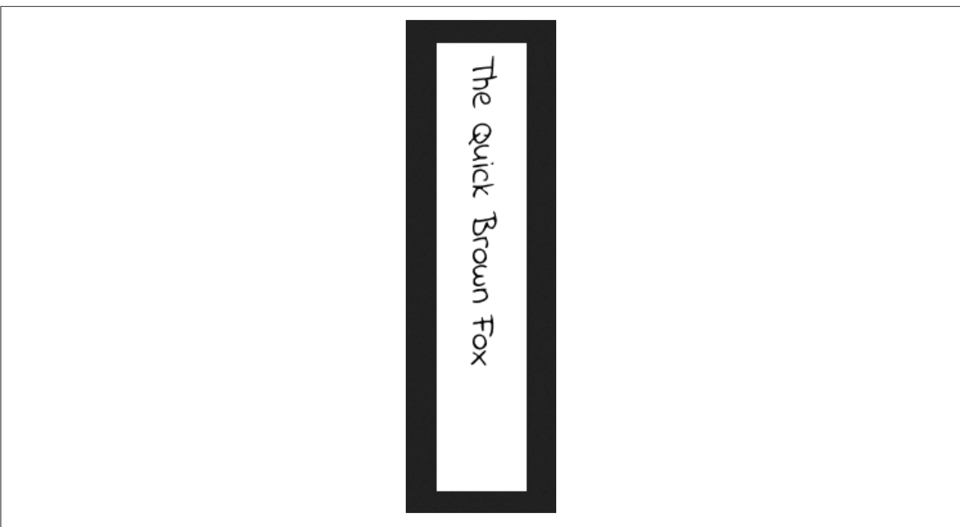



*Figure 10-6. Vertical TrueType text*

## Dynamically Generated Buttons

Creating images for buttons on the fly is one popular use for generating images (this topic was introduced in Chapter 1). Typically, this involves compositing text over a preexisting background image, as shown in Example 10-8.

*Example 10-8. Creating a dynamic button*

```
<?php
$font = "c:/wamp64/www/bookcode/chapter_10/IndieFlower.ttf" ;
$size = isset($_GET['size']) ? $_GET['size'] : 12;
$text = isset($_GET['text']) ? $_GET['text'] : 'some text';

$image = imagecreatefrompng("button.png");
$black = imagecolorallocate($image, 0, 0, 0);

if ($text) {
 // calculate position of text
 $tsize = imagettfbbox($size, 0, $font, $text);
 $dx = abs($tsize[2] - $tsize[0]);
 $dy = abs($tsize[5] - $tsize[3]);
 $x = (imagesx($image) - $dx ) / 2;
 $y = (imagesy($image) - $dy ) / 2 + $dy;

 // draw text
 imagettftext($image, $size, 0, $x, $y, $black, $font, $text);
}
```

```
header("Content-Type: image/png");
imagepng($image);
```

In this case, the blank button (*button.png*) is overwritten with the default text, as shown in Figure 10-7.

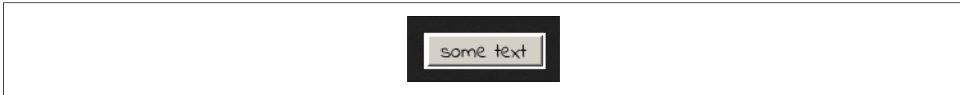



*Figure 10-7. Dynamic button with default text*

The script in Example 10-8 can be called from a page like this:

```
<img src="button.php?text=PHP+Button" />
```

This HTML generates the button shown in Figure 10-8.

*Figure 10-8. Button with generated text label*

The + character in the URL is the encoded form of a space. Spaces are illegal in URLs and must be encoded. Use PHP's `urlencode()` function to encode your button strings. For example:

```
<img src="button.php?text=<?= urlencode("PHP Button"); ?>" />
```

## Caching the Dynamically Generated Buttons

It is somewhat slower to generate an image than to send a static image. For buttons that will always look the same when called with the same text argument, you can implement a simple cache mechanism.

Example 10-9 generates the button only when no cache file for that button is found. The `$path` variable holds a directory, writable by the web server user, where buttons can be cached; make sure it can be reached from where you run this code. The `filesize()` function returns the size of a file, and `readfile()` sends the contents of a file to the browser. Because this script uses the text form parameter as the filename, it is very insecure. (Chapter 14, which covers security issues, explains why and how to fix it.)

*Example 10-9. Caching dynamic buttons*

```
<?php

$font = "c:/wamp64/www/bookcode/chapter_10/IndieFlower.ttf";
$size = isset($_GET['size']) ? $_GET['size'] : 12;
$text = isset($_GET['text']) ? $_GET['text'] : 'some text';

$path = "/tmp/buttons"; // button cache directory

// send cached version

if ($bytes = @filesize("{$path}/button.png")) {
 header("Content-Type: image/png");
 header("Content-Length: {$bytes}");
 readfile("{$path}/button.png");

 exit;
}

// otherwise, we have to build it, cache it, and return it
$image = imagecreatefrompng("button.png");
$black = imagecolorallocate($image, 0, 0, 0);

if ($text) {
 // calculate position of text
 $tsize = imagettfbbox($size, 0, $font, $text);
 $dx = abs($tsize[2] - $tsize[0]);
 $dy = abs($tsize[5] - $tsize[3]);
 $x = (imagesx($image) - $dx ) / 2;
 $y = (imagesy($image) - $dy ) / 2 + $dy;

 // draw text
 imagettftext($image, $size, 0, $x, $y, $black, $font, $text);

 // save image to file
 imagepng($image, "{$path}/{$text}.png");
}

header("Content-Type: image/png");
imagepng($image);
```

## A Faster Cache

Example 10-9 is still not as quick as it could be. Using Apache directives, you can bypass the PHP script entirely and load the cached image directly once it is created.

First, create a *buttons* directory somewhere under your web server's `DocumentRoot` and make sure that your web server user has permissions to write to this directory. For example, if the `DocumentRoot` directory is */var/www/html*, create */var/www/html/buttons*.

Second, edit your Apache *httpd.conf* file and add the following block:

```
<Location /buttons/>
 ErrorDocument 404 /button.php
</Location>
```

This tells Apache that requests for nonexistent files in the *buttons* directory should be sent to your *button.php* script.

Third, save Example 10-10 as *button.php*. This script creates new buttons, saving them to the cache and sending them to the browser. There are several differences from Example 10-9, though. We don't have form parameters in `$_GET`, because Apache handles error pages as redirections. Instead, we have to pull apart values in `$_SERVER` to find out which button we're generating. While we're at it, we delete the `'..'` in the filename to fix the security hole from Example 10-9.

Once *button.php* is installed, when a request comes in for something like *http://your.site/buttons/php.png*, the web server checks whether the *buttons/php.png* file exists. If it does not, the request is redirected to the *button.php* script, which creates the image (with the text "php") and saves it to *buttons/php.png*. Any subsequent requests for this file are served up directly without a line of PHP being run.

*Example 10-10. More efficient caching of dynamic buttons*

```
<?php
// bring in redirected URL parameters, if any
parse_str($_SERVER['REDIRECT_QUERY_STRING']);

$cacheDir = "/buttons/";
$url = $_SERVER['REDIRECT_URL'];

// pick out the extension
$extension = substr($url, strrpos($url, '.'));

// remove directory and extension from $url string
$file = substr($url, strlen($cacheDir), -strlen($extension));

// security - don't allow '..' in filename
$file = str_replace('..', '', $file);

// text to display in button
$text = urldecode($file);

$font = "c:/wamp64/www/bookcode/chapter_10/IndieFlower.ttf" ;

// build it, cache it, and return it
$image = imagecreatefrompng("button.png");
$black = imagecolorallocate($image, 0, 0, 0);

if ($text) {
```

```
  // calculate position of text
  $tsize = imagettfbbox($size, 0, $font, $text);
  $dx = abs($tsize[2] - $tsize[0]);
  $dy = abs($tsize[5] - $tsize[3]);
  $x = (imagesx($image) - $dx ) / 2;
  $y = (imagesy($image) - $dy ) / 2 + $dy;

  // draw text
  imagettftext($image, $size, 0, $x, $y, $black, $font, $text);

  // save image to file
  imagepng($image, "{$_SERVER['DOCUMENT_ROOT']}{$cacheDir}{$file}.png");
}

header("Content-Type: image/png");
imagepng($image);
```

One significant drawback to the mechanism in Example 10-10 is that the button text cannot contain any characters that are illegal in a filename. Nonetheless, this is the most efficient way to cache dynamically generated images. If you change the look of your buttons and you need to regenerate the cached images, simply delete all the images in your *buttons* directory, and they will be re-created as they are requested.

You can also take this a step further and get your *button.php* script to support multiple image types. Simply check `$extension` and call the appropriate `imagepng()`, `imagejpeg()`, or `imagegif()` function at the end of the script. You can also parse the filename and add modifiers such as color, size, and font, or pass them right in the URL. Because of the `parse_str()` call in the example, a URL such as *http://your.site/buttons/php.png?size=16* displays "php" in a font size of 16.

# Scaling Images

There are two ways to change the size of an image. The `imagecopyresized()` function is fast but crude, and may produce jagged edges in your new images. The `imagecopyresampled()` function is slower, but uses pixel interpolation to generate smooth edges and give clarity to the resized image. Both functions take the same arguments:

```
imagecopyresized(dest, src, dx, dy, sx, sy, dw, dh, sw, sh);
imagecopyresampled(dest, src, dx, dy, sx, sy, dw, dh, sw, sh);
```

The *dest* and *src* parameters are image handles. The point (*dx*, *dy*) is the point in the destination image where the region will be copied. The point (*sx*, *sy*) is the upper-left corner of the source image. The *sw*, *sh*, *dw*, and *dh* parameters give the width and height of the copy regions in the source and destination.

Example 10-11 takes the *php.jpg* image shown in Figure 10-9 and smoothly scales it down to one-quarter of its size, yielding the image in Figure 10-10.

*Example 10-11. Resizing with imagecopyresampled()*

```
<?php
$source = imagecreatefromjpeg("php_logo_big.jpg");

$width = imagesx($source);
$height = imagesy($source);
$x = $width / 2;
$y = $height / 2;

$destination = imagecreatetruecolor($x, $y);
imagecopyresampled($destination, $source, 0, 0, 0, 0, $x, $y, $width, $height);

header("Content-Type: image/png");
imagepng($destination);
```

*Figure 10-9. Original php.jpg image*

*Figure 10-10. Resulting 1/4-sized image*

Dividing the height and the width by 4 instead of 2 produces the output shown in Figure 10-11.

*Figure 10-11. Resulting 1/16-sized image*

## Color Handling

The GD library supports both 8-bit palette (256 color) images and true color images with alpha channel transparency.

To create an 8-bit palette image, use the `imagecreate()` function. The image's background is subsequently filled with the first color you allocate using `imagecolorallocate()`:

```
$width = 128;
$height = 256;

$image = imagecreate($width, $height);
$white = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
```

To create a true color image with a 7-bit alpha channel, use the `imagecreatetruecolor()` function:

```
$image = imagecreatetruecolor(width, height);
```

Use `imagecolorallocatealpha()` to create a color index that includes transparency:

```
$color = imagecolorallocatealpha(image, red, green, blue, alpha);
```

The *alpha* value is between 0 (opaque) and 127 (transparent).

While most people are used to an 8-bit (0–255) alpha channel, it is actually quite handy that GD's is 7-bit (0–127). Each pixel is represented by a 32-bit signed integer, with the four 8-bit bytes arranged like this:

```
High Byte Low Byte
{Alpha Channel} {Red} {Green} {Blue}
```

For a signed integer, the leftmost bit, or the highest bit, is used to indicate whether the value is negative, thus leaving only 31 bits of actual information. PHP's default integer value is a signed long into which we can store a single GD palette entry. Whether that integer is positive or negative tells us whether antialiasing is enabled for that palette entry.

Unlike with palette images, with true color images the first color you allocate does not automatically become your background color. Instead, the image is initially filled with fully transparent pixels. Call `imagefilledrectangle()` to fill the image with any background color you want.

Example 10-12 creates a true color image and draws a semitransparent orange ellipse on a white background.

*Example 10-12. A simple orange ellipse on a white background*

```
<?php
$image = imagecreatetruecolor(150, 150);
$white = imagecolorallocate($image, 255, 255, 255);

imagealphablending($image, false);
imagefilledrectangle($image, 0, 0, 150, 150, $white);

$red = imagecolorallocatealpha($image, 255, 50, 0, 50);
imagefilledellipse($image, 75, 75, 80, 63, $red);

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-12 shows the output of Example 10-12.

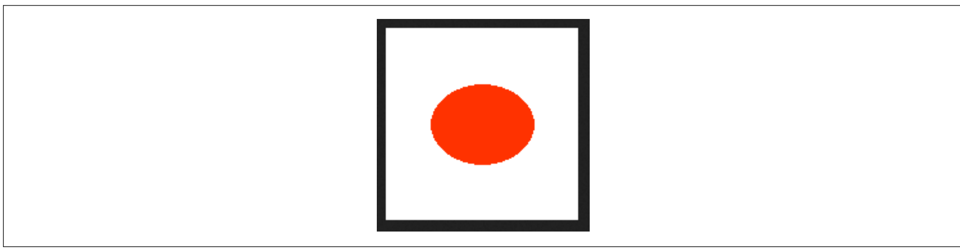


*Figure 10-12. An orange ellipse on a white background*

You can use the `imagetruecolortopalette()` function to convert a true color image to one with a color index (also known as a *paletted* image).

## Using the Alpha Channel

In Example 10-12, we turned off *alpha blending* before drawing our background and our ellipse. Alpha blending is a toggle that determines whether the alpha channel, if present, should be applied when the image is drawn. If alpha blending is off, the old pixel is replaced with the new pixel. If an alpha channel exists for the new pixel, it is maintained, but all pixel information for the original pixel being overwritten is lost.

Example 10-13 illustrates alpha blending by drawing a gray rectangle with a 50% alpha channel over an orange ellipse.

*Example 10-13. A gray rectangle with a 50% alpha channel overlaid*

```
<?php
$image = imagecreatetruecolor(150, 150);
imagealphablending($image, false);

$white = imagecolorallocate($image, 255, 255, 255);
```

```
imagefilledrectangle($image, 0, 0, 150, 150, $white);

$red = imagecolorallocatealpha($image, 255, 50, 0, 63);
imagefilledellipse($image, 75, 75, 80, 50, $red);

imagealphablending($image, false);

$gray = imagecolorallocatealpha($image, 70, 70, 70, 63);
imagefilledrectangle($image, 60, 60, 120, 120, $gray);

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-13 shows the output of Example 10-13 (alpha blending is still turned off).

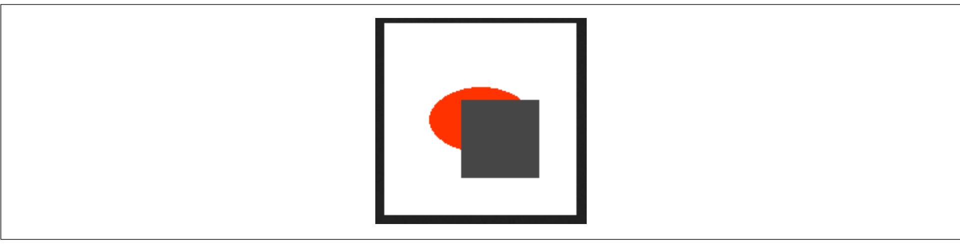


*Figure 10-13. A gray rectangle over the orange ellipse*

If we change Example 10-13 to enable alpha blending just before the call to `imagefilledrectangle()`, we get the image shown in Figure 10-14.

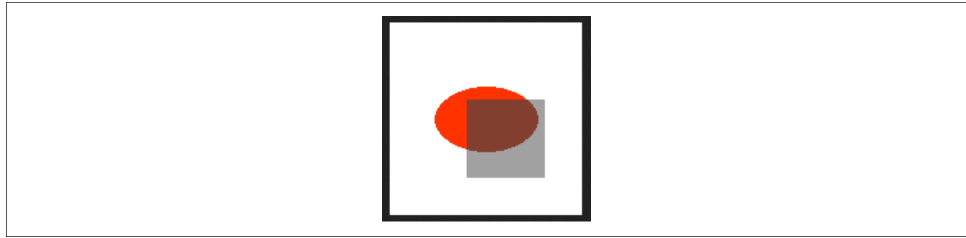


*Figure 10-14. Image with alpha blending enabled*

## Identifying Colors

To check the color index for a specific pixel in an image, use `imagecolorat()`:

```
$color = imagecolorat(image, x, y);
```

For images with an 8-bit color palette, the function returns a color index that you then pass to `imagecolorsforindex()` to get the actual RGB values:

```
$values = imagecolorsforindex(image, index);
```

The array returned by `imagecolorsforindex()` has the keys `'red'`, `'green'`, and `'blue'`. If you call `imagecolorsforindex()` on a color from a true color image, the returned array also has a value for the key `'alpha'`. The values for these keys correspond to the 0–255 color values and the 0–127 alpha value used when calling `imagecolorallocate()` and `imagecolorallocatealpha()`.

## True Color Indexes

The color index returned by `imagecolorallocatealpha()` is really a 32-bit signed long, with the first three bytes holding the red, green, and blue values, respectively. The next bit indicates whether antialiasing is enabled for this color, and the remaining seven bits hold the transparency value.

For example:

```
$green = imagecolorallocatealpha($image, 0, 0, 255, 127);
```

This code sets `$green` to `2130771712`, which in hex is `0x7F00FF00` and in binary is `01111111000000001111111100000000`.

This is equivalent to the following `imagecolorresolvealpha()` call:

```
$green = (127 << 24) | (0 << 16) | (255 << 8) | 0;
```

You can also drop the two `0` entries in this example and just make it:

```
$green = (127 << 24) | (255 << 8);
```

To deconstruct this value, you can use something like this:

```
$a = ($col & 0x7F000000) >> 24;
$r = ($col & 0x00FF0000) >> 16;
$g = ($col & 0x0000FF00) >> 8;
$b = ($col & 0x000000FF);
```

Direct manipulation of color values like this is rarely necessary. One application is to generate a color-testing image that shows the pure shades of red, green, and blue. For example:

```
$image = imagecreatetruecolor(256, 60);

for ($x = 0; $x < 256; $x++) {
 imageline($image, $x, 0, $x, 19, $x);
 imageline($image, 255 - $x, 20, 255 - $x, 39, $x << 8);
 imageline($image, $x, 40, $x, 59, $x<<16);
}

header("Content-Type: image/png");
imagepng($image);
```

Figure 10-15 shows the output of the color-testing program.

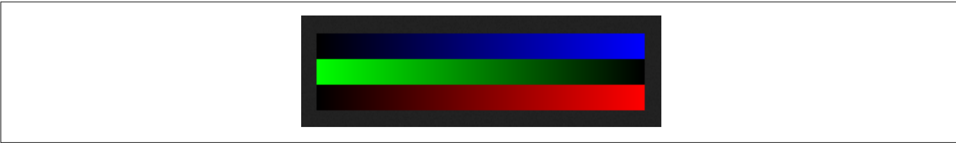


*Figure 10-15. The color test*

Obviously it will be much more colorful than what we can show you here in black and white print, so try this example for yourself. In this particular example, it is much easier to simply calculate the pixel color than to call `imagecolorallocatealpha()` for every color.

## Text Representation of an Image

An interesting use of the `imagecolorat()` function is to loop through each pixel in an image and do something with that color data. Example 10-14 prints # for each pixel in the image *php-tiny.jpg* in that pixel's color.

*Example 10-14. Converting an image to text*

```
<html><body bgcolor="#000000">

<tt><?php
$image = imagecreatefromjpeg("php_logo_tiny.jpg");

$dx = imagesx($image);
$dy = imagesy($image);

for ($y = 0; $y < $dy; $y++) {
 for ($x = 0; $x < $dx; $x++) {
 $colorIndex = imagecolorat($image, $x, $y);
 $rgb = imagecolorsforindex($image, $colorIndex);

 printf('<font color=#%02x%02x%02x>#</font>',
 $rgb['red'], $rgb['green'], $rgb['blue']);
 }

 echo "<br>\n";
} ?></tt>

</body></html>
```

The result is an ASCII representation of the image, as shown in Figure 10-16.

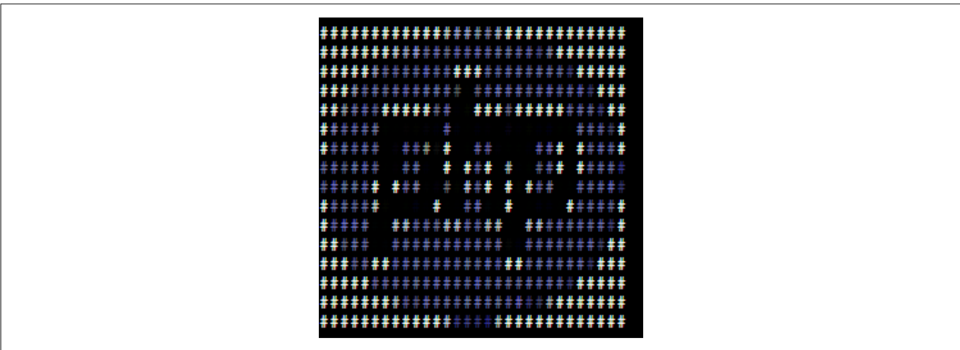


*Figure 10-16. ASCII representation of an image*

## What's Next

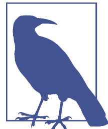
There are many different ways to manipulate images on the fly with PHP. This certainly dispels the myth that PHP is useful only for generating web HTML content. If you have the time and desire to explore what's possible in more depth, feel free to experiment with the code samples here. In the next chapter we'll be looking at another myth-buster in generating dynamic PDF documents. Stay tuned!

# CHAPTER 11
# PDF

Adobe's Portable Document Format (PDF) is a popular way to get a consistent look, both on screen and in print, for documents. This chapter shows you how to dynamically create PDF files with text, graphics, links, and more. Doing so opens the door to many applications. You can create almost any kind of business document, including form letters, invoices, and receipts. In addition, you can automate most paperwork by overlaying text onto a scan of the paper form and saving the result as a PDF file.

## PDF Extensions

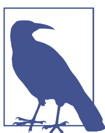
PHP has several libraries for generating PDF documents. This chapter's examples use the popular FPDF library, a set of PHP code you include in your scripts with the `require()` function—it doesn't require any server-side configuration or support, so you can use it even without support from your host. The basic concepts, structure, and features of a PDF file should be common to all the PDF libraries, however.



Another PDF-generating library, TCPDF, is better at handling HTML special characters and UTF-8 multilanguage output than FPDF. Look it up if you need that capability. The methods you'll use are `writeHTMLCell()` and `writeHTML()`.

## Documents and Pages

A PDF document is made up of a number of pages, each of which contains text and/or images. This section shows you how to create a document, add pages in that document, write text to the pages, and send the pages back to the browser when you're done.



The examples in this chapter assume that you have at least the Adobe PDF document viewer installed as an add-on to your web browser. These examples will not work otherwise. You can get the add-on from the Adobe website.

## A Simple Example

Let's start with a simple PDF document. Example 11-1 writes the text "Hello Out There!" to a page and then displays the resulting PDF document.

*Example 11-1. "Hello Out There!" in PDF*

```
<?php

require("../fpdf/fpdf.php"); // path to fpdf.php

$pdf = new FPDF();
$pdf->addPage();

$pdf->setFont("Arial", 'B', 16);
$pdf->cell(40, 10, "Hello Out There!");

$pdf->output();
```

Example 11-1 follows the basic steps involved in creating a PDF document: creating a new PDF object instance, creating a page, setting a valid font for the PDF text, and writing the text to a "cell" on the page. Figure 11-1 shows the output of Example 11-1.

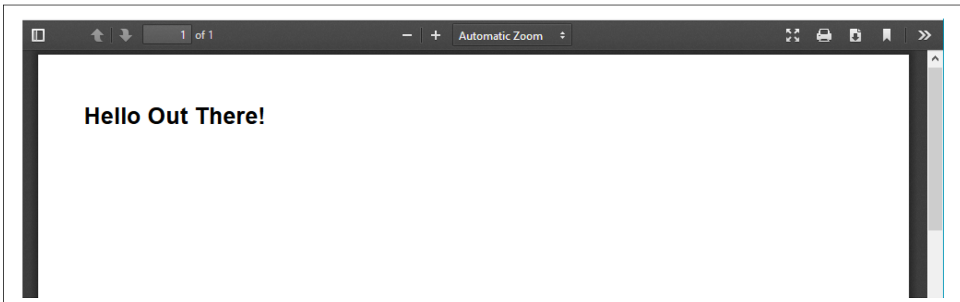



*Figure 11-1. "Hello Out There!" PDF example*

## Initializing the Document

In Example 11-1, we started by making a reference to the FPDF library with the `require()` function. Then the code created a new instance of the FPDF object. Note that all the calls to the new FPDF instance are object-oriented calls to methods in that object. (Refer to Chapter 6 if you have trouble with the examples in this chapter.)

After you have created the new instance of the FPDF object, you'll need to add at least one page to the object, so the `AddPage()` method is called. Next, you need to set the font for the output you are about to generate with the `SetFont()` call. Then, using the `cell()` method call, you can send the output to your created document. To send all your work to the browser, simply use the `output()` method.

## Outputting Basic Text Cells

In the FPDF library, a *cell* is a rectangular area on the page that you can create and control. This cell can have a height, width, and border, and of course it can contain text. The basic syntax for the `cell()` method is as follows:

```
cell(float w [, float h [, string txt [, mixed border
 [, int ln [, string align [, int fill [, mixed link]]]]]]])
```

The first option is the width, then the height, and then the text to be output. This is followed by the border, the new line control, its alignment, any fill color for the text, and finally whether you want the text to be an HTML link. So, for example, if we want to change our original example to have a border and be center-aligned, we would change the cell code to the following:

```
$pdf->cell(90, 10, "Hello Out There!", 1, 0, 'C');
```

You'll use the `cell()` method extensively when generating PDF documents with FPDF, so you'd be well served by taking some time to learn the ins and outs of this method. We will cover most of them in this chapter.

# Text

Text is the heart of a PDF file. Accordingly, there are many options for changing its appearance and layout. In this section, we'll discuss the coordinate system used in PDF documents, functions for inserting text and changing text attributes, and font usage.

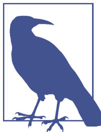
## Coordinates

The origin (0, 0) in a PDF document with the FPDF library is in the top-left corner of the defined page. All of the measurements are specified in points, millimeters, inches, or centimeters. A point (the default) is equal to 1/72 of an inch, or 0.35 mm. In Example 11-2, we change the defaults of the page dimensions to inches with the `FPDF()` class instantiation-constructor method. The other options with this call are the orientation of the page (portrait or landscape) and the page size (typically Legal or Letter). The full options of this instantiation are shown in Table 11-1.

*Table 11-1. FPDF options*

| FPDF() constructor parameters | Parameter options |
|---|---|
| Orientation | P (portrait; default)<br>L (landscape) |
| Units of measurement | pt (point, or 1/72 of an inch; default)<br>in (inch)<br>mm (millimeter)<br>cm (centimeter) |
| Page size | Letter (default)<br>Legal<br>A5<br>A3<br>A4 or a customizable size (see FPDF documentation) |

Also in Example 11-2, we use the `ln()` method call to manage what line of the page we are on. The `ln()` method can take an optional argument, instructing it how many units (i.e., the unit of measurement defined in the constructor call) to move. In our case, we've defined the page to be in inches, so we're moving through the document in measurement units of inches. Further, since we've defined the page to be in inches, the coordinates for the `cell()` method are also rendered in inches.



This is not really the ideal approach for building a PDF page because you don't have as fine-grained control with inches as you would with points or millimeters. We've used inches in this instance so that the examples can be seen more clearly.

Example 11-2 puts text in the corners and center of a page.

*Example 11-2. Demonstrating coordinates and line management*

```
<?php
require("../fpdf/fpdf.php");

$pdf = new FPDF('P', 'in', 'Letter');
$pdf->addPage();

$pdf->setFont('Arial', 'B', 24);

$pdf->cell(0, 0, "Top Left!", 0, 1, 'L');
$pdf->cell(6, 0.5, "Top Right!", 1, 0, 'R');
$pdf->ln(4.5);

$pdf->cell(0, 0, "This is the middle!", 0, 0, 'C');
$pdf->ln(5.3);
```

```
$pdf->cell(0, 0, "Bottom Left!", 0, 0, 'L');
$pdf->cell(0, 0, "Bottom Right!", 0, 0, 'R');

$pdf->output();
```

The output of Example 11-2 is shown in Figure 11-2.

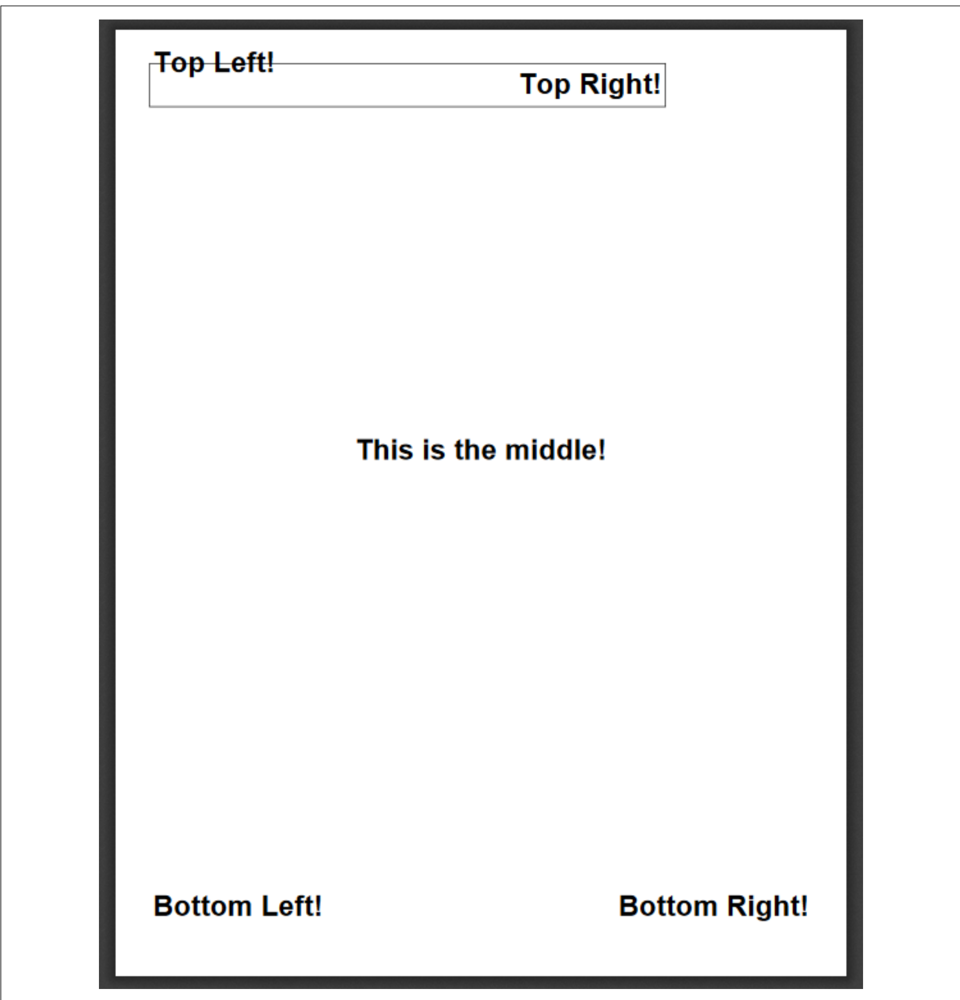



*Figure 11-2. Coordinate and line control demo output*

So let's analyze this code a little. After we define the page with the constructor, we see these lines of code:

```
$pdf->cell(0, 0, "Top Left!", 0, 1, 'L');
$pdf->cell(6, 0.5, "Top Right!", 1, 0, 'R');
$pdf->ln(4.5);
```

The first `cell()` method call tells the PDF class to start at the top coordinates (0,0) and write out the left-justified text "Top Left!" with no border, and to insert a line break at the end of the output. The next `cell()` method call prompts the creation of a cell six inches wide, again starting on the lefthand side of the page, with a half-inch-high border and the right-justified text "Top Right!" We then tell the PDF class to move down 4½ inches on the page with the `ln(4.5)` statement, and continue the output generation from that point. As you can see, there are a lot of possible combinations with the `cell()` and `ln()` methods alone. But that is not all that the FPDF library can do.

## Text Attributes

There are three common ways to alter the appearance of text: bold, underline, and italics. In Example 11-3 the `SetFont()` method (introduced earlier in the chapter) is used to alter the formatting of the outgoing text. Note that these alterations in the text's appearance are not exclusive (i.e., you can use them in any combination) and that the font name is changed in the last `SetFont()` call.

*Example 11-3. Demonstrating font attributes*

```
<?php
require("../fpdf/fpdf.php");

$pdf = new FPDF();
$pdf->addPage();

$pdf->setFont("Arial", '', 12);
$pdf->cell(0, 5, "Regular normal Arial Text here, size 12", 0, 1, 'L');
$pdf->ln();

$pdf->setFont("Arial", 'IBU', 20);
$pdf->cell(0, 15, "This is Bold, Underlined, Italicised Text size 20", 0, 0, 'L');
$pdf->ln();

$pdf->setFont("Times", 'IU', 15);
$pdf->cell(0, 5, "This is Underlined Italicised 15pt Times", 0, 0, 'L');

$pdf->output();
```

Also, in this code the constructor has been called with no attributes passed into it, using the default values of portrait, points, and letter. The output of Example 11-3 is shown in Figure 11-3.

Regular normal Arial Text here, size 12

***This is Bold, Underlined, Italicised Text size 20***

*This is Underlined Italicised 15pt Times*

*Figure 11-3. Changing font types, sizes, and attributes*

The available font styles that come with FPDF are:

- `Courier` (fixed-width)
- `Helvetica` or `Arial` (synonymous; sans serif)
- `Times` (serif)
- `Symbol` (symbols)
- `ZapfDingbats` (symbols)

You can include any other font family for which you have the definition file by using the `AddFont()` method.

Of course, this wouldn't be any fun at all if you couldn't change the color of the text that you're outputting to the PDF definition. Enter the `SetTextColor()` method. This method takes the existing font definition and simply changes the color of the text. Be sure to call this method before you use the `cell()` method so that the content of the cell can be changed. The color parameters are combinations of red, green, and blue numeric constants from `0` (none) to `255` (full color). If you do not pass in the second and third parameters, then the first number will be a shade of gray with red, green, and blue values equal to the single passed value. Example 11-4 shows how this can be employed.

*Example 11-4. Demonstrating color attributes*

```
<?php
require("../fpdf/fpdf.php");

$pdf = new FPDF();
$pdf->addPage();

$pdf->setFont("Times", 'U', 15);
$pdf->setTextColor(128);
$pdf->cell(0, 5, "Times font, Underlined and shade of Grey Text", 0, 0, 'L');
```

```
$pdf->ln(6);

$pdf->setTextColor(255, 0, 0);
$pdf->cell(0, 5, "Times font, Underlined and Red Text", 0, 0, 'L');

$pdf->output();
```

Figure 11-4 is the result of the code in Example 11-4.

Times font, Underlined and shade of Grey Text
Times font, Underlined and Red Text

*Figure 11-4. Adding color to the text output*

## Page Headers, Footers, and Class Extension

So far we've looked only at what can be output to the PDF page in small quantities. We did this intentionally, to show you the variety of what you can do within a controlled environment. Now we need to expand what the FPDF library can do. Remember that this library actually is just a class definition provided for your use and extension, the latter of which we'll look at now. Since FPDF is indeed a class definition, all we have to do to extend it is to use the object command that is native to PHP, like this:

```
class MyPDF extends FPDF
```

Here we take the `FPDF` class and extend it with a new name of `MyPDF`. Then we can extend any of the methods in the object. We can even add more methods to our class extension if we so desire, but more on that later. The first two methods that we'll look at are extensions of existing empty methods that are predefined in the parent of the `FPDF` class: `header()` and `footer()`. These methods, as their names imply, generate page headers and footers for each page in your PDF document. Example 11-5, which is rather long, shows the definition of these two methods. You will notice only a few newly used methods; the most significant is `AliasNbPages()`, which is used simply to track the overall page count in the PDF document before it is sent to the browser.

*Example 11-5. Defining header and footer methods*

```
<?php
require("../fpdf/fpdf.php");

class MyPDF extends FPDF
```

```
{
 function header()
 {
 global $title;

 $this->setFont("Times", '', 12);
 $this->setDrawColor(0, 0, 180);
 $this->setFillColor(230, 0, 230);
 $this->setTextColor(0, 0, 255);
 $this->setLineWidth(1);

 $width = $this->getStringWidth($title) + 150;
 $this->cell($width, 9, $title, 1, 1, 'C', 1);
 $this->ln(10);
 }

 function footer()
 {
 //Position at 1.5 cm from bottom
 $this->setY(-15);
 $this->setFont("Arial", 'I', 8);
 $this->cell(0, 10,
 "This is the page footer -> Page {$this->pageNo()}/{nb}", 0, 0, 'C');
 }
}

$title = "FPDF Library Page Header";

$pdf = new MyPDF('P', 'mm', 'Letter');
$pdf->aliasNbPages();
$pdf->addPage();

$pdf->setFont("Times", '', 24);
$pdf->cell(0, 0, "some text at the top of the page", 0, 0, 'L');
$pdf->ln(225);

$pdf->cell(0, 0, "More text toward the bottom", 0, 0, 'C');

$pdf->addPage();
$pdf->setFont("Arial", 'B', 15);

$pdf->cell(0, 0, "Top of page 2 after header", 0, 1, 'C');

$pdf->output();
```

The results of Example 11-5 are shown in Figure 11-5. This is a shot of both pages side by side to show you the page count in the footers and the page number at the top of the page(s) after page 1. The header has a cell with some coloring (for cosmetic effect); of course, you don't have to use colors if you don't want to.

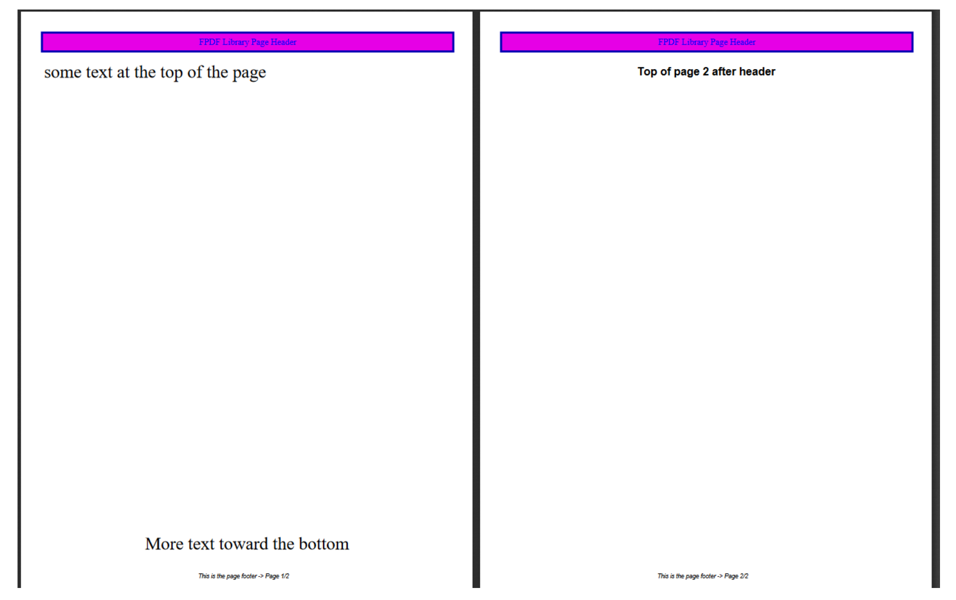



*Figure 11-5. FPDF header and footer addition*

## Images and Links

The FPDF library can also handle image insertion and control links within the PDF document or externally to outside web addresses. Let's first look at how FPDF allows you to insert graphics into your document. Perhaps you're building a PDF document that uses your company logo and you want to make a banner to print at the top of each page. We can use the `header()` and `footer()` methods that we defined in the previous section to do this. Once we have an image file to use, we simply call the `image()` method to place the image in the PDF document.

The new `header()` method code looks like this:

```
function header()
{
 global $title;

 $this->setFont("Times", '', 12);
 $this->setDrawColor(0, 0, 180);
 $this->setFillColor(230, 0, 230);
 $this->setTextColor(0, 0, 255);
 $this->setLineWidth(0.5);

 $width = $this->getStringWidth($title) + 120;


 $this->image("php_logo_big.jpg", 10, 10.5, 15, 8.5);
```

```
    $this->cell($width, 9, $title, 1, 1, 'C');
    $this->ln(10);
  }
```

As you can see, the `image()` method's parameters are the filename of the image to use, the *x* coordinate at which to start the image output, the *y* coordinate, and the width and height of the image. If you don't specify the width and height, then FPDF will do its best to render the image at the *x* and *y* coordinates that you specified. The code has changed a little in other areas as well. We removed the fill color parameter from the `cell()` method call even though we still have the fill color method called. This makes the box area around the header cell white so that we can insert the image without hassle.

The output of this new header with the image inserted is shown in Figure 11-6.

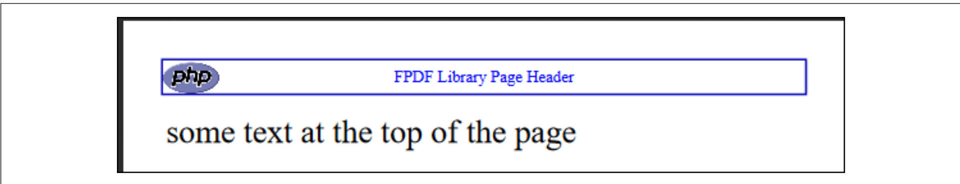



*Figure 11-6. PDF page header with inserted image file*

This section also has links in its title, so now let's turn our attention to how to use FPDF to add links to PDF documents. FPDF can create two kinds of links: an *internal* link (i.e., one within the PDF document to another location within the same document, such as two pages later) and an *external* link to a web URL.

An internal link is created in two parts. First you define the starting point, or origin, for the link, and then you set the anchor, or destination, for where the link will go when it is clicked. To set a link's origin, use the `addLink()` method. This method will return a handle that you need to use when creating the destination portion of the link. To set the destination, use the `setLink()` method, which takes the origin's link handle as its parameter so that it can perform the join between the two steps.

An external URL type link can be created in two ways. If you are using an image as a link, you will need to use the `image()` method. If you want to use straight text as a link, you'll need to use the `cell()` or `write()` method. We use the `write()` method in this example.

Both internal and external links are shown in Example 11-6.

*Example 11-6. Creating internal and external links*

```
<?php
require("../fpdf/fpdf.php");
```

```
$pdf = new FPDF();

// First page
$pdf->addPage();
$pdf->setFont("Times", '', 14);

$pdf->write(5, "For a link to the next page - Click");
$pdf->setFont('', 'U');
$pdf->setTextColor(0, 0, 255);
$linkToPage2 = $pdf->addLink();
$pdf->write(5, "here", $linkToPage2);
$pdf->setFont('');

// Second page
$pdf->addPage();
$pdf->setLink($linkToPage2);
$pdf->image("php-tiny.jpg", 10, 10, 30, 0, '', "http://www.php.net");
$pdf->ln(20);

$pdf->setTextColor(1);
$pdf->cell(0, 5, "Click the following link, or click on the image", 0, 1, 'L');
$pdf->setFont('', 'U');
$pdf->setTextColor(0,0,255);
$pdf->write(5, "www.oreilly.com", "http://www.oreilly.com");

$pdf->output();
```

The two-page output that this code produces is shown in Figures 11-7 and 11-8.

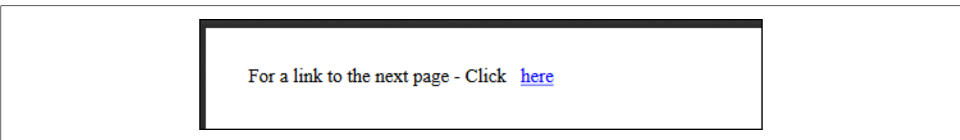
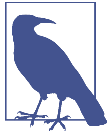



*Figure 11-7. First page of linked PDF document*

*Figure 11-8. Second page of linked PDF document with URL links*

## Tables and Data

So far we've looked only at PDF materials that are static in nature. But PHP, being what it is, does so much more than static processes. In this section, we'll combine some data from a database (using a MySQL example of the database information from Chapter 9) and FPDF's ability to generate tables.



Be sure to reference the database file structures available in Chapter 9 to follow along in this section.

Example 11-7 is, again, a little lengthy. However, it is well commented, so read through it here first; we'll cover the highlights after the listing.

*Example 11-7. Generating a table*

```
<?php
require("../fpdf/fpdf.php");

class TablePDF extends FPDF
{
 function buildTable($header, $data)
 {
 $this->setFillColor(255, 0, 0);
 $this->setTextColor(255);
 $this->setDrawColor(128, 0, 0);
 $this->setLineWidth(0.3);
 $this->setFont('', 'B');

//Header
// make an array for the column widths
 $widths = array(85, 40, 15);
// send the headers to the PDF document
 for($i = 0; $i < count($header); $i++) {
 $this->cell($widths[$i], 7, $header[$i], 1, 0, 'C', 1);
 }

 $this->ln();

// Color and font restoration
 $this->setFillColor(175);
 $this->setTextColor(0);
 $this->setFont('');

// now spool out the data from the $data array
 $fill = 0;// used to alternate row color backgrounds
 $url = "http://www.oreilly.com";
```

```
 foreach($data as $row)
 {
 $this->cell($widths[0], 6, $row[0], 'LR', 0, 'L', $fill);

// set colors to show a URL style link
 $this->setTextColor(0, 0, 255);
 $this->setFont('', 'U');
 $this->cell($widths[1], 6, $row[1], 'LR', 0, 'L', $fill, $url);

// restore normal color settings
 $this->setTextColor(0);
 $this->setFont('');
 $this->cell($widths[2], 6, $row[2], 'LR', 0, 'C', $fill);

 $this->ln();

 $fill = ($fill) ? 0 : 1;
 }
 $this->cell(array_sum($widths), 0, '', 'T');
 }
}

//connect to database
$dbconn = new mysqli('localhost', 'dbusername', 'dbpassword', 'library');
$sql = "SELECT * FROM books ORDER BY title";
$result = $dbconn->query($sql);

// build the data array from the database records.
while ($row = $result->fetch_assoc()) {
 $data[] = array($row['title'], $row['ISBN'], $row['pub_year']);
}

// start and build the PDF document
$pdf = new TablePDF();

// Column titles
$header = array("Title", "ISBN", "Year");

$pdf->setFont("Arial", '', 14);

$pdf->addPage();
$pdf->buildTable($header, $data);

$pdf->output();
```

We are using the database connection and building two arrays to send to the `buildTable()` custom method of this extended class. Inside the `buildTable()` method, we set colors and font attributes for the table header. Then, we send out the headers based on the first passed-in array. There is another array called `$width` used to set the column widths in the calls to `cell()`.

After the table header is sent out, we use the `$data` array containing the database information and walk through that array with a `foreach` loop. Notice here that the `cell()` method is using `'LR'` for its `border` parameter. This inserts borders on the left and right of the cell in question, thus effectively adding the sides to the table rows. We also add a URL link to the second column just to show you that it can be done in concert with the table row construction. Lastly, we use a `$fill` variable to flip back and forth so that the background color will alternate as the table is built row by row.

The last call to the `cell()` method in this `buildTable()` method is used to draw the bottom of the table and close off the columns.

The result of this code is shown in Figure 11-9.

| Title | ISBN | Year |
|---|---|---|
| Executive Orders | 0-425-15863-2 | 1996 |
| Exploring the Earth and the Cosmos | 0-517-546671 | 1982 |
| Forward the Foundation | 0-553-56507-9 | 1993 |
| Foundation | 0-553-80371-9 | 1951 |
| Foundation and Empire | 0-553-29337-0 | 1952 |
| Foundation's Edge | 0-553-29338-9 | 1982 |
| I, Robot | 0-553-29438-5 | 1950 |
| Isaac Asimov: Gold | 0-06-055652-8 | 1995 |
| Rainbow Six | 0-425-17034-9 | 1998 |
| Red Rabbit | 0-399-14870-1 | 2000 |
| Roots | 0-440-17464-3 | 1974 |
| Second Foundation | 0-553-29336-2 | 1953 |
| Teeth of the Tiger | 0-399-15079-X | 2003 |
| The Best of Isaac Asimov | 0-449-20829-X | 1973 |
| The Hobbit | 0-261-10221-4 | 1937 |
| The Return of The King | 0-261-10237-0 | 1955 |
| The Sum of All Fears | 0-425-13354-0 | 1991 |
| The Two Towers | 0-261-10236-2 | 1954 |

*Figure 11-9. FPDF-generated table based on database information with active URL links*

## What's Next

There are quite a few other features of FPDF that are not covered in this chapter. Be sure to go to the library's website to see other examples of what it can help you accomplish. There are code snippets and fully functional scripts available there as well as a discussion forum—all designed to help you become an FPDF expert.

In the next chapter we'll be switching gears a little to explore the interactions between PHP and XML. We will be covering some of the techniques that can be used to "consume" XML and how to parse it with a built-in library called SimpleXML.

# CHAPTER 12
# XML

XML, the Extensible Markup Language, is a standardized data format. It looks a little like HTML, with tags (`<example>like this</example>`) and entities (`&`). Unlike HTML, however, XML is designed to be easy to programmatically parse, and there are rules for what you can and cannot do in an XML document. XML is now the standard data format in fields as diverse as publishing, engineering, and medicine. It's used for remote procedure calls, databases, purchase orders, and much more.

There are many scenarios where you might want to use XML. Because it is a common format for data transfer, other programs can emit XML files for you to either extract information from (*parse*) or display in HTML (*transform*). This chapter shows you how to use the XML parser bundled with PHP, as well as how to use the optional XSLT extension to transform XML. We also briefly cover generating XML.

Recently, XML has been used in remote procedure calls (XML-RPC). A client encodes a function name and parameter values in XML and sends them via HTTP to a server. The server decodes the function name and values, decides what to do, and returns a response value encoded in XML. XML-RPC has proved a useful way to integrate application components written in different languages. We'll show you how to write XML-RPC servers and clients in Chapter 16, but for now let's look at the basics of XML.

## Lightning Guide to XML

Most XML consists of elements (like HTML tags), entities, and regular data. For example:

```
<book isbn="1-56592-610-2">
 <title>Programming PHP</title>
 <authors>
 <author>Rasmus Lerdorf</author>
```

```
 <author>Kevin Tatroe</author>
 <author>Peter MacIntyre</author>
 </authors>
</book>
```

In HTML, you often have an open tag without a close tag. The most common example of this is:

```
<br>
```

In XML, that is illegal. XML requires that every open tag be closed. For tags that don't enclose anything, such as the line break `<br>`, XML adds this syntax:

```
<br />
```

Tags can be nested but cannot overlap. For example, this is valid:

```
<book><title>Programming PHP</title></book>
```

This, however, is not valid, because the `<book>` and `<title>` tags overlap:

```
<book><title>Programming PHP</book></title>
```

XML also requires that the document begin with a processing instruction that identifies the version of XML being used (and possibly other things, such as the text encoding used). For example:

```
<?xml version="1.0" ?>
```

The final requirement of a well-formed XML document is that there be only one element at the top level of the file. For example, this is well formed:

```
<?xml version="1.0" ?>
<library>
 <title>Programming PHP</title>
 <title>Programming Perl</title>
 <title>Programming C#</title>
</library>
```

This is not well formed, as there are three elements at the top level of the file:

```
<?xml version="1.0" ?>
<title>Programming PHP</title>
<title>Programming Perl</title>
<title>Programming C#</title>
```

XML documents generally are not completely ad hoc. The specific tags, attributes, and entities in an XML document, and the rules governing how they nest, compose the structure of the document. There are two ways to write down this structure: the *document type definition* (DTD) and the *schema*. DTDs and schemas are used to validate documents—that is, to ensure that they follow the rules for their type of document.

Most XML documents don't include a DTD; in these cases, the document is considered valid merely if it's valid XML. Others identify the DTD as an external entity with a line that gives the name and location (file or URL) of the DTD:

```
<!DOCTYPE rss PUBLIC 'My DTD Identifier' 'http://www.example.com/my.dtd'>
```

Sometimes it's convenient to encapsulate one XML document in another. For example, an XML document representing a mail message might have an `attachment` element that surrounds an attached file. If the attached file is XML, it's a nested XML document. What if the mail message document has a `body` element (the subject of the message), and the attached file is an XML representation of a dissection that also has a `body` element, but this element has completely different DTD rules? How can you possibly validate or make sense of the document if the meaning of `body` changes partway through?

This problem is solved with the use of namespaces. Namespaces let you qualify the XML tag—for example, `email:body` and `human:body`.

There's a lot more to XML than we have time to go into here. For a gentle introduction to XML, read *Learning XML* (O'Reilly) by Erik Ray. For a complete reference to XML syntax and standards, see *XML in a Nutshell* (O'Reilly) by Elliotte Rusty Harold and W. Scott Means.

# Generating XML

Just as PHP can be used to generate dynamic HTML, it can also be used to generate dynamic XML. You can generate XML for other programs to make use of based on forms, database queries, or anything else you can do in PHP. One application for dynamic XML is *Rich Site Summary* (RSS), a file format for syndicating news sites. You can read an article's information from a database or from HTML files and emit an XML summary file based on that information.

Generating an XML document from a PHP script is simple. Simply change the MIME type of the document, using the `header()` function, to `"text/xml"`. To emit the `<?xml ... ?>` declaration without it being interpreted as a malformed PHP tag, simply `echo` the line from within PHP code:

```
echo '<?xml version="1.0" encoding="ISO-8859-1" ?>';
```

Example 12-1 generates an RSS document using PHP. An RSS file is an XML document containing several `channel` elements, each of which contains some news `item` elements. Each news `item` can have a title, a description, and a link to the article itself. More properties of an `item` are supported by RSS than Example 12-1 creates. Just as there are no special functions for generating HTML from PHP, there are no special functions for generating XML. You just `echo` it!

*Example 12-1. Generating an XML document*

```
<?php
header('Content-Type: text/xml');
echo "<?xml version=\"1.0\" encoding=\"ISO-8859-1\" ?>";
?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
 "http://my.netscape.com/publish/formats/rss-0.91.dtd">

<rss version="0.91">
 <channel>
 <?php
 // news items to produce RSS for
 $items = array(
 array(
 'title' => "Man Bites Dog",
 'link' => "http://www.example.com/dog.php",
 'desc' => "Ironic turnaround!"
 ),
 array(
 'title' => "Medical Breakthrough!",
 'link' => "http://www.example.com/doc.php",
 'desc' => "Doctors announced a cure for me."
 )
 );

 foreach($items as $item) {
 echo "<item>\n";
 echo " <title>{$item['title']}</title>\n";
 echo " <link>{$item['link']}</link>\n";
 echo " <description>{$item['desc']}</description>\n";
 echo " <language>en-us</language>\n";
 echo "</item>\n\n";
 } ?>
 </channel>
</rss>
```

This script generates output such as the following:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
 "http://my.netscape.com/publish/formats/rss-0.91.dtd">
<rss version="0.91">
 <channel>
<item>
 <title>Man Bites Dog</title>
 <link>http://www.example.com/dog.php</link>
 <description>Ironic turnaround!</description>
 <language>en-us</language>
</item>
```

```
<item>
 <title>Medical Breakthrough!</title>
 <link>http://www.example.com/doc.php</link>
 <description>Doctors announced a cure for me.</description>
 <language>en-us</language>
</item>
 </channel>
</rss>
```

# Parsing XML

Say you have a set of XML files, each containing information about a book, and you want to build an index showing the document title and its author for the collection. You need to parse the XML files to recognize the `title` and `author` elements and their contents. You could do this by hand with regular expressions and string functions such as `strtok()`, but it's a lot more complex than it seems. In addition, such methods are prone to breakage even with valid XML documents. The easiest and quickest solution is to use one of the XML parsers that ship with PHP.

PHP includes three XML parsers: one event-driven library based on the Expat C library, one DOM-based library, and one for parsing simple XML documents named, appropriately, SimpleXML.

The most commonly used parser is the event-based library, which lets you parse but not validate XML documents. This means you can find out which XML tags are present and what they surround, but you can't find out if they're the right XML tags in the right structure for this type of document. In practice, this isn't generally a big problem. PHP's event-based XML parser calls various handler functions you provide while it reads the document as it encounters certain *events*, such as the beginning or end of an element.

In the following sections, we discuss the handlers you can provide, the functions to set the handlers, and the events that trigger the calls to those handlers. We also provide sample functions for creating a parser to generate a map of the XML document in memory, tied together in a sample application that pretty-prints XML.

## Element Handlers

When the parser encounters the beginning or end of an element, it calls the start and end element handlers. You set the handlers through the `xml_set_element_handler()` function:

```
xml_set_element_handler(parser, start_element, end_element);
```

The *start_element* and *end_element* parameters are the names of the handler functions.

The start element handler is called when the XML parser encounters the beginning of an element:

```
startElementHandler(parser, element, &attributes);
```

The start element handler is passed three parameters: a reference to the XML parser calling the handler, the name of the element that was opened, and an array containing any attributes the parser encountered for the element. The `$attribute` array is passed by reference for speed.

Example 12-2 contains the code for a start element handler, `startElement()`. This handler simply prints the element name in bold and the attributes in gray.

*Example 12-2. Start element handler*

```
function startElement($parser, $name, $attributes) {
 $outputAttributes = array();

 if (count($attributes)) {
 foreach($attributes as $key => $value) {
 $outputAttributes[] = "<font color=\"gray\">{$key}=\"{$value}\"</font>";
 }
 }

 echo "&lt;<b>{$name}</b> " . join(' ', $outputAttributes) . '&gt;';
}
```

The end element handler is called when the parser encounters the end of an element:

```
endElementHandler(parser, element);
```

It takes two parameters: a reference to the XML parser calling the handler, and the name of the element that is closing.

Example 12-3 shows an end element handler that formats the element.

*Example 12-3. End element handler*

```
function endElement($parser, $name) {
 echo "&lt;<b>/{$name}</b>&gt;";
}
```

## Character Data Handler

All of the text between elements (*character data*, or *CDATA* in XML terminology) is handled by the character data handler. The handler you set with the `xml_set_character_data_handler()` function is called after each block of character data:

```
xml_set_character_data_handler(parser, handler);
```

The character data handler takes in a reference to the XML parser that triggered the handler and a string containing the character data itself:

```
characterDataHandler(parser, cdata);
```

Here's a simple character data handler that simply prints the data:

```
function characterData($parser, $data) {
 echo $data;
}
```

## Processing Instructions

Processing instructions are used in XML to embed scripts or other code into a document. PHP itself can be seen as a processing instruction and, with the <?php ... ?> tag style, follows the XML format for demarking the code. The XML parser calls the processing instruction handler when it encounters a processing instruction. Set the handler with the xml_set_processing_instruction_handler() function:

```
xml_set_processing_instruction_handler(parser, handler);
```

A processing instruction looks like:

```
<? target instructions ?>
```

The processing instruction handler takes in a reference to the XML parser that triggered the handler, the name of the target (for example, 'php'), and the processing instructions:

```
processingInstructionHandler(parser, target, instructions);
```

What you do with a processing instruction is up to you. One trick is to embed PHP code in an XML document and, as you parse that document, execute the PHP code with the eval() function. Example 12-4 does just that. Of course, you have to trust the documents you're processing if you include the eval() code in them. eval() will run any code given to it—even code that destroys files or mails passwords to a cracker. In practice, executing arbitrary code like this is extremely dangerous.

*Example 12-4. Processing instruction handler*

```
function processing_instruction($parser, $target, $code) {
 if ($target === 'php') {
 eval($code);
 }
}
```

## Entity Handlers

Entities in XML are placeholders. XML provides five standard entities (&, >, <, ", and '), but XML documents can define their own entities. Most

entity definitions do not trigger events, and the XML parser expands most entities in documents before calling the other handlers.

Two types of entities, external and unparsed, have special support in PHP's XML library. An *external* entity is one whose replacement text is identified by a filename or URL rather than explicitly given in the XML file. You can define a handler to be called for occurrences of external entities in character data, but it's up to you to parse the contents of the file or URL yourself if that's what you want.

An *unparsed* entity must be accompanied by a notation declaration, and while you can define handlers for declarations of unparsed entities and notations, occurrences of unparsed entities are deleted from the text before the character data handler is called.

#### External entities

External entity references allow XML documents to include other XML documents. Typically, an external entity reference handler opens the referenced file, parses the file, and includes the results in the current document. Set the handler with `xml_set_external_entity_ref_handler()`, which takes in a reference to the XML parser and the name of the handler function:

```
xml_set_external_entity_ref_handler(parser, handler);
```

The external entity reference handler takes five parameters: the parser triggering the handler, the entity's name, the base Uniform Resource Identifier (URI) for resolving the identifier of the entity (which is currently always empty), the system identifier (such as the filename), and the public identifier for the entity, as defined in the entity's declaration. For example:

```
externalEntityHandler(parser, entity, base, system, public);
```

If your external entity reference handler returns `false` (which it will if it returns no value), XML parsing stops with an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error. If it returns `true`, parsing continues.

Example 12-5 shows how you would parse externally referenced XML documents. Define two functions, `createParser()` and `parse()`, to do the actual work of creating and feeding the XML parser. You can use them both to parse the top-level document and any documents included via external references. Such functions are described in the section "Using the Parser". The external entity reference handler simply identifies the right file to send to those functions.

*Example 12-5. External entity reference handler*

```
function externalEntityReference($parser, $names, $base, $systemID, $publicID) {
 if ($systemID) {
```

```
 if (!list ($parser, $fp) = createParser($systemID)) {
 echo "Error opening external entity {$systemID}\n";

 return false;
 }

 return parse($parser, $fp);
 }

 return false;
}
```

### Unparsed entities

An unparsed entity declaration must be accompanied by a notation declaration:

```
<!DOCTYPE doc [
 <!NOTATION jpeg SYSTEM "image/jpeg">
 <!ENTITY logo SYSTEM "php-tiny.jpg" NDATA jpeg>
]>
```

Register a notation declaration handler with `xml_set_notation_decl_handler()`:

```
xml_set_notation_decl_handler(parser, handler);
```

The handler will be called with five parameters:

```
notationHandler(parser, notation, base, system, public);
```

The *base* parameter is the base URI for resolving the identifier of the notation (which is currently always empty). Either the *system* identifier or the *public* identifier for the notation will be set, but not both.

Use the `xml_set_unparsed_entity_decl_handler()` function to register an unparsed entity declaration:

```
xml_set_unparsed_entity_decl_handler(parser, handler);
```

The handler will be called with six parameters:

```
unparsedEntityHandler(parser, entity, base, system, public, notation);
```

The *notation* parameter identifies the notation declaration with which this unparsed entity is associated.

## Default Handler

For any other event, such as the XML declaration and the XML document type, the default handler is called. Call the `xml_set_default_handler()` function to set the default handler:

```
xml_set_default_handler(parser, handler);
```

The handler will be called with two parameters:

```
defaultHandler(parser, text);
```

The *text* parameter will have different values depending on the kind of event triggering the default handler. Example 12-6 just prints out the given string when the default handler is called.

*Example 12-6. Default handler*

```
function default($parser, $data) {
 echo "<font color=\"red\">XML: Default handler called with '{$data}'</font>\n";
}
```

## Options

The XML parser has several options you can set to control the source and target encodings and case folding. Use `xml_parser_set_option()` to set an option:

```
xml_parser_set_option(parser, option, value);
```

Similarly, use `xml_parser_get_option()` to interrogate a parser about its options:

```
$value = xml_parser_get_option(parser, option);
```

### Character encoding

The XML parser used by PHP supports Unicode data in a number of different character encodings. Internally, PHP's strings are always encoded in UTF-8, but documents parsed by the XML parser can be in ISO-8859-1, US-ASCII, or UTF-8. UTF-16 is not supported.

When creating an XML parser, you can give it an encoding format to use for the file to be parsed. If omitted, the source is assumed to be in ISO-8859-1. If a character outside the possible range in the source encoding is encountered, the XML parser will return an error and immediately stop processing the document.

The target encoding for the parser is the encoding in which the XML parser passes data to the handler functions; normally, this is the same as the source encoding. At any time during the XML parser's lifetime, the target encoding can be changed. The parser demotes any characters outside the target encoding's character range by replacing them with a question mark character (`?`).

Use the constant `XML_OPTION_TARGET_ENCODING` to get or set the encoding of the text passed to callbacks. Allowable values are `"ISO-8859-1"` (the default), `"US-ASCII"`, and `"UTF-8"`.

### Case folding

By default, element and attribute names in XML documents are converted to all uppercase. You can turn off this behavior (and get case-sensitive element names) by

setting the XML_OPTION_CASE_FOLDING option to false with the xml_parser _set_option() function:

```
xml_parser_set_option(XML_OPTION_CASE_FOLDING, false);
```

### Skipping whitespace-only

Set the XML_OPTION_SKIP_WHITE option to ignore values consisting entirely of whitespace characters.

```
xml_parser_set_option(XML_OPTION_SKIP_WHITE, true);
```

### Truncating tag names

When creating a parser, you can optionally have it truncate characters at the start of each tag name. To truncate the start of each tag by a number of characters, provide that value in the XML_OPTION_SKIP_TAGSTART option:

```
xml_parser_set_option(XML_OPTION_SKIP_TAGSTART, 4);
// <xsl:name> truncates to "name"
```

In this case, the tag name will be truncated by four characters.

## Using the Parser

To use the XML parser, create a parser with xml_parser_create(), set handlers and options on the parser, and then hand chunks of data to the parser with the xml_parse() function until either the data runs out or the parser returns an error. Once the processing is complete, free the parser by calling xml_parser_free().

The xml_parser_create() function returns an XML parser:

```
$parser = xml_parser_create([encoding]);
```

The optional *encoding* parameter specifies the text encoding ("ISO-8859-1", "US-ASCII", or "UTF-8") of the file being parsed.

The xml_parse() function returns true if the parse was successful and false if it was not:

```
$success = xml_parse(parser, data[, final ]);
```

The *data* argument is a string of XML to process. The optional *final* parameter should be true for the last piece of data to be parsed.

To easily deal with nested documents, write functions that create the parser and set its options and handlers for you. This puts the options and handler settings in one place, rather than duplicating them in the external entity reference handler. Example 12-7 shows such a function.

*Example 12-7. Creating a parser*

```
function createParser($filename) {
 $fh = fopen($filename, 'r');
 $parser = xml_parser_create();

 xml_set_element_handler($parser, "startElement", "endElement");
 xml_set_character_data_handler($parser, "characterData");
 xml_set_processing_instruction_handler($parser, "processingInstruction");
 xml_set_default_handler($parser, "default");

 return array($parser, $fh);
}

function parse($parser, $fh) {
 $blockSize = 4 * 1024; // read in 4 KB chunks

 while ($data = fread($fh, $blockSize)) {
 if (!xml_parse($parser, $data, feof($fh))) {
 // an error occurred; tell the user where
 echo 'Parse error: ' . xml_error_string($parser) . " at line " .
 xml_get_current_line_number($parser);

 return false;
 }
 }

 return true;
}

if (list ($parser, $fh) = createParser("test.xml")) {
 parse($parser, $fh);
 fclose($fh);

 xml_parser_free($parser);
}
```

## Errors

The `xml_parse()` function returns `true` if the parse completed successfully, and `false` if there was an error. If something did go wrong, use `xml_get_error_code()` to fetch a code identifying the error:

```
$error = xml_get_error_code($parser);
```

The error code corresponds to one of these error constants:

```
XML_ERROR_NONE
XML_ERROR_NO_MEMORY
XML_ERROR_SYNTAX
XML_ERROR_NO_ELEMENTS
XML_ERROR_INVALID_TOKEN
```

```
XML_ERROR_UNCLOSED_TOKEN
XML_ERROR_PARTIAL_CHAR
XML_ERROR_TAG_MISMATCH
XML_ERROR_DUPLICATE_ATTRIBUTE
XML_ERROR_JUNK_AFTER_DOC_ELEMENT
XML_ERROR_PARAM_ENTITY_REF
XML_ERROR_UNDEFINED_ENTITY
XML_ERROR_RECURSIVE_ENTITY_REF
XML_ERROR_ASYNC_ENTITY
XML_ERROR_BAD_CHAR_REF
XML_ERROR_BINARY_ENTITY_REF
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
XML_ERROR_MISPLACED_XML_PI
XML_ERROR_UNKNOWN_ENCODING
XML_ERROR_INCORRECT_ENCODING
XML_ERROR_UNCLOSED_CDATA_SECTION
XML_ERROR_EXTERNAL_ENTITY_HANDLING
```

The constants generally aren't very useful. Use `xml_error_string()` to turn an error code into a string that you can use when you report the error:

```
$message = xml_error_string(code);
```

For example:

```
$error = xml_get_error_code($parser);

if ($error != XML_ERROR_NONE) {
 die(xml_error_string($error));
}
```

## Methods as Handlers

Because functions and variables are global in PHP, any component of an application that requires several functions and variables is a candidate for object-oriented design. XML parsing typically requires you to keep track of where you are in the parsing (e.g., "just saw an opening `title` element, so keep track of character data until you see a closing `title` element") with variables, and of course you must write several handler functions to manipulate the state and actually do something. Wrapping these functions and variables into a class enables you to keep them separate from the rest of your program and easily reuse the functionality later.

Use the `xml_set_object()` function to register an object with a parser. After you do so, the XML parser looks for the handlers as methods on that object, rather than as global functions:

```
xml_set_object(object);
```

## Sample Parsing Application

Let's develop a program to parse an XML file and display different types of information from it. The XML file given in Example 12-8 contains information on a set of books.

*Example 12-8. books.xml file*

```
<?xml version="1.0" ?>
<library>
 <book>
 <title>Programming PHP</title>
 <authors>
 <author>Rasmus Lerdorf</author>
 <author>Kevin Tatroe</author>
 <author>Peter MacIntyre</author>
 </authors>
 <isbn>1-56592-610-2</isbn>
 <comment>A great book!</comment>
 </book>
 <book>
 <title>PHP Pocket Reference</title>
 <authors>
 <author>Rasmus Lerdorf</author>
 </authors>
 <isbn>1-56592-769-9</isbn>
 <comment>It really does fit in your pocket</comment>
 </book>
 <book>
 <title>Perl Cookbook</title>
 <authors>
 <author>Tom Christiansen</author>
 <author whereabouts="fishing">Nathan Torkington</author>
 </authors>
 <isbn>1-56592-243-3</isbn>
 <comment>Hundreds of useful techniques, most
 applicable to PHP as well as Perl</comment>
 </book>
</library>
```

The PHP application parses the file and presents the user with a list of books, showing just the titles and authors. This menu is shown in Figure 12-1. The titles are links to a page showing the complete information for a book. A page of detailed information for *Programming PHP* is shown in Figure 12-2.

We define a class, `BookList`, whose constructor parses the XML file and builds a list of records. There are two methods on a `BookList` that generate output from that list of records. The `showMenu()` method generates the book menu, and the `showBook()` method displays detailed information on a particular book.

Parsing the file involves keeping track of the record, which element we're in, and which elements correspond to records (`book`) and fields (`title`, `author`, `isbn`, and `comment`). The `$record` property holds the current record as it's being built, and `$currentField` holds the name of the field we're currently processing (e.g., `title`). The `$records` property is an array of all the records we've read so far.

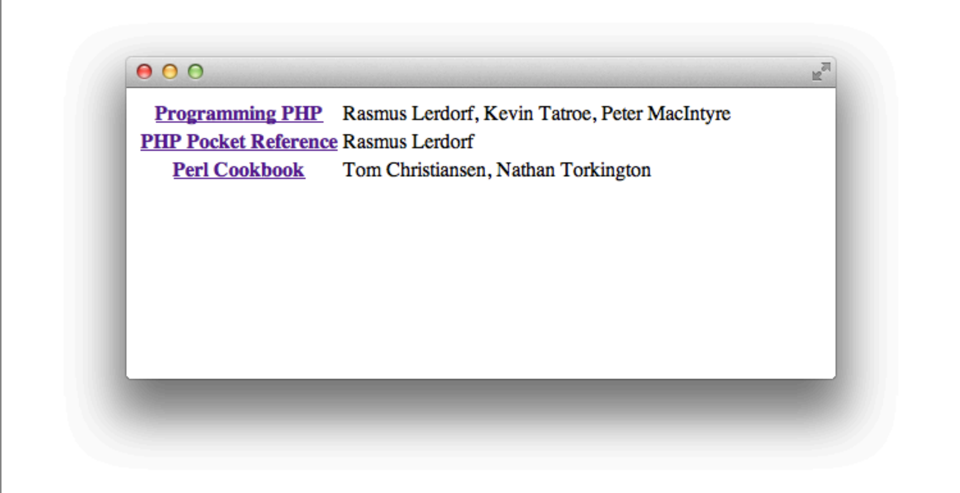



*Figure 12-1. Book menu*

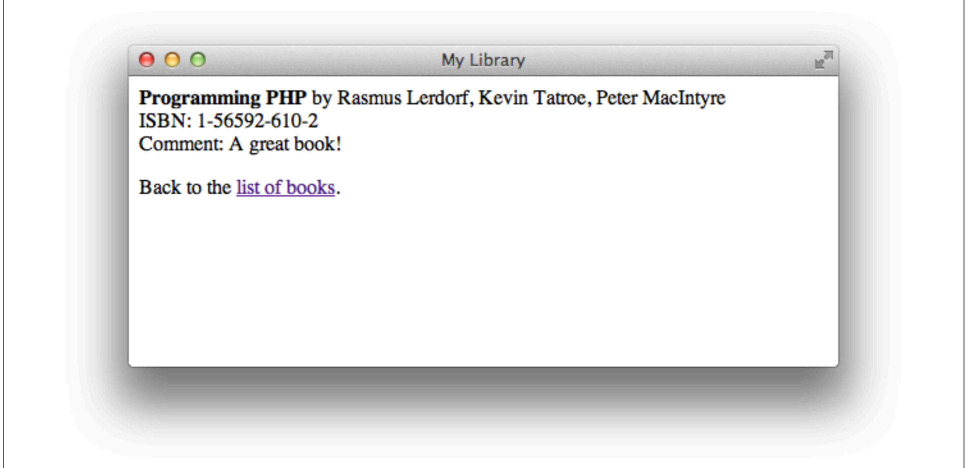



*Figure 12-2. Book details*

Two associative arrays, `$fieldType` and `$endsRecord`, tell us which elements correspond to fields in a record and which closing element signals the end of a record. Values in `$fieldType` are either `1` or `2`, corresponding to a simple scalar field (e.g.,

title) or an array of values (e.g., author), respectively. We initialize those arrays in the constructor.

The handlers themselves are fairly straightforward. When we see the start of an element, we work out whether it corresponds to a field we're interested in. If it is, we set the $currentField property to be that field name so when we see the character data (e.g., the title of the book), we know which field it's the value for. When we get character data, we add it to the appropriate field of the current record if $currentField says we're in a field. When we see the end of an element, we check to see if it's the end of a record; if so, we add the current record to the array of completed records.

One PHP script, given in Example 12-9, handles both the book menu and book details pages. The entries in the book menu link back to the menu URL with a GET parameter identifying the ISBN of the book to display.

*Example 12-9. bookparse.php*

```
<html>
 <head>
 <title>My Library</title>
 </head>

 <body>
 <?php
 class BookList {
 const FIELD_TYPE_SINGLE = 1;
 const FIELD_TYPE_ARRAY = 2;
 const FIELD_TYPE_CONTAINER = 3;

 var $parser;
 var $record;
 var $currentField = '';
 var $fieldType;
 var $endsRecord;
 var $records;

 function __construct($filename) {
 $this->parser = xml_parser_create();
 xml_set_object($this->parser, $this);
 xml_set_element_handler($this->parser, "elementStarted", "elementEnded");
 xml_set_character_data_handler($this->parser, "handleCdata");

 $this->fieldType = array(
 'title' => self::FIELD_TYPE_SINGLE,
 'author' => self::FIELD_TYPE_ARRAY,
 'isbn' => self::FIELD_TYPE_SINGLE,
 'comment' => self::FIELD_TYPE_SINGLE,
 );
```

```
$this->endsRecord = array('book' => true);

$xml = join('', file($filename));
xml_parse($this->parser, $xml);

xml_parser_free($this->parser);
}

function elementStarted($parser, $element, &$attributes) {
$element = strtolower($element);

if ($this->fieldType[$element] != 0) {
$this->currentField = $element;
}
else {
$this->currentField = '';
}
}

function elementEnded($parser, $element) {
$element = strtolower($element);

if ($this->endsRecord[$element]) {
$this->records[] = $this->record;
$this->record = array();
}

$this->currentField = '';
}

function handleCdata($parser, $text) {
if ($this->fieldType[$this->currentField] == self::FIELD_TYPE_SINGLE) {
$this->record[$this->currentField] .= $text;
}
else if ($this->fieldType[$this->currentField] == self::FIELD_TYPE_ARRAY) {
$this->record[$this->currentField][] = $text;
}
}

function showMenu() {
echo "<table>\n";

foreach ($this->records as $book) {
echo "<tr>";
echo "<th><a href=\"{$_SERVER['PHP_SELF']}?isbn={$book['isbn']}\">";
echo "{$book['title']}</a></th>";
echo "<td>" . join(', ', $book['author']) . "</td>\n";
echo "</tr>\n";
}

echo "</table>\n";
}
```

```
function showBook($isbn) {
foreach ($this->records as $book) {
if ($book['isbn'] !== $isbn) {
continue;
}

echo "<p><b>{$book['title']}</b> by " . join(', ', $book['author']) . "<br />";
echo "ISBN: {$book['isbn']}<br />";
echo "Comment: {$book['comment']}</p>\n";
}

echo "<p>Back to the <a href=\"{$_SERVER['PHP_SELF']}\">list of books</a>.</p>";
}
}

$library = new BookList("books.xml");

if (isset($_GET['isbn'])) {
// return info on one book
$library->showBook($_GET['isbn']);
}
else {
// show menu of books
$library->showMenu();
} ?>
</body>
</html>
```

## Parsing XML with the DOM

The DOM parser provided in PHP is much simpler to use, but what you take out in complexity comes back in memory usage—in spades. Instead of firing events and allowing you to handle the document as it is being parsed, the DOM parser takes an XML document and returns an entire tree of nodes and elements:

```
$parser = new DOMDocument();
$parser->load("books.xml");
processNodes($parser->documentElement);

function processNodes($node) {
 foreach ($node->childNodes as $child) {
 if ($child->nodeType == XML_TEXT_NODE) {
 echo $child->nodeValue;
 }
 else if ($child->nodeType == XML_ELEMENT_NODE) {
 processNodes($child);
 }
 }
}
```

## Parsing XML with SimpleXML

If you're consuming very simple XML documents, you might consider the third library provided by PHP, SimpleXML. SimpleXML doesn't have the ability to generate documents as the DOM extension does, and isn't as flexible or memory-efficient as the event-driven extension, but it makes it very easy to read, parse, and traverse simple XML documents.

SimpleXML takes a file, string, or DOM document (produced using the DOM extension) and generates an object. Properties on that object are arrays providing access to elements in each node. With those arrays, you can access elements using numeric indices and attributes using non-numeric indices. Finally, you can use string conversion on any value you retrieve to get the text value of the item.

For example, we could display all the titles of the books in our *books.xml* document using:

```
$document = simplexml_load_file("books.xml");

foreach ($document->book as $book) {
 echo $book->title . "\r\n";
}
```

Using the `children()` method on the object, you can iterate over the child nodes of a given node; likewise, you can use the `attributes()` method on the object to iterate over the attributes of the node:

```
$document = simplexml_load_file("books.xml");

foreach ($document->book as $node) {
 foreach ($node->attributes() as $attribute) {
 echo "{$attribute}\n";
 }
}
```

Finally, using the `asXml()` method on the object, you can retrieve the XML of the document in XML format. This lets you change values in your document and write it back out to disk easily:

```
$document = simplexml_load_file("books.xml");

foreach ($document->children() as $book) {
 $book->title = "New Title";
}

file_put_contents("books.xml", $document->asXml());
```

## Transforming XML with XSLT

Extensible Stylesheet Language Transformations (XSLT) is a language for transforming XML documents into different XML, HTML, or any other format. For example, many websites offer several formats of their content—HTML, printable HTML, and WML (Wireless Markup Language) are common. The easiest way to present these multiple views of the same information is to maintain one form of the content in XML and use XSLT to produce the HTML, printable HTML, and WML.

PHP's XSLT extension uses the Libxslt C library to provide XSLT support.

Three documents are involved in an XSLT transformation: the original XML document, the XSLT document containing transformation rules, and the resulting document. The final document doesn't have to be in XML; in fact, it's common to use XSLT to generate HTML from XML. To do an XSLT transformation in PHP, you create an XSLT processor, give it some input to transform, and then destroy the processor.

Create a processor by creating a new `XsltProcessor` object:

```
$processor = new XsltProcessor;
```

Parse the XML and XSL files into DOM objects:

```
$xml = new DomDocument;
$xml->load($filename);

$xsl = new DomDocument;
$xsl->load($filename);
```

Attach the XML rules to the object:

```
$processor->importStyleSheet($xsl);
```

Process a file with the `transformToDoc()`, `transformToUri()`, or `transformToXml()` methods:

```
$result = $processor->transformToXml($xml);
```

Each takes the DOM object representing the XML document as a parameter.

Example 12-10 is the XML document we're going to transform. It is in a similar format to many of the news documents you find on the web.

*Example 12-10. XML document*

```
<?xml version="1.0" ?>

<news xmlns:news="http://slashdot.org/backslash.dtd">
 <story>
 <title>O'Reilly Publishes Programming PHP</title>
```

```
<url>http://example.org/article.php?id=20020430/458566</url>
<time>2002-04-30 09:04:23</time>
<author>Rasmus and some others</author>
</story>

<story>
<title>Transforming XML with PHP Simplified</title>
<url>http://example.org/article.php?id=20020430/458566</url>
<time>2002-04-30 09:04:23</time>
<author>k.tatroe</author>
<teaser>Check it out</teaser>
</story>
</news>
```

Example 12-11 is the XSL document we'll use to transform the XML document into HTML. Each `xsl:template` element contains a rule for dealing with part of the input document.

*Example 12-11. News XSL transform*

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes" encoding="utf-8" />

<xsl:template match="/news">
 <html>
 <head>
 <title>Current Stories</title>
 </head>
 <body bgcolor="white" >
 <xsl:call-template name="stories"/>
 </body>
 </html>
</xsl:template>

<xsl:template name="stories">
 <xsl:for-each select="story">
 <h1><xsl:value-of select="title" /></h1>

 <p>
 <xsl:value-of select="author"/> (<xsl:value-of select="time"/>)<br />
 <xsl:value-of select="teaser"/>
 [ <a href="{url}">More</a> ]
 </p>

 <hr />
 </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Example 12-12 is the very small amount of code necessary to transform the XML document into an HTML document using the XSL stylesheet. We create a processor, run the files through it, and print the result.

*Example 12-12. XSL transformation from files*

```
<?php
$processor = new XsltProcessor;

$xsl = new DOMDocument;
$xsl->load("rules.xsl");
$processor->importStyleSheet($xsl);

$xml = new DomDocument;
$xml->load("feed.xml");
$result = $processor->transformToXml($xml);

echo "<pre>{$result}</pre>";
```

Although it doesn't specifically discuss PHP, Doug Tidwell's book *XSLT* (O'Reilly) provides a detailed guide to the syntax of XSLT stylesheets.

## What's Next

While XML remains a major format for sharing data, a simplified version of JavaScript data encapsulation, known as JSON, has rapidly become the de facto standard for simple, readable, and terse sharing of web service responses and other data. That's the subject we'll turn to in the next chapter.

CHAPTER 13

# JSON

Similar to XML, JavaScript Object Notation (JSON) was designed as a standardized data-interchange format. However, unlike XML, JSON is extremely lightweight and human-readable. While it takes many syntax cues from JavaScript, JSON is designed to be language-independent.

JSON is built on two structures: collections of name/value pairs called *objects* (equivalent to PHP's associative arrays) and ordered lists of values called *arrays* (equivalent to PHP's indexed arrays). Each value can be one of a number of types: an object, an array, a string, a number, the Boolean values `TRUE` or `FALSE`, or `NULL` (indicating a lack of a value).

## Using JSON

The *json* extension, included by default in PHP installations, natively supports converting data to JSON format from PHP variables and vice versa.

To get a JSON representation of a PHP variable, use `json_encode()`:

```
$data = array(1, 2, "three");
$jsonData = json_encode($data);
echo $jsonData;
[1, 2, "three"]
```

Similarly, if you have a string containing JSON data, you can turn it into a PHP variable using `json_decode()`:

```
$jsonData = "[1, 2, [3, 4], \"five\"]";
$data = json_decode($jsonData);
print_r($data);
Array( [0] => 1 [1] => 2 [2] => Array( [0] => 3 [1] => 4 ) [3] => five)
```

If the string is invalid JSON, or if the string is not encoded in UTF-8 format, a single `NULL` value is returned instead.

The value types in JSON are converted to PHP equivalents as follows:

`object`
: An associative array containing the object's key-value pairs. Each value is converted into its PHP equivalent as well.

`array`
: An indexed array containing the contained values, each converted into its PHP equivalent as well.

`string`
: Converts directly into a PHP string.

`number`
: Returns a number. If the value is too large to be represented by PHP's number value, it returns `NULL`, unless `json_decode()` is called with the `JSON_BIGINT_AS_STRING` (in which case, a string is returned).

`boolean`
: The Boolean `true` value is converted to `TRUE`; the Boolean `false` value is converted to `FALSE`.

`null`
: The `null` value, and any value that cannot be decoded, is converted to `NULL`.

# Serializing PHP Objects

Despite the similar names, there is no direct translation between PHP objects and JSON objects—what JSON calls an "object" is really an associative array. To convert JSON data into an instance of a PHP object class, you must write code to do so based on the format returned by the API.

However, the `JsonSerializable` interface allows you to convert objects *into* JSON data however you like. If an object class does not implement the interface, `json_encode()` simply creates a JSON object containing keys and values corresponding to the object's data members.

Otherwise, `json_encode()` calls the `jsonSerialize()` method on the class and uses that to serialize the object's data.

Example 13-1 adds the `JsonSerializable` interface to the `Book` and `Author` classes.

*Example 13-1. Book and Author JSON serialization*

```
class Book implements JsonSerializable {
 public $id;
 public $name;
 public $edition;

 public function __construct($id) {
 $this->id = $id;
 }

 public function jsonSerialize() {
 $data = array(
 'id' => $this->id,
 'name' => $this->name,
 'edition' => $this->edition,
 );

 return $data;
 }
}

class Author implements JsonSerializable {
 public $id;
 public $name;
 public $books = array();

 public function __construct($id) {
 $this->id = $id;
 }

 public function jsonSerialize() {
 $data = array(
 'id' => $this->id,
 'name' => $this->name,
 'books' => $this->books,
 );

 return $data;
 }
}
```

Creating a PHP object from JSON data requires you to write code to perform the translation.

Example 13-2 shows a class implementing factory-style transformation of JSON data into `Book` and `Author` instances into PHP objects.

*Example 13-2. Book and Author JSON serialization by factory*

```
class ResourceFactory {
 static public function authorFromJSON($jsonData) {
 $author = new Author($jsonData['id']);
 $author->name = $jsonData['name'];

 foreach ($jsonData['books'] as $bookIdentifier) {
 $this->books[] = new Book($bookIdentifier);
 }

 return $author;
 }

 static public function bookFromJSON($jsonData) {
 $book = new Book($jsonData['id']);
 $book->name = $jsonData['name'];
 $book->edition = (int) $jsonData['edition'];

 return $book;
 }
}
```

## Options

The JSON parser functions have several options you can set to control the conversion process.

For `json_decode()`, the most common options include:

`JSON_BIGINT_AS_STRING`
: When decoding a number that is too large to be represented as a PHP number type, returns that value as a string instead.

`JSON_OBJECT_AS_ARRAY`
: Decodes JSON objects as PHP arrays.

For `json_encode()`, the most common options include:

`JSON_FORCE_OBJECT`
: Encodes indexed arrays from the PHP values as JSON objects instead of JSON arrays.

`JSON_NUMERIC_CHECK`
: Encodes strings that represent number values as JSON numbers, rather than as JSON strings. In practice, you're better off converting manually, so you're aware of what the types are.

`JSON_PRETTY_PRINT`
: Uses whitespace to format the returned data to something more human-readable. Not strictly necessary, but makes debugging simpler.

Finally, the following options can be used for both `json_encode()` and `json_decode()`:

`JSON_INVALID_UTF8_IGNORE`
: Ignores invalid UTF-8 characters. If `JSON_INVALID_UTF8_SUBSTITUTE` is also set, replaces them; otherwise, drops them in the resulting string.

`JSON_INVALID_UTF8_SUBSTITUTE`
: Replaces invalid UTF-8 characters with `\0xfffd` (the Unicode character `'REPLACEMENT CHARACTER'`).

`JSON_THROW_ON_ERROR`
: Throws an error instead of populating the global last error state when an error occurs.

## What's Next

When you're writing PHP, one of the most important things to consider is the security of your code, from how well the code can absorb and deflect attacks to how you keep your own and your users' data safe. The next chapter provides guidance and best practices to help you avert security-related disasters.

# CHAPTER 14
# Security

PHP is a flexible language with hooks into just about every API offered on the machines on which it runs. Because it was designed to be a forms-processing language for HTML pages, PHP makes it easy to use form data sent to a script. Convenience is a double-edged sword, however. The very features that allow you to quickly write programs in PHP can open doors for those who would break into your systems.

PHP itself is neither secure nor insecure. The security of your web applications is entirely determined by the code you write. For example, if a script opens a file whose name is passed to the script as a form parameter, that script could be given a remote URL, an absolute pathname, or even a relative path, allowing it to open a file outside the site's document root. This could expose your password file or other sensitive information.

Web application security is still a relatively young and evolving discipline. A single chapter on security cannot sufficiently prepare you for the onslaught of attacks your applications are sure to receive. This chapter takes a pragmatic approach and covers a distilled selection of topics related to security, including how to protect your applications from the most common and dangerous attacks. The chapter concludes with a list of further resources as well as a brief recap with a few additional tips.

## Safeguards

One of the most fundamental things you need to understand when developing a secure site is that all information not generated within the application itself is potentially tainted, or at least suspect. This includes data from forms, files, and databases. There should always be protections or safeguards in place.

## Filtering Input

When data is described as being tainted, this doesn't necessarily mean it's malicious. It means it *might be* malicious. You can't trust the source, so you should inspect it to make sure it's valid. This inspection process is called *filtering*, and you only want to allow valid data to enter your application.

There are a few best practices for the filtering process:

- Use a whitelist approach. This means you err on the side of caution and assume data is invalid unless you can prove it to be valid.
- Never correct invalid data. History has proven that attempts to correct invalid data often result in security vulnerabilities due to errors.
- Use a naming convention to help distinguish between filtered and tainted data. Filtering is useless if you can't reliably determine whether something has been filtered.

In order to solidify these concepts, consider a simple HTML form allowing a user to select among three colors:

```
<form action="process.php" method="POST">
 <p>Please select a color:

 <select name="color">
 <option value="red">red</option>
 <option value="green">green</option>
 <option value="blue">blue</option>
 </select>

 <input type="submit" /></p>
</form>
```

It's easy to appreciate the desire to trust `$_POST['color']` in *process.php*. After all, the form seemingly restricts what a user can enter. However, experienced developers know that HTTP requests have no restriction on the fields they contain—client-side validation is never sufficient by itself. There are numerous ways malicious data can be sent to your application, and your only defense is to trust nothing and filter your input:

```
$clean = array();

switch($_POST['color']) {
 case 'red':
 case 'green':
 case 'blue':
 $clean['color'] = $_POST['color'];
 break;
```

```
  default:
  /* ERROR */
  break;
}
```

This example demonstrates a simple naming convention. You initialize an array called `$clean`. For each input field, validate the input and store the validated input in the array. This reduces the likelihood of tainted data being mistaken for filtered data, because you should always err on the side of caution and consider everything not stored in this array to be tainted.

Your filtering logic depends entirely upon the type of data you're inspecting, and the more restrictive you can be, the better. For example, consider a registration form that asks the user to provide a desired username. Clearly, there are many possible usernames, so the previous example doesn't help. In these cases, the best approach is to filter based on format. If you want to require a username to be alphanumeric (consisting of only alphabetic and numeric characters), your filtering logic can enforce this:

```
$clean = array();

if (ctype_alnum($_POST['username'])) {
 $clean['username'] = $_POST['username'];
}
else {
 /* ERROR */
}
```

Of course, this doesn't ensure any particular length. Use `mb_strlen()` to inspect a string's length and enforce a minimum and maximum:

```
$clean = array();

$length = mb_strlen($_POST['username']);

if (ctype_alnum($_POST['username']) && ($length > 0) && ($length <= 32)) {
 $clean['username'] = $_POST['username'];
}
else {
 /* ERROR */
}
```

Frequently, the characters you want to allow don't all belong to a single group (such as alphanumeric), and this is where regular expressions can help. For example, consider the following filtering logic for a last name:

```
$clean = array();

if (preg_match("/[^A-Za-z \'\-]/", $_POST['last_name'])) {
 /* ERROR */
}
```

```
else {
 $clean['last_name'] = $_POST['last_name'];
}
```

This filter allows only alphabetic characters, spaces, hyphens, and single quotes (apostrophes), and it uses a whitelist approach as described earlier. In this case, the whitelist is the list of valid characters.

In general, filtering is a process that ensures the integrity of your data. But while many web application security vulnerabilities can be prevented by filtering, most are due to a failure to escape data, and neither safeguard is a substitute for the other.

## Escaping Output Data

*Escaping* is a technique that preserves data as it enters another context. PHP is frequently used as a bridge between disparate data sources, and when you send data to a remote source, it's your responsibility to prepare it properly so that it's not misinterpreted.

For example, `O'Reilly` is represented as `O\'Reilly` when used in an SQL query to be sent to a MySQL database. The backslash preserves the single quote (apostrophe) in the context of the SQL query. The single quote is part of the data, not part of the query, and the escaping guarantees this interpretation.

The two predominant remote sources to which PHP applications send data are HTTP clients (web browsers) that interpret HTML, JavaScript, and other client-side technologies, and databases that interpret SQL. For the former, PHP provides `htmlentities()`:

```
$html = array();
$html['username'] = htmlentities($clean['username'], ENT_QUOTES, 'UTF-8');

echo "<p>Welcome back, {$html['username']}.</p>";
```

This example demonstrates the use of another naming convention. The `$html` array is similar to the `$clean` array, except that its purpose is to hold data that is safe to be used in the context of HTML.

URLs are sometimes embedded in HTML as links:

```
<a href="http://host/script.php?var={$value}">Click Here</a>
```

In this particular example, `$value` exists within nested contexts. It's within the query string of a URL that is embedded in HTML as a link. Because it's alphabetic in this case, it's safe to be used in both contexts. However, when the value of `$var` cannot be guaranteed to be safe in these contexts, it must be escaped twice:

```
$url = array(
 'value' => urlencode($value),
);
```

```
$link = "http://host/script.php?var={$url['value']}";

$html = array(
 'link' => htmlentities($link, ENT_QUOTES, "UTF-8"),
);

echo "<a href=\"{$html['link']}\">Click Here</a>";
```

This ensures that the link is safe to be used in the context of HTML, and when it is used as a URL (such as when the user clicks the link), the URL encoding ensures that the value of `$var` is preserved.

For most databases, there is a native escaping function specific to the database. For example, the MySQL extension provides `mysqli_real_escape_string()`:

```
$mysql = array(
 'username' => mysqli_real_escape_string($clean['username']),
);

$sql = "SELECT * FROM profile
 WHERE username = '{$mysql['username']}'";

$result = mysql_query($sql);
```

An even safer alternative is to use a database abstraction library that handles the escaping for you. The following illustrates this concept with `PEAR::DB`:

```
$sql = "INSERT INTO users (last_name) VALUES (?)";

$db->query($sql, array($clean['last_name']));
```

Although this is not a complete example, it highlights the use of a placeholder (the question mark) in the SQL query. `PEAR::DB` properly quotes and escapes the data according to the requirements of your database. Take a look at Chapter 9 for more in-depth coverage of placeholder techniques.

A more complete output-escaping solution would include context-aware escaping for HTML elements, HTML attributes, JavaScript, CSS, and URL content, and would do so in a Unicode-safe manner. Example 14-1 shows a sample class for escaping output in a variety of contexts, based on the content-escaping rules defined by the Open Web Application Security Project.

*Example 14-1. Escaping output for multiple contexts*

```
class Encoder
{
 const ENCODE_STYLE_HTML = 0;
 const ENCODE_STYLE_JAVASCRIPT = 1;
 const ENCODE_STYLE_CSS = 2;
 const ENCODE_STYLE_URL = 3;
```

```
const ENCODE_STYLE_URL_SPECIAL = 4;

private static $URL_UNRESERVED_CHARS =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcedfghijklmnopqrstuvwxyz-_.~';

public function encodeForHTML($value) {
$value = str_replace('&', '&amp;', $value);
$value = str_replace('<', '&lt;', $value);
$value = str_replace('>', '&gt;', $value);
$value = str_replace('"', '&quot;', $value);
$value = str_replace('\'', '&#x27;', $value); // &apos; is not recommended
$value = str_replace('/', '&#x2F;', $value); // forward slash can help end
HTML entity

return $value;
}

public function encodeForHTMLAttribute($value) {
return $this->_encodeString($value);
}

public function encodeForJavascript($value) {
return $this->_encodeString($value, self::ENCODE_STYLE_JAVASCRIPT);
}

public function encodeForURL($value) {
return $this->_encodeString($value, self::ENCODE_STYLE_URL_SPECIAL);
}

public function encodeForCSS($value) {
return $this->_encodeString($value, self::ENCODE_STYLE_CSS);
}

/**
* Encodes any special characters in the path portion of the URL. Does not
* modify the forward slash used to denote directories. If your directory
* names contain slashes (rare), use the plain urlencode on each directory
* component and then join them together with a forward slash.
*
* Based on http://en.wikipedia.org/wiki/Percent-encoding and
* http://tools.ietf.org/html/rfc3986
*/
public function encodeURLPath($value) {
$length = mb_strlen($value);

if ($length == 0) {
return $value;
}

$output = '';

for ($i = 0; $i < $length; $i++) {
```

```
$char = mb_substr($value, $i, 1);

if ($char == '/') {
// Slashes are allowed in paths.
$output .= $char;
}
else if (mb_strpos(self::$URL_UNRESERVED_CHARS, $char) == false) {
// It's not in the unreserved list so it needs to be encoded.
$output .= $this->_encodeCharacter($char, self::ENCODE_STYLE_URL);
}
else {
// It's in the unreserved list so let it through.
$output .= $char;
}
}

return $output;
}

private function _encodeString($value, $style = self::ENCODE_STYLE_HTML) {
if (mb_strlen($value) == 0) {
return $value;
}

$characters = preg_split('/(?<!^)(?!$)/u', $value);
$output = '';

foreach ($characters as $c) {
$output .= $this->_encodeCharacter($c, $style);
}

return $output;
}

private function _encodeCharacter($c, $style = self::ENCODE_STYLE_HTML) {
if (ctype_alnum($c)) {
return $c;
}

if (($style === self::ENCODE_STYLE_URL_SPECIAL) && ($c == '/' || $c == ':')) {
return $c;
}

$charCode = $this->_unicodeOrdinal($c);

$prefixes = array(
self::ENCODE_STYLE_HTML => array('&#x', '&#x'),
self::ENCODE_STYLE_JAVASCRIPT => array('\\x', '\\u'),
self::ENCODE_STYLE_CSS => array('\\', '\\'),
self::ENCODE_STYLE_URL => array('%', '%'),
self::ENCODE_STYLE_URL_SPECIAL => array('%', '%'),
);
```

```
$suffixes = array(
self::ENCODE_STYLE_HTML => ';',
self::ENCODE_STYLE_JAVASCRIPT => '',
self::ENCODE_STYLE_CSS => '',
self::ENCODE_STYLE_URL => '',
self::ENCODE_STYLE_URL_SPECIAL => '',
);

// if ASCII, encode with \\xHH
if ($charCode < 256) {
$prefix = $prefixes[$style][0];
$suffix = $suffixes[$style];

return $prefix . str_pad(strtoupper(dechex($charCode)), 2, '0') . $suffix;
}

// otherwise encode with \\uHHHH
$prefix = $prefixes[$style][1];
$suffix = $suffixes[$style];

return $prefix . str_pad(strtoupper(dechex($charCode)), 4, '0') . $suffix;
}

private function _unicodeOrdinal($u) {
$c = mb_convert_encoding($u, 'UCS-2LE', 'UTF-8');
$c1 = ord(substr($c, 0, 1));
$c2 = ord(substr($c, 1, 1));

return $c2 * 256 + $c1;
}
}
```

# Security Vulnerabilities

Now that we've explored the two primary safeguarding approaches, let's turn to some of the common security vulnerabilities they seek to address.

## Cross-Site Scripting

Cross-site scripting (XSS) has become the most common web application security vulnerability, and with the rising popularity of Ajax technologies, XSS attacks are likely to become more advanced and to occur more frequently.

The term *cross-site scripting* derives from an old exploit and is no longer very descriptive or accurate for most modern attacks, and this has caused some confusion. Simply put, your code is vulnerable whenever you output data not properly escaped to the output's context. For example:

```
echo $_POST['username'];
```

This is an extreme example, because `$_POST` is obviously neither filtered nor escaped, but it demonstrates the vulnerability.

XSS attacks are limited to only what is possible with client-side technologies. Historically, XSS has been used to capture a victim's cookies by taking advantage of the fact that `document.cookie` contains this information.

In order to prevent XSS, you simply need to properly escape your output for the output context:

```
$html = array(
 'username' => htmlentities($_POST['username'], ENT_QUOTES, "UTF-8"),
);

echo $html['username'];
```

You should also always filter your input, which can offer a redundant safeguard in some cases (implementing redundant safeguards adheres to a security principle known as *Defense in Depth*). For example, if you inspect a username to ensure that it's alphabetic and also only output the filtered username, no XSS vulnerability exists. Just be sure that you don't depend upon filtering as your primary safeguard against XSS, because it doesn't address the root cause of the problem.

## SQL Injection

The second most common web application vulnerability is SQL injection, an attack very similar to XSS. The difference is that SQL injection vulnerabilities exist wherever you use unescaped data in an SQL query. (If these names were more consistent, XSS would probably be called "HTML injection.")

The following example demonstrates an SQL injection vulnerability:

```
$hash = hash($_POST['password']);

$sql = "SELECT count(*) FROM users
 WHERE username = '{$_POST['username']}' AND password = '{$hash}'";

$result = mysql_query($sql);
```

The problem is that if the username is not escaped, its value can manipulate the format of the SQL query. Because this particular vulnerability is so common, many attackers try usernames such as the following when trying to log in to a target site:

```
chris' --
```

Attackers love this username, because it allows access to the account with the username `chris'` without them having to know that account's password. After interpolation, the SQL query becomes:

```
SELECT count(*)
FROM users
WHERE username = 'chris' --'
AND password = '...'";
```

Because two consecutive hyphens (`--`) indicate the beginning of an SQL comment, this query is identical to:

```
SELECT count(*)
FROM users
WHERE username = 'chris'
```

If the code containing this snippet of code assumes a successful login when `$result` is nonzero, this SQL injection would allow an attacker to log in to any account without having to know or guess the password.

Safeguarding your applications against SQL injection is primarily accomplished by escaping output:

```
$mysql = array();

$hash = hash($_POST['password']);
$mysql['username'] = mysql_real_escape_string($clean['username']);

$sql = "SELECT count(*) FROM users
 WHERE username = '{$mysql['username']}' AND password = '{$hash}'";

$result = mysql_query($sql);
```

However, this only ensures that the data you escape is interpreted as data. You still need to filter data because characters like the percent sign (`%`) have a special meaning in SQL but don't need to be escaped.

The best protection against SQL injection is the use of *bound parameters*. The following example demonstrates the use of bound parameters with PHP's PDO extension and an Oracle database:

```
$sql = $db->prepare("SELECT count(*) FROM users
 WHERE username = :username AND password = :hash");

$sql->bindParam(":username", $clean['username'], PDO::PARAM_STRING, 32);
$sql->bindParam(":hash", hash($_POST['password']), PDO::PARAM_STRING, 32);
```

Because bound parameters ensure that the data never enters a context where it can be considered anything but data (i.e., it's never misinterpreted), no escaping of the username and password is necessary.

## Filename Vulnerabilities

It's fairly easy to construct a filename that refers to something other than what you intended. For example, say you have a `$username` variable that contains the name the

user wants to be called, which the user has specified through a form field. Now let's say you want to store a welcome message for each user in the directory */usr/local/lib/greetings* so that you can output the message any time the user logs in to your application. The code to print the current user's greeting is:

```
include("/usr/local/lib/greetings/{$username}");
```

This seems harmless enough, but what if the user chose the username `"../../../../etc/passwd"`? The code to include the greeting now includes this relative path instead: */etc/passwd*. Relative paths are a common trick used by hackers against unsuspecting scripts.

Another trap for the unwary programmer lies in the way that, by default, PHP can open remote files with the same functions that open local files. The `fopen()` function and anything that uses it—such as `include()` and `require()`—can be passed an HTTP or FTP URL as a filename, and the document identified by the URL will be opened. For example:

```
chdir("/usr/local/lib/greetings");
$fp = fopen($username, 'r');
```

If `$username` is set to *https://www.example.com/myfile*, a remote file is opened, not a local one.

The situation is even worse if you let the user tell you which file to `include()`:

```
$file = $_REQUEST['theme'];
include($file);
```

If the user passes a `theme` parameter of *https://www.example.com/badcode.inc* and your `variables_order` includes `GET` or `POST`, your PHP script will happily load and run the remote code. Never use parameters as filenames like this.

There are several solutions to the problem of checking filenames. You can disable remote file access, check filenames with `realpath()` and `basename()` (as described next), and use the `open_basedir` option to restrict filesystem access outside your site's document root.

### Check for relative paths

When you need to allow the user to specify a filename in your application, you can use a combination of the `realpath()` and `basename()` functions to ensure that the filename is what it ought to be. The `realpath()` function resolves special markers (such as `.` and `..`). After a call to `realpath()`, the resulting path is a full path on which you can then use `basename()`. The `basename()` function returns just the filename portion of the path.

Going back to our welcome message scenario, here's an example of `realpath()` and `basename()` in action:

```
$filename = $_POST['username'];
$vetted = basename(realpath($filename));

if ($filename !== $vetted) {
 die("{$filename} is not a good username");
}
```

In this case, we've resolved `$filename` to its full path and then extracted just the filename. If this value doesn't match the original value of `$filename`, we've got a bad filename that we don't want to use.

Once you have the completely bare filename, you can reconstruct what the file path ought to be, based on where legal files should go, and add a file extension based on the actual contents of the file:

```
include("/usr/local/lib/greetings/{$filename}");
```

## Session Fixation

A very popular attack that targets sessions is session fixation. The primary reason behind its popularity is that it's the easiest method by which an attacker can obtain a valid session identifier. As such, it is intended as a stepping-stone to a session hijacking attack, in which an attacker impersonates a user by presenting the user's session identifier.

Session fixation is any approach that causes a victim to use a session identifier chosen by an attacker. The simplest example is a link with an embedded session identifier:

```
<a href="http://host/login.php?PHPSESSID=1234">Log In</a>
```

A victim who clicks this link will resume the session identified as `1234`, and if the victim proceeds to log in, the attacker can hijack the victim's session to escalate the level of privilege.

There are a few variants of this attack, including some that use cookies for this same purpose. Luckily, the safeguard is simple, straightforward, and consistent. Whenever there is a change in the level of privilege, such as when a user logs in, regenerate the session identifier with `session_regenerate_id()`:

```
if (check_auth($_POST['username'], $_POST['password'])) {
 $_SESSION['auth'] = TRUE;
 session_regenerate_id(TRUE);
}
```

This effectively prevents session fixation attacks by ensuring that any user who logs in (or otherwise escalates the privilege level in any way) is assigned a fresh, random session identifier.

## File Upload Traps

File uploads combine two dangers we've already discussed: user-modifiable data and the filesystem. While PHP 7 itself is secure in how it handles uploaded files, there are several potential traps for unwary programmers.

### Distrust browser-supplied filenames

Be careful using the filename sent by the browser. If possible, do not use it as the name of the file on your filesystem. It's easy to make the browser send a file identified as */etc/passwd* or */home/kevin/.forward*. You can use the browser-supplied name for all user interaction, but generate a unique name yourself to actually call the file. For example:

```
$browserName = $_FILES['image']['name'];
$tempName = $_FILES['image']['tmp_name'];

echo "Thanks for sending me {$browserName}.";

$counter++; // persistent variable
$filename = "image_{$counter}";

if (is_uploaded_file($tempName)) {
 move_uploaded_file($tempName, "/web/images/{$filename}");
}
else {
 die("There was a problem processing the file.");
}
```

### Beware of filling your filesystem

Another trap is the size of uploaded files. Although you can tell the browser the maximum size of file to upload, this is only a recommendation and does not ensure your script won't be handed a file of a larger size. Attackers can perform a denial-of-service attack by sending files large enough to fill up your server's filesystem.

Set the `post_max_size` configuration option in *php.ini* to the maximum size (in bytes) that you want:

```
post_max_size = 1024768; // one megabyte
```

PHP will ignore requests with data payloads larger than this size. The default 10 MB is probably larger than most sites require.

### Account for EGPCS settings

The default `variables_order` (EGPCS: environment, `GET`, `POST`, cookie, server) processes `GET` and `POST` parameters before cookies. This makes it possible for the user to send a cookie that overwrites the global variable you think contains information on

your uploaded file. To avoid being tricked like this, check that the given file was actually an uploaded file using the `is_uploaded_file()` function. For example:

```
$uploadFilepath = $_FILES['uploaded']['tmp_name'];

if (is_uploaded_file($uploadFilepath)) {
 $fp = fopen($uploadFilepath, 'r');

 if ($fp) {
 $text = fread($fp, filesize($uploadFilepath));
 fclose($fp);

 // do something with the file's contents
 }
}
```

PHP provides a `move_uploaded_file()` function that moves the file only if it was an uploaded file. This is preferable to moving the file directly with a system-level function or PHP's `copy()` function. For example, the following code cannot be fooled by cookies:

```
move_uploaded_file($_REQUEST['file'], "/new/name.txt");
```

## Unauthorized File Access

If only you and people you trust can log in to your web server, you don't need to worry about file permissions for files used by or created by your PHP programs. However, most websites are hosted on an ISP's machines, and there's a risk that non-trusted users can read files that your PHP program creates. There are a number of techniques that you can use to deal with file permissions issues.

### Restrict filesystem access to a specific directory

You can set the `open_basedir` option to restrict access from your PHP scripts to a specific directory. If `open_basedir` is set in your *php.ini*, PHP limits filesystem and I/O functions so that they can operate only within that directory or any of its subdirectories. For example:

```
open_basedir = /some/path
```

With this configuration in effect, the following function calls succeed:

```
unlink("/some/path/unwanted.exe");
include("/some/path/less/travelled.inc");
```

But these generate runtime errors:

```
$fp = fopen("/some/other/file.exe", 'r');
$dp = opendir("/some/path/../other/file.exe");
```

Of course, one web server can run many applications, and each application typically stores files in its own directory. You can configure `open_basedir` on a per-virtual-host basis in your *httpd.conf* file like this:

```
<VirtualHost 1.2.3.4>
 ServerName domainA.com
 DocumentRoot /web/sites/domainA
 php_admin_value open_basedir /web/sites/domainA
</VirtualHost>
```

Similarly, you can configure it per directory or per URL in *httpd.conf*:

```
# by directory
<Directory /home/httpd/html/app1>
 php_admin_value open_basedir /home/httpd/html/app1
</Directory>

# by URL
<Location /app2>
 php_admin_value open_basedir /home/httpd/html/app2
</Location>
```

The `open_basedir` directory can be set only in the *httpd.conf* file, not in *.htaccess* files, and you must use `php_admin_value` to set it.

### Get permissions right the first time

Do not create a file and then change its permissions. This creates a *race condition*, where a lucky user can open the file once it's created but before it's locked down. Instead, use the `umask()` function to strip off unnecessary permissions. For example:

```
umask(077); // disable ---rwxrwx
$fh = fopen("/tmp/myfile", 'w');
```

By default, the `fopen()` function attempts to create a file with permission 0666 (`rw-rw-rw-`). Calling `umask()` first disables the group and other bits, leaving only 0600 (`rw-------`). Now, when `fopen()` is called, the file is created with those permissions.

### Don't use files

Because all scripts running on a machine run as the same user, a file that one script creates can be read by another, regardless of which user wrote the script. All a script needs to know to read a file is the name of that file.

There is no way to change this, so the best solution is to not use files to store data that should be protected; the most secure place to store data is in a database.

A complex workaround is to run a separate Apache daemon for each user. If you add a reverse proxy such as *haproxy* in front of the pool of Apache instances, you may be able to serve 100+ users on a single machine. Few sites do this, however, because the

complexity and cost are much greater than those for the typical situation, where one Apache daemon can serve web pages for thousands of users.

### Protect session files

With PHP's built-in session support, session information is stored in files. Each file is named `/tmp/sess_id`, where *id* is the name of the session and is owned by the web server user ID, usually `nobody`.

Because all PHP scripts run as the same user through the web server, this means that any PHP script hosted on a server can read any session files for any other PHP site. In situations where your PHP code is stored on an ISP's server that is shared with other users' PHP scripts, variables you store in your sessions are visible to other PHP scripts.

Even worse, other users on the server can create files in the session directory */tmp*. There's nothing preventing attackers from creating a fake session file that has any variables and values they want in it. They can then have the browser send your script a cookie containing the name of the faked session, and your script will happily load the variables stored in the fake session file.

One workaround is to ask your service provider to configure their server to place your session files in your own directory. Typically, this means that your `VirtualHost` block in the Apache *httpd.conf* file will contain:

```
php_value session.save_path /some/path
```

If you have *.htaccess* capabilities on your server and Apache is configured to let you override options, you can make the change yourself.

### Conceal PHP libraries

Many a hacker has learned of weaknesses by downloading include files or data that is stored alongside HTML and PHP files in the web server's document root. To prevent this from happening to you, all you need to do is store code libraries and data outside the server's document root.

For example, if the document root is */home/httpd/html*, everything below that directory can be downloaded through a URL. It is a simple matter to put your library code, configuration files, logfiles, and other data outside that directory (e.g., in */usr/local/lib/myapp*). This doesn't prevent other users on the web server from accessing those files (see "Don't use files" on page 329), but it does prevent the files from being downloaded by remote users.

If you must store these auxiliary files in your document root, you should configure the web server to deny requests for those files. For example, this tells Apache to deny

requests for any file with the *.inc* extension, a common extension for PHP include files:

```
<Files ~ "\.inc$">
 Order allow,deny
 Deny from all
</Files>
```

A better and more preferred way to prevent downloading of PHP source files is to always use the *.php* extension.

If you store code libraries in a different directory from the PHP pages that use them, you'll need to tell PHP where the libraries are. Either give a path to the code in each `include()` or `require()`, or change `include_path` in *php.ini*:

```
include_path = ".:/usr/local/php:/usr/local/lib/myapp";
```

## PHP Code Issues

With the `eval()` function, PHP allows a script to execute arbitrary PHP code. Although it can be useful in a few limited cases, allowing any user-supplied data to go into an `eval()` call is just begging to be hacked. For instance, the following code is a security nightmare:

```
<html>
 <head>
 <title>Here are the keys...</title>
 </head>

 <body>
 <?php if ($_REQUEST['code']) {
 echo "Executing code...";

 eval(stripslashes($_REQUEST['code'])); // BAD!
 } ?>

 <form action="<?php echo $_SERVER['PHP_SELF']; ?>">
 <input type="text" name="code" />
 <input type="submit" name="Execute Code" />
 </form>
 </body>
</html>
```

This page takes some arbitrary PHP code from a form and runs it as part of the script. The running code has access to all of the global variables for, and runs with the same privileges as, the script. It's not hard to see why this is a problem. Type this into the form:

```
include("/etc/passwd");
```

Never do this. There is no practical way to ensure such a script can ever be secure.

You can globally disable particular function calls by listing them, separated by commas, in the `disable_functions` configuration option in *php.ini*. For example, you may never have need for the `system()` function, so you can disable it entirely with:

```
disable_functions = system
```

This doesn't make `eval()` any safer, though, as there's no way to prevent important variables from being changed or built-in constructs such as `echo()` from being called.

In the case of `include`, `require`, `include_once`, and `require_once`, your best bet is to turn off remote file access using `allow_url_fopen`.

Any use of `eval()` and the `/e` option with `preg_replace()` is dangerous, especially if you use any user-entered data in the calls. Consider the following:

```
eval("2 + {$userInput}");
```

It seems pretty innocuous. However, suppose the user enters the following value:

```
2; mail("l33t@somewhere.com", "Some passwords", "/bin/cat /etc/passwd");
```

In this case, both the expected command and the one you'd rather avoid will be executed. The only viable solution is to never give user-supplied data to `eval()`.

## Shell Command Weaknesses

Be very wary of using the `exec()`, `system()`, `passthru()`, and `popen()` functions and the backtick operator (`` ` ``) in your code. The shell is a problem because it recognizes special characters (e.g., semicolons to separate commands). For example, suppose your script contains this line:

```
system("ls {$directory}");
```

If the user passes the value `"/tmp;cat /etc/passwd"` as the `$directory` parameter, your password file is displayed because `system()` executes the following command:

```
ls /tmp;cat /etc/passwd
```

In cases where you must pass user-supplied arguments to a shell command, use `escapeshellarg()` on the string to escape any sequences that have special meaning to shells:

```
$cleanedArg = escapeshellarg($directory);
system("ls {$cleanedArg}");
```

Now, if the user passes `"/tmp;cat /etc/passwd"`, the command that's actually run is:

```
ls '/tmp;cat /etc/passwd'
```

The easiest way to avoid the shell is to do the work of whatever program you're trying to call in PHP code, rather than calling out to the shell. Built-in functions are likely to be more secure than anything involving the shell.

## Data Encryption Concerns

One last topic to cover is encrypting data that you want to ensure is not viewable in its native form. This mostly applies to website passwords, but there are other examples, such as Social Security numbers (Social Insurance numbers in Canada), credit card numbers, and bank account numbers.

Check out the discussion on the FAQ page of the PHP website to find the best approach for your specific data encryption needs.

# Further Resources

The following resources can help you expand on this brief introduction to code security:

- *Essential PHP Security* (O'Reilly) by Chris Shiflett and its companion website
- The Open Web Application Security Project

# Security Recap

Because security is such an important issue, we want to reiterate the main points of this chapter as well as provide a few additional tips:

- Filter input to be sure that all data you receive from remote sources is the data you expect. Remember, the stricter your filtering logic, the safer your application.
- Escape output in a context-aware manner to be sure that your data isn't misinterpreted by a remote system.
- Always initialize your variables. This is especially important when the `register_globals` directive is enabled.
- Disable `register_globals`, `magic_quotes_gpc`, and `allow_url_fopen`. See the PHP website for details on these directives.
- Whenever you construct a filename, check the components with `basename()` and `realpath()`.
- Store include files outside of the document root. It is better to not name your include files with the *.inc* extension. Name them with a *.php* extension, or some other less obvious extension.
- Always call `session_regenerate_id()` whenever a user's privilege level changes.
- Whenever you construct a filename from a user-supplied component, check the components with `basename()` and `realpath()`.

- Don't create a file and then change its permissions. Instead, set `umask()` so that the file is created with the correct permissions.
- Don't use user-supplied data with `eval()`, `preg_replace()` with the `/e` option, or any of the system commands—`exec()`, `system()`, `popen()`, `passthru()`, and the backtick operator (`` ` ``).

## What's Next

With potential vulnerabilities like these, you might be wondering why you should do this "web development thing" at all. There are almost daily reports of web security breaches at banks and investment houses with massive data loss and identity theft. At the very least, if you are going to become a good web developer you *must* always embrace security and keep in mind that it is a changing landscape. Don't ever assume that you are 100% secure.

Coming in the next chapter is a discussion on application development techniques. This is another area where web developers can really shine and save themselves a lot of headaches. The use of code libraries, error handling, and performance tuning are among the topics we'll cover.

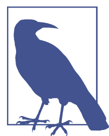
# CHAPTER 15
# Application Techniques

By now, you should have a solid understanding of the details of the PHP language and its use in a variety of common situations. Now we're going to show you some techniques you may find useful in your PHP applications, such as code libraries, templating systems, efficient output handling, error handling, and performance tuning.

## Code Libraries

As you've seen, PHP ships with numerous extension libraries that combine useful functionality into distinct packages that you can access from your scripts. We covered using the GD, FPDF, and Libxslt extension libraries in Chapters 10, 11, and 12, respectively.

In addition to using the extensions that ship with PHP, you can create libraries of your own code that you can use in more than one part of your website. The general technique is to store a collection of related functions in a PHP file. Then, when you need to use that functionality in a page, you can use `require_once()` to insert the contents of the file into your current script.



Note that there are three other inclusion type functions that can also be employed. They are `require()`, `include_once()`, and `include()`. Chapter 2 discusses these functions in detail.

For example, say you have a collection of functions that help create HTML form elements in valid HTML: one function in your collection creates a text field or a `textarea` (depending on how many characters you set as the maximum), another creates a series of pop ups from which to set a date and time, and so on. Rather than copying

the code into many pages—which is tedious, leads to errors, and makes it difficult to fix any bugs found in the functions—creating a function library is the sensible choice.

When you are combining functions into a code library, be careful to maintain a balance between grouping related functions and including functions that are not often used. When you include a code library in a page, all of the functions in that library are parsed, whether you use them all or not. PHP's parser is quick, but not parsing a function is even faster. At the same time, you don't want to split your functions across too many libraries, causing you to have to include lots of files in each page, because file access is slow.

# Templating Systems

A *templating system* provides a way of separating the code in a web page from the layout of that page. In larger projects, templates can be used to allow designers to deal exclusively with designing web pages and programmers to deal (more or less) exclusively with programming. The basic idea of a templating system is that the web page itself contains special markers that are replaced with dynamic content. A web designer can create the HTML for a page and simply worry about the layout, using the appropriate markers for different kinds of dynamic content that are needed. The programmer, on the other hand, is responsible for creating the code that generates the dynamic content for the markers.

To make this more concrete, let's look at a simple example. Consider the following web page, which asks the user to supply a name and then, if a name is provided, thanks the user:

```
<html>
 <head>
 <title>User Information</title>
 </head>

 <body>
 <?php if (!empty($_GET['name'])) {
 // do something with the supplied values ?>

 <p><font face="helvetica,arial">Thank you for filling out the form,
 <?php echo $_GET['name'] ?>.</font></p>
 <?php }
else { ?>
 <p><font face="helvetica,arial">Please enter the following information:
 </font></p>

 <form action="<?php echo $_SERVER['PHP_SELF'] ?>">
 <table>
 <tr>
 <td>Name:</td>
 <td>
```

```
    <input type="text" name="name" />
    <input type="submit" />
    </td>
    </tr>
    </table>
    </form>
  <?php } ?>
  </body>
  </html>
```

The placement of the different PHP elements within various layout tags, such as the `font` and `table` elements, is better left to a designer, especially as the page gets more complex. Using a templating system, we can split this page into separate files, some containing PHP code and some containing the layout. The HTML pages will then contain special markers where dynamic content should be placed. Example 15-1 shows the new HTML template page for our simple form, which is stored in the file *user.template*. It uses the `{DESTINATION}` marker to indicate the script that should process the form.

*Example 15-1. HTML template for user input form*

```
<html>
 <head>
 <title>User Information</title>
 </head>

 <body>
 <p>Please enter the following information:</p>

 <form action="{DESTINATION}">
 <table>
 <tr>
 <td>Name:</td>
 <td><input type="text" name="name" /></td>
 </tr>
 </table>
 </form>
 </body>
</html>
```

Example 15-2 shows the template for the thank-you page, called *thankyou.template*, which is displayed after the user has filled out the form. This page uses the `{NAME}` marker to include the value of the user's name.

*Example 15-2. HTML template for thank-you page*

```
<html>
 <head>
 <title>Thank You</title>
```

```
 </head>

 <body>
 <p>Thank you for filling out the form, {NAME}.</p>
 </body>
</html>
```

Now we need a script that can process these template pages, filling in the appropriate information for the various markers. Example 15-3 shows the PHP script that uses these templates (one for before the user has given us information and one for after). The PHP code uses the `fillTemplate()` function to join our values and the template files. This file is called *form_template.php*.

*Example 15-3. Template script*

```
<?php
$bindings["DESTINATION"] = $_SERVER["PHP_SELF"];
$name = $_GET["name"];

if (!empty($name)) {
 // do something with the supplied values
 $template = "thankyou.template";
 $bindings["NAME"] = $name;
}
else {
 $template = "user.template";
}

echo fillTemplate($template, $bindings);
```

Example 15-4 shows the `fillTemplate()` function used by the script in Example 15-3. The function takes a template filename (relative to a directory named *templates* located in the document root), an array of values, and an optional instruction denoting what to do if a marker is found for which no value is given. The possible values are `delete`, which deletes the marker; `comment`, which replaces the marker with a comment noting that the value is missing; or anything else, which just leaves the marker alone. This file is called *func_template.php*.

*Example 15-4. The fillTemplate() function*

```
<?php
function fillTemplate($name, $values = array(), $unhandled = "delete") {
 $templateFile = "{$_SERVER['DOCUMENT_ROOT']}/templates/{$name}";

 if ($file = fopen($templateFile, 'r')) {
 $template = fread($file, filesize($templateFile));
 fclose($file);
 }
```

```
$keys = array_keys($values);

foreach ($keys as $key) {
// look for and replace the key everywhere it occurs in the template
$template = str_replace("{{$key}}", $values[$key], $template);
}

if ($unhandled == "delete") {
// remove remaining keys
$template = preg_replace("/{[^ }]*}/i", "", $template);
}
else if ($unhandled == "comment") {
// comment remaining keys
$template = preg_replace("/{([^ }]*)}/i", "<!-- \\1 undefined -->", $template);
}

return $template;
}
```

Clearly, this example of a templating system is somewhat contrived. But if you think of a large PHP application that displays hundreds of news articles, you can imagine how a templating system that used markers such as `{HEADLINE}`, `{BYLINE}`, and `{ARTICLE}` might be useful, as it would allow designers to create the layout for article pages without needing to worry about the actual content.

While templates may reduce the amount of PHP code that designers have to see, there is a performance trade-off, as every request incurs the cost of building a page from the template. Performing pattern matches on every outgoing page can really slow down a popular site. Andrei Zmievski's Smarty is an efficient templating system that neatly side-steps much of this performance hit by turning the template into straight PHP code and caching it. Instead of doing the template replacement on every request, it does it only when the template file is changed.

# Handling Output

PHP is all about displaying output in the web browser. Accordingly, there are a few different techniques that you can use to handle output more efficiently or conveniently.

## Output Buffering

By default, PHP sends the results of `echo` and similar commands to the browser after each command is executed. Alternately, you can use PHP's output buffering functions to gather the information that would normally be sent to the browser into a buffer and send it later (or kill it entirely). This allows you to specify the content length of

your output after it is generated, capture the output of a function, or discard the output of a built-in function.

You turn on output buffering with the `ob_start()` function:

```
ob_start([callback]);
```

The optional *`callback`* parameter is the name of a function that postprocesses the output. If specified, this function is passed the collected output when the buffer is flushed, and it should return a string of output to send to the browser. You can use this, for instance, to turn all occurrences of *http://www.yoursite.com* to *http://www.mysite.com*.

While output buffering is enabled, all output is stored in an internal buffer. To get the current length and contents of the buffer, use `ob_get_length()` and `ob_get_contents()`:

```
$len = ob_get_length();
$contents = ob_get_contents();
```

If buffering isn't enabled, these functions return `false`.

There are two ways to throw away the data in the buffer. The `ob_clean()` function erases the output buffer but does not turn off buffering for subsequent output. The `ob_end_clean()` function erases the output buffer and ends output buffering.

There are three ways to send the collected output to the browser (this action is known as *flushing* the buffer). The `ob_flush()` function sends the output data to the web server and clears the buffer, but doesn't terminate output buffering. The `flush()` function not only flushes and clears the output buffer, but also tries to make the web server send the data to the browser immediately. The `ob_end_flush()` function sends the output data to the web server and ends output buffering. In all cases, if you specified a callback with `ob_start()`, that function is called to decide exactly what gets sent to the server.

If your script ends with output buffering still enabled—that is, if you haven't called `ob_end_flush()` or `ob_end_clean()`—PHP calls `ob_end_flush()` for you.

The following code collects the output of the `phpinfo()` function and uses it to determine whether you have the GD graphics module installed:

```
ob_start();
 phpinfo();
 $phpinfo = ob_get_contents();
ob_end_clean();

if (strpos($phpinfo, "module_gd") === false) {
 echo "You do not have GD Graphics support in your PHP, sorry.";
}
else {
```

```
 echo "Congratulations, you have GD Graphics support!";
}
```

Of course, a quicker and simpler approach to check if a certain extension is available is to pick a function that you know the extension provides and check if it exists. For the GD extension, you might do:

```
if (function_exists("imagecreate")) {
 // do something useful
}
```

To change all references in a document from *http://www.yoursite.com* to *http://www.mysite.com*, simply wrap the page like this:

```
ob_start(); ?>

Visit <a href="http://www.yoursite.com/foo/bar">our site</a> now!

<?php $contents = ob_get_contents();
ob_end_clean();
echo str_replace("http://www.yoursite.com/",
"http://www.mysite.com/", $contents);
?>

Visit <a href="http://www.mysite.com/foo/bar">our site</a> now!
```

Another way to do this is with a callback. Here, the `rewrite()` callback changes the text of the page:

```
function rewrite($text) {
 return str_replace("http://www.yoursite.com/",
"http://www.mysite.com/", $text);
}

ob_start("rewrite"); ?>

Visit <a href="http://www.yoursite.com/foo/bar">our site</a> now!
Visit <a href="http://www.mysite.com/foo/bar">our site</a> now!
```

## Output Compression

Recent browsers support compressing the text of web pages; the server sends compressed text and the browser decompresses it. To automatically compress your web page, wrap it like this:

```
ob_start("ob_gzhandler");
```

The built-in `ob_gzhandler()` function can be used as the callback for a call to `ob_start()`. It compresses the buffered page according to the `Accept-Encoding` header sent by the browser. Possible compression techniques are *gzip*, *deflate*, or none.

It rarely makes sense to compress short pages, as the time for compression and decompression exceeds the time it would take to simply send the uncompressed text. It does make sense to compress large (greater than 5 KB) web pages, however.

Instead of adding the `ob_start()` call to the top of every page, you can set the `output_handler` option in your *php.ini* file to a callback to be made on every page. For compression, this is `ob_gzhandler`.

# Performance Tuning

Before thinking much about performance tuning, take the time to get your code working properly. Once you have sound working code, you can locate the slower sections, or *bottlenecks*. If you try to optimize your code while writing it, you'll discover that optimized code tends to be more difficult to read and generally takes more time to write. If you spend that time on a section of code that isn't actually causing a problem, that's time wasted, especially down the road when you need to maintain that code and you can no longer read it.

Once you get your code working, you may find that it needs some optimization. Optimizing code tends to fall within one of two areas: shortening execution times and reducing memory requirements.

Before you begin optimization, ask yourself whether you need to optimize at all. Too many programmers have wasted hours wondering whether a complex series of string function calls are faster or slower than a single Perl regular expression, when the page where this code is located is viewed once every five minutes. Optimization is necessary only when a page takes so long to load that the user perceives it as slow. Often this is a symptom of a very popular site—if requests for a page come in fast enough, the time it takes to generate that page can mean the difference between prompt delivery and server overload. With a possible long wait on your site, you can bet that your web visitors won't take long to decide to look elsewhere for their information.

Once you've decided that your page needs optimization (this can best be done with some end user testing and observation), you can move on to working out exactly what is slow. You can use the techniques in the section "Profiling" to time the various subroutines or logical units of your page. This will give you an idea of which parts of your page are taking the longest time to produce—these parts are where you should focus your optimization efforts. If a page is taking 5 seconds to produce, you'll never get it down to 2 seconds by optimizing a function that accounts for only 0.25 seconds of the total time. Identify the biggest time-wasting blocks of code and focus on them. Time the page and the pieces you're optimizing to make sure your changes are having a positive, and not a negative, effect.

Finally, know when to quit. Sometimes there is an absolute limit for the speed at which you can get something to run. In these circumstances, the only way to get better performance is to throw new hardware at the problem. The solution might turn out to be faster machines or more web servers with a reverse-proxy cache in front of them.

## Benchmarking

If you're using Apache, you can use the Apache benchmarking utility, `ab`, to do high-level performance testing. To use it, run:

```
$ /usr/local/apache/bin/ab -c 10 -n 1000 http://localhost/info.php
```

This command tests the speed of the PHP script *info.php* 1,000 times, with 10 concurrent requests running at any given time. The benchmarking tool returns various information about the test, including the slowest, fastest, and average load times. You can compare those values to a static HTML page to see how quickly your script performs.

For example, here's the output from 1,000 fetches of a page that simply calls `phpinfo()`:

```
This is ApacheBench, Version 1.3d <$Revision: 1.2 $> apache-1.3
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd,
http://www.zeustech.net/
Copyright (c) 1998-2001 The Apache Group, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Finished 1000 requests
Server Software: Apache/1.3.22
Server Hostname: localhost
Server Port: 80

Document Path: /info.php
Document Length: 49414 bytes

Concurrency Level: 10
Time taken for tests: 8.198 seconds
Complete requests: 1000
Failed requests: 0
Broken pipe errors: 0
```

```
Total transferred: 49900378 bytes
HTML transferred: 49679845 bytes
Requests per second: 121.98 [#/sec] (mean)
Time per request: 81.98 [ms] (mean)
Time per request: 8.20 [ms] (mean, across all concurrent requests)
Transfer rate: 6086.90 [Kbytes/sec] received

Connnection Times (ms)
 min mean[+/-sd] median max
Connect: 0 12 16.9 1 72
Processing: 7 69 68.5 58 596
Waiting: 0 64 69.4 50 596
Total: 7 81 66.5 79 596

Percentage of the requests served within a certain time (ms)
 50% 79
 66% 80
 75% 83
 80% 84
 90% 158
 95% 221
 98% 268
 99% 288
 100% 596 (last request)
```

If your PHP script uses sessions, the results you get from `ab` will not be representative of the real-world performance of the scripts. Since a session is locked across a request, results from the concurrent requests run by `ab` will be extremely poor. However, in normal usage, a session is typically associated with a single user, who isn't likely to make concurrent requests.

Using `ab` tells you the overall speed of your page but gives you no information on the speed of individual functions of blocks of code within the page. Use `ab` to test changes you make to your code as you attempt to improve its speed. We show you how to time individual portions of a page in the next section, but ultimately these microbenchmarks don't matter if the overall page is still slow to load and run. The ultimate proof that your performance optimizations have been successful comes from the numbers that `ab` reports.

## Profiling

PHP does not have a built-in profiler, but there are some techniques you can use to investigate code that you think has performance issues. One technique is to call the `microtime()` function to get an accurate representation of the amount of time that elapses. You can surround the code you're profiling with calls to `microtime()` and use the values it returns to calculate how long the code took.

For instance, here's some code you can use to find out just how long it takes to produce the `phpinfo()` output:

```
ob_start();
$start = microtime(true);

phpinfo();

$end = microtime(true);
ob_end_clean();

echo "phpinfo() took " . ($end - $start) . " seconds to run.\n";
```

Reload this page several times, and you'll see the number fluctuate slightly. Reload it often enough, and you'll see it fluctuate quite a lot. The danger of timing a single run of a piece of code is that you may not get a representative machine load—the server might be paging as a user starts *emacs*, or it may have removed the source file from its cache. The best way to get an accurate representation of the time it takes to do something is to time repeated runs and look at the average of those times.

The `Benchmark` class available in PEAR makes it easy to repeatedly time sections of your script. Here is a simple example that shows how you can use it:

```
require_once 'Benchmark/Timer.php';

$timer = new Benchmark_Timer;

$timer->start();
 sleep(1);
 $timer->setMarker('Marker 1');
 sleep(2);
$timer->stop();

$profiling = $timer->getProfiling();

foreach ($profiling as $time) {
 echo $time["name"] . ": " . $time["diff"] . "<br>\n";
}

echo "Total: " . $time["total"] . "<br>\n";
```

The output from this program is:

```
Start: -
Marker 1: 1.0006979703903
Stop: 2.0100029706955
Total: 3.0107009410858
```

That is, it took 1.0006979703903 seconds to get to Marker 1, which is set right after our `sleep(1)` call, so it is what you would expect. It took just over two seconds to get from Marker 1 to the end, and the entire script took just over three seconds to run.

You can add as many markers as you like and thereby time various parts of your script.

## Optimizing Execution Time

Here are some tips for shortening the execution times of your scripts:

- Avoid `printf()` when `echo` is all you need.
- Avoid recomputing values inside a loop, as PHP's parser does not remove loop invariants. For example, don't do this if the size of $array doesn't change:

  ```
  for ($i = 0; $i < count($array); $i++) { /* do something */ }
  ```

  Instead, do this:

  ```
  $num = count($array);
  for ($i = 0; $i < $num; $i++) { /* do something */ }
  ```

- Include only files that you need. Split included files to include only functions that you are sure will be used together. Although the code may be a bit more difficult to maintain, parsing code you don't use is expensive.
- If you are using a database, use persistent database connections—setting up and tearing down database connections can be slow.
- Don't use a regular expression when a simple string-manipulation function will do the job. For example, to turn one character into another in a string, use `str_replace()`, not `preg_replace()`.

## Optimizing Memory Requirements

Here are some techniques for reducing the memory requirements of your scripts:

- Use numbers instead of strings whenever possible:

  ```
  for ($i = "0"; $i < "10"; $i++) // bad
  for ($i = 0; $i < 10; $i++) // good
  ```

- When you're done with a large string, set the variable holding the string to an empty string. This frees up the memory to be reused.
- Only include or require files that you need. Use `include_once()` and `require_once()` instead of `include()` and `require()`.
- Release MySQL or other database result sets as soon as you are done with them. There is no benefit to keeping result sets in memory beyond their use.

## Reverse Proxies and Replication

Adding hardware is often the quickest route to better performance. It's better to benchmark your software first, though, as it's generally cheaper to fix software than to buy new hardware. Three common solutions to the problem of scaling traffic are reverse-proxy caches, load-balancing servers, and database replication.

### Reverse-proxy caches

A *reverse proxy* is a program that sits in front of your web server and handles all connections from client browsers. Proxies are optimized to serve up static files quickly, and despite appearances and implementation, most dynamic sites can be cached for short periods of time without loss of service. Normally, you'll run the proxy on a separate machine from your web server.

Take, for example, a busy site whose front page is hit 50 times per second. If this first page is built from two database queries and the database changes as often as twice a minute, you can avoid 5,994 database queries per minute by using a `Cache-Control` header to tell the reverse proxy to cache the page for 30 seconds. The worst-case scenario is that there will be a 30-second delay from database update to a user seeing this new data. For most applications that's not a very long delay, and it gives significant performance benefits.

Proxy caches can even intelligently cache content that is personalized or tailored to the browser type, accepted language, or similar feature. The typical solution is to send a `Vary` header telling the cache exactly which request parameters affect the caching.

There are hardware proxy caches available, but there are also very good software implementations. For a high-quality and extremely flexible open source proxy cache, have a look at Squid. See the book *Web Caching* (O'Reilly) by Duane Wessels for more information on proxy caches and how to tune a website to work with one.

### Load balancing and redirection

One way to boost performance is to spread the load over a number of machines. A *load-balancing system* does this by either evenly distributing the load or sending incoming requests to the least-loaded machine. A *redirector* is a program that rewrites incoming URLs, allowing fine-grained control over the distribution of requests to individual server machines.

Again, there are hardware HTTP redirectors and load balancers, but redirection and load balancing can also be done effectively in software. By adding redirection logic to Squid through a tool like SquidGuard, you can improve performance in a number of ways.

### MySQL replication

Sometimes the database server is the bottleneck—many simultaneous queries can bog down a database server, resulting in sluggish performance. Replication is one of the best solutions. Take everything that happens to one database and quickly bring one or more other databases in sync, so you end up with multiple identical databases. This lets you spread your queries across many database servers instead of loading down only one.

The most effective model is to use one-way replication, where you have a single master database that gets replicated to a number of slave databases. Database writes go to the master server, and database reads are load-balanced across multiple slave databases. This technique is aimed at architectures that do a lot more reads than writes. Most web applications fit this scenario nicely.

Figure 15-1 shows the relationship between the master and slave databases during replication.

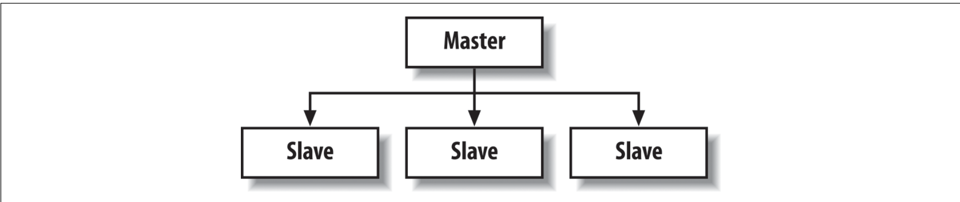



*Figure 15-1. Database replication relationship*

Many databases support replication, including MySQL, PostgreSQL, and Oracle.

### Putting it all together

For a really high-powered architecture, pull all these concepts together into a configuration like the one shown in Figure 15-2.

Using five separate machines—one for the reverse proxy and redirector, three web servers, and one master database server—this architecture can handle a huge number of requests. The exact number depends only on the two bottlenecks—the single Squid proxy and the single master database server. With a bit of creativity, either or both of these could be split across multiple servers as well, but as it is, if your application is somewhat cacheable and heavy on database reads, this is a nice approach.

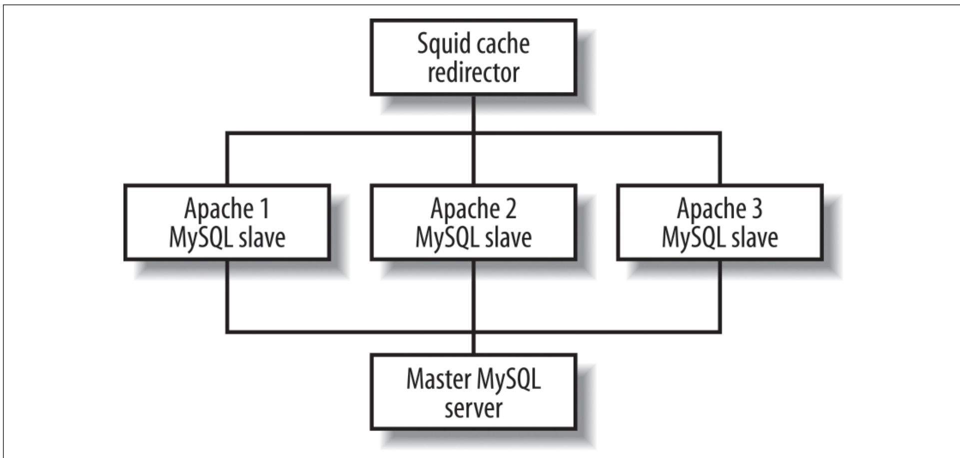



*Figure 15-2. Putting it all together*

Each Apache server gets its own read-only MySQL database, so all read requests from your PHP scripts go over a Unix-domain local socket to a dedicated MySQL instance. You can add as many of these Apache/PHP/MySQL servers as you need under this framework. Any database writes from your PHP applications will go over a Transmission Control Protocol (TCP) socket to the master MySQL server.

## What's Next

In the next chapter, we'll dive deeper into using PHP to develop and deploy web services.

CHAPTER 16

# Web Services

Historically, every time there's been a need for two systems to communicate, a new protocol has been created (for example, SMTP for sending mail, POP3 for receiving mail, and the numerous protocols that database clients and servers use). The idea of web services is to remove the need to create new protocols by providing a standardized mechanism for remote procedure calls, based on XML and HTTP.

Web services make it easy to integrate heterogeneous systems. Say you're writing a web interface to a library system that already exists. It has a complex system of database tables, and lots of business logic embedded in the program code that manipulates those tables. And it's written in C++. You could reimplement the business logic in PHP, writing a lot of code to manipulate tables in the correct way, or you could write a little code in C++ to expose the library operations (e.g., check out a book to a user, see when this book is due back, see what the overdue fines are for this user) as a web service. Now your PHP code simply has to handle the web frontend; it can use the library service to do all the heavy lifting.

## REST Clients

A *RESTful web service* is a loose description of web APIs implemented using HTTP and the principles of Representational State Transfer (REST). It refers to a collection of resources, along with basic operations a client can perform on those resources through the API.

For example, an API might describe a collection of authors and the books to which those authors have contributed. The data within each object type is arbitrary. In this case, a *resource* is each individual author, each individual book, and the collections of all authors, all books, and the books to which each author has contributed. Each

resource must have a unique identifier so calls into the API know what resource is being retrieved or acted upon.

You might represent a simple set of classes to represent the book and author resources, as in Example 16-1.

*Example 16-1. Book and Author classes*

```
class Book {
 public $id;
 public $name;
 public $edition;

 public function __construct($id) {
 $this->id = $id;
 }
}

class Author {
 public $id;
 public $name;
 public $books = array();

 public function __construct($id) {
 $this->id = $id;
 }
}
```

Because HTTP was built with the REST architecture in mind, it provides a set of *verbs* that you use to interact with the API. We've already seen `GET` and `POST` verbs, which websites often use to represent "retrieve data" and "perform an action," respectively. RESTful web services introduce two additional verbs, `PUT` and `DELETE`:

`GET`
: Retrieve information about a resource or collection of resources.

`POST`
: Create a new resource.

`PUT`
: Update a resource with new data, or replace a collection of resources with new ones.

`DELETE`
: Delete a resource or a collection of resources.

For example, the `Books` and `Authors` API might consist of the following REST endpoints, based on the data contained within the object classes:

`GET /api/authors`
: Return a list of identifiers for each author in the collection.

`POST /api/authors`
: Given information about a new author, create a new author in the collection.

`GET /api/authors/`*`id`*
: Retrieve the author with identifier *`id`* from the collection and return it.

`PUT /api/authors/`*`id`*
: Given updated information about an author with identifier *`id`*, update that author's information in the collection.

`DELETE /api/authors/`*`id`*
: Delete the author with identifier *`id`* from the collection.

`GET /api/authors/`*`id`*`/books`
: Retrieve a list of identifiers for each book to which the author with identifier *`id`* has contributed.

`POST /api/authors/`*`id`*`/books`
: Given information about a new book, create a new book in the collection under the author with identifier *`id`*.

`GET /api/books/`*`id`*
: Retrieve the book with identifier *`id`* from the collection and return it.

The `GET`, `POST`, `PUT`, and `DELETE` verbs provided by RESTful web services can be thought of as roughly equivalent to the *create*, *retrieve*, *update*, and *delete* (CRUD) operations typical to a database, although they can correlate to collections, not just entities as is typical with CRUD implementations.

## Responses

In each of the preceding API endpoints, the HTTP status code is used to provide the result of the request. HTTP provides a long list of standard status codes: for example, `201 Created` would be returned when you create a resource, and `501 Not Implemented` would be returned when you send a request to an endpoint that doesn't exist.

While the full list of HTTP codes is beyond the scope of this chapter, some common ones include:

`200 OK`
: The request was successfully completed.

`201 Created`
: A request for creating a new resource was completed successfully.

`400 Bad Request`
: The request hit a valid endpoint, but was malformed and could not be completed.

`401 Unauthorized`
: Along with `403 Forbidden`, represents a valid request, but one that could not be completed due to a lack of permissions. Typically, this response indicates that authorization is required but has not yet been provided.

`403 Forbidden`
: Similar to `401 Unauthorized`, this response indicates a valid request, but one that could not be completed due to a lack of permissions. Typically, this response indicates that authorization was available but that the user lacks permission to perform the requested action.

`404 Not Found`
: The resource was not found (for example, attempting to delete an author with an ID that does not exist).

`500 Internal Server Error`
: An error occurred on the server side.

These codes are mere guidelines and typical responses; the exact responses provided by a RESTful API are detailed by the API itself.

## Retrieving Resources

Retrieving information for a resource involves a straightforward `GET` request. Example 16-2 uses the *curl* extension to format an HTTP request, set parameters on it, send the request, and get the returned information.

*Example 16-2. Retrieving author data*

```
$authorID = "ktatroe";
$url = "http://example.com/api/authors/{$authorID}";

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);

$response = curl_exec($ch);
$resultInfo = curl_getinfo($ch);

curl_close($ch);

// decode the JSON and use a Factory to instantiate an Author object
$authorJSON = json_decode($response);
$author = ResourceFactory::authorFromJSON($authorJSON);
```

To retrieve information about an author, this script first constructs a URL representing the endpoint for the resource. Then, it initializes a curl resource and provides the constructed URL to it. Finally, the curl object is executed, which sends the HTTP request, waits for the response, and returns it.

In this case, the response is JSON data, which is decoded and handed off to a `Factory` method of `Author` to construct an instance of the `Author` class.

## Updating Resources

Updating an existing resource is a bit trickier than retrieving information about a resource. In this case, you need to use the `PUT` verb. As `PUT` was originally intended to handle file uploads, `PUT` requests require that you stream data to the remote service from a file.

Rather than creating a file on disk and streaming from it, the script in Example 16-3 uses the `'memory'` stream provided by PHP, first filling it with the data to send, then rewinding it to the start of the data it just wrote, and finally pointing the curl object at the file.

*Example 16-3. Updating book data*

```
$bookID = "ProgrammingPHP";
$url = "http://example.com/api/books/{$bookID}";

$data = json_encode(array(
 'edition' => 4,
));

$requestData = http_build_query($data, '', '&');

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);

$fh = fopen("php://memory", 'rw');
fwrite($fh, $requestData);
rewind($fh);

curl_setopt($ch, CURLOPT_INFILE, $fh);
curl_setopt($ch, CURLOPT_INFILESIZE, mb_strlen($requestData));
curl_setopt($ch, CURLOPT_PUT, true);

$response = curl_exec($ch);
$resultInfo = curl_getinfo($ch);

curl_close($ch);
fclose($fh);
```

## Creating Resources

To create a new resource, call the appropriate endpoint with the POST verb. The data for the request is put into the typical key-value form for POST requests.

In Example 16-4, the Author API endpoint for creating a new author takes the information to create the new author as a JSON-formatted object under the key 'data'.

*Example 16-4. Creating an author*

```
<?php $newAuthor = new Author('pbmacintyre');
$newAuthor->name = "Peter Macintyre";

$url = "http://example.com/api/authors";

$data = array(
 'data' => json_encode($newAuthor)
);

$requestData = http_build_query($data, '', '&');

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);

curl_setopt($ch, CURLOPT_POSTFIELDS, $requestData);
curl_setopt($ch, CURLOPT_POST, true);

$response = curl_exec($ch);
$resultInfo = curl_getinfo($ch);

curl_close($ch);
```

This script first constructs a new Author instance and encodes its values as a JSON-formatted string. Then, it constructs the key-value data in the appropriate format, provides that data to the curl object, and sends the request.

## Deleting Resources

Deleting a resource is similarly straightforward. Example 16-5 creates a request, sets the verb on that request to 'DELETE' via the curl_setopt() function, and sends it.

*Example 16-5. Deleting a book*

```
<?php $authorID = "ktatroe";
$bookID = "ProgrammingPHP";
$url = "http://example.com/api/authors/{$authorID}/books/{$bookID}";

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
```

```
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'DELETE');

$result = curl_exec($ch);
$resultInfo = curl_getinfo($ch);

curl_close($ch);
```

# XML-RPC

While less popular nowadays than REST, XML-RPC and SOAP are two older standard protocols used to create web services. XML-RPC is the older and simpler of the two, while SOAP is newer and more complex.

PHP provides access to both SOAP and XML-RPC through the *xmlrpc* extension, which is based on the xmlrpc-epi project. The *xmlrpc* extension is not compiled in by default, so you'll need to add `--with-xmlrpc` to your `configure` line when you compile PHP.

## Servers

Example 16-6 shows a very basic XML-RPC server that exposes only one function (which XML-RPC calls a "method"). That function, `multiply()`, multiplies two numbers and returns the result. It's not a very exciting example, but it shows the basic structure of an XML-RPC server.

*Example 16-6. Multiplier XML-RPC server*

```
<?php
// expose this function via RPC as "multiply()"
function times ($method, $args) {
 return $args[0] * $args[1];
}

$request = $HTTP_RAW_POST_DATA;

if (!$request) {
 $requestXml = $_POST['xml'];
}

$server = xmlrpc_server_create() or die("Couldn't create server");
xmlrpc_server_register_method($server, "multiply", "times");

$options = array(
 'output_type' => 'xml',
 'version' => 'auto',
);
```

```
echo xmlrpc_server_call_method($server, $request, null, $options);

xmlrpc_server_destroy($server);
```

The *xmlrpc* extension handles the dispatch for you. That is, it works out which method the client was trying to call, decodes the arguments, and calls the corresponding PHP function. It then returns an XML response that encodes any values returned by the function that can be decoded by an XML-RPC client.

Create a server with `xmlrpc_server_create()`:

```
$server = xmlrpc_server_create();
```

Once you've created a server, expose functions through the XML-RPC dispatch mechanism using `xmlrpc_server_register_method()`:

```
xmlrpc_server_register_method(server, method, function);
```

The *`method`* parameter is the name the XML-RPC client knows. The *`function`* parameter is the PHP function implementing that XML-RPC method. In the case of Example 16-6, the `multiply()` XML-RPC client method is implemented by the `times()` function in PHP. Often a server will call `xmlrpc_server_register_method()` many times to expose many functions.

When you've registered all your methods, call `xmlrpc_server_call_method()` to dispatch the incoming request to the appropriate function:

```
$response = xmlrpc_server_call_method(server, request, user_data [, options]);
```

The *`request`* is the XML-RPC request, which is typically sent as HTTP `POST` data. We fetch that through the `$HTTP_RAW_POST_DATA` variable. It contains the name of the method to be called and parameters to that method. The parameters are decoded into PHP data types, and the function (`times()`, in this case) is called.

A function exposed as an XML-RPC method takes two or three parameters:

```
$retval = exposedFunction(method, args [, user_data]);
```

The *`method`* parameter contains the name of the XML-RPC method (so you can have one PHP function exposed under many names). The arguments to the method are passed in the array *`args`*, and the optional *`user_data`* parameter is whatever the `xmlrpc_server_call_method()` function's *`user_data`* parameter was.

The *`options`* parameter to `xmlrpc_server_call_method()` is an array mapping option names to their values. The options are:

`output_type`
: Controls the data encoding used. Permissible values are `"php"` or `"xml"` (default).

`verbosity`
: Controls how much whitespace is added to the output XML to make it readable to humans. Permissible values are `"no_white_space"`, `"newlines_only"`, and `"pretty"` (default).

`escaping`
: Controls which characters are escaped and how they are escaped. Multiple values may be given as a subarray. Permissible values are `"cdata"`, `"non-ascii"` (default), `"non-print"` (default), and `"markup"` (default).

`versioning`
: Controls which web service system to use. Permissible values are `"simple"`, `"soap 1.1"`, `"xmlrpc"` (default for clients), and `"auto"` (default for servers, meaning "whatever format the request came in").

`encoding`
: Controls the character encoding of the data. Permissible values include any valid encoding identifiers, but you'll rarely want to change it from `"iso-8859-1"` (the default).

## Clients

An XML-RPC client issues an HTTP request and parses the response. The *xmlrpc* extension that ships with PHP can work with the XML that encodes an XML-RPC request, but it doesn't know how to issue HTTP requests. For that functionality, you must download the xmlrpc-epi distribution and install the *sample/utils/utils.php* file. This file contains a function to perform the HTTP request.

Example 16-7 shows a client for the `multiply` XML-RPC service.

*Example 16-7. Multiply XML-RPC client*

```
<?php
require_once("utils.php");

$options = array('output_type' => "xml", 'version' => "xmlrpc");

$result = xu_rpc_http_concise(
 array(
 'method' => "multiply",
 'args' => array(5, 6),
 'host' => "192.168.0.1",
 'uri' => "/~gnat/test/ch11/xmlrpc-server.php",
 'options' => $options,
 )
);
```

```
echo "5 * 6 is {$result}";
```

We begin by loading the XML-RPC convenience utilities library. This gives us the `xu_rpc_http_concise()` function, which constructs a `POST` request for us:

```
$response = xu_rpc_http_concise(hash);
```

The *hash* array contains the various attributes of the XML-RPC call as an associative array:

`method`
: Name of the method to call.

`args`
: Array of arguments to the method.

`host`
: Hostname of the web service offering the method.

`url`
: URL path to the web service.

`options`
: Associative array of options, as for the server.

`debug`
: If nonzero, prints debugging information (default is `0`).

The value returned by `xu_rpc_http_concise()` is the decoded return value from the called method.

There are several features of XML-RPC we haven't covered. For example, XML-RPC's data types do not always map precisely onto those of PHP, and there are ways to encode values as a particular data type rather than as the *xmlrpc* extension's best guess. Also, there are features of the *xmlrpc* extension we haven't covered, such as SOAP faults. See the *xmlrpc* extension's documentation for the full details.

For more information on XML-RPC, see *Programming Web Services in XML-RPC* (O'Reilly) by Simon St. Laurent et al. See *Programming Web Services with SOAP* (O'Reilly) by James Snell et al. for more information on SOAP.

## What's Next

Now that we've covered the majority of the syntax, features, and application of PHP, the next chapter explores what to do when things go wrong: how to debug problems that arise in your PHP applications and scripts.

CHAPTER 17

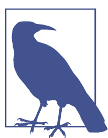
# Debugging PHP

Debugging is an acquired skill. As is often said in the development world, "You are given all the rope you should ever need; just attempt to tie a pretty bow with it rather than getting yourself hanged." It naturally stands to reason that the more debugging you do, the more proficient you will become. Of course, you will also get some excellent hints from your server environment when your code does not deliver what you were expecting. Before we get too deep into debugging concepts, however, we need to look at the bigger picture and discuss these programming environments. Every development shop has its own setup and its own way of doing things, so what we'll be covering here reflects the ideal conditions, also known as best practices.

PHP development in a utopian world has at least three separate environments in which work is being done: development, staging, and production. We'll explore each in turn in the following sections.

## The Development Environment

The development environment is a place where the raw code is created without fear of server crashes or peer ridicule. This should be a place where concepts and theories are proven or disproven, where code can be created experimentally. Therefore, the error-reporting environmental feedback should be as verbose as possible. All error reporting should be logged and at the same time also sent to the output device (the browser). All warnings should be as sensitive and descriptive as possible.

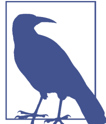


Later in this chapter, Table 17-1 compares the recommended server settings for each of the three environments as it relates to debugging and error reporting.

The location of this development environment can be debated. However, if your company has the resources, then a separate server should be established for this purpose with full code management (e.g., SVN, aka Subversion, or Git) in place. If the resources are not available, then a development PC can serve this purpose via a `localhost`-style setup. This `localhost` environment can be advantageous in and of itself in the sense that you may want to try something completely off-the-wall, and by coding on a standalone PC you can be fully experimental without affecting a common development server or anyone else's code base.

You can create `localhost` environments with the Apache web server, or Microsoft's Internet Information Services (IIS), as a manual process. There are a few all-in-one environments that can be utilized as well; Zend Server CE (Community Edition) is a great example.

No matter what setup you have for raw development, be sure to give your developers full freedom to do what they want without fear of reprimand. This gives them the confidence to be as innovative as possible, and no one gets "hurt."

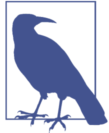


There are at least two alternatives to setting up a local environment on your own PC. The first one is, as of PHP 5.4, a built-in web server. This option saves on downloading and installing full Apache or IIS web server products for `localhost` purposes.

Second, there are now hosts (pun intended) of sites that allow for cloud development. Zend offers one for free as a testing and development environment.

# The Staging Environment

The staging environment should mimic the production environment as closely as possible. Although this is sometimes hard to achieve, the more closely you can mimic the production environment, the better. You will be able to see how your code reacts in an area that is protected but also simulates the real production environment. The staging environment is often where the end user or client can test out new features or functionality, giving feedback and stress-testing code, without fear of affecting production code.

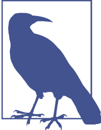


As testing and experimentation progress, your staging area (at least from a data perspective) will eventually grow more distinct from the production environment. So it is a good practice to have procedures in place that will replace the staging area with production information from time to time. The set times will be different for each company or development shop depending on features being created, release cycles, and so on.

If resources permit, you should consider having two separate staging environments: one for developers (coding peers) and the other for client testing. Feedback from these two types of users is quite often very different and very telling. Server error reporting and feedback should be kept to a minimum here as well, to duplicate production as closely as possible.

## The Production Environment

The production environment, from an error-reporting perspective, needs to be as tightly controlled as possible. You want to fully control what the end user sees and experiences. Things like SQL failures and code syntax warnings should never be seen by the client, if at all possible. Your code base, of course, should be well mitigated by this time (assuming you've been using the two aforementioned environments properly and religiously), but sometimes errors and bugs can still get through to production. If you're going to fail in production, you want to fail as gracefully and as *quietly* as possible.



Consider using 404 page redirects and `try...catch` structures to redirect errors and failures to a safe landing area in the production environment. See Chapter 2 for proper coding styles of the `try...catch` syntax.

At the very least, all error reporting should be suppressed and sent to the logfiles in the production environment.

## php.ini Settings

There are a few environment-wide settings to consider for each type of server you're using to develop your code. First, we'll offer a brief summary of what these are, and then we'll list the recommended settings for each of the three coding environments.

`display_errors`
: An on-off toggle that controls the display of any errors encountered by PHP. This should be set to `0` (off) for production environments.

`error_reporting`
: This is a setting of predefined constants that will report to the error log and/or the web browser any errors that PHP encounters. There are 16 different individual constants that can be set within this directive, and certain ones can be used collectively. The most common ones are `E_ALL`, for reporting all errors and warnings of any kind; `E_WARNING`, for only showing warnings (nonfatal errors) to the browser; and `E_DEPRECATED`, to display runtime notice warnings about code that will fail in future versions of PHP because some functionality is scheduled to be

ended (like `register_globals` was). An example of these being used in combination is `E_ALL & ~E_NOTICE`, which tells PHP to report all errors except the generated notices. A full listing of these defined constants can be found on the PHP website.

`error_log`
: The path to the location of the error log. The error log is a text-based file located on the server at the path location that records all errors in text form. This could be *apache2/logs* in the case of an Apache server.

`variables_order`
: Sets the order of precedence in which the superglobal arrays are loaded with information. The default order is `EGPCS`, meaning the environment (`$_ENV`) array is loaded first, then the `GET` (`$_GET`) array, then the `POST` (`$_POST`) array, then the cookie (`$_COOKIE`) array, and finally the server (`$_SERVER`) array.

`request_order`
: Describes the order in which PHP registers `GET`, `POST`, and cookie variables into the `$_REQUEST` array. Registration is done from left to right, and newer values override older values.

`zend.assertions`
: Determines whether assertions are run and throw errors. When disabled, the conditions in calls to `assert()` are never run (thus, any side effects they might have do not happen).

`assert.exception`
: Determines whether the exception system is enabled. By default, this is on in both development and production environments, and is generally the preferred way to handle error conditions.

Additional settings can be used as well; for example, you can use `ignore_repeated_errors` if you are concerned with your logfile getting too large. This directive can suppress repeating errors being logged, but only from the same line of code in the same file. This could be useful if you are debugging a looping section of code and an error is occurring somewhere within it.

PHP also allows you to alter certain INI settings from their server-wide settings during the execution of your code. This can be a quick way to turn on some error reporting and display the results on screen, but it is still not recommended in a production environment. You could do this in the staging environment if desired. One example is to turn on all the error reporting and display any reported errors to the browser in a single suspect file. To do so, insert the following two commands at the top of the file:

```
error_reporting(E_ALL);
ini_set("display_errors", 1);
```

The `error_reporting()` function allows you to override the level of reported errors, and the `ini_set()` function allows you to change *php.ini* settings. Again, not all INI settings can be altered, so be sure to check the PHP website for what can and cannot be changed at runtime.

As promised earlier, Table 17-1 lists the PHP directives and their recommendations for each of the three basic server environments.

*Table 17-1. PHP error directives for server environments*

| PHP directive | Development | Staging | Production |
|---|---|---|---|
| `display_errors` | On | Either setting, depending on desired outcome | Off |
| `error_reporting` | `E_ALL` | `E_ALL & ~E_WARNING & ~E_DEPRECATED` | `E_ALL & ~E_DEPRECATED & ~E_STRICT` |
| `error_log` | */logs* folder | */logs* folder | */logs* folder |
| `variables_order` | EGPCS | GPCS | GPCS |
| `request_order` | GP | GP | GP |

# Error Handling

Error handling is an important part of any real-world application. PHP provides a number of mechanisms that you can use to handle errors, both during the development process and once your application is in a production environment.

## Error Reporting

Normally, when an error occurs in a PHP script, the error message is inserted into the script's output. If the error is fatal, the script execution stops.

There are three levels of conditions: notices, warnings, and errors. A *notice* that occurs during a script's execution might indicate an error, but it could also occur during normal execution (e.g., a script trying to access a variable that has not been set). A *warning* indicates a nonfatal error condition; typically, warnings are displayed when you call a function with invalid arguments. Scripts will continue executing after issuing a warning. An *error* indicates a fatal condition from which the script cannot recover. A *parse error* is a specific kind of error that occurs when a script is syntactically incorrect. All errors except parse errors are runtime errors.

It's recommended that you treat all notices, warnings, and errors as if they were errors; this helps prevent mistakes such as using variables before they have legitimate values.

By default, all conditions except runtime notices are caught and displayed to the user. You can change this behavior globally in your *php.ini* file with the `error_reporting` option. You can also locally change the error-reporting behavior in a script using the `error_reporting()` function.

With both the `error_reporting` option and the `error_reporting()` function, you specify the conditions that are caught and displayed by using the various bitwise operators to combine different constant values, as listed in Table 17-2. For example, this indicates all error-level options:

```
(E_ERROR | E_PARSE | E_CORE_ERROR | E_COMPILE_ERROR | E_USER_ERROR)
```

while this indicates all options except runtime notices:

```
(E_ALL & ~E_NOTICE)
```

If you set the `track_errors` option on in your *php.ini* file, a description of the current error is stored in `$PHP_ERRORMSG`.

*Table 17-2. Error-reporting values*

| Value | Meaning |
|---|---|
| E_ERROR | Runtime errors |
| E_WARNING | Runtime warnings |
| E_PARSE | Compile-time parse errors |
| E_NOTICE | Runtime notices |
| E_CORE_ERROR | Errors generated internally by PHP |
| E_CORE_WARNING | Warnings generated internally by PHP |
| E_COMPILE_ERROR | Errors generated internally by the Zend scripting engine |
| E_COMPILE_WARNING | Warnings generated internally by the Zend scripting engine |
| E_USER_ERROR | Runtime errors generated by a call to `trigger_error()` |
| E_USER_WARNING | Runtime warnings generated by a call to `trigger_error()` |
| E_USER_NOTICE | Runtime notices generated by a call to `trigger_error()` |
| E_ALL | All of the above options |

## Exceptions

Many PHP functions now throw exceptions instead of fatally exiting operation. Exceptions allow a script to continue execution even after an error—when the exception occurs, an object that's a subclass of the `BaseException` class is created, then thrown. A thrown exception must be "caught" by code following the throwing code.

```
try {
 $result = eval($code);
} catch {\ParseException $exception) {
 // handle the exception
}
```

You should include an exception handler to catch exceptions from any method that throws them. Any uncaught exceptions will cause the script to cease execution.

## Error Suppression

You can disable error messages for a single expression by putting the error suppression operator `@` before the expression. For example:

```
$value = @(2 / 0);
```

Without the error suppression operator, the expression would normally halt execution of the script with a "divide by zero" error. As shown here, the expression does nothing, although in other cases, your program might be in an unknown state if you simply ignore errors that would otherwise cause the program to halt. The error suppression operator cannot trap parse errors, only the various types of runtime errors.

Of course, the downside to suppressing errors is that you won't know they're there. You're much better off handling potential error conditions properly; see "Triggering Errors" for an example.

To turn off error reporting entirely, use:

```
error_reporting(0);
```

This function ensures that, regardless of the errors PHP encounters while processing and executing your script, no errors will be sent to the client (except parse errors, which cannot be suppressed). Of course, it doesn't stop those errors from occurring. Better options for controlling which error messages are displayed in the client are shown in the section "Defining Error Handlers".

## Triggering Errors

You can throw an error from within a script with the `assertion()` function:

```
assert (mixed $expression [, mixed $message]);
```

The first parameter is the condition that must be `true` to not trigger the assertion; the second (optional) parameter is the message.

Triggering errors is useful when you're writing your own functions for sanity-checking the parameters. For example, here's a function that divides one number by another and throws an error if the second parameter is `0`:

```
function divider($a, $b) {
 assert($b != 0, '$b cannot be 0');

 return($a / $b);
}

echo divider(200, 3);
```

```
echo divider(10, 0);
66.666666666667
Fatal error: $b cannot be 0 in page.php on line 5
```

When a call to `assert()` is triggered, an `AssertionException`—an exception extending `ErrorException` with a severity of `E_ERROR`—is thrown. In some cases, you might want to throw an error of a type that extends `AssertionException`. You can do so by providing an exception as the message parameter instead of a string:

```
class DividerParameterException extends AssertionException { }

function divider($a, $b) {
 assert($b != 0, new DividerParameterException('$b cannot be 0'));

 return($a / $b);
}
```

## Defining Error Handlers

If you want better error control than just hiding any errors (and you usually do), you can supply PHP with an error handler. The error handler is called when a condition of any kind is encountered, and can do anything you want it to, from logging information to a file to pretty-printing the error message. The basic process is to create an error-handling function and register it with `set_error_handler()`.

The function you declare can take in either two or five parameters. The first two parameters are the error code and a string describing the error. The final three parameters, if your function accepts them, are the filename in which the error occurred, the line number at which the error occurred, and a copy of the active symbol table at the time the error occurred. Your error handler should check the current level of errors being reported with `error_reporting()` and act appropriately.

The call to `set_error_handler()` returns the current error handler. You can restore the previous error handler either by calling `set_error_handler()` with the returned value when your script is done with its own error handler, or by calling the `restore_error_handler()` function.

The following code shows how to use an error handler to format and print errors:

```
function displayError($error, $errorString, $filename, $line, $symbols)
{
 echo "<p>Error '<b>{$errorString}</b>' occurred.<br />";
 echo "-- in file '<i>{$filename}</i>', line $line.</p>";
}

set_error_handler('displayError');
$value = 4 / 0; // divide by zero error
```

```
<p>Error '<b>Division by zero</b>' occurred.
-- in file '<i>err-2.php</i>', line 8.</p>
```

### Logging in error handlers

PHP provides the built-in function `error_log()` to log errors to the myriad places where administrators like to put them:

```
error_log(message, type [, destination [, extra_headers ]]);
```

The first parameter is the error message. The second parameter specifies where the error is logged: a value of `0` logs the error via PHP's standard error-logging mechanism; a value of `1` emails the error to the *destination* address, optionally adding any *extra_headers* to the message; a value of `3` appends the error to the *destination* file.

To save an error using PHP's logging mechanism, call `error_log()` with a type of `0`. By changing the value of `error_log` in your *php.ini* file, you can change which file to log into. If you set `error_log` to `syslog`, the system logger is used instead. For example:

```
error_log('A connection to the database could not be opened.', 0);
```

To send an error via email, call `error_log()` with a type of `1`. The third parameter is the email address to which to send the error message, and an optional fourth parameter can be used to specify additional email headers. Here's how to send an error message by email:

```
error_log('A connection to the database could not be opened.',
 1, 'errors@php.net');
```

Finally, to log to a file, call `error_log()` with a type of `3`. The third parameter specifies the name of the file to log into:

```
error_log('A connection to the database could not be opened.',
 3, '/var/log/php_errors.log');
```

Example 17-1 shows an example of an error handler that writes logs into a file and rotates the logfile when it gets above 1 KB.

*Example 17-1. Log-rolling error handler*

```
function logRoller($error, $errorString) {
 $file = '/var/log/php_errors.log';

 if (filesize($file) > 1024) {
 rename($file, $file . (string) time());
 clearstatcache();
 }

 error_log($errorString, 3, $file);
}
```

```
set_error_handler('logRoller');

for ($i = 0; $i < 5000; $i++) {
 trigger_error(time() . ": Just an error, ma'am.\n");
}

restore_error_handler();
```

Generally, while you are working on a site, you will want errors shown directly in the pages in which they occur. However, once the site goes live, it doesn't make much sense to show internal error messages to visitors. A common approach is to use something like this in your *php.ini* file once your site goes live:

```
display_errors = Off
log_errors = On
error_log = /tmp/errors.log
```

This tells PHP to never show any errors, but instead to log them to the location specified by the `error_log` directive.

### Output buffering in error handlers

Using a combination of output buffering and an error handler, you can send different content to the user depending on whether various error conditions occur. For example, if a script needs to connect to a database, you can suppress output of the page until the script successfully connects to the database.

Example 17-2 shows the use of output buffering to delay output of a page until it has been generated successfully.

*Example 17-2. Output buffering to handle errors*

```
<html>
 <head>
 <title>Results!</title>
 </head>

 <body>
 <?php function handle_errors ($error, $message, $filename, $line) {
 ob_end_clean();
 echo "<b>{$message}</b><br/> in line {$line}<br/> of ";
 echo "<i>{$filename}</i></body></html>";

 exit;
 }

 set_error_handler('handle_errors');
 ob_start(); ?>
```

```
  <h1>Results!</h1>

  <p>Here are the results of your search:</p>

  <table border="1">
  <?php require_once('DB.php');
  $db = DB::connect('mysql://gnat:waldus@localhost/webdb');

  if (DB::iserror($db)) {
  die($db->getMessage());
  } ?>
  </table>
  </body>
</html>
```

In Example 17-2, after we start the `<body>` element, we register the error handler and begin output buffering. If we cannot connect to the database (or if anything else goes wrong in the subsequent PHP code), the heading and table are not displayed. Instead, the user sees only the error message. If no errors are raised by the PHP code, however, the user simply sees the HTML page.

## Manual Debugging

Once you get a few good years of development time under your belt, you should be able to get at least 75% of your debugging done on a purely visual basis. What of the other 25%, and the more difficult segments of code that you need to work through? You can tackle some of it by using a great code development environment like Zend Studio for Eclipse or Komodo. These advanced IDEs can help with syntax checking and some simple logical problems and warnings.

You can do the next level of debugging (again, you'll do most of this in the development environment) by echoing values out onto the screen. This will catch a lot of logic errors that may be dependent on the contents of variables. For example, how would you be able to easily see the value of the third iteration of a `for...next` loop? Consider the following code:

```
for ($j = 0; $j < 10; $j++) {
 $sample[] = $j * 12;
}
```

The easiest way is to interrupt the loop conditionally and echo out the value at the time; alternatively, you can wait until the loop is completed, as in this case since the loop is building an array. Here are some examples of how to determine that third iteration value (remember that array keys start with 0):

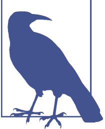
```
for ($j = 0; $j < 10; $j++) {
 $sample[] = $j * 12;

 if ($j == 2) {
 echo $sample[2];
 }
}
24
```

Here we are simply inserting a test (if statement) that will send a particular value to the browser when that condition is met. If you are having SQL syntax problems or failures, you can also echo the raw statement out to the browser and copy it into the SQL interface (*phpMyAdmin*, for example) and execute the code that way to see if any SQL error messages are returned.

If we want to see the entire array at the end of this loop, and what values it contains in each of its elements, we can still use the echo statement, but it would be tedious and cumbersome to write echo statements for each one. Rather, we can use the var_dump() function. The extra advantage of var_dump() is that it also tells us the data type of each element of the array. The output is not necessarily pretty, but it is informative. You can copy the output into a text editor and use it to clean up the look of the output.

Of course you can use echo and var_dump() in concert as the need arises. Here is an example of the raw var_dump() output:

```
for ($j = 0; $j < 10; $j++) {
 $sample[] = $j * 12;
}

var_dump($sample);
array(10) { [0] => int(0) [1] => int(12) [2] => int(24) [3] => int(36) [4] =>
int(48) [5] => int(60) [6] => int(72) [7] => int(84) [8] => int(96) [9] =>
int(108)}
```



There are two other ways to send simple data to the browser: the `print` language construct and the `print_r()` function. `print` is merely an alternative to `echo` (except that it returns a value of `1`), while `print_r()` sends information to the browser in a human-readable format. You can think of `print_r()` as an alternative to `var_dump()`, except that the output on an array would not send out each element's data type. The output for this code:

```
<?php
for ($j = 0; $j < 10; $j++) {
 $sample[] = $j * 12;
}
?>
<pre><?php print_r($sample); ?></pre>
```

would look like this (notice the formatting accomplished by the `<pre>` tags):

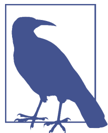
```
Array( [0] => 0 [1] => 12 [2] => 24 [3] => 36 [4] => 48
[5] => 60 [6] => 72 [7] => 84 [8] => 96 [9] => 108)
```

# Error Logs

You will find many helpful descriptions in the error logfile. As mentioned previously, you should be able to locate the file under the web server's installation folder in a folder called *logs*. You should make it part of your debugging routine to check this file for helpful clues as to what might be amiss. Here is just a sample of the verbosity of an error logfile:

```
[20-Apr-2012 15:10:55] PHP Notice: Undefined variable: size in C:\Program Files
(x86)
[20-Apr-2012 15:10:55] PHP Notice: Undefined index: p in C:\Program Files
(x86)\Zend
[20-Apr-2012 15:10:55] PHP Warning: number_format() expects parameter 1 to be
double
[20-Apr-2012 15:10:55] PHP Warning: number_format() expects parameter 1 to be
double
[20-Apr-2012 15:10:55] PHP Deprecated: Function split() is deprecated in
C:\Program
[20-Apr-2012 15:10:55] PHP Deprecated: Function split() is deprecated in
C:\Program
[26-Apr-2012 13:18:38] PHP Fatal error: Maximum execution time of 30 seconds
exceeded
```

As you can see, there are a few different types of errors being reported here—notices, warnings, deprecation notices, and a fatal error—with their respective timestamps, file locations, and the line on which the error occurred.



Depending on your environment, some commercial server space providers do not grant access for security reasons, so you may not have access to the logfile. Be sure to select a production provider that grants you access to the logfile. Additionally, note that the log can be and often is moved outside the web server's installation folder. On Ubuntu, for example, the default is in */var/logs/apache2/*.log*. Check the web server's configuration if you can't locate the log.

## IDE Debugging

For more complex debugging issues, you would be best served to use a debugger that can be found in a good integrated development environment (IDE). We will be showing you a debug session example with Zend Studio for Eclipse. Other IDEs, like Komodo and PhpED, have built-in debuggers, so they can also be used for this purpose.

Zend Studio has an entire Debug Perspective setup for debugging purposes, as shown in Figure 17-1.

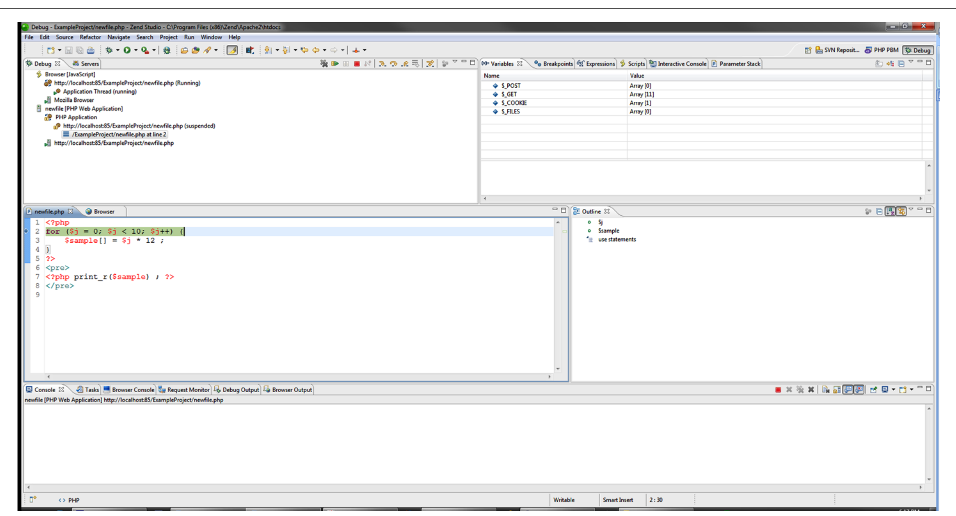



*Figure 17-1. The default Debug Perspective in Zend Studio*

To get your bearings with this debugger, open the Run menu. It shows all the options you can try when in the debug process—stepping into and over code segments, running to a cursor location, restarting the session from the beginning, and just simply letting your code run until it fails or ends, to name a few.

In Zend Studio for Eclipse, you can even debug JavaScript code with the right setup!

Check the many debug views in this product as well; you can watch the variables (both superglobals and user-defined) as they change over the course of code execution.

Breakpoints can also be set (and suspended) anywhere in the PHP code, so you can run to a certain location in your code and view the overall situation at that particular moment. Two other handy views are Debug Output and Browser Output, which present the output of the code as the debugger runs through it. The Debug Output view presents the output in the format you would see if you had selected View Source in a browser, showing the raw HTML as it is being generated. The Browser Output view displays the executing code as it would appear in a browser. The neat thing about both of these views is that they're populated as the code executes, so if you are stopped at a breakpoint halfway through your code file, they display only the information generated up to that point.

Figure 17-2 shows an example of the sample code from earlier in this chapter (with an added `echo` statement within the `for` loop so that you can see the output as it is being created) run in the debugger. The two main variables (`$j` and `$sample`) are being tracked in the Expressions view, and the Browser Output and Debug Output views display their content at a stopped location in the code.

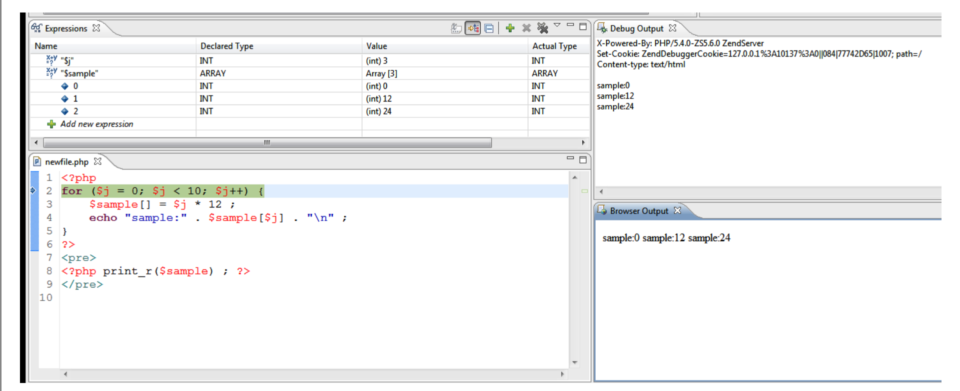



*Figure 17-2. The debugger in action with watch expressions defined*

## Additional Debugging Techniques

There are more advanced techniques that can be used for debugging, but they are beyond the scope of this chapter. Two such techniques are profiling and unit testing. If you have a large web system that requires a lot of server resources, you should certainly look into the benefits of these two techniques, as they can make your code base more fault-tolerant and efficient.

## What's Next

Up next, we'll explore writing Unix and Windows cross-platform scripts, and provide a brief introduction to hosting your PHP sites on Windows servers.

# CHAPTER 18
# PHP on Disparate Platforms

There are many reasons to use PHP on a Windows system, but the most common is that you want to develop web applications on your Windows desktop. PHP development on Windows is just as doable these days as it is on a Unix platform. PHP plays very well on Windows, and PHP's supporting cast of server and add-on tools is just as Windows-friendly. Having a PHP system working on any of its supported platforms is simply a matter of preference. Setting up and developing with a PHP environment on Windows is very easy to do, as PHP is extremely cross-platform friendly, and installation and configuration are becoming simpler all the time. The relatively recent appearance on the market of Zend Server CE (Community Edition) for multiple platforms has been a wonderful help in establishing a common installation platform on all the major operating systems.

## Writing Portable Code for Windows and Unix

One of the main reasons for running PHP on Windows is to develop locally before deploying in a production environment. As many production servers are Unix-based, it is important to consider writing your applications so that they can operate on any operating platform with minimal fuss.

Potential problem areas include applications that rely on external libraries, use native file I/O and security features, access system devices, fork or spawn threads, communicate via sockets, use signals, spawn external executables, or generate platform-specific graphical user interfaces.

The good news is that cross-platform development has been a major goal as PHP has evolved. For the most part, PHP scripts should be ported from Windows to Unix with few problems. However, there are instances where you can run into trouble when porting your scripts. For instance, some functions that were implemented very early

in the life of PHP had to be mimicked for use under Windows. Other functions may be specific to the web server under which PHP is running.

## Determining the Platform

To design with portability in mind, you may want to first test for the platform on which the script is running. PHP defines the constant `PHP_OS`, which contains the name of the operating system on which the PHP parser is executing. Possible values for the `PHP_OS` constant include `"HP-UX"`, `"Darwin"` (macOS), `"Linux"`, `"SunOS"`, `"WIN32"`, and `"WINNT"`. You may also want to consider the `php_uname()` built-in function; it returns even more operating system information.

The following code shows how to test for a Windows platform:

```
if (PHP_OS == 'WIN32' || PHP_OS == 'WINNT') {
 echo "You are on a Windows System";
}
else {
 // some other platform
 echo "You are NOT on a Windows System";
}
```

Here is an example of the output for the `php_uname()` function as executed on a Windows 7 i5 laptop:

```
Windows NT PALADIN-LAPTO 6.1 build 7601 (Windows 7 Home Premium Edition Service
Pack 1) i586
```

## Handling Paths Across Platforms

PHP understands the use of backward or forward slashes on Windows platforms, and can even handle paths that use both. PHP also recognizes the forward slash when accessing Windows Universal Naming Convention (UNC) paths (i.e., *//machine_name/path/to/file*). For example, these two lines are equivalent:

```
$fh = fopen("c:/planning/schedule.txt", 'r');
$fh = fopen("c:\\planning\\schedule.txt", 'r');
```

## Navigating the Server Environment

The constant superglobal array `$_SERVER` provides server and execution environment information. Here is a partial listing of what it contains:

```
["PROCESSOR_ARCHITECTURE"] => string(3) "x86"
["PROCESSOR_ARCHITEW6432"] => string(5) "AMD64"
["PROCESSOR_IDENTIFIER"] => string(50) "Intel64 Family 6 Model 42 Stepping 7,
GenuineIntel"
["PROCESSOR_LEVEL"] => string(1) "6"
["PROCESSOR_REVISION"] => string(4) "2a07"
["ProgramData"] => string(14) "C:\ProgramData"
```

```
["ProgramFiles"] => string(22) "C:\Program Files (x86)"
["ProgramFiles(x86)"] => string(22) "C:\Program Files (x86)"
["ProgramW6432"] => string(16) "C:\Program Files"
["PSModulePath"] => string(51)
 "C:\Windows\system32\WindowsPowerShell\v1.0\Modules\"
["PUBLIC"] => string(15) "C:\Users\Public"
["SystemDrive"] => string(2) "C:"
["SystemRoot"] => string(10) "C:\Windows"
```

To see all of the information available within this global array, check out its documentation.

Once you know the specific information you are looking for, you can request it directly like so:

```
echo "The windows Dir is: {$_SERVER['WINDIR']}";
The windows Dir is: C:\Windows
```

## Sending Mail

On Unix systems, you can configure the `mail()` function to use *sendmail* or *Qmail* to send messages. When running PHP under Windows, you can use sendmail by installing it and setting the `sendmail_path` in *php.ini* to point at the executable. It is likely more convenient, however, to simply point the Windows version of PHP to an SMTP server that will accept you as a known mail client:

```
[mail function]
SMTP = mail.example.com ;URL or IP number to known mail server
sendmail_from = test@example.com
```

For an even simpler email solution, you can use the comprehensive PHPMailer library, which not only simplifies sending email from Windows platforms but is completely cross-platform and works on Unix systems as well.

```
$mail = new PHPMailer(true);

try {
 //Server settings
 $mail->SMTPDebug = SMTP::DEBUG_SERVER;
 $mail->isSMTP();
 $mail->Host = 'smtp1.example.com';
 $mail->SMTPSecure = PHPMailer::ENCRYPTION_STARTTLS;
 $mail->Port = 587;

 $mail->setFrom('from@example.com', 'Mailer');
 $mail->addAddress('joe@example.net');

 $mail->isHTML(false);
 $mail->Subject = 'Here is the subject';
 $mail->Body = 'And here is the body.';

 $mail->send();
```

```
 echo 'Message has been sent';
} catch (Exception $e) {
 echo "Message could not be sent. Mailer Error: {$mail->ErrorInfo}";
}
```

## End-of-Line Handling

Windows text files have lines that end in `\r\n`, whereas Unix text files have lines that end in `\n`. PHP processes files in binary mode, so it does not automatically convert from Windows line terminators to their Unix equivalents.

PHP on Windows sets the standard output, standard input, and standard error file handlers to binary mode and thus does not do any translations for you. This is important for handling the binary input often associated with `POST` messages from web servers.

Your program's output goes to standard output, and you will have to specifically place Windows line terminators in the output stream if you want them there. One way to handle this is to define an end-of-line (EOL) constant and output functions that use it:

```
if (PHP_OS == "WIN32" || PHP_OS == "WINNT") {
 define('EOL', "\r\n");
}
else if (PHP_OS == "Linux") {
 define('EOL', "\n");
}
else {
 define('EOL', "\n");
}

function ln($out) {
 echo $out . EOL;
}

ln("this line will have the server platform's EOL character");
```

A simpler way of handling this is through the `PHP_EOL` constant, which automatically determines the end-of-line string for the server's system. (Note, however, that the server system and the desired EOL marker may not be the same in all cases.)

```
function ln($out) {
 echo $out . PHP_EOL;
}
```

## End-of-File Handling

Windows text files end in a Control-Z (`\x1A`), whereas Unix stores file-length information separately from the file's data. PHP recognizes the end-of-file (EOF) character

of the platform on which it is running; thus, the `feof()` function works for reading Windows text files.

## Using External Commands

PHP uses the default command shell of Windows for process manipulation. Only rudimentary Unix shell redirections and pipes are available under Windows (e.g., separate redirection of standard output and standard error is not possible), and the quoting rules are entirely different. The Windows shell does not *glob* (i.e., replace arguments containing wildcard markers with the list of files that match the wildcards). Whereas on Unix you can say `system("someprog php*.php")`, on Windows you must build the list of filenames yourself using `opendir()` and `readdir()`.

## Accessing Platform-Specific Extensions

There are currently well over 80 extensions for PHP covering a wide range of services and functionality. Only about half of these are available for both Windows and Unix platforms. Only a handful of extensions, such as the COM, .NET, and IIS extensions, are specific to Windows. If an extension you use in your scripts is not currently available under Windows, you need to either port that extension or convert your scripts to use an extension that is available under Windows.

In some cases, some functions are not available under Windows even though the module as a whole is available.

Windows PHP does not support signal handling, forking, or multithreaded scripts. A Unix PHP script that uses these features cannot be ported to Windows. Instead, you should rewrite the script to not depend on those features.

# Interfacing with COM

COM allows you to control other Windows applications. You can send file data to Excel, have it draw a graph, and export the graph as a GIF image. You could also use Word to format the information you receive from a form and then print an invoice as a record. After a brief introduction to COM terminology, this section shows you how to interact with both Word and Excel.

## Background

COM is a remote procedure call (RPC) mechanism with a few object-oriented features. It provides a way for the calling program (the *controller*) to talk to another program (the COM server, or *object*), regardless of where it resides. If the underlying code is local to the same machine, the technology is COM; if it's remote, it's Distributed COM (DCOM). If the underlying code is a dynamic link library (DLL), and

the code is loaded into the same process space, the COM server is referred to as an in-process, or *inproc*, server. If the code is a complete application that runs in its own process space, it's known as an out-of-process server, or *local server application*.

Object Linking and Embedding (OLE) is the overall marketing term for Microsoft's early technology that allowed one object to embed another object. For instance, you could embed an Excel spreadsheet in a Word document. Developed during the days of Windows 3.1, OLE 1.0 was limited because it used a technology known as Dynamic Data Exchange (DDE) to communicate between programs. DDE wasn't very powerful, and if you wanted to edit an Excel spreadsheet embedded in a Word file, Excel had to be open and running.

OLE 2.0 replaced DDE with COM as the underlying communication method. Using OLE 2.0, you can now paste an Excel spreadsheet right into a Word document and edit the Excel data inline. Using OLE 2.0, the controller can pass complex messages to the COM server. For our examples, the controller will be our PHP script, and the COM server will be one of the typical MS Office applications. In the following sections, we will provide some tools for approaching this type of integration.

To whet your appetite and show you how powerful COM can be, Example 18-1 shows how you would start Word and add "Hello World" to the initially empty document.

*Example 18-1. Creating a Word file in PHP (word_com_sample.php)*

```
// starting word
$word = new COM("word.application") or die("Unable to start Word app");
echo "Found and Loaded Word, version {$word->Version}\n";

//open an empty document
$word->Documents->add();

//do some weird stuff
$word->Selection->typeText("Hello World");
$word->Documents[1]->saveAs("c:/php_com_test.doc");

//closing word
$word->quit();

//free the object
$word = null;

echo "all done!";
```

This code file will have to be executed from the command line in order to work correctly, as shown in Figure 18-1. Once you see the output string of `all done!`, you can look for the file in the Save As folder and open it with Word to see what it looks like.

```
Administrator: C:\Windows\system32\cmd.exe

C:\Program Files (x86)\Zend\Apache2\htdocs\current\code\ch14>php testing_code_1.
php
Found and Loaded Word, version 12.0
all done !

C:\Program Files (x86)\Zend\Apache2\htdocs\current\code\ch14>
```

*Figure 18-1. Calling the Word sample in the command window*

The actual Word file should look something like Figure 18-2.

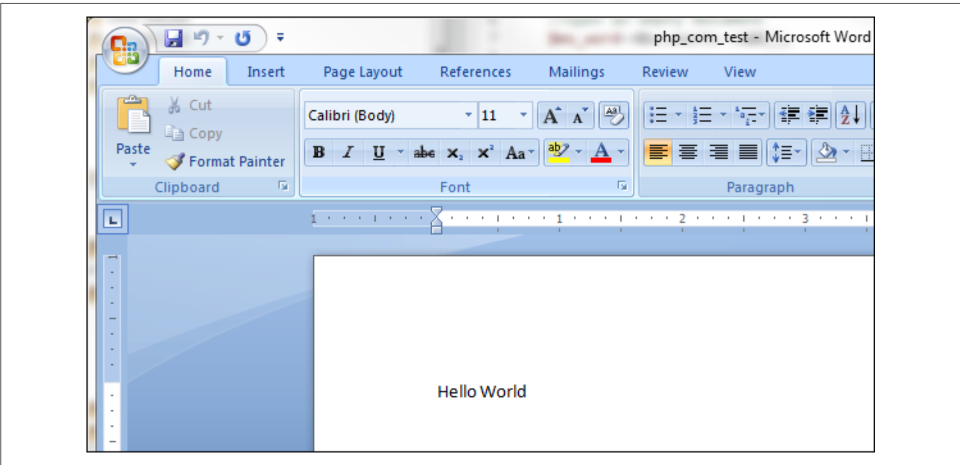



*Figure 18-2. The Word file as created by PHP*

## PHP Functions

PHP provides an interface into COM through a small set of function calls. Most of these are low-level functions that require detailed knowledge of COM that is beyond the scope of this chapter. An object of the `COM` class represents a connection to a COM server:

```
$word = new COM("word.application") or die("Unable to start Word app");
```

For most OLE automation, the most difficult task is converting a Visual Basic method call to something similar in PHP. For instance, this is VBScript to insert text into a Word document:

```
Selection.TypeText Text := "This is a test"
```

The same line in PHP is:

```
$word->Selection->typetext("This is a test");
```

## API Specifications

To determine object hierarchy and parameters for a product such as Word, you might visit the Microsoft developer site and search for the specification for the Word object that interests you. Another alternative is to use both Microsoft's online VB scripting help and Word's supported macro language. Using these together will help you understand the order of parameters, as well as the desired values for a given task.

# APPENDIX
# Function Reference

This appendix describes the functions available in the built-in PHP extensions. These are the extensions that PHP is built with if you provide no `--with` or `--enable` options to `configure`, and they cannot be removed via configuration options.

For each function, we've provided the function signature, showing the data types of the various arguments and which are mandatory or optional, as well as a brief description of the side effects, errors, and returned data structures.

## PHP Functions by Category

This section gives a list of functions provided by PHP's built-in extensions, grouped by extension category.

### Arrays

`array_change_key_case`
`array_chunk`
`array_combine`
`array_count_values`
`array_diff`
`array_diff_assoc`
`array_diff_key`
`array_diff_uassoc`
`array_diff_ukey`
`array_fill`
`array_fill_keys`
`array_filter`
`array_flip`
`array_intersect`
`array_intersect_assoc`
`array_intersect_key`
`array_intersect_uassoc`
`array_intersect_ukey`
`array_key_exists`
`array_keys`
`array_map`
`array_merge`

- array_merge_recursive
- array_multisort
- array_pad
- array_pop
- array_product
- array_push
- array_rand
- array_reduce
- array_replace
- array_replace_recursive
- array_reverse
- array_search
- array_shift
- array_slice
- array_splice
- array_sum
- array_udiff
- array_udiff_assoc
- array_udiff_uassoc
- array_uintersect
- array_uintersect_assoc
- array_uintersect_uassoc
- array_unique
- array_unshift
- array_values
- array_walk
- array_walk_recursive
- arsort
- asort
- compact
- count
- current
- each
- end
- extract
- in_array
- is_countable
- key
- krsort
- ksort
- list
- natcasesort
- natsort
- next
- prev
- range
- reset
- rsort
- shuffle
- sort
- uasort
- uksort
- usort

## Classes and Objects

- class_alias
- class_exists
- get_called_class
- get_class
- get_class_methods
- get_class_vars
- get_declared_classes
- get_declared_interfaces
- get_declared_traits
- get_object_vars

get_parent_class
interface_exists
is_a
is_subclass_of
method_exists
property_exists
trait_exists

## Data Filtering

filter_has_var
filter_id
filter_input_array
filter_var
filter_input
filter_list
filter_var_array

## Date and Time

checkdate
date
date_default_timezone_get
date_default_timezone_set
date_parse
date_parse_from_format
date_sun_info
date_sunrise
date_sunset
getdate
gettimeofday
gmdate
gmmktime
gmstrftime
hrtime
idate
localtime
microtime
mktime
strftime
strptime
strtotime
time
timezone_name_from_abbr
timezone_version_get

## Directories

chdir
chroot
closedir
dir
getcwd
opendir
readdir
rewinddir
scandir

## Errors and Logging

debug_backtrace
debug_print_backtrace
error_clear_last
error_get_last
error_log
error_reporting
restore_error_handler
restore_exception_handler
set_error_handler
set_exception_handler
trigger_error

## Filesystem

basename
chgrp
chmod
chown
clearstatcache
copy
dirname
disk_free_space
disk_total_space
fclose
feof
fflush
fgetc
fgetcsv
fgets
fgetss
file
file_exists
file_get_contents
file_put_contents
fileatime
filectime
filegroup
fileinode
filemtime
fileowner
fileperms
filesize
filetype
flock
fnmatch
fopen
fpassthru
fputcsv
fread
fscanf
fseek
fstat
ftell
ftruncate
fwrite
glob
is_dir
is_executable
is_file
is_link
is_readable
is_uploaded_file
is_writable
lchgrp

lchown
link
linkinfo
lstat
mkdir
move_uploaded_file
parse_ini_file
parse_ini_string
pathinfo
pclose
popen
readfile
readlink
realpath_cache_get
realpath_cache_size
realpath
rename
rewind
rmdir
stat
symlink
tempnam
tmpfile
touch
umask
unlink

## Functions

call_user_func
call_user_func_array
create_function
forward_static_call
forward_static_call_array
func_get_arg
func_get_args
func_num_args
function_exists
get_defined_functions
register_shutdown_function
register_tick_function
unregister_tick_function

## Mail

mail

## Math

abs
acos
acosh
asin
asinh
atan2
atan
atanh
base_convert
bindec
ceil
cos
cosh
decbin

dechex
decoct
deg2rad
exp
expm1
floor
fmod
getrandmax
hexdec
hypot
is_finite
is_infinite
is_nan
lcg_value
log10
log1p
log
max
min
mt_getrandmax
mt_rand
mt_srand
octdec
pi
pow
rad2deg
rand
random_int
round
sin
sinh
sqrt
srand
tan
tanh

## Miscellaneous Functions

connection_aborted
connection_status
constant
define
defined
get_browser
highlight_file
highlight_string
ignore_user_abort
pack
php_strip_whitespace
sleep
sys_getloadavg
time_nanosleep
time_sleep_until
uniqid
unpack
usleep

## Network

checkdnsrr
closelog
fsockopen
gethostbyaddr
gethostbyname
gethostbynamel

gethostname
getmxrr
getprotobyname
getprotobynumber
getservbyname
getservbyport
header
header_remove
headers_list
headers_sent
inet_ntop
inet_pton
ip2long
long2ip
openlog
pfsockopen
setcookie
setrawcookie
syslog

## Output Buffering

flush
ob_clean
ob_end_clean
ob_end_flush
ob_flush
ob_get_clean
ob_get_contents
ob_get_flush
ob_get_length
ob_get_level
ob_get_status
ob_gzhandler
ob_implicit_flush
ob_list_handlers
ob_start
output_add_rewrite_var
output_reset_rewrite_vars

## PHP Language Tokenizer

token_get_all
token_name

## PHP Options/Info

assert_options
assert
extension_loaded
gc_collect_cycles
gc_disable
gc_enable
gc_enabled
get_cfg_var
get_current_user
get_defined_constants
get_extension_funcs
get_include_path
get_included_files
get_loaded_extensions
getenv
getlastmod

getmygid
getmyinode
getmypid
getmyuid
getopt
getrusage
ini_get_all
ini_get
ini_restore
ini_set
memory_get_peak_usage
memory_get_usage
php_ini_loaded_file
php_ini_scanned_files
php_logo_guid
php_sapi_name
php_uname
phpcredits
phpinfo
phpversion
putenv
set_include_path
set_time_limit
sys_get_temp_dir
version_compare
zend_logo_guid
zend_thread_id
zend_version

## Program Execution

escapeshellarg
escapeshellcmd
exec
passthru
proc_close
proc_get_status
proc_nice
proc_open
proc_terminate
shell_exec
system

## Session Handling

session_cache_expire
session_cache_limiter
session_decode
session_destroy
session_encode
session_get_cookie_params
session_id
session_module_name
session_name
session_regenerate_id
session_register_shutdown
session_save_path
session_set_cookie_params
session_set_save_handler
session_start
session_status
session_unset
session_write_close

## Streams

stream_bucket_append
stream_bucket_make_writeable
stream_bucket_new
stream_bucket_prepend
stream_context_create
stream_context_get_default
stream_context_get_options
stream_context_get_params
stream_context_set_default
stream_context_set_option
stream_context_set_params
stream_copy_to_stream
stream_encoding
stream_filter_append
stream_filter_prepend
stream_filter_register
stream_filter_remove
stream_get_contents
stream_get_filters
stream_get_line
stream_get_meta_data
stream_get_transports
stream_get_wrappers
stream_is_local
stream_notification_callback
stream_resolve_include_path
stream_select
stream_set_blocking
stream_set_chunk_size
stream_set_read_buffer
stream_set_timeout
stream_set_write_buffer
stream_socket_accept
stream_socket_client
stream_socket_enable_crypto
stream_socket_get_name
stream_socket_pair
stream_socket_recvfrom
stream_socket_sendto
stream_socket_server
stream_socket_shutdown
stream_supports_lock
stream_wrapper_register
stream_wrapper_restore
stream_wrapper_unregister

## Strings

addcslashes
addslashes
bin2hex
chr
chunk_split
convert_cyr_string
convert_uudecode
convert_uuencode
count_chars
crc32
crypt
echo
explode
fprintf
get_html_translation_table
hebrev

hex2bin

html_entity_decode

htmlentities

htmlspecialchars

htmlspecialchars_decode

implode

lcfirst

levenshtein

localeconv

ltrim

md5

md5_file

metaphone

nl_langinfo

nl2br

number_format

ord

parse_str

printf

quoted_printable_decode

quoted_printable_encode

quotemeta

random_bytes

rtrim

setlocale

sha1

sha1_file

similar_text

soundex

sprintf

sscanf

str_getcsv

str_ireplace

str_pad

str_repeat

str_replace

str_rot13

str_shuffle

str_split

str_word_count

strcasecmp

strcmp

strcoll

strcspn

strip_tags

stripcslashes

stripos

stripslashes

stristr

strlen

strnatcasecmp

strnatcmp

strncasecmp

strncmp

strpbrk

strpos

strrchr

strrev

strripos

strrpos

strspn

strstr

strtok

strtolower

strtoupper

strtr

substr

substr_compare

substr_count
substr_replace
trim
ucfirst
ucwords
vfprintf
vprintf
vsprintf
wordwrap

## URLs

base64_decode
base64_encode
get_headers
get_meta_tags
http_build_query
parse_url
rawurldecode
rawurlencode
urldecode
urlencode

## Variables

debug_zval_dump
empty
floatval
get_defined_vars
get_resource_type
gettype
intval
is_array
is_bool
is_callable
is_float
is_int
is_null
is_numeric
is_object
is_resource
is_scalar
is_string
isset
print_r
serialize
settype
strval
unserialize
unset
var_dump
var_export

## Zlib

deflate_add
deflate_init
inflate_add
inflate_init

## Alphabetical Listing of PHP Functions

**abs.** `int abs(int` *number*`)` `float abs(float` *number*`)`

Returns the absolute value of *number* in the same type (float or integer) as the argument.

**acos.** `float acos(float` *value*`)`

Returns the arc cosine of *value* in radians.

**acosh.** `float acosh(float` *value*`)`

Returns the inverse hyberbolic cosine of *value*.

**addcslashes.** `string addcslashes(string` *string*`, string` *characters*`)`

Returns escaped instances of *characters* in *string* by adding a backslash before them. You can specify ranges of characters by separating them with two periods—for example, to escape characters between `a` and `q`, use `"a..q"`. Multiple characters and ranges can be specified in *characters*. The `addcslashes()` function is the inverse of `stripcslashes()`.

**addslashes.** `string addslashes(string` *string*`)`

Returns escaped characters in *string* that have special meaning in SQL database queries. Single quotes (`''`), double quotes (`""`), backslashes (`\`), and the NUL-byte (`\0`) are escaped. The `stripslashes()` function is the inverse for this function.

**array_change_key_case.** `array array_change_key_case(array` *array*`[, CASE_UPPER|CASE_LOWER])`

Returns an array whose elements' keys are changed to all uppercase or all lowercase. Numeric indices are unchanged. If the optional case parameter is left off, the keys are changed to lowercase.

**array_chunk.** `array array_chunk(array` *array*`, int` *size*`[, int` *preserve_keys*`])`

Splits *array* into a series of arrays, each containing *size* elements, and returns them in an array. If *preserve_keys* is `true` (default is `false`), the original keys are preserved in the resulting arrays; otherwise, the values are ordered with numeric indices starting at 0.

**array_combine.** `array array_combine(array` *keys*`, array` *values*`)`

Returns an array created by using each element in the *keys* array as the key and the element in the *values* array as the value. If either array has no elements, if the number of elements in each array differs, or if an element exists in one array but not in the other, `false` is returned.

**array_count_values.** `array array_count_values(array` *array*`)`

Returns an array whose elements' keys are the input array's values. The value of each key is the number of times that key appears in the input array as a value.

**array_diff.** `array array_diff(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array that contains all of the values from the first array that are not present in any of the other provided arrays. The keys of the values are preserved.

**array_diff_assoc.** `array array_diff_assoc(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array containing all the values in *array1* that are not present in any of the other provided arrays. Unlike in `array_diff()`, both the keys and values must match to be considered identical. The keys of the values are preserved.

**array_diff_key.** `array array_diff_key(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array that contains all of the values from the first array whose keys are not present in any of the other provided arrays. The keys of the values are preserved.

**array_diff_uassoc.** `array array_diff_uassoc(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], callable` *function*`)`

Returns an array containing all the values in *array1* that are not present in any of the other provided arrays. Unlike in `array_diff()`, both the keys and values must match to be considered identical. The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the values to compare. It should return an integer less than 0 if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than 0 if the first argument is greater than the second. The keys of the values are preserved.

**array_diff_ukey.** `array array_diff_ukey(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], callable` *function*`)`

Returns an array containing all the values in *array1* whose keys are not present in any of the other provided arrays. The function *function* is used to compare the keys of the elements for equality. The function is called with two parameters—the keys to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_fill.** `array array_fill(int` *start*`, int` *count*`, mixed` *value*`)`

Returns an array with *count* elements with the value *value*. Numeric indices are used, starting at *start* and counting upward by 1 for each element. If *count* is zero or less, an error is produced.

**array_fill_keys.** `array array_fill_keys(array` *keys*`, mixed` *value*`)`

Returns an array containing values for each item in *keys*, using the elements in *keys* for each element's key and *value* for each element's value.

**array_filter.** `array array_filter(array` *array*`, mixed` *callback*`)`

Creates an array containing all values from the original array for which the given callback function returns `true`. If the input array is an associative array, the keys are preserved. For example:

```
function isBig($inValue)
{
 return($inValue > 10);
}

$array = array(7, 8, 9, 10, 11, 12, 13, 14);
$newArray = array_filter($array, "isBig"); // contains (11, 12, 13, 14)
```

**array_flip.** `array array_flip(array` *array*`)`

Returns an array in which the elements' keys are the original array's values, and vice versa. If multiple values are found, the last one encountered is retained. If any of the values in the original array are any type except strings and integers, `array_flip()` will issue a warning, and the key-value pair in question will not be included in the result. `array_flip()` returns `NULL` on failure.

**array_intersect.** `array array_intersect(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array consisting of every element in *array1* that also exists in every other array.

**array_intersect_assoc.** `array array_intersect_assoc(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array containing all the values present in all of the given arrays. Unlike in `array_intersect()`, both the keys and values must match to be considered identical. The keys of the values are preserved.

**array_intersect_key.** `array array_intersect_key(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array consisting of every element in *array1* whose keys also exist in every other array.

**array_intersect_uassoc.** `array array_intersect_uassoc(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], callable` *function*`)`

Returns an array containing all the values present in all of the given arrays.

The function *function* is used to compare the keys of the elements for equality. The function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_intersect_ukey.** `array array_intersect_ukey(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], callable` *function*`)`

Returns an array consisting of every element in *array1* whose keys also exist in every other array.

The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the keys to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second.

**array_key_exists.** `bool array_key_exists(mixed` *key*`, array` *array*`)`

Returns `true` if *array* contains a key with the value *key*. If no such key is available, returns `false`.

**array_keys.** `array array_keys(array` *array*`[, mixed` *value*`[, bool` *strict*`]])`

Returns an array containing all of the keys in the given array. If the second parameter is provided, only keys whose values match *value* are returned in the array. If *strict* is specified and is `true`, a matched element is returned only when it is of the same type and value as *value*.

**array_map.** `array array_map(mixed` *callback*`, array` *array1*`[, ... array` *arrayN*`])`

Creates an array by applying the callback function referenced in the first parameter to the remaining parameters (provided arrays); the callback function should take as parameters a number of values equal to the number of arrays passed into `array_map()`. For example:

```
function multiply($inOne, $inTwo) {
 return $inOne * $inTwo;
}
$first = (1, 2, 3, 4);
$second = (10, 9, 8, 7);
$array = array_map("multiply", $first, $second); // contains (10, 18, 24, 28)
```

**array_merge.** `array array_merge(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array created by appending the elements of every provided array to the previous. If any array has a value with the same string key, the last value encountered for the key is returned in the array; any elements with identical numeric keys are inserted into the resulting array.

**array_merge_recursive.** `array array_merge_recursive(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Like `array_merge()`, creates and returns an array by appending each input array to the previous. However, unlike in `array_merge()`, when multiple elements have the same string key, an array containing each value is inserted into the resulting array.

**array_multisort.** `bool array_multisort(array` *array1*`[, SORT_ASC|SORT_DESC [, SORT_REGULAR|SORT_NUMERIC|SORT_STRING]] [, array` *array2*`[, SORT_ASC|SORT_DESC [, SORT_REGULAR|SORT_NUMERIC|SORT_STRING]], ...])`

Used to sort several arrays simultaneously, or to sort a multidimensional array in one or more dimensions. The input arrays are treated as columns in a table to be sorted by rows—the first array is the primary sort. Any values that compare the same according to that sort are sorted by the next input array, and so on.

The first argument is an array; following that, each argument may be an array or one of the following order flags (the order flags are used to change the default order of the sort):

| | |
|---|---|
| `SORT_ASC` (default) | Sort in ascending order |
| `SORT_DESC` | Sort in descending order |

After that, a sorting type from the following list can be specified:

| | |
|---|---|
| `SORT_REGULAR` (default) | Compare items normally |
| `SORT_NUMERIC` | Compare items numerically |
| `SORT_STRING` | Compare items as strings |

The sorting flags apply only to the immediately preceding array, and they revert to `SORT_ASC` and `SORT_REGULAR` before each new array argument.

This function returns `true` if the operation was successful and `false` otherwise.

**array_pad.** `array array_pad(array` *input*`, int` *size*`[, mixed` *padding*`])`

Returns a copy of the input array padded to the length specified by *size*. Any new elements added to the array have the value of the optional third value. You can add elements to the beginning of the array by specifying a negative size—in this case, the new size of the array is the absolute value of the size.

If the array already has the specified number of elements or more, no padding takes place and an exact copy of the original array is returned.

**array_pop.** `mixed array_pop(array &`*stack*`)`

Removes the last value from the given array and returns it. If the array is empty (or the argument is not an array), returns `NULL`. Note that the array pointer is reset on the provided array.

**array_product.** `number array_product(array` *array*`)`

Returns the product of every element in *array*. If each value in *array* is an integer, the resulting product is an integer; otherwise, the resulting product is a float.

**array_push.** `int array_push(array &`*array*`, mixed` *value1*`[, ... mixed` *valueN*`])`

Adds the given values to the end of the array specified in the first argument and returns the new size of the array. Performs the same function as calling `$array[] = $value` for each of the values in the list.

**array_rand.** `mixed array_rand(array` *array*`[, int` *count*`])`

Picks a random element from the given array. The second (optional) parameter can be given to specify a number of elements to pick and return. If more than one element is returned, an array of keys is returned, rather than the element's value.

**array_reduce.** `mixed array_reduce(array` *array*`, mixed` *callback*`[, int` *initial*`])`

Returns a value derived by iteratively calling the given callback function with pairs of values from the array. If the third parameter is supplied, it, along with the first element in the array, is passed to the callback function for the initial call.

**array_replace.** `array array_replace(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array created by replacing values in *array1* with values from the other arrays. Elements in *array1* with keys matching in the replacement arrays are replaced with the values of those elements.

If multiple replacement arrays are provided, they are processed in order. Any elements in *array1* whose keys do not match any keys in the replacement arrays are preserved.

**array_replace_recursive.** `array array_replace_recursive(array` *array1*`, array` *array2*`[, ... array` *arrayN*`])`

Returns an array created by replacing values in *array1* with values from the other arrays. Elements in *array1* with keys matching in the replacement arrays are replaced with the values of those elements.

If the value in both *array1* and a replacement array for a particular key are arrays, those values in those arrays are recursively merged using the same process.

If multiple replacement arrays are provided, they are processed in order. Any elements in *array1* whose keys do not match any keys in the replacement arrays are preserved.

**array_reverse.** `array array_reverse(array` *array*`[, bool` *preserve_keys*`])`

Returns an array containing the same elements as the input array, but whose order is reversed. If `preserve_keys` is set to `true`, then numeric keys are preserved. Non-numeric keys are not affected by this parameter and are always preserved.

**array_search.** `mixed array_search(mixed` *value*`, array` *array*`[, bool` *strict*`])`

Performs a search for a value in an array, as with `in_array()`. If the value is found, the key of the matching element is returned; `NULL` is returned if the value is not

found. If *strict* is specified and is `true`, a matched element is returned only when it is of the same type and value as *value*.

**array_shift.** `mixed array_shift(array` *stack*`)`

Similar to `array_pop()`, but instead of removing and returning the last element in the array, it removes and returns the first element in the array. If the array is empty, or if the argument is not an array, returns `NULL`.

**array_slice.** `array array_slice(array` *array*`, int` *offset*`[, int` *length*`][, bool keepkeys])`

Returns an array containing a set of elements pulled from the given array. If *offset* is a positive number, elements starting from that index onward are used; if *offset* is a negative number, elements starting that many elements from the end of the array are used. If the third argument is provided and is a positive number, that many elements are returned; if negative, the sequence stops that many elements from the end of the array. If the third argument is omitted, the sequence returned contains all elements from the offset to the end of the array. If `keepkeys`, the fourth argument, is `true`, then the order of numeric keys will be preserved; otherwise, they will be renumbered and resorted.

**array_splice.** `array array_splice(array` *array*`, int` *offset*`[, int` *length*`[, array` *replacement*`]])`

Selects a sequence of elements using the same rules as `array_slice()`, but instead of being returned, those elements are either removed or, if the fourth argument is provided, replaced with that array. An array containing the removed (or replaced) elements is returned.

**array_sum.** `number array_sum(array` *array*`)`

Returns the sum of every element in the array. If all of the values are integers, an integer is returned. If any of the values are floats, a float is returned.

**array_udiff.** `array array_udiff(array` *array1*`, array` *array2*`[, ... array` *arrayN*`], string` *function*`)`

Returns an array containing all the values in *array1* that are not present in any of the other arrays. Only the values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are considered equal. The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_udiff_assoc.** `array array_udiff_assoc(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], string` *function*`)`

Returns an array containing all the values in *array1* that are not present in any of the other arrays. Both keys and values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are not considered equal. The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_udiff_uassoc.** `array array_udiff_uassoc(array` *array1*`, array` *array2*`[, ... array` *arrayN*`], string` *function1*`, string` *function2*`)`

Returns an array containing all the values in *array1* that are not present in any of the other arrays. Both keys and values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are not considered equal. The function *function1* is used to compare the values of the elements for equality. The function *function2* is used to compare the values of the keys for equality. Each function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_uintersect.** `array array_uintersect(array` *array1*`, array` *array2* `[, ... array` *arrayN*`], string` *function*`)`

Returns an array containing all the values in *array1* that are present in all of the other arrays. Only the values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are considered equal. The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_uintersect_assoc.** `array array_uintersect_assoc(array` *array1*`, array` *array2*`[, ... array` *arrayN*`], string` *function*`)`

Returns an array containing all the values in *array1* that are present in all of the other arrays. Both keys and values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are not considered equal. The function *function* is used to compare the values of the elements for equality. The function is called with two parameters—the values to

compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_uintersect_uassoc.** `array array_uintersect_uassoc(array` *array1*`, array` *array2*`[, ... array` *arrayN*`], string` *function1*`, string` *function2*`)`

Returns an array containing all the values in the first array that are also present in all of the other arrays. Both keys and values are used to check for equality; that is, `"a" => 1` and `"b" => 1` are not considered equal. The function *function1* is used to compare the values of the elements for equality. The function *function2* is used to compare the values of the keys for equality. Each function is called with two parameters—the values to compare. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The keys of the values are preserved.

**array_unique.** `array array_unique(array` *array*`[, int sort_flags])`

Creates and returns an array containing each element in the given array. If any values are duplicated, the later values are ignored. The `sort_flags` optional argument can be used to alter the sorting methods with constants: `SORT_REGULAR`, `SORT_NUMERIC`, `SORT_STRING` (default), and `SORT_LOCALE_STRING`. Keys from the original array are preserved.

**array_unshift.** `int array_unshift(array` *stack*`, mixed` *value1*`[, ... mixed` *valueN*`])`

Returns a copy of the given array with the additional arguments added to the beginning of the array; the added elements are added as a whole, so the elements as they appear in the array are in the same order as they appear in the argument list. Returns the number of elements in the new array.

**array_values.** `array array_values(array` *array*`)`

Returns an array containing all of the values from the input array. The keys for those values are not retained.

**array_walk.** `bool array_walk(array` *input*`, string` *callback*`[, mixed` *user_data*`])`

Calls the named function for each element in the array. The function is called with the element's value, key, and optional user data as arguments. To ensure that the function works directly on the values of the array, define the first parameter of the function by reference. Returns `true` on success, and `false` on failure.

**array_walk_recursive.** `bool array_walk_recursive(array` *input*`, string` *function*`[, mixed` *user_data*`])`

Like `array_walk()`, calls the named function for each element in the array. Unlike in `array_walk()`, if an element's value is an array, the function is called for each element in that array as well. The function is called with the element's value, key, and optional user data as arguments. To ensure that the function works directly on the values of the array, define the first parameter of the function by reference. Returns `true` on success, and `false` on failure.

**arsort.** `bool arsort(array` *array*`[, int` *flags*`])`

Sorts an array in reverse order, maintaining the keys for the array values. The optional second parameter contains additional sorting flags. Returns `true` on success, and `false` on failure. See Chapter 5 and `sort` for more information on using this function.

**asin.** `float asin(float` *value*`)`

Returns the arc sine of *value* in radians.

**asinh.** `float asinh(float` *value*`)`

Returns the inverse hyperbolic sine of *value*.

**asort.** `bool asort(array` *array*`[, int` *flags*`])`

Sorts an array, maintaining the keys for the array values. The optional second parameter contains additional sorting flags. Returns `true` on success, and `false` on failure. See Chapter 5 and `sort` for more information on using this function.

**assert.** `bool assert(string|bool` *assertion*`[, string description])`

If *assertion* is `true`, generates a warning in executing the code. If *assertion* is a string, `assert()` evaluates that string as PHP code. The optional second argument allows for additional text to be added in with the failure message. Check the `assert_options()` function to see its related connection.

**assert_options.** `mixed assert_options(int` *option*`[, mixed` *value*`])`

If *value* is specified, sets the assert control option *option* to *value* and returns the previous setting. If *value* is not specified, returns the current value of *option*. The following values for *option* are allowed:

| | |
|---|---|
| `ASSERT_ACTIVE` | Enable assertions |
| `ASSERT_WARNING` | Have assertions generate warnings |
| `ASSERT_BAIL` | Have execution of the script halt on an assertion |
| `ASSERT_QUIET_EVAL` | Disable error reporting while evaluating assertion code given to the `assert()` function |
| `ASSERT_CALLBACK` | Call the specified user function to handle an assertion. Assertion callbacks are called with three arguments: the file, the line, and the expression where the assertion failed |

**atan.** `float atan(float` *value*`)`

Returns the arc tangent of *value* in radians.

**atan2.** `float atan2(float` *y*`, float` *x*`)`

Using the signs of both parameters to determine the quadrant the value is in, returns the arc tangent of *x* and *y* in radians.

**atanh.** `float atanh(float` *value*`)`

Returns the inverse hyperbolic tangent of *value*.

**base_convert.** `string base_convert(string` *number*`, int` *from*`, int` *to*`)`

Converts *number* from one base to another. The base the number is currently in is *from*, and the base to convert to is *to*. The bases to convert from and to must be between 2 and 36. Digits in a base higher than 10 are represented with the letters `a` (10) through `z` (35). Up to a 32-bit number, or 2,147,483,647 decimal, can be converted.

**base64_decode.** `string base64_decode(string` *data*`)`

Decodes *data*, which is base-64-encoded data, into a string (which may contain binary data). For more information on base-64 encoding, see RFC 2045.

**base64_encode.** `string base64_encode(string` *data*`)`

Returns a base-64-encoded version of *data*. MIME base-64 encoding is designed to allow binary or other 8-bit data to survive transition through protocols that may not be 8-bit safe, such as email messages.

**basename.** `string basename(string` *path*`[, string` *suffix*`])`

Returns the filename component from the full path *path*. If the file's name ends in *suffix*, that string is removed from the name. For example:

```
$path = "/usr/local/httpd/index.html";
echo(basename($path)); // index.html
echo(basename($path, '.html')); // index
```

**bin2hex.** `string bin2hex(string` *binary*`)`

Converts *binary* to a hexadecimal (base-16) value. Up to a 32-bit number, or 2,147,483,647 decimal, can be converted.

**bindec.** `number bindec(string` *binary*`)`

Converts *binary* to a decimal value. Up to a 32-bit number, or 2,147,483,647 decimal, can be converted.

**call_user_func.** `mixed call_user_func(string` *function*`[, mixed` *parameter1*`[, ... mixed` *parameterN*`]])`

Calls the function given in the first parameter. Additional parameters are used as such when calling the function. The comparison to check for a matching function is case-insensitive. Returns the value returned by the function.

**call_user_func_array.** `mixed call_user_func_array(string` *function*`, array` *parameters*`)`

Similar to `call_user_func()`, this function calls the function named *function* with the parameters in the array *parameters*. The comparison to check for a matching function is case-insensitive. Returns the value returned by the function.

**ceil.** `float ceil(float` *number*`)`

Returns the next highest value to *number*, rounding upward if needed.

**chdir.** `bool chdir(string` *path*`)`

Sets the current working directory to *path*; returns `true` if the operation was successful and `false` if not.

**checkdate.** `bool checkdate(int` *month*`, int` *day*`, int` *year*`)`

Returns `true` if the month, date, and year as given in the parameters are valid (Gregorian), and `false` if not. A date is considered valid if the year falls between 1 and 32,767 inclusive, the month is between 1 and 12 inclusive, and the day is within the number of days the specified month has (including leap years).

**checkdnsrr.** `bool checkdnsrr(string` *host*`[, string` *type*`])`

Searches DNS records for a host having the given type. Returns `true` if any records are found, and `false` if none are found. The host type can take any of the following values (if no value is specified, `MX` is the default):

| | |
|---|---|
| `A` | IP address |
| `MX` (default) | Mail exchanger |
| `NS` | Name server |
| `SOA` | Start of authority |
| `PTR` | Pointer to information |
| `CNAME` | Canonical name |
| `AAAA` | 128-bit IPv6 address |
| `A6` | Defined as part of early IPv6 but downgraded to experimental |
| `SRV` | Generalized service location record |
| `NAPTR` | Regular expression–based rewriting of domain names |
| `TXT` | Originally for human-readable text. However, this record also carries machine-readable data |
| `ANY` | Any of the above |

Check the DNS record entry on Wikipedia for more details.

**chgrp.** `bool chgrp(string` *path*`, mixed` *group*`)`

Changes the group for the file *path* to *group*; PHP must have appropriate privileges for this function to work. Returns `true` if the change was successful and `false` if not.

**chmod.** `bool chmod(string` *path*`, int` *mode*`)`

Attempts to change the permissions of *path* to *mode*. *mode* is expected to be an octal number, such as `0755`. An integer value such as `755` or a string value such as `"u+x"` will not work as expected. Returns `true` if the operation was successful and `false` if not.

**chown.** `bool chown(string` *path*`, mixed` *user*`)`

Changes ownership for the file *path* to the user named *user*. PHP must have appropriate privileges (generally, root for this function) for the function to operate. Returns `true` if the change was successful and `false` if not.

**chr.** `string chr(int` *char*`)`

Returns a string consisting of the single ASCII character *char*.

**chroot.** `bool chroot(string` *path*`)`

Changes the root directory of the current process to *path*. You cannot use `chroot()` to restore the root directory to `/` when running PHP in a web server environment. Returns `true` if the change was successful and `false` if not.

**chunk_split.** `string chunk_split(string` *string*`[, int` *size*`[, string` *postfix*`]])`

Inserts *postfix* into *string* after every *size* characters and at the end of the string; returns the resulting string. If not specified, *postfix* defaults to `\r\n` and *size* defaults to `76`. This function is most useful for encoding data to the RPF 2045 standard. For example:

```
$data = "...some long data...";
$converted = chunk_split(base64_encode($data));
```

**class_alias.** `bool class_alias(string` *name*`, string` *alias*`)`

Creates an alias to the class *name*. From then on, you can reference the class (for example, to instantiate objects) with either *name* or *alias*. Returns `true` if the alias could be created; if not, it returns `false`.

**class_exists.** `bool class_exists(string` *name*`[, bool` *autoload_class*`])`

Returns `true` if a class with the same name as the string has been defined; if not, it returns `false`. The comparison for class names is case-insensitive. If *autoload*`_class` is set and is `true`, the class is loaded through the class's `__autoload()` function before getting the interfaces it implements.

**class_implements.** `array class_implements(mixed` *class*`[, bool` *autoload_class*`])`

If *class* is an object, returns an array containing the names of the interfaces implemented by *class*'s object class. If *class* is a string, returns an array containing the names of the interfaces implemented by the class named *class*. Returns `false` if *class* is neither an object nor a string, or if *class* is a string but no object class of that name exists. If *autoload_class* is set and is `true`, the class is loaded through the class's `__autoload()` function before getting the interfaces it implements.

**class_parents.** `array class_parents(mixed` *class*`[, bool` *autoload_class*`])`

If *class* is an object, returns an array containing the names of the parents of *class*'s object class. If *class* is a string, returns an array containing the class names of the parents of the class named *class*. Returns `false` if *class* is neither an object nor a string, or if *class* is a string but no object class of that name exists. If *autoload_class* is set and is `true`, the class is loaded through the class's `__autoload()` function before getting its parents.

**clearstatcache.** `void clearstatcache([bool` *clear_realpath_cache*`[, string` *file*`]])`

Clears the file status functions cache. The next call to any of the file status functions will retrieve the information from the disk. The *clear_realpath_cache* parameter allows for clearing the *realpath* cache. The `file` parameter allows for the clearing of the *realpath* and stat caches for a specific filename only, and it can be used only if *clear_realpath_cache* is `true`.

**closedir.** `void closedir([int` *handle*`])`

Closes the directory stream referenced by *handle*. See `opendir()` for more information on directory streams. If *handle* is not specified, the most recently opened directory stream is closed.

**closelog.** `int closelog()`

Closes the file descriptor used to write to the system logger after an `openlog()` call. Returns `true` if the change was successful and `false` if not.

**compact.** `array compact(mixed` *variable1*`[, ... mixed` *variableN*`])`

Creates an array by retrieving the values of the variables named in the parameters. If any of the parameters are arrays, the values of variables named in the arrays are also retrieved. The array returned is an associative array, with the keys being the arguments provided to the function and the values being the values of the named variables. This function is the opposite of `extract()`.

**connection_aborted.** `int connection_aborted()`

Returns `true` (`1`) if the client disconnected (for example, clicked Stop in the browser) at any point before the function is called. Returns `false` (`0`) if the client is still connected.

**connection_status.** `int connection_status()`

Returns the status of the connection as a bitfield with three states: `NORMAL` (`0`), `ABORTED` (`1`), and `TIMEOUT` (`2`).

**constant.** `mixed constant(string` *name*`)`

Returns the value of the constant called *name*.

**convert_cyr_string.** `string convert_cyr_string(string` *value*`, string` *from*`, string` *to*`)`

Converts *value* from one Cyrillic set to another. The *from* and *to* parameters are single-character strings representing the set and have the following valid values:

| | |
|---|---|
| `k` | koi8-r |
| `w` | Windows-1251 |
| `i` | ISO 8859-5 |
| `a` or `d` | x-cp866 |
| `m` | x-mac-cyrillic |

**convert_uudecode.** `string convert_uudecode(string` *value*`)`

Decodes the `uuencoded` string *value* and returns it.

**convert_uuencode.** `string convert_uuencode(string` *value*`)`

Encodes the string *value* using `uuencode` and returns it.

**copy.** `int copy(string` *path*`, string` *destination*`[, resource` *context*`])`

Copies the file at *path* to *destination*. If the operation succeeds, the function returns `true`; otherwise, it returns `false`. If the file at the destination exists, it will be replaced. The optional *context* parameter can make use of a valid context resource created with the `stream_context_create()` function.

**cos.** `float cos(float` *value*`)`

Returns the cosine of *value* in radians.

**cosh.** `float cosh(float` *value*`)`

Returns the hyperbolic cosine of *value*.

**count.** `int count(mixed` *value*`[, int` *mode*`])`

Returns the number of elements in the *value*; for arrays or objects, this is the number of elements; for any other *value*, this is `1`. If the parameter is a variable and the variable is not set, `0` is returned. If *mode* is set and is `COUNT_RECURSIVE`, the number of elements is counted recursively, counting the number of values in arrays inside arrays.

**count_chars.** `mixed count_chars(string` *string*`[, int` *mode*`])`

Returns the number of occurrences of each byte value from 0 to 255 in *string*; *mode* determines the form of the result. The possible values of *mode* are:

| `0` (default) | Returns an associative array with each byte value as a key and the frequency of that byte value as the value |
|---|---|
| `1` | Same as above, except that only byte values with a nonzero frequency are listed |
| `2` | Same as above, except that only byte values with a frequency of zero are listed |
| `3` | Returns a string containing all byte values with a nonzero frequency |
| `4` | Returns a string containing all byte values with a frequency of zero |

**crc32.** `int crc32(string` *value*`)`

Calculates and returns the *cyclic redundancy checksum* (CRC) for *value*.

**create_function.** `string create_function(string` *arguments*`, string` *code*`)`

Creates an anonymous function with the given *arguments* and *code*; returns a generated name for the function. Such anonymous functions (also called *lambda functions*) are useful for short-term callback functions, such as when using `usort()`.

**crypt.** `string crypt(string` *string*`[, string` *salt*`])`

Encrypts *string* using the DES encryption algorithm seeded with the two-character salt value *salt*. If *salt* is not supplied, a random *salt* value is generated the first time `crypt()` is called in a script; this value is used on subsequent calls to `crypt()`. Returns the encrypted string.

**current.** `mixed current(array` *array*`)`

Returns the value of the element to which the internal pointer is set. The first time that `current()` is called, or when `current()` is called after `reset`, the pointer is set to the first element in the array.

**date.** `string date(string` *format*`[, int` *timestamp*`])`

Formats a time and date according to the *format* string provided in the first parameter. If the second parameter is not specified, the current time and date is used. The following characters are recognized in the *format* string:

| `a` | "am" or "pm" |
|---|---|
| `A` | "AM" or "PM" |
| `B` | Swatch internet time |
| `d` | Day of the month as two digits, including a leading zero if necessary (e.g., "01" through "31") |

| | |
|---|---|
| `D` | Name of the day of the week as a three-letter abbreviation (e.g., "Mon") |
| `F` | Name of the month (e.g., "August") |
| `g` | Hour in 12-hour format (e.g., "1" through "12") |
| `G` | Hour in 24-hour format (e.g., "0" through "23") |
| `h` | Hour in 12-hour format, including a leading zero if necessary; e.g., "01" through "12" |
| `H` | Hour in 24-hour format, including a leading zero if necessary (e.g., "00" through "23") |
| `i` | Minutes, including a leading zero if necessary (e.g., "00" through "59") |
| `I` | "1" if Daylight Saving Time; "0" otherwise |
| `j` | Day of the month (e.g., "1" through "31") |
| `l` | Name of the day of the week (e.g., "Monday") |
| `L` | "0" if the year is not a leap year; "1" if it is |
| `m` | Month, including a leading zero if necessary (e.g., "01" through "12") |
| `M` | Name of the month as a three-letter abbreviation (e.g., "Aug") |
| `n` | Month without leading zeros (e.g., "1" to "12") |
| `r` | Date formatted according to RFC 822 (e.g., "Thu, 21 Jun 2001 21:27:19 +0600") |
| `s` | Seconds, including a leading zero if necessary (e.g., "00" through "59") |
| `S` | English ordinal suffix for the day of the month; either "st", "nd", or "th" |
| `t` | Number of days in the month, from "28" to "31" |
| `T` | Time zone setting of the machine running PHP (e.g., "MST") |
| `u` | Seconds since the Unix epoch |
| `w` | Numeric day of the week, starting with "0" for Sunday |
| `W` | Numeric week of the year according to ISO 8601 |
| `Y` | Year with four digits (e.g., "1998") |
| `y` | Year with two digits (e.g., "98") |
| `z` | Day of the year, from "0" through "365" |
| `Z` | Time zone offset in seconds, from "–43200" (far west of UTC) to "43200" (far east of UTC) |

Any characters in the *format* string not matching one of the above will be kept in the resulting string as is. If a non-numeric value is provided for `timestamp`, then `false` is returned and a warning is issued.

**date_default_timezone_get.** `string date_default_timezone_get()`

Returns the current default time zone, set previously by the `date_default_timezone_set()` function or via the `date.timezone` option in the *php.ini* file. Returns `"UTC"` if neither is set.

**date_default_timezone_set.** `string date_default_timezone_set(string` *timezone*`)`

Sets the current default time zone.

**date_parse.** `array date_parse(string` *time*`)`

Converts an English description of a time and date into an array describing that time and date. Returns `false` if the value could not be converted into a valid date. The returned array contains the same values as returned from `date_parse_from_format()`.

**date_parse_from_format.** `array date_parse_from_format(string` *format*`, string` *time*`)`

Parses *time* into an associative array representing a date. The string *time* is given in the format specified by *format*, using the same character codes as described in `date()`. The returned array contains the following entries:

| | |
|---|---|
| `year` | Year |
| `month` | Month |
| `day` | Day of the month |
| `hour` | Hours |
| `minute` | Minutes |
| `second` | Seconds |
| `fraction` | Fractions of seconds |
| `warning_count` | Number of warnings that occurred during parsing |
| `warnings` | An array of warnings that occurred during parsing |
| `error_count` | Number of errors that occurred during parsing |
| `errors` | An array of errors that occurred during parsing |
| `is_localtime` | True if the time represents a time in the current default time zone |
| `zone_type` | The type of time zone `zone` represents |
| `zone` | The time zone the time is in |
| `is_dst` | True if the time represents a time in Daylight Saving Time |

**date_sun_info.** `array date_sun_info(int` *timestamp*`, float` *latitude*`, float` *longitude*`)`

Returns information as an associative array about the times of sunrise and sunset, and the times twilight begins and ends, at a given latitude and longitude. The resulting array contains the following keys:

| | |
|---|---|
| `sunrise` | The time sunrise occurs |
| `sunset` | The time sunset occurs |
| `transit` | The time the sun is at its zenith |
| `civil_twilight_begin` | The time civil twilight begins |
| `civil_twilight_end` | The time civil twilight ends |

| | |
|---|---|
| `nautical_twilight_begin` | The time nautical twilight begins |
| `nautical_twilight_end` | The time nautical twilight ends |
| `astronomical_twilight_begin` | The time astronomical twilight begins |
| `astronomical_twilight_end` | The time astronomical twilight ends |

**date_sunrise.** `mixed date_sunrise(int` *timestamp*`[, int` *format*`[, float` *latitude*`[, float` *longitude* `[, float` *zenith*`[, float` *gmt_offset*`]]]]])`

Returns the time of the sunrise for the day in *timestamp*; `false` on failure. The *format* parameter determines the format the time is returned as (with a default of `SUNFUNCS_RET_STRING`), while the *latitude*, *longitude*, *zenith*, and *gmt_offset* parameters provide a specific location. They default to values given in the PHP configuration options (*php.ini*). Parameters include:

| | |
|---|---|
| `SUNFUNCS_RET_STRING` | Returns the value as a string; for example, "06:14" |
| `SUNFUNCS_RET_DOUBLE` | Returns the value as a float; for example, 6.233 |
| `SUNFUNCS_RET_TIMESTAMP` | Returns the value as a Unix epochal timestamp |

**date_sunset.** `mixed date_sunset(int` *timestamp*`[, int` *format*`[, float` *latitude*`[, float` *longitude* `[, float` *zenith*`[, float` *gmt_offset*`]]]]])`

Returns the time of the sunset for the day in *timestamp*; `false` on failure. The *format* parameter determines the format the time is returned as (with a default of `SUNFUNCS_RET_STRING`), while the *latitude*, *longitude*, *zenith*, and *gmt_offset* parameters provide a specific location. They default to values given in the PHP configuration options (*php.ini*). Parameters include:

| | |
|---|---|
| `SUNFUNCS_RET_STRING` | Returns the value as a string; for example, "19:02" |
| `SUNFUNCS_RET_DOUBLE` | Returns the value as a float; for example, 19.033 |
| `SUNFUNCS_RET_TIMESTAMP` | Returns the value as a Unix epochal timestamp |

**debug_backtrace.** `array debug_backtrace([ int` *options* `[, int` *limit*`]])`

Returns an array of associative arrays containing a backtrace of where PHP is currently executing. One element is included per function or file include, with the following elements:

| | |
|---|---|
| `function` | If in a function, the function's name as a string |
| `line` | The line number within the file where the current function or file include is located |
| `file` | The name of the file the element is in |
| `class` | If in an object instance or class method, the name of the class the element is in |

| | |
|---|---|
| `object` | If in an object, that object's name |
| `type` | The current call type: `::` if a static method; `->` if a method; nothing if a function |
| `args` | If in a function, the arguments used to call that function; if in a file include, the include file's name |

Each function call or file include generates a new element in the array. The innermost function call or file include is the element with an index of 0; further elements are less deep function calls or file includes.

**debug_print_backtrace.** `void debug_print_backtrace()`

Prints the current debug backtrace (see `debug_backtrace`) to the client.

**decbin.** `string decbin(int` *decimal*`)`

Converts the provided *decimal* value to a binary representation of it. Up to a 32-bit number, or 2,147,483,647 decimal, can be converted.

**dechex.** `string dechex(int` *decimal*`)`

Converts *decimal* to a hexadecimal (base-16) representation of it. Up to a 32-bit number, or 2,147,483,647 decimal (0x7FFFFFFF hexadecimal), can be converted.

**decoct.** `string decoct(int` *decimal*`)`

Converts *decimal* to an octal (base-8) representation of it. Up to a 32-bit number, or 2,147,483,647 decimal (017777777777 octal), can be converted.

**define.** `bool define(string` *name*`, mixed` *value*`[, int` *case_insensitive*`])`

Defines a constant named *name* and sets its value to *value*. If *case_insensitive* is set and is `true`, the operation fails if a constant with the same name, compared case-insensitively, is previously defined. Otherwise, the check for existing constants is done case-sensitively. Returns `true` if the constant could be created, and `false` if a constant with the given name already exists.

**define_syslog_variables.** `void define_syslog_variables()`

Initializes all variables and constants used by the syslog functions `openlog()`, `syslog()`, and `closelog()`. This function should be called before using any of the syslog functions.

**defined.** `bool defined(string` *name*`)`

Returns `true` if a constant with the name *name* exists, and `false` if a constant with that name does not exist.

**deflate_add.** `void deflate_init(resource` *context*`, string` *data*`[, int` *flush_mode*`])`

Adds *data* to the deflate context *context*, and checks if the context should be flushed based on *flush_mode*, which is one of `ZLIB_BLOCK`, `ZLIB_NO_FLUSH`, `ZLIB_PARTIAL_FLUSH`, `ZLIB_SYNC_FLUSH` (the default), `ZLIB_FULL_FLUSH`, or `ZLIB_FINISH`. When adding most chunks of data, choose `ZLIB_NO_FLUSH` to maximize compression attempts. After the last chunk has been added, use `ZLIB_FINISH` to indicate the context is complete.

**deflate_init.** `void deflate_init(int` *encoding*`[, array` *options*`])`

Initializes and returns an incremental deflation context. This context can be used to incrementally deflate data using calls to `deflate_add()` using that context.

| | |
|---|---|
| `level` | The compression range from –1 through 9 |
| `memory` | The compression memory level from 1 through 9 |
| `window` | The zlib window size from 8 through 15 |
| `strategy` | The compression strategy to use; either `ZLIB_FILTERED`, `ZLIB_HUFFMAN_ONLY`, `ZLIB_RLE`, `ZLIB_FIXED`, or `ZLIB_DEFAULT_STRATEGY` (default) |
| `dictionary` | A string or array of strings of the compression preset dictionary |

**deg2rad.** `float deg2rad(float` *number*`)`

Converts *number* from degrees to radians and returns the result.

**dir.** `directory dir(string` *path*`[, resource` *context*`])`

Returns an instance of the `directory` class initialized to the given *path*. You can use the `read()`, `rewind()`, and `close()` methods on the object as equivalent to the `readdir()`, `rewinddir()`, and `closedir()` procedural functions.

**dirname.** `string dirname(string` *path*`)`

Returns the directory component of *path*. This includes everything up to the filename portion (see `basename`) and doesn't include the trailing path separator.

**disk_free_space.** `float disk_free_space(string` *path*`)`

Returns the number of bytes of free space available on the disk partition or filesystem at *path*.

**disk_total_space.** `float disk_total_space(string` *path*`)`

Returns the number of bytes of total space available (including both used and free) on the disk partition or filesystem at *path*.

**each.** `array each(array &`*array*`)`

Creates an array containing the keys and values of the element currently pointed at by the array's internal pointer. The array contains four elements: elements with the keys `0` and *key* containing the key of the element, and elements with the keys `1` and *value* containing the value of the element.

If the internal pointer of the array points beyond the end of the array, `each()` returns `false`.

**echo.** `void echo string` *string*`[, string` *string2*`[, string` *stringN* `...]]`

Outputs the given strings. `echo` is a language construct, and enclosing the parameters in parentheses is optional, unless multiple parameters are given—in which case, you cannot use parentheses.

**empty.** `bool empty(mixed` *value*`)`

Returns `true` if *value* is either `0` or not set, and `false` otherwise.

**end.** `mixed end(array &`*array*`)`

Advances the array's internal pointer to the last element and returns the element's value.

**error_clear_last.** `array error_clear_last()`

Clears the most recent error; it will no longer be returned by `error_get_last()`.

**error_get_last.** `array error_get_last()`

Returns an associative array of information about the most recent error that occurred, or `NULL` if no errors have yet occurred while processing the current script. The following values are included in the array:

| | |
|---|---|
| `type` | The type of error |
| `message` | Printable version of the error |
| `file` | The full path to the file where the error occurred |
| `line` | The line number within the file where the error occurred |

**error_log.** `bool error_log(string` *message*`, int` *type*`[, string` *destination*`[, string` *headers*`]])`

Records an error message to the web server's error log, to an email address, or to a file. The first parameter is the message to log. The type is one of the following:

| | |
|---|---|
| 0 | `message` is sent to the PHP system log; the message is put into the file pointed at by the `error_log` configuration directive. |
| 1 | `message` is sent to the email address destination. If specified, *headers* provides optional headers to use when creating the message (see `mail` for more information on the optional headers). |
| 3 | Appends *message* to the file *destination*. |
| 4 | `message` is sent directly to the Server Application Programming Interface (SAPI) logging handler. |

**error_reporting.** `int error_reporting([int` *level*`])`

Sets the level of errors reported by PHP to *level* and returns the current level; if *level* is omitted, the current level of error reporting is returned. The following values are available for the function:

| | |
|---|---|
| `E_ERROR` | Fatal runtime errors (script execution halts) |
| `E_WARNING` | Runtime warnings |
| `E_PARSE` | Compile-time parse errors |
| `E_NOTICE` | Runtime notices |
| `E_CORE_ERROR` | Errors generated internally by PHP |
| `E_CORE_WARNING` | Warnings generated internally by PHP |
| `E_COMPILE_ERROR` | Errors generated internally by the Zend scripting engine |
| `E_COMPILE_WARNING` | Warnings generated internally by the Zend scripting engine |
| `E_USER_ERROR` | Runtime errors generated by a call to `trigger_error()` |
| `E_USER_WARNING` | Runtime warnings generated by a call to `trigger_error()` |
| `E_STRICT` | Direct PHP to suggest code changes to assist with forward compatibility |
| `E_RECOVERABLE_ERROR` | If a potentially fatal error has occurred, was caught, and properly handled, the code can continue execution |
| `E_DEPRECATED` | If enabled, warnings will be issued about deprecated code that will eventually not work properly |
| `E_USER_DEPRECATED` | If enabled, any warning message triggered by deprecated code can be user-generated with the `trigger_error()` function |
| `E_ALL` | All of the above options |

Any number of these options can be ORed (bitwise OR, `|`) together, so that errors in each of the levels are reported. For example, the following code turns off user errors and warnings, performs some actions, and then restores the original level:

```
<$level = error_reporting();
 error_reporting($level & ~(E_USER_ERROR | E_USER_WARNING));
 // do some stuff
 error_reporting($level);>
```

**escapeshellarg.** `string escapeshellarg(string` *argument*`)`

Properly escapes *argument* so it can be used as a safe argument to a shell function. When directly passing user input (such as from forms) to a shell command, you should use this function to escape the data to ensure that the argument isn't a security risk.

**escapeshellcmd.** `string escapeshellcmd(string` *command*`)`

Escapes any characters in *command* that could cause a shell command to run additional commands. When directly passing user input (such as from forms) to the `exec()` or `system()` functions, you should use this function to escape the data to ensure that the argument isn't a security risk.

**exec.** `string exec(string` *command*`[, array` *output*`[, int` *return*`]])`

Executes *command* via the shell and returns the last line of output from the command's result. If *output* is specified, it is filled with the lines returned by the command. If *return* is specified, it is set to the return status of the command.

If you want to have the results of the command output into the PHP page, use `passthru()`.

**exp.** `float exp(float` *number*`)`

Returns *e* raised to the *number* power.

**explode.** `array explode(string` *separator*`, string` *string*`[, int` *limit*`])`

Returns an array of substrings created by splitting *string* wherever *separator* is found. If supplied, a maximum of *limit* substrings will be returned, with the last substring returned containing the remainder of the string. If *separator* is not found, returns the original string.

**expm1.** `float expm1(float` *number*`)`

Returns `exp(`*number*`) - 1`, computed such that the returned value is accurate even when *number* is near 0.

**extension_loaded.** `bool extension_loaded(string` *name*`)`

Returns `true` if the *name*d extension is loaded, and `false` if it is not.

**extract.** `int extract(array` *array*`[, int` *type*`[, string` *prefix*`]])`

Sets the value of variables to the values of elements from an array. For each element in the array, the key is used to determine the variable name to set, and that variable is set to the value of the element.

The second argument, if given, takes one of the following values to determine behavior if the values in the array have the same name as variables that already exist in the local scope:

| | |
|---|---|
| `EXTR_OVERWRITE` (default) | Overwrite the existing variable |
| `EXTR_SKIP` | Don't overwrite the existing variable (ignore the value provided in the array) |
| `EXTR_PREFIX_SAME` | Prefix the variable name with the string given as the third argument |
| `EXTR_PREFIX_ALL` | Prefix all variable names with the string given as the third argument |
| `EXTR_PREFIX_INVALID` | Prefix any invalid or numeric variable names with the string given as the third argument |
| `EXTR_IF_EXISTS` | Replace variable only if it exists in the current symbol table |
| `EXTR_PREFIX_IF_EXISTS` | Create prefixed variable names only if the nonprefixed version of the same variable exists |
| `EXTR_REFS` | Extract variables as references |

The function returns the number of successfully set variables.

**fclose.** `bool fclose(int` *handle*`)`

Closes the file referenced by *handle*; returns `true` if successful and `false` if not.

**feof.** `bool feof(int` *handle*`)`

Returns `true` if the marker for the file referenced by *handle* is at the end of the file (EOF) or if an error occurs. If the marker is not at EOF, returns `false`.

**fflush.** `bool fflush(int` *handle*`)`

Commits any changes to the file referenced by *handle* to disk, ensuring that the file contents are on disk and not just in a disk buffer. If the operation succeeds, the function returns `true`; otherwise, it returns `false`.

**fgetc.** `string fgetc(int` *handle*`)`

Returns the character at the marker for the file referenced by *handle* and moves the marker to the next character. If the marker is at the end of the file, the function returns `false`.

**fgetcsv.** `array fgetcsv(resource` *handle*`[, int` *length*`[, string` *delimiter*`[, string` *enclosure* `[, string` *escape*`]]]])`

Reads the next line from the file referenced by *handle* and parses the line as a comma-separated values (CSV) line. The longest line to read is given by *length*. If *delimiter* is supplied, it is used to delimit the values for the line instead of commas. If supplied, *enclosure* is a single character that is used to enclose values (by default, the double quote character, "). *escape* sets the escape character to use; the default is backslash \; only one character can be specified. For example, to read and display all lines from a file containing tab-separated values, use:

```
$fp = fopen("somefile.tab", "r");

while($line = fgetcsv($fp, 1024, "\t")) {
 print "<p>" . count($line) . "fields:</p>";
 print_r($line);
}
fclose($fp);
```

**fgets.** `string fgets(resource` *handle* `[, int` *length*`])`

Reads a string from the file referenced by *handle*; a string of no more than *length* characters is returned, but the read ends at *length* `- 1` (for the end-of-line character) characters, at an end-of-line character, or at EOF. Returns `false` if any error occurs.

**fgetss.** `string fgetss(resource` *handle* `[, int` *length*`[, string` *tags*`]])`

Reads a string from the file referenced by *handle*; a string of no more than *length* characters is returned, but the read ends at *length* `- 1` (for the end-of-line character) characters, at an end-of-line character, or at EOF. Any PHP and HTML tags in the string, except those listed in *tags*, are stripped before returning it. Returns `false` if any error occurs.

**file.** `array file(string` *filename*`[, int` *flags* `[, resource` *context*`]])`

Reads the *file* into an array. *flags* can be one or more of the following constants:

| | |
|---|---|
| `FILE_USE_INCLUDE_PATH` | Search for the file in the include path as set in the *php.ini* file |
| `FILE_IGNORE_NEW_LINES` | Do not add a newline at the end of the array elements |
| `FILE_SKIP_EMPTY_LINES` | Skip any empty lines |

**file_exists.** `bool file_exists(string` *path*`)`

Returns `true` if the file at *path* exists and `false` if not.

**fileatime.** `int fileatime(string` *path*`)`

Returns the last access time, as a Unix timestamp value, for the file *path*. Because of the cost involved in retrieving this information from the filesystem, this information is cached; you can clear the cache with `clearstatcache()`.

**filectime.** `int filectime(string` *path*`)`

Returns the inode change time value for the file at *path*. Because of the cost involved in retrieving this information from the filesystem, this information is cached; you can clear the cache with `clearstatcache()`.

**file_get_contents.** `string file_get_contents(string` *path*`[, bool` *include* `[, resource` *context* `[, int` *offset* `[, int` *maxlen*`]]]])`

Reads the file at *path* and returns its contents as a string, optionally starting at *offset*. If *include* is specified and is `true`, the include path is searched for the file. The length of the returned string can also be controlled with the *maxlen* parameter.

**filegroup.** `int filegroup(string` *path*`)`

Returns the group ID of the group owning the file *path*. Because of the cost involved in retrieving this information from the filesystem, this information is cached; you can clear the cache with `clearstatcache()`.

**fileinode.** `int fileinode(string` *path*`)`

Returns the inode number of the file *path*, or `false` if an error occurs. This information is cached; see `clearstatcache`.

**filemtime.** `int filemtime(string` *path*`)`

Returns the last-modified time, as a Unix timestamp value, for the file *path*. This information is cached; you can clear the cache with `clearstatcache()`.

**fileowner.** `int fileowner(string` *path*`)`

Returns the user ID of the owner of the file *path*, or `false` if an error occurs. This information is cached; you can clear the cache with `clearstatcache()`.

**fileperms.** `int fileperms(string` *path*`)`

Returns the file permissions for the file *path*, or `false` if an error occurs. This information is cached; you can clear the cache with `clearstatcache()`.

**file_put_contents.** `int file_put_contents(string` *path*`, mixed` *string* `[, int` *flags*`[, resource` *context*`]])`

Opens the file specified by *path*, writes *string* to the file, and then closes the file. Returns the number of bytes written to the file, or `-1` on error. The *flags* argument is a bitfield with two possible values:

| | |
|---|---|
| `FILE_USE_INCLUDE_PATH` | If specified, the include path is searched for the file and the file is written at the first location where it already exists |
| `FILE_APPEND` | If specified and if the file indicated by `path` already exists, `string` is appended to the existing contents of the file |
| `LOCK_EX` | Exclusively lock the file before writing to it |

**filesize.** `int filesize(string` *path*`)`

Returns the size, in bytes, of the file *path*. If the file does not exist or any other error occurs, the function returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**filetype.** `string filetype(string` *path*`)`

Returns the type of file given in *path*. The possible types are:

| | |
|---|---|
| `Fifo` | The file is a FIFO pipe |
| `Char` | The file is a text file |
| `Dir` | *path* is a directory |
| `Block` | A block reserved for use by the filesystem |
| `Link` | The file is a symbolic link |
| `File` | The file contains binary data |
| `Socket` | A socket interface |
| `Unknown` | The file's type could not be determined |

**filter_has_var.** `bool filter_has_var(int` *context*`, string` *name*`)`

Returns `true` if a value named *name* exists in the specified *context*, and `false` if it doesn't. The context is one of `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, or `INPUT_ENV`.

**filter_id.** `int filter_id(string` *name*`)`

Returns the ID for the filter identified by *name*, or `false` if no such filter exists.

**filter_input.** `mixed filter_input(mixed` *var*`[, int` *filter_id*`[, mixed` *options*`]])`

Performs the filter identified by ID *filter_id* on *var* in the given context and returns the result. The context is one of `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, or `INPUT_ENV`. If *filter_id* is not specified, the default filter is used. The *options* parameter can either be a bitfield of flags or an associative array of options appropriate to the filter. See Chapter 4 for more information on using filters.

**filter_input_array.** `mixed filter_input_array(array` *variables*`[, mixed` *filters*`])`

Performs a series of filters against variables in the associative array *variables* and returns the results as an associative array. The context is one of `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, or `INPUT_ENV`.

The optional parameter is an associative array where each element's key is a variable name, with the associated value defining the filter and options to use to filter that variable's value. The definition is either the ID of the filter to use or an array containing one or more of the following elements:

| | |
|---|---|
| `filter` | The ID of the filter to apply |
| `flags` | A bitfield of flags |
| `options` | An associative array of options specific to the filter |

**filter_list.** `array filter_list()`

Returns an array of the name of each available filter; these names can be passed into `filter_id()` to obtain a filter ID for use in the other filtering functions.

**filter_var.** `mixed filter_var(mixed` *var*`[, int` *filter_id*`[, mixed` *options*`]])`

Performs the filter identified by ID *filter_id* on *var* and returns the result. If *filter_id* is not specified, the default filter is used. The *options* parameter can either be a bitfield of flags or an associative array of options appropriate to the filter. See Chapter 4 for more information on using filters.

**filter_var_array.** `mixed filter_var_array(mixed` *var*`[, mixed` *options*`])`

Performs a series of filters against variables in the specified context and returns the results as an associative array. The context is one of `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, or `INPUT_ENV`.

The *options* parameter is an associative array where each element's key is a variable name, with the associated value defining the filter and options to use to filter that variable's value. The definition is either the ID of the filter to use or an array containing one or more of the following elements:

| | |
|---|---|
| `filter` | The ID of the filter to apply |
| `flags` | A bitfield of flags |
| `options` | An associative array of options specific to the filter |

**floatval.** `float floatval(mixed` *value*`)`

Returns the float value for *value*. If value is a nonscalar (object or array), `1` is returned.

**flock.** `bool flock(resource` *handle*`, int` *operation*`[, int` *would_block*`])`

Attempts to lock the file path of the file specified by *handle*. The operation is one of the following values:

| | |
|---|---|
| `LOCK_SH` | Shared lock (reader) |
| `LOCK_EX` | Exclusive lock (writer) |
| `LOCK_UN` | Release a lock (either shared or exclusive) |
| `LOCK_NB` | Add to `LOCK_SH` or `LOCK_EX` to obtain a nonblocking lock |

If specified, *would_block* is set to `true` if the operation would cause a block on the file. The function returns `false` if the lock could not be obtained, and `true` if the operation succeeded.

Because file locking is implemented at the process level on most systems, `flock()` cannot prevent two PHP scripts running in the same web server process from accessing a file at the same time.

**floor.** `float floor(float` *number*`)`

Returns the largest integer value less than or equal to *number*.

**flush.** `void flush()`

Sends the current output buffer to the client and empties the output buffer. See Chapter 15 for more information on using the output buffer.

**fmod.** `float fmod(float` *x*`, float` *y*`)`

Returns the floating-point modulo of the division of *x* by *y*.

**fnmatch.** `bool fnmatch(string` *pattern*`, string` *string*`[, int` *flags*`])`

Returns `true` if *string* matches the shell wildcard pattern given in *pattern*. See `glob` for the pattern-matching rules. The flags value is a bitwise OR of any of the following values:

| | |
|---|---|
| `FNM_NOESCAPE` | Treat backslashes in `pattern` as backslashes, rather than as the start of an escape sequence |
| `FNM_PATHNAME` | Slash characters in `string` must be matched explicitly by slashes in `pattern` |
| `FNM_PERIOD` | A period at the beginning of the string, or before any slash if `FNM_PATHNAME` is also specified, must be explicitly matched by periods in `pattern` |
| `FNM_CASEFOLD` | Ignore case when matching `string` to `pattern` |

**fopen.** `resource fopen(string` *path*`, string` *mode*`[, bool` *include* `[, resource` *context*`]])`

Opens the file specified by *path* and returns a file resource handle to the open file. If *path* begins with `http://`, an HTTP connection is opened and a file pointer to the start of the response is returned. If *path* begins with `ftp://`, an FTP connection is opened and a file pointer to the start of the file is returned; the remote server must support passive FTP.

If *path* is `php://stdin`, `php://stdout`, or `php://stderr`, a file pointer to the appropriate stream is returned.

The parameter *mode* specifies the permissions to open the file with. It must be one of the following:

| | |
|---|---|
| `r` | Open the file for reading; file pointer will be at beginning of file. |
| `r+` | Open the file for reading and writing; file pointer will be at beginning of file. |
| `w` | Open the file for writing. If the file exists, it will be truncated to zero length; if the file doesn't already exist, it will be created. |
| `w+` | Open the file for reading and writing. If the file exists, it will be truncated to zero length; if the file doesn't already exist, it will be created. The file pointer starts at the beginning of the file. |
| `a` | Open the file for writing. If the file exists, the file pointer will be at the end of the file; if the file does not exist, it is created. |
| `a+` | Open the file for reading and writing. If the file exists, the file pointer will be at the end of the file; if the file does not exist, it is created. |
| `x` | Create and open file for writing only; place the file pointer at the beginning of the file. |
| `x+` | Create and open file for reading and writing. |
| `c` | Open the file for writing only. If the file does not exist, it is created. If it exists, it is not truncated (as is the case with `w`), nor does the call to this function fail (as is the case with `x`). The file pointer is positioned at the beginning of the file. |
| `c+` | Open the file for reading and writing. |

If *include* is specified and is `true`, `fopen()` tries to locate the file in the current *include* path.

If any error occurs while attempting to open the file, `false` is returned.

**forward_static_call.** `mixed forward_static_call(callable` *function*`[, mixed` *parameter1*`[, ... mixed` *parameterN*`]])`

Calls the function named *function* in the current object's context with the parameters provided. If *function* includes a class name, it uses late static binding to find the appropriate class for the method. Returns the value returned by the function.

**forward_static_call_array.** `mixed forward_static_call_array(callable` *function*`, array` *parameters*`)`

Calls the function named *function* in the current object's context with the parameters in the array *parameters*. If *function* includes a class name, it uses late static binding to find the appropriate class for the method. Returns the value returned by the function.

**fpassthru.** `int fpassthru(resource` *handle*`)`

Outputs the file pointed to by *handle* and closes the file. The file is output from the current file pointer location to EOF. If any error occurs, `false` is returned; if the operation is successful, `true` is returned.

**fprintf.** `int fprintf(resource` *handle*`, string` *format*`[, mixed` *value1*`[, ...` *valueN*`]])`

Writes a string created by filling *format* with the given arguments to the stream resource *handle*. See `printf()` for more information on using this function.

**fputcsv.** `int fputcsv(resource` *handle*`[, array` *fields*`[, string` *delimiter*`[, string` *enclosure*`]]])`

Formats the items contained in *fields* in comma-separated values (CSV) format and writes the result to the file handle *handle*. If supplied, *delimiter* is a single character used to delimit the values for the line instead of commas. If supplied, *enclosure* is a single character that is used to enclose values (by default, the double quote character, "). Returns the length of the string written, or `false` if a failure occurred.

**fread.** `string fread(int` *handle*`, int` *length*`)`

Reads *length* bytes from the file referenced by *handle* and returns them as a string. If fewer than *length* bytes are available before EOF is reached, the bytes up to EOF are returned.

**fscanf.** `mixed fscanf(resource` *handle*`, string` *format*`[, string` *name1*`[, ... string` *nameN*`]])`

Reads data from the file referenced by *handle* and returns a value from it based on *format*. For more information on how to use this function, see `sscanf`.

If the optional *name1* through *nameN* parameters are not given, the values scanned from the file are returned as an array; otherwise, they are put into the variables named by *name1* through *nameN*.

**fseek.** `int fseek(resource` *handle*`, int` *offset*`[, int` *from*`])`

Moves the file pointer in *handle* to the byte *offset*. If *from* is specified, it determines how to move the file pointer. *from* must be one of the following values:

| | |
|---|---|
| `SEEK_SET` | Sets the file pointer to the byte *offset* (the default) |
| `SEEK_CUR` | Sets the file pointer to the current location plus *offset* bytes |
| `SEEK_END` | Sets the file pointer to EOF minus *offset* bytes |

This function returns `0` if the function was successful and `-1` if the operation failed.

**fsockopen.** `resource fsockopen(string` *host*`, int` *port*`[, int` *error*`[, string` *message*`[, float` *timeout*`]]])`

Opens a TCP or UDP connection to a remote *host* on a specific *port*. By default, TCP is used; to connect via UDP, *host* must begin with the protocol `udp://`. If specified, *timeout* indicates the length of time in seconds to wait before timing out.

If the connection is successful, a virtual file pointer is returned, which can be used with functions such as `fgets()` and `fputs()`. If the connection fails, `false` is returned. If *error* and *message* are supplied, they are set to the error number and error string, respectively.

**fstat.** `array fstat(resource` *handle*`)`

Returns an associative array of information about the file referenced by *handle*. The following values (given here with their numeric and key indices) are included in the array:

| | |
|---|---|
| `dev` (0) | The device on which the file resides |
| `ino` (1) | The file's inode |
| `mode` (2) | The mode with which the file was opened |
| `nlink` (3) | The number of links to this file |
| `uid` (4) | The user ID of the file's owner |
| `gid` (5) | The group ID of the file's owner |

| | |
|---|---|
| `rdev` (6) | The device type (if the file is on an inode device) |
| `size` (7) | The file's size (in bytes) |
| `atime` (8) | The time of last access (in Unix timestamp format) |
| `mtime` (9) | The time of last modification (in Unix timestamp format) |
| `ctime` (10) | The time the file was created (in Unix timestamp format) |
| `blksize` (11) | The blocksize (in bytes) for the filesystem |
| `blocks` (12) | The number of blocks allocated to the file |

**ftell.** `int ftell(resource` *handle*`)`

Returns the byte offset to which the file referenced by *handle* is set. If an error occurs, returns `false`.

**ftruncate.** `bool ftruncate(resource` *handle*`, int` *length*`)`

Truncates the file referenced by *handle* to *length* bytes. Returns `true` if the operation is successful and `false` if not.

**func_get_arg.** `mixed func_get_arg(int` *index*`)`

Returns the *index* element in the function argument array. If called outside a function, or if *index* is greater than the number of arguments in the argument array, `func_get_arg()` generates a warning and returns `false`.

**func_get_args.** `array func_get_args()`

Returns the array of arguments given to the function as an indexed array. If called outside a function, `func_get_args()` returns `false` and generates a warning.

**func_num_args.** `int func_num_args()`

Returns the number of arguments passed to the current user-defined function. If called outside a function, `func_num_args()` returns `false` and generates a warning.

**function_exists.** `bool function_exists(string` *function*`)`

Returns `true` if a function with *function* has been defined (both user-defined and built-in functions are checked), and `false` otherwise. The comparison to check for a matching function is case-insensitive.

**fwrite.** `int fwrite(resource` *handle*`, string` *string*`[, int` *length*`])`

Writes *string* to the file referenced by *handle*. The file must be open with write privileges. If *length* is given, only that many bytes of the string will be written. Returns the number of bytes written, or `–1` on error.

**gc_collect_cycles.** `int gc_collect_cycles()`

Performs a garbage collection cycle and returns the number of references that were freed. Does nothing if garbage collection is not currently enabled.

**gc_disable.** `void gc_disable()`

Disables the garbage collector. If the garbage collector was on, performs a collection prior to disabling it.

**gc_enable.** `void gc_enable()`

Enables the garbage collector; typically, only very long-running scripts can benefit from the garbage collector.

**gc_enabled.** `bool gc_enabled()`

Returns `true` if the garbage collector is currently enabled, and `false` if it's disabled.

**get_browser.** `mixed get_browser([string` *name*`[, bool` *return_array*`]])`

Returns an object containing information about the user's current browser, as found in `$HTTP_USER_AGENT`, or the browser identified by the user agent *name*. The information is gleaned from the *browscap.ini* file. The version of the browser and various capabilities of the browser—such as whether or not the browser supports frames, cookies, and so on—are returned in the object. If `return_array` is `true`, an array will be returned rather than an object.

**get_called_class.** `string get_called_class()`

Returns the name of the class that a static method was called on via late static binding, or `false` if called outside a class static method.

**get_cfg_var.** `string get_cfg_var(string` *name*`)`

Returns the value of the PHP configuration variable *name*. If *name* does not exist, `get_cfg_var()` returns `false`. Only those configuration variables set in a configuration file, as returned by `cfg_file_path()`, are returned by this function; compile-time settings and Apache configuration file variables are not returned.

**get_class.** `string get_class(object` *object*`)`

Returns the name of the class of which the given object is an instance. The class name is returned as a lowercase string. If *object* is not an object, then `false` is returned.

**get_class_methods.** `array get_class_methods(mixed` *class*`)`

If the parameter is a string, returns an array containing the names of each method defined for the specified *class*. If the parameter is an object, this function returns the methods defined in the class of which the object is an instance.

**get_class_vars.** `array get_class_vars(string` *class*`)`

Returns an associative array of default properties for the given *class*. For each property, an element with a key of the property name and a value of the default value is added to the array. Properties that do not have default values are not returned in the array.

**get_current_user.** `string get_current_user()`

Returns the name of the user under whose privileges the current PHP script is executing.

**get_declared_classes.** `array get_declared_classes()`

Returns an array containing the name of each defined class. This includes any classes defined in extensions currently loaded in PHP.

**get_declared_interfaces.** `array get_declared_interfaces()`

Returns an array containing the name of each declared interface. This includes any interfaces declared in extensions currently loaded in PHP and built-in interfaces.

**get_declared_traits.** `array get_declared_traits()`

Returns an array containing the name of each defined trait. This includes any traits defined in extensions currently loaded in PHP.

**get_defined_constants.** `array get_defined_constants([bool` *categories*`])`

Returns an associative array of all constants defined by extensions and the `define()` function and their values. If *categories* is set and is `true`, the associative array contains subarrays, one for each category of constant.

**get_defined_functions.** `array get_defined_functions()`

Returns an array containing the name of each defined function. The returned array is an associative array with two keys, `internal` and `user`. The value of the first key is an array containing the names of all internal PHP functions; the value of the second key is an array containing the names of all user-defined functions.

**get_defined_vars.** `array get_defined_vars()`

Returns an array of all variables defined in the environment, server, global, and local scopes.

**get_extension_funcs.** `array get_extension_funcs(string` *name*`)`

Returns an array of functions provided by the extension specified by *name*.

**get_headers.** `array get_headers(string` *url*`[, int` *format*`])`

Returns an array of headers that are sent by the remote server for the page given in *url*. If *format* is `0` or not set, the headers are returned in a simple array, with each entry in the array corresponding to a single header. If `format` is set and is `1`, an associative array is returned with keys and values corresponding to the header fields.

**get_html_translation_table.** `array get_html_translation_table([int` *which*`[, int` *style*`[, string` *encoding*`]]])`

Returns the translation table used by either `htmlspecialchars()` or `htmlentities()`. If *which* is `HTML_ENTITIES`, the table used by `htmlentities()` is returned; if *which* is `HTML_SPECIALCHARS`, the table used by `htmlspecialchars()` is returned. Optionally, you can specify which quotes style you want returned; the possible values are the same as those in the translation functions:

| | |
|---|---|
| `ENT_COMPAT` (default) | Converts double quotes, but not single quotes |
| `ENT_NOQUOTES` | Does not convert either double quotes or single quotes |
| `ENT_QUOTES` | Converts both single and double quotes |
| `ENT_HTML401` | Table for HTML 4.01 entities |
| `ENT_XML1` | Table for XML 1 entities |
| `ENT_XHTML` | Table for XHTML entities |
| `ENT_HTML5` | Table for HTML 5 entities |

The `encoding` optional parameter has the following possible selections:

| | |
|---|---|
| `ISO-8859-1` | Western European, Latin-1. |
| `ISO-8859-5` | Cyrillic charset (Latin/Cyrillic), rarely used. |
| `ISO-8859-15` | Western European, Latin-9. Adds the Euro sign, French and Finnish letters missing in Latin-1. |
| `UTF-8` | ASCII compatible multibyte 8-bit Unicode. |
| `cp866` | DOS-specific Cyrillic charset. |
| `cp1251` | Windows-specific Cyrillic charset. |
| `cp1252` | Windows-specific charset for Western European. |
| `KOI8-R` | Russian. |

| | |
|---|---|
| BIG5 | Traditional Chinese, mainly used in Taiwan. |
| GB2312 | Simplified Chinese, national standard character set. |
| BIG5-HKSCS | Big5 with Hong Kong extensions, Traditional Chinese. |
| Shift_JIS | Japanese. |
| EUC-JP | Japanese. |
| MacRoman | Charset that was used by macOS. |
| "" | An empty string activates detection from script encoding (Zend multibyte), default_charset, and current locale, in this order. Not recommended. |

**get_included_files.** `array get_included_files()`

Returns an array of the files included into the current script by `include()`, `include_once()`, `require()`, and `require_once()`.

**get_include_path.** `string get_include_path()`

Returns the value of the include path configuration option, giving you a list of include path locations. If you want to split the returned value into individual entries, be sure to split on the `PATH_SEPARATOR` constant, which is set separately for Unix and Windows compiles:

```
$paths = split(PATH_SEPARATOR, get_include_path());
```

**get_loaded_extensions.** `array get_loaded_extensions([ bool` *zend_extensions*`])`

Returns an array containing the names of every extension compiled and loaded into PHP. If the `zend_extensions` option is `true`, only return the Zend extensions; it defaults to `false`.

**get_meta_tags.** `array get_meta_tags(string` *path*`[, int` *include*`])`

Parses the file *path* and extracts any HTML meta tags it locates. Returns an associative array, the keys of which are `name` attributes for the meta tags, and the values of which are the appropriate values for the tags. The keys are in lowercase regardless of the case of the original attributes. If *include* is specified and `true`, the function searches for *path* in the include path.

**getmygid.** `int getmygid()`

Returns the group ID for the PHP process executing the current script. If the group ID cannot be determined, `false` is returned.

**getmyuid.** `int getmyuid()`

Returns the user ID for the PHP process executing the current script. If the user ID cannot be determined, `false` is returned.

**get_object_vars.** `array get_object_vars(object` *object*`)`

Returns an associative array of the properties for the given *object*. For each property, an element with a key of the property name and a value of the current value is added to the array. Properties that do not have current values are not returned in the array, even if they are defined in the class.

**get_parent_class.** `string get_parent_class(mixed` *object*`)`

Returns the name of the parent class for the given *object*. If the object does not inherit from another class, returns an empty string.

**get_resource_type.** `string get_resource_type(resource` *handle*`)`

Returns a string representing the type of the specified resource *handle*. If *handle* is not a valid resource, the function generates an error and returns `false`. The kinds of resources available are dependent on the extensions loaded, but include `file`, `mysql link`, and so on.

**getcwd.** `string getcwd()`

Returns the path of the PHP process's current working directory.

**getdate.** `array getdate([int` *timestamp*`])`

Returns an associative array containing values for various components for the given *timestamp* time and date. If no *timestamp* is given, the current date and time is used. A variation of the `date()` function. The array contains the following keys and values:

| | |
|---|---|
| `seconds` | Seconds |
| `minutes` | Minutes |
| `hours` | Hours |
| `mday` | Day of the month |
| `wday` | Numeric day of the week (Sunday is `0`) |
| `mon` | Month |
| `year` | Year |
| `yday` | Day of the year |
| `weekday` | Name of the day of the week (Sunday through Saturday) |
| `month` | Name of the month (January through December) |

**getenv.** `string getenv(string` *name*`)`

Returns the value of the environment variable *name*. If *name* does not exist, `getenv()` returns `false`.

**gethostbyaddr.** `string gethostbyaddr(string` *address*`)`

Returns the hostname of the machine with the IP address *address*. If no such address can be found, or if *address* doesn't resolve to a hostname, *address* is returned.

**gethostbyname.** `string gethostbyname(string` *host*`)`

Returns the IP address for *host*. If no such host exists, *host* is returned.

**gethostbynamel.** `array gethostbynamel(string` *host*`)`

Returns an array of IP addresses for *host*. If no such host exists, returns `false`.

**gethostname.** `string gethostname()`

Returns the hostname of the machine running the current script.

**getlastmod.** `int getlastmod()`

Returns the Unix timestamp value for the last modification date of the file containing the current script. If an error occurs while retrieving the information, returns `false`.

**getmxrr.** `bool getmxrr(string` *host*`, array &`*hosts*`[, array &`*weights*`])`

Searches DNS for all Mail Exchanger (MX) records for *host*. The results are put into the array *hosts*. If given, the weights for each MX record are put into *weights*. Returns `true` if any records are found and `false` if none are found.

**getmyinode.** `int getmyinode()`

Returns the inode value of the file containing the current script. If an error occurs, returns `false`.

**getmypid.** `int getmypid()`

Returns the process ID for the PHP process executing the current script. When PHP runs as a server module, any number of scripts may share the same process ID, so it is not necessarily a unique number.

**getopt.** `array getopt(string` *short_options*`[, array` *long_options*`])`

Parses the command-line arguments list used to invoke the current script and returns an associative array of optional name/value pairs. The *short_options* and *long_options* parameters define the command-line arguments to parse.

The *short_options* parameter is a single string, with each character representing a single argument passed into the script via a single hyphen. For example, the short options string `"ar"` matches the command-line arguments `-a -r`. Any character followed by a single colon `:` requires a value to match, while any character followed by two colons `::` optionally includes a value to match. For example, `"a:r::x"` would match the command-line arguments `-aTest -r -x` but not `-a -r -x`.

The *long_options* parameter is an array of strings, with each element representing a single argument passed into the script via a double hyphen. For example, the element `"verbose"` matches the command-line argument `--verbose`. All parameters specified in the *long_options* parameter optionally match values in the command line separated from the option name with an equals sign. For example, `"verbose"` will match both `--verbose` and `--verbose=1`.

**getprotobyname.** `int getprotobyname(string` *name*`)`

Returns the protocol number associated with *name* in */etc/protocols*.

**getprotobynumber.** `string getprotobynumber(int` *protocol*`)`

Returns the protocol name associated with *protocol* in */etc/protocols*.

**getrandmax.** `int getrandmax()`

Returns the largest value that can be returned by `rand()`.

**getrusage.** `array getrusage([int` *who*`])`

Returns an associative array of information describing the resources being used by the process running the current script. If *who* is specified and is equal to `1`, information about the process's children is returned. A list of the keys and descriptions of the values can be found under the `getrusage(2)` Unix command.

**getservbyname.** `int getservbyname(string` *service*`, string` *protocol*`)`

Returns the port associated with *service* in */etc/services*. *protocol* must be either TCP or UDP.

**getservbyport.** `string getservbyport(int port, string protocol)`

Returns the service name associated with *port* and *protocol* in */etc/services*. *protocol* must be either TCP or UDP.

**gettimeofday.** `mixed gettimeofday([ bool return_float])`

Returns an associative array containing information about the current time, as obtained through `gettimeofday(2)`. When `return_float` is set to `true`, a float is returned rather than an array.

The array contains the following keys and values:

| | |
|---|---|
| `sec` | The current number of seconds since the Unix epoch |
| `usec` | The current number of microseconds to add to the number of seconds |
| `minuteswest` | The number of minutes west of Greenwich the current time zone is |
| `dsttime` | The type of Daylight Saving Time correction to apply (during the appropriate time of year, a positive number if the time zone observes Daylight Saving Time) |

**gettype.** `string gettype(mixed value)`

Returns a string description of the type of *value*. The possible values for *value* are `"boolean"`, `"integer"`, `"float"`, `"string"`, `"array"`, `"object"`, `"resource"`, `"NULL"`, and `"unknown type"`.

**glob.** `globarray(string pattern[, int flags])`

Returns a list of filenames matching the shell wildcard pattern given in *pattern*. The following characters and sequences make matches:

| | |
|---|---|
| `*` | Matches any number of any character (equivalent to the regex pattern `.*`) |
| `?` | Matches any one character (equivalent to the regex pattern `.`) |

For example, to process every JPEG file in a particular directory, you might write:

```
foreach(glob("/tmp/images/*.jpg") as $filename) {
 // do something with $filename
}
```

The *flags* value is a bitwise OR of any of the following values:

| | |
|---|---|
| `GLOB_MARK` | Adds a slash to each item returned |
| `GLOB_NOSORT` | Returns files in the same order as found in the directory itself. If this is not specified, the names are sorted by ASCII value |
| `GLOB_NOCHECK` | If no files matching *pattern* are found, *pattern* is returned |
| `GLOB_NOESCAPE` | Treat backslashes in *pattern* as backslashes, rather than as the start of an escape sequence |

| | |
|---|---|
| GLOB_BRACE | In addition to the normal matches, strings in the form {foo, bar, baz} match either "foo", "bar", or "baz" |
| GLOB_ONLYDIR | Returns only directories matching *pattern* |
| GLOB_ERR | Stop on read errors |

**gmdate.** `string gmdate(string` *format*`[, int` *timestamp*`])`

Returns a formatted string for a timestamp date and time. Identical to `date()`, except that it always uses Greenwich Mean Time (GMT) rather than the time zone specified on the local machine.

**gmmktime.** `int gmmktime(int` *hour*`, int` *minutes*`, int` *seconds*`, int` *month*`, int` *day*`, int` *year*`, int` *is_dst*`)`

Returns a timestamp date and time value from the provided set of values. Identical to `mktime()`, except that the values represent a GMT time and date rather than one in the local time zone.

**gmstrftime.** `string gmstrftime(string` *format*`[, int` *timestamp*`])`

Formats a GMT timestamp. See `strftime` for more information on how to use this function.

**hash.** `string hash(string` *algorithm*`, string` *data* `[, bool` *output*`])`

Generates a hash value on the provided *data* based on the given *algorithm*. When *output* is set to `true`, defaults to `false`; the returned hash value is raw binary data. *Algorithm* values can be `md5`, `sha1`, `sha256`, and so on. See `hash_algos` for more algorithm information.

**hash_algos.** `array hash_algos(void)`

Returns a numerically indexed array of all the supported hash algorithms.

**hash_file.** `string hash_file(string` *algorithm*`, string` *filename* `[, bool` *output*`])`

Generates a hash value string on the contents of *filename* (URL for location of the file) based on the given *algorithm*. When *output* is set to `true`, defaults to `false`; the returned hash value is raw binary data. *Algorithm* values can be `md5`, `sha1`, `sha256`, and so on.

**header.** `void header(string` *header*`[, bool` *replace* `[, int` *http_response_code*`]])`

Sends *header* as a raw HTTP header string; must be called before any output is generated (including blank lines—a common mistake). If the *header* is a `Location` header, PHP also generates the appropriate `REDIRECT` status code. If *replace* is specified and `false`, the header does not replace a header of the same name; otherwise, the header replaces any header of the same name.

**header_remove.** `void header_remove([string` *header*`])`

If *header* is specified, removes the HTTP header with named *header* from the current response. If *header* is not specified, or is an empty string, removes all headers generated by the `header()` function from the current response. Note that the headers cannot be removed if they have already been sent to the client.

**headers_list.** `array headers_list()`

Returns an array of the HTTP response headers that have been prepared for sending (or have been sent) to the client.

**headers_sent.** `bool headers_sent([ string &`*file* `[, int &`*line*`]])`

Returns `true` if the HTTP headers have already been sent. If they have not yet been sent, the function returns `false`. If *file* and *line* options are provided, the filename and the line number where the output began are placed in *file* and *line* variables.

**hebrev.** `string hebrev(string` *string*`[, int` *size*`])`

Converts the logical Hebrew text *string* to visual Hebrew text. If the second parameter is specified, each line will contain no more than *size* characters; the function attempts to avoid breaking words.

**hex2bin.** `string hex2bin(string` *hex*`)`

Converts *hex* to its binary value.

**hexdec.** `number hexdec(string` *hex*`)`

Converts *hex* to its decimal value. Up to a 32-bit number, or 2,147,483,647 decimal (0x7FFFFFFF hexadecimal), can be converted.

**highlight_file.** `mixed highlight_file(string` *filename* `[, bool` *return*`])`

Prints a syntax-colored version of the PHP source file *filename* using PHP's built-in syntax highlighter. Returns `true` if *filename* exists and is a PHP source file; otherwise,

returns `false`. If *return* is `true`, the highlighted code is returned as a string rather than being sent to the output device.

**highlight_string.** `mixed highlight_string(string` *source* `[, bool` *return*`])`

Prints a syntax-colored version of the string *source* using PHP's built-in syntax highlighter. Returns `true` if successful; otherwise, returns `false`. If *return* is `true`, then the highlighted code is returned as a string rather than being sent to the output device.

**hrtime.** `mixed hrtime([bool` *get_as_number*`])`

Returns the system's high-resolution time as an array, counted from an arbitrary point in time. The delivered timestamp is monotonic and cannot be adjusted. *get_as_number* returns as an array (`false`) or a number (`true`); defaults to `false`.

**htmlentities.** `string htmlentities(string` *string*`[, int` *style*`[, string` *encoding* `[, bool` *double_encode*`]]])`

Converts all characters in *string* that have special meaning in HTML and returns the resulting string. All entities defined in the HTML standard are converted. If supplied, *style* determines the manner in which quotes are translated. The possible values for *style* are:

| | |
|---|---|
| `ENT_COMPAT` (default) | Converts double quotes, but not single quotes |
| `ENT_NOQUOTES` | Does not convert either double quotes or single quotes |
| `ENT_QUOTES` | Converts both single and double quotes |
| `ENT_SUBSTITUTE` | Replace invalid code unit sequences with a Unicode Replacement Character |
| `ENT_DISALLOWED` | Replace invalid code points for the given document type with a Unicode Replacement Character |
| `ENT_HTML401` | Handle code as HTML 4.01 |
| `ENT_XML1` | Handle code as XML 1 |
| `ENT_XHTML` | Handle code as XHTML |
| `ENT_HTML5` | Handle code as HTML 5 |

If supplied, *encoding* determines the final encoding for the characters. The possible values for *encoding* are:

| | |
|---|---|
| `ISO-8859-1` | Western European, Latin-1 |
| `ISO-8859-5` | Cyrillic charset (Latin/Cyrillic), rarely used |
| `ISO-8859-15` | Western European, Latin-9. Adds the Euro sign, French and Finnish letters missing in Latin-1. |
| `UTF-8` | ASCII-compatible multi-byte 8-bit Unicode |
| `cp866` | DOS-specific Cyrillic charset |
| `cp1251` | Windows-specific Cyrillic charset |
| `cp1252` | Windows-specific charset for Western European |

| KOI8-R | Russian |
|---|---|
| BIG5 | Traditional Chinese, mainly used in Taiwan |
| GB2312 | Simplified Chinese, national standard character set |
| BIG5-HKSCS | Big5 with Hong Kong extensions, Traditional Chinese |
| Shift_JIS | Japanese |
| EUC-JP | Japanese |
| MacRoman | Charset that was used by Mac OS |
| "" | An empty string activates detection from script encoding (Zend multibyte), `default_charset`, and current locale, in this order. Not recommended. |

**html_entity_decode.** `string html_entity_decode(string` *string*`[, int` *style*`[, string` *encoding*`]])`

Converts all HTML entities in *string* to the equivalent character. All entities defined in the HTML standard are converted. If supplied, *style* determines the manner in which quotes are translated. The possible values for *style* are the same as those for *htmlentities*.

If supplied, *encoding* determines the final encoding for the characters. The possible values for *encoding* are the same as those for *htmlentities*.

**htmlspecialchars.** `string htmlspecialchars(string` *string*`[, int` *style*`[, string` *encoding*`[, bool` *double_encode*`]]])`

Converts characters in *string* that have special meaning in HTML and returns the resulting string. A subset of all HTML entities covering the most common characters is used to perform the translation. If supplied, *style* determines the manner in which quotes are translated. The characters translated are:

- Ampersand (&) becomes `&`
- Double quotes (") become `"`
- Single quote (') becomes `'`
- Less than sign (<) becomes `<`
- Greater than sign (>) becomes `>`

The possible values for *style* are the same as those for *htmlentities*. If supplied, *encoding* determines the final encoding for the characters. The possible values for *encoding* are the same as those for *htmlentities*. When *double_encode* is turned off, PHP will not encode existing *htmlentities*.

**htmlspecialchars_decode.** `string htmlspecialchars_decode(string` *string*`[, int` *style*`])`

Converts HTML entities in *string* to characters. A subset of all HTML entities covering the most common characters is used to perform the translation. If supplied, *style* determines the manner in which quotes are translated. See `htmlentities()` for the possible values for *style*. The characters translated are those found in `htmlspecialchars()`.

**http_build_query.** `string http_build_query(mixed` *values*`[, string` *prefix* `[, string` *arg_separator* `[, int` *enc_type*`]]])`

Returns a URL-encoded query string from *values*. The array values can be either numerically indexed or associative (or a combination). Because strictly numeric names may be illegal in some languages interpreting the query string on the other side (PHP, for example), if you use numeric indices in values, you should also provide *prefix*. The value of *prefix* is prepended to all numeric names in the resulting query string. The *arg_separator* allows for assigning a customized delimiter and the *enc_type* option allows for selecting different encoding types.

**hypot.** `float hypot(float` *x*`, float` *y*`)`

Calculates and returns the length of the hypotenuse of a right-angle triangle whose other sides have lengths *x* and *y*.

**idate.** `int idate(string` *format*`[, int` *timestamp*`])`

Formats a time and date as an integer according to the *format* string provided in the first parameter. If the second parameter is not specified, the current time and date is used. The following characters are recognized in the *format* string:

| | |
|---|---|
| `B` | Swatch internet time |
| `d` | Day of the month |
| `h` | Hour in 12-hour format |
| `H` | Hour in 24-hour format |
| `i` | Minutes |
| `I` | `1` if Daylight Saving Time; `0` otherwise |
| `j` | Day of the month (e.g., 1 through 31) |
| `L` | `0` if the year is not a leap year; `1` if it is |
| `m` | Month (1 through 12) |
| `s` | Seconds |
| `t` | Number of days in the month, from 28 to 31 |
| `U` | Seconds since the Unix epoch |

| | |
|---|---|
| w | Numeric day of the week, starting with 0 for Sunday |
| W | Numeric week of the year according to ISO 8601 |
| Y | Year with four digits (e.g., 1998) |
| y | Year with one or two digits (e.g., 98) |
| z | Day of the year, from 1 through 365 |
| Z | Time zone offset in seconds, from –43200 (far west of UTC) to 43200 (far east of UTC) |

Any characters in the *format* string not matching one of the above are ignored. Although the character strings used in `idate` are similar to those in `date`, because `idate` returns an integer, in places where `date` would return a two-digit number with leading zero, the leading zero is not preserved; for example, `date('y');` will return `05` for a timestamp in 2005, while `idate('y');` will return `5`.

**ignore_user_abort.** `int ignore_user_abort([string` *ignore*`])`

Sets whether the client disconnecting from the script should stop processing of the PHP script. If *ignore* is `true`, the script will continue processing, even after a client disconnect. Returns the current value; if *ignore* is not given, the current value is returned without a new value being set.

**implode.** `string implode(string` *separator*`, array` *strings*`)`

Returns a string created by joining every element in *strings* with *separator*.

**inet_ntop.** `string inet_ntop(string` *address*`)`

Unpacks the packed IPv4 or IPv6 IP address *address* and returns it as a human-readable string.

**inet_pton.** `string inet_pton(string` *address*`)`

Packs the human-readable IP address *address* into a 32- or 128-bit value and returns it.

**in_array.** `bool in_array(mixed` *value*`, array` *array*`[, bool` *strict*`])`

Returns `true` if the given *value* exists in the *array*. If the third argument is provided and is `true`, the function will return `true` only if the element exists in the array and has the same type as the provided value (that is, `"1.23"` in the array will not match `1.23` as the argument). If the argument is not found in the array, the function returns `false`.

**ini_get.** `string ini_get(string` *variable*`)`

Returns the value for the configuration option *variable*. If *variable* does not exist, returns `false`.

**ini_get_all.** `array ini_get_all([string` *extension* `[, bool` *details*`]])`

Returns all configuration options as an associative array. If a valid *extension* is specified then only values pertaining to that named *extension* are returned. If *details* is `true` (default), then detail settings are retrieved. Each value returned in the array is an associative array with three keys:

| | |
|---|---|
| `global_value` | The global value for the configuration option, as set in *php.ini* |
| `local_value` | The local override for the configuration option, as set through `ini_set()`, for example |
| `access` | A bitmask with the levels at which the value can be set (see `ini_set` for more information on access levels) |

**ini_restore.** `void ini_restore(string` *variable*`)`

Restores the value for the configuration option *variable*. This is done automatically when a script completes execution for all configuration options set using `ini_set()` during the script.

**ini_set.** `string ini_set(string` *variable*`, string` *value*`)`

Sets the configuration option *variable* to *value*. Returns the previous value if successful, or `false` if not. The new value is kept for the duration of the current script and is restored after the script ends.

**intdiv.** `int intdiv (int` *dividend*`, int` *vdivisor*`)`

Returns the quotient of the division of *dividend* by *divisor*. The quotient is returned as an integer.

**interface_exists.** `bool interface_exists(string` *name* `[, bool` *autoload_interface*`])`

Returns `true` if an interface named *name* has been defined and `false` otherwise. By default, the function will call `__autoload()` on the interface; if `autoload_interface` is set and is `false`, `__autoload()` will not be called.

**intval.** `int intval(mixed` *value*`[, int` *base*`])`

Returns the integer value for *value* using the optional base *base* (if unspecified, base-10 is used). If *value* is a nonscalar value (object or array), the function returns `0`.

**ip2long.** `int ip2long(string` *address*`)`

Converts a dotted (standard format) IP address to an IPv4 address.

**is_a.** `bool is_a(object` *object*`, string` *class* `[, bool` *allow_string*`])`

Returns `true` if *object* is of the class *class*, or if its class has *class* as one of its parents; otherwise, returns `false`. If *allow_string* is `false`, then string *class* name as *object* is not allowed.

**is_array.** `bool is_array(mixed` *value*`)`

Returns `true` if *value* is an array; otherwise, returns `false`.

**is_bool.** `bool is_bool(mixed` *value*`)`

Returns `true` if *value* is a boolean; otherwise, returns `false`.

**is_callable.** `int is_callable(callable` *callback*`[, int` *lazy*`[, string` *name*`]])`

Returns `true` if *callback* is a valid callback, `false` otherwise. To be valid, *callback* must either be the name of a function or an array containing two values—an object and the name of a method on that object. If *lazy* is given and is `true`, the actual existence of the function in the first form, or that the first element in callback is an object with a method named the second element, is not checked. The arguments merely have to have the correct kind of values to qualify as `true`. If supplied, the final argument is filled with the callable name for the function—though in the case of the callback being a method on an object, the resulting name in *name* is not actually usable to call the function directly.

**is_countable.** `bool is_countable(mixed` *variable*`)`

Verify that the contents of *variable* is an array or an object implementing `Countable`.

**is_dir.** `bool is_dir(string` *path*`)`

Returns `true` if *path* exists and is a directory; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**is_executable.** `bool is_executable(string` *path*`)`

Returns `true` if *path* exists and is executable; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**is_file.** `bool is_file(string` *path*`)`

Returns `true` if *path* exists and is a file; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**is_finite.** `bool is_finite(float` *value*`)`

Returns `true` if *value* is not positive or negative infinity, and `false` otherwise.

**is_float.** `bool is_float(mixed` *value*`)`

Returns `true` if *value* is a float; otherwise, returns `false`.

**is_infinite.** `bool is_infinite(float` *value*`)`

Returns `true` if *value* is positive or negative infinity, and `false` otherwise.

**is_int.** `bool is_int(mixed` *value*`)`

Returns `true` if *value* is an integer; otherwise, returns `false`.

**is_iterable.** `bool is_iterable(mixed` *value*`)`

Returns `true` if *value* is an iterable pseudotype, an array, or a traversable object; otherwise, returns `false`.

**is_link.** `bool is_link(string` *path*`)`

Returns `true` if *path* exists and is a symbolic link file; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**is_nan.** `bool is_nan(float` *value*`)`

Returns `true` if *value* is a "not a number" value, or `false` if *value* is a number.

**is_null.** `bool is_null(mixed` *value*`)`

Returns `true` if *value* is null (that is, the keyword `NULL`); otherwise, returns `false`.

**is_numeric.** `bool is_numeric(mixed` *value*`)`

Returns `true` if *value* is an integer, a floating-point value, or a string containing a number; otherwise, returns `false`.

**is_object.** `bool is_object(mixed` *value*`)`

Returns `true` if *value* is an object; otherwise, returns `false`.

**is_readable.** `bool is_readable(string` *path*`)`

Returns `true` if *path* exists and is readable; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**is_resource.** `bool is_resource(mixed` *value*`)`

Returns `true` if *value* is a resource; otherwise, returns `false`.

**is_scalar.** `bool is_scalar(mixed` *value*`)`

Returns `true` if *value* is a scalar value—an integer, boolean, floating-point value, resource, or string. If *value* is not a scalar value, the function returns `false`.

**is_string.** `bool is_string(mixed` *value*`)`

Returns `true` if *value* is a string; otherwise, returns `false`.

**is_subclass_of.** `bool is_subclass_of(object` *object*`, string` *class* `[, bool` *allow_string*`])`

Returns `true` if *object* is an instance of the class *class* or an instance of a subclass of *class*. If not, the function returns `false`. If the *allow_string* parameter is set to `false`, *class* "as object" is not allowed.

**is_uploaded_file.** `bool is_uploaded_file(string` *path*`)`

Returns `true` if *path* exists and was uploaded to the web server using the `file` element in a web page form; otherwise, returns `false`. See Chapter 8 for more information on using uploaded files.

**is_writable.** `bool is_writable(string` *path*`)`

Returns `true` if *path* exists and is a directory; otherwise, returns `false`. This information is cached; you can clear the cache with `clearstatcache()`.

**isset.** `bool isset(mixed` *value1*`)[, ... mixed` *valueN*`])`

Returns `true` if *value*, a variable, has been set; if the variable has never been set or has been `unset()`, the function returns `false`. If multiple *values* are provided, then `isset` will return `true` only if they are all set.

**json_decode.** `mixed json_decode(string` *json*`[, bool` *assoc* `[, int` *depth* `[, int` *options*`]]])`

Takes a JSON-encoded string, *json*, and returns it as a converted PHP variable. If the JSON cannot be decoded, then `NULL` is returned. When *assoc* is `true`, objects will be converted into associative arrays. *depth* is user-controlled recursion level. *options* controls how some of the provided data in the string can be alternatively returned.

**json_encode.** `mixed json_encode(mixed` *value* `[, int` *options* `[, int` *depth*`]])`

Returns a string containing the JSON representation of *value*. *options* controls how some of the provided data in the string can be alternatively returned. If *depth* is used, it must be greater than zero.

**key.** `mixed key(array &`*array*`)`

Returns the key for the element currently pointed to by the internal array pointer.

**krsort.** `int krsort(array` *array*`[, int` *flags*`])`

Sorts an array by key in reverse order, maintaining the keys for the array values. The optional second parameter contains additional sorting flags. See Chapter 5 and `sort` for more information on using this function.

**ksort.** `int ksort(array` *array*`[, int` *flags*`])`

Sorts an array by key, maintaining the keys for the array values. The optional second parameter contains additional sorting flags. See Chapter 5 and `sort` for more information on using this function.

**lcfirst.** `string lcfirst(string` *string*`)`

Returns *string* with the first character, if alphabetic, converted to lowercase. The table used for converting characters is locale-specific.

**lcg_value.** `float lcg_value()`

Returns a pseudorandom float number between 0 and 1, inclusive, using a linear congruential number generator.

**lchgrp.** `bool lchgrp(string` *path*`, mixed` *group*`)`

Changes the group for the symlink *path* to *group*; PHP must have appropriate privileges for this function to work. Returns `true` if the change was successful and `false` if not.

**lchown.** `bool lchown(string` *path*`, mixed` *user*`)`

Changes ownership for the symlink *path* to the user named *user*. PHP must have appropriate privileges (generally, root) for the function to operate. Returns `true` if the change was successful and `false` if not.

**levenshtein.** `int levenshtein(string` *one*`, string` *two*`[, int` *insert*`, int` *replace*`,int` *delete*`]) int levenshtein(string` *one*`, string` *two*`[, mixed` *callback*`])`

Calculates the Levenshtein distance between two strings. This is the number of characters you have to replace, insert, or delete to transform *one* into *two*. By default, replacements, inserts, and deletes have the same cost, but you can specify different costs with *insert*, *replace*, and *delete*. In the second form, just the total cost of inserts, replaces, and deletes are returned, not broken down.

**link.** `bool link(string` *path*`, string` *new*`)`

Creates a hard link to *path* at the path *new*. Returns `true` if the link was successfully created and `false` if not.

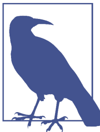
**linkinfo.** `int linkinfo(string` *path*`)`

Returns `true` if *path* is a link and if the file referenced by *path* exists. Returns `false` if *path* is not a link, if the file referenced by it does not exist, or if an error occurs.

**list.** `array list(mixed` *value1*`[, ...` *valueN*`])`

Assigns a set of variables from elements in an array. For example:

```
list($first, $second) = array(1, 2); // $first = 1, $second = 2
```



`list` is actually a language construct.

**localeconv.** `array localeconv()`

Returns an associative array of information about the current locale's numeric and monetary formatting. The array contains the following elements:

| | |
|---|---|
| `decimal_point` | Decimal-point character |
| `thousands_sep` | Separator character for thousands |
| `grouping` | Array of numeric groupings; indicates where the number should be separated using the thousands separator character |

| | |
|---|---|
| `int_curr_symbol` | International currency symbol (e.g., USD) |
| `currency_symbol` | Local currency symbol (e.g., $) |
| `mon_decimal_point` | Decimal-point character for monetary values |
| `mon_thousands_sep` | Separator character for thousands in monetary values |
| `positive_sign` | Sign for positive values |
| `negative_sign` | Sign for negative values |
| `int_frac_digits` | International fractional digits |
| `frac_digits` | Local fractional digits |
| `p_cs_precedes` | `true` if the local currency symbol precedes a positive value; `false` if it follows the value |
| `p_sep_by_space` | `true` if a space separates the local currency symbol from a positive value |
| `p_sign_posn` | `0` if parentheses surround the value and currency symbol for positive values, `1` if the sign precedes the currency symbol and value, `2` if the sign follows the currency symbol and value, `3` if the sign precedes the currency symbol, and `4` if the sign follows the currency symbol |
| `n_cs_precedes` | `true` if the local currency symbol precedes a negative value; `false` if it follows the value |
| `n_sep_by_space` | `true` if a space separates the local currency symbol from a negative value |
| `n_sign_posn` | `0` if parentheses surround the value and currency symbol for negative values, `1` if the sign precedes the currency symbol and value, `2` if the sign follows the currency symbol and value, `3` if the sign precedes the currency symbol, and `4` if the sign follows the currency symbol |

**localtime.** `array localtime([int` *timestamp*`[, bool` *associative*`]])`

Returns an array of values as given by the C function of the same name. The first argument is the timestamp; if the second argument is provided and is `true`, the values are returned as an associative array. If the second argument is not provided or is `false`, a numeric array is returned. The keys and values returned are:

| | |
|---|---|
| `tm_sec` | Seconds |
| `tm_min` | Minutes |
| `tm_hour` | Hour |
| `tm_mday` | Day of the month |
| `tm_mon` | Month of the year |
| `tm_year` | Number of years since 1900 |
| `tm_wday` | Day of the week |
| `tm_yday` | Day of the year |
| `tm_isdst` | `1` if Daylight Saving Time was in effect at the date and time |

If a numeric array is returned, the values are in the order given above.

**log.** `float log(float` *number* `[, float` *base*`])`

Returns the natural log of *number*. The *base* option controls the logarithmic base that will be used; it defaults to *e*, which is a natural logarithm.

**log10.** `float log10(float` *number*`)`

Returns the base-10 logarithm of *number*.

**log1p.** `float log1p(float` *number*`)`

Returns the `log(1 +` *number*`)`, computed in such a way that the returned value is accurate even when *number* is close to zero.

**long2ip.** `string long2ip(string` *address*`)`

Converts an IPv4 address to a dotted (standard format) address.

**lstat.** `array lstat(string` *path*`)`

Returns an associative array of information about the file *path*. If *path* is a symbolic link, information about *path* is returned, rather than information about the file to which *path* points. See `fstat` for a list of the values returned and their meanings.

**ltrim.** `string ltrim(string` *string*`[, string` *characters*`])`

Returns *string* with all characters in *characters* stripped from the beginning. If *characters* is not specified, the characters stripped are `\n`, `\r`, `\t`, `\v`, `\0`, and spaces.

**mail.** `bool mail(string` *recipient*`, string` *subject*`, string` *message*`[, string` *headers* `[, string` *parameters*`]])`

Sends *message* to *recipient* via email with the subject *subject* and returns `true` if the message was successfully sent and `false` if it wasn't. If given, *headers* is added to the end of the headers generated for the message, allowing you to add cc:, bcc:, and other headers. To add multiple headers, separate them with `\n` characters (or `\r\n` characters on Windows servers). Finally, if specified, *parameters* is added to the parameters of the call to the mailer program used to send the mail.

**max.** `mixed max(mixed` *value1*`[, mixed` *value2*`[, ... mixed` *valueN*`]])`

If *value1* is an array, returns the largest number found in the values of the array. If not, returns the largest number found in the arguments.

**md5.** `string md5(string` *string* `[, bool` *binary*`])`

Calculates the MD5 encryption hash of *string* and returns it. If the *binary* option is `true`, then the `MD5` hash returned is in raw binary format (length of 16); *binary* defaults to `false`, thus making `md5` return a full 32-character hex string.

**md5_file.** `string md5_file(string` *path*`[, bool` *binary*`])`

Calculates and returns the MD5 encryption hash for the file at *path*. An MD5 hash is a 32-character hexadecimal value that can be used to checksum a file's data. If *binary* is supplied and is `true`, the result is sent as a 16-bit binary value instead.

**memory_get_peak_usage.** `int memory_get_peak_usage([bool` *actual*`])`

Returns the peak memory usage so far, in bytes, of the currently running script. If *actual* is specified and `true`, returns the actual bytes allocated; otherwise, it returns the bytes allocated through PHP's internal memory allocation routines.

**memory_get_usage.** `int memory_get_usage([bool` *actual*`])`

Returns the current memory usage, in bytes, of the currently running script. If *actual* is specified and `true`, returns the actual bytes allocated; otherwise, it returns the bytes allocated through PHP's internal memory allocation routines.

**metaphone.** `string metaphone(string` *string*`, int` *max_phonemes*`)`

Calculates the metaphone key for *string*. The maximum number of phonemes to use in calculating the value is given in *max_phonemes*. Similar-sounding English words generate the same key.

**method_exists.** `bool method_exists(object` *object*`, string` *name*`)`

Returns `true` if the object contains a method with the name given in the second parameter, and `false` otherwise. The method may be defined in the class of which the object is an instance, or in any superclass of that class.

**microtime.** `mixed microtime([ bool` *get_as_float*`])`

Returns a string in the format *microseconds seconds*, where *seconds* is the number of seconds since the Unix epoch (January 1, 1970), and *microseconds* is the microseconds portion of the time since the Unix epoch. If *get_as_float* is `true`, a float will be returned instead of a string.

**min.** `mixed min(mixed` *value1*`[, mixed` *value2*`[, ... mixed` *valueN*`]])`

If *value1* is an array, returns the smallest number found in the values of the array. If not, returns the smallest number found in the arguments.

**mkdir.** `bool mkdir(string` *path*`[, int` *mode* `[, bool` *recursive* `[, resource` *context*`]]])`

Creates the directory *path* with *mode* permissions. The mode is expected to be an octal number such as `0755`. An integer value such as `755` or a string value such as `"u+x"` will not work as expected. Returns `true` if the operation was successful and `false` if not. If recursive is used, it allows for the creation of nested directories.

**mktime.** `int mktime(int` *hours*`, int` *minutes*`, int` *seconds*`, int` *month*`, int` *day*`, int` *year* `[, int` *is_dst*`])`

Returns the Unix timestamp value corresponding to the parameters, which are supplied in the order *hours*, *minutes*, *seconds*, *month*, *day*, *year*, and (optionally) whether the value is in Daylight Saving Time. This timestamp is the number of seconds elapsed between the Unix epoch and the given date and time.

The order of the parameters is different from that of the standard Unix `mktime()` call, to make it simpler to leave out unneeded arguments. Any arguments left out are given the current local date and time.

**move_uploaded_file.** `bool move_uploaded_file(string` *from*`, string` *to*`)`

Moves the file *from* to the new location *to*. The function moves the file only if *from* was uploaded by an HTTP `POST`. If *from* does not exist or is not an uploaded file, or if any other error occurs, `false` is returned; if the operation is successful, `true` is returned.

**mt_getrandmax.** `int mt_getrandmax()`

Returns the largest value that can be returned by `mt_rand()`.

**mt_rand.** `int mt_rand([int` *min*`, int` *max*`])`

Returns a random number from *min* to *max*, inclusive, generated using the Mersenne Twister pseudorandom number generator. If *min* and *max* are not provided, returns a random number from 0 to the value returned by `mt_getrandmax()`.

**mt_srand.** `void mt_srand(int` *seed*`)`

Seeds the Mersenne Twister generator with *seed*. You should call this function with a varying number, such as that returned by `time()`, before making calls to `mt_rand()`.

**natcasesort.** `void natcasesort(array` *array*`)`

Sorts the elements in the given array using a case-insensitive *natural order* algorithm; see `natsort` for more information.

**natsort.** `bool natsort(array` *array*`)`

Sorts the values of the array using "natural order": numeric values are sorted in the manner expected by language, rather than the often bizarre order in which computers insist on putting them (ASCII ordered). For example:

```
$array = array("1.jpg", "4.jpg", "12.jpg", "2,.jpg", "20.jpg");
$first = sort($array); // ("1.jpg", "12.jpg", "2.jpg", "20.jpg", "4.jpg")
$second = natsort($array); // ("1.jpg", "2.jpg", "4.jpg", "12.jpg", "20.jpg")
```

**next.** `mixed next(array` *array*`)`

Increments the internal pointer to the element after the current element and returns the value of the element to which the internal pointer is now set. If the internal pointer already points beyond the last element in the array, the function returns `false`.

Be careful when iterating over an array using this function—if an array contains an empty element or an element with a key value of `0`, a value equivalent to `false` is returned, causing the loop to end. If an array might contain empty elements or an element with a key of `0`, use the `each` function instead of a loop with `next`.

**nl_langinfo.** `string nl_langinfo(int` *item*`)`

Returns the string containing the information for *item* in the current locale; *item* is one of a number of different values such as day names, time format strings, and so on. The actual possible values are different on different implementations of the C library; see `<langinfo.h>` on your machine for the values on your OS.

**nl2br.** `string nl2br(string` *string* `[, bool` *xhtml_lb*`])`

Returns a string created by inserting `<br />` before all newline characters in *string*. If *xhtml_lb* is `true`, then `nl2br` will use XHTML-compatible line breaks.

**number_format.** `string number_format(float` *number*`[, int` *precision*`[, string` *decimal_separator*`, string` *thousands_separator*`]])`

Creates a string representation of *number*. If *precision* is given, the number is rounded to that many decimal places; the default is no decimal places, creating an integer. If *decimal_separator* and *thousands_separator* are provided, they are used as the decimal-place character and thousands separator, respectively. They default to the English locale versions (`.` and `,`). For example:

```
$number = 7123.456;
$english = number_format($number, 2); // 7,123.45
$francais = number_format($number, 2, ',', ' '); // 7 123,45
$deutsche = number_format($number, 2, ',', '.'); // 7.123,45
```

If rounding occurs, proper rounding is performed, which may not be what you expect (see `round`).

**ob_clean.** `void ob_clean()`

Discards the contents of the output buffer. Unlike `ob_end_clean()`, the output buffer is not closed.

**ob_end_clean.** `bool ob_end_clean()`

Turns off output buffering and empties the current buffer without sending it to the client. See Chapter 15 for more information on using the output buffer.

**ob_end_flush.** `bool ob_end_flush()`

Sends the current output buffer to the client and stops output buffering. See Chapter 15 for more information on using the output buffer.

**ob_flush.** `void ob_flush()`

Sends the contents of the output buffer to the client and discards the contents. Unlike calling `ob_end_flush()`, the output buffer itself is not closed.

**ob_get_clean.** `string ob_get_clean()`

Returns the contents of the output buffer and ends output buffering.

**ob_get_contents.** `string ob_get_contents()`

Returns the current contents of the output buffer; if buffering has not been enabled with a previous call to `ob_start()`, returns `false`. See Chapter 15 for more information on using the output buffer.

**ob_get_flush.** `string ob_get_flush()`

Returns the contents of the output buffer, flushes the output buffer to the client, and ends output buffering.

**ob_get_length.** `int ob_get_length()`

Returns the length of the current output buffer, or `false` if output buffering isn't enabled. See Chapter 15 for more information on using the output buffer.

**ob_get_level.** `int ob_get_level()`

Returns the count of nested output buffers, or `0` if output buffering is not currently active.

**ob_get_status.** `array ob_get_status([bool` *verbose*`])`

Returns status information about the current output buffer. If *verbose* is supplied and is `true`, returns information about all nested output buffers.

**ob_gzhandler.** `string ob_gzhandler(string` *buffer*`[, int` *mode*`])`

This function *gzip*-compresses output before it is sent to the browser. You don't call this function directly. Rather, it is used as a handler for output buffering using the `ob_start()` function. To enable *gzip*-compression, call `ob_start()` with this function's name:

```
<ob_start("ob_gzhandler");>
```

**ob_implicit_flush.** `void ob_implicit_flush([int` *flag*`])`

If *flag* is `true` or unspecified, turns on output buffering with implicit flushing. When implicit flushing is enabled, the output buffer is cleared and sent to the client after any output (such as the `printf()` and `echo()` functions). See Chapter 15 for more information on using the output buffer.

**ob_list_handlers.** `array ob_list_handlers()`

Returns an array with the names of the active output handlers. If PHP's built-in output buffering is enabled, the array contains the value `default output handler`. If no output handlers are active, it returns an empty array.

**ob_start.** `bool ob_start([string` *callback* `[, int` *chunk* `[, bool` *erase*`]]])`

Turns on output buffering, which causes all output to be accumulated in a buffer instead of being sent directly to the browser. If *callback* is specified, it is a function (called before sending the output buffer to the client) that can modify the data in any way; the `ob_gzhandler()` function is provided to compress the output buffer in a client-aware manner. The *chunk* option can be used to trigger the flushing of the buffer when the buffer size equals the chunk number. If the *erase* option is set to `false`, then the buffer will not be deleted until the end of the script. See Chapter 15 for more information on using the output buffer.

**octdec.** `number octdec(string` *octal*`)`

Converts *octal* to its decimal value. Up to a 32-bit number, or 2,147,483,647 decimal (017777777777 octal), can be converted.

**opendir.** `resource opendir(string` *path*`[, resource context])`

Opens the directory *path* and returns a directory handle for the path that is suitable for use in subsequent `readdir()`, `rewinddir()`, and `closedir()` calls. If *path* is not a valid directory, if permissions do not allow the PHP process to read the directory, or if any other error occurs, `false` is returned.

**openlog.** `bool openlog(string` *identity*`, int` *options*`, int` *facility*`)`

Opens a connection to the system logger. Each message sent to the logger with a subsequent call to `syslog()` is prepended by *identity*. Various options can be specified by *options*; `OR` any options you want to include. The valid options are:

| | |
|---|---|
| `LOG_CONS` | If an error occurs while writing to the system log, write the error to the system console |
| `LOG_NDELAY` | Open the system log immediately |
| `LOG_ODELAY` | Delay opening the system log until the first message is written to it |
| `LOG_PERROR` | Print this message to standard error in addition to writing it to the system log |
| `LOG_PID` | Include the PID in each message |

The third parameter, *facility*, tells the system log what kind of program is logging to the system log. The following facilities are available:

| | |
|---|---|
| `LOG_AUTH` | Security and authorization errors (deprecated; if `LOG_AUTHPRIV` is available, use it instead) |
| `LOG_AUTHPRIV` | Security and authorization errors |
| `LOG_CRON` | Clock daemon (*cron* and *at*) errors |
| `LOG_DAEMON` | Errors for system daemons not given their own codes |
| `LOG_KERN` | Kernel errors |
| `LOG_LPR` | Line printer subsystem errors |
| `LOG_MAIL` | Mail errors |
| `LOG_NEWS` | USENET news system errors |
| `LOG_SYSLOG` | Errors generated internally by *syslogd* |
| `LOG_AUTHPRIV` | Security and authorization errors |
| `LOG_USER` | Generic user-level errors |
| `LOG_UUCP` | UUCP errors |

**ord.** `int ord(string` *string*`)`

Returns the ASCII value of the first character in *string*.

**output_add_rewrite_var.** `bool output_add_rewrite_var(string name, string value)`

Begins using the value rewriting output handler by appending the name and value to all HTML anchor elements and forms. For example:

```
output_add_rewrite_var('sender', 'php');

echo "<a href=\"foo.php\">\n";
echo '<form action="bar.php"></form>';

// outputs:
// <a href="foo.php?sender=php">
// <form action="bar.php"><input type="hidden" name="sender" value="php" />
// </form>
```

**output_reset_rewrite_vars.** `bool output_reset_rewrite_vars()`

Resets the value writing output handler; if the value writing output handler was in effect, any still unflushed output will no longer be affected by rewriting even if put into the buffer before this call.

**pack.** `string pack(string format, mixed arg1[, mixed arg2[, ... mixed argN]])`

Creates a binary string containing packed versions of the given arguments according to format. Each character may be followed by a number of arguments to use in that format, or an asterisk (*), which uses all arguments to the end of the input data. If no repeater argument is specified, a single argument is used for the format character. The following characters are meaningful in the *format* string:

| | |
|---|---|
| a | NUL-byte-padded string |
| A | Space-padded string |
| h | Hexadecimal string, with the low nibble first |
| H | Hexadecimal string, with the high nibble first |
| c | Signed char |
| C | Unsigned char |
| s | 16-bit, machine-dependent byte-ordered signed short |
| S | 16-bit, machine-dependent byte-ordered unsigned short |
| n | 16-bit, big-endian byte-ordered unsigned short |
| v | 16-bit, little-endian byte-ordered unsigned short |
| i | Machine-dependent size and byte-ordered signed integer |
| I | Machine-dependent size and byte-ordered unsigned integer |
| l | 32-bit, machine-dependent byte-ordered signed long |
| L | 32-bit, machine-dependent byte-ordered unsigned long |

| | |
|---|---|
| N | 32-bit, big-endian byte-ordered unsigned long |
| V | 32-bit, little-endian byte-ordered unsigned long |
| f | Float in machine-dependent size and representation |
| d | Double in machine-dependent size and representation |
| x | NUL-byte |
| X | Back up one byte |
| @ | Fill to absolute position (given by the repeater argument) with NUL-bytes |

**parse_ini_file.** `array parse_ini_file(string` *filename*`[, bool` *process_sections*`[, int` *scanner_mode*`]])`

Loads *filename*—which must be a file in the standard *php.ini* format—and returns the values contained in it as an associative array, or `false` if the file could not be parsed. If *process_sections* is set and is `true`, a multidimensional array with values for the sections in the file is returned. The *scanner_mode* option is either `INI_SCANNER_NORMAL`, the default, or `INI_SCANNER_RAW`, indicating that the function should not parse option values.

**parse_ini_string.** `array parse_ini_string(string` *config*`[, bool` *process_sections*`[, int` *scanner_mode*`]])`

Parses a string in the *php.ini* format and returns the values contained in it in an associative array, or `false` if the string could not be parsed. If *process_sections* is set and is `true`, a multidimensional array with values for the sections in the file is returned. The *scanner_mode* option is either `INI_SCANNER_NORMAL`, the default, or `INI_SCANNER_RAW`, indicating that the function should not parse option values.

**parse_str.** `void parse_str(string` *string*`[, array` *variables*`])`

Parses *string* as if coming from an HTTP `POST` request, setting variables in the local scope to the values found in the string. If *variables* is given, the array is set with keys and values from the string.

**parse_url.** `mixed parse_url(string` *url*`)[, int` *component*`])`

Returns an associative array of the component parts of *url*. The array contains the following values:

| | |
|---|---|
| `fragment` | The named anchor in the URL |
| `host` | The host |
| `pass` | The user's password |
| `path` | The requested path (which may be a directory or a file) |
| `port` | The port to use for the protocol |

| | |
|---|---|
| query | The query information |
| scheme | The protocol in the URL, such as "http" |
| user | The user given in the URL |

The array will not contain values for components not specified in the URL. For example:

```
$url = "http://www.oreilly.net/search.php#place?name=php&type=book";
$array = parse_url($url);
print_r($array); // contains values for "scheme", "host", "path", "query",
 // and "fragment"
```

If the component option is provided, then just that particular component of the URL will be returned.

**passthru.** `void passthru(string` *command*`[, int` *return*`])`

Executes *command* via the shell and outputs the results of the command into the page. If *return* is specified, it is set to the return status of the command. If you want to capture the results of the command, use `exec()`.

**pathinfo.** `mixed pathinfo(string` *path*`[, int` *options*`])`

Returns an associative array containing information about *path*. If the *options* parameter is given, it specifies a particular element to be returned. `PATHINFO_DIRNAME`, `PATHINFO_BASENAME`, `PATHINFO_EXTENSION`, and `PATHINFO_FILENAME` are valid *options* values.

The following elements are in the returned array:

| | |
|---|---|
| dirname | The directory in which *path* is contained. |
| basename | The basename (see `basename`) of *path*, including the file's extension. |
| extension | The extension, if any, on the file's name. Does not include the period at the beginning of the extension. |

**pclose.** `int pclose(resource` *handle*`)`

Closes the pipe referenced by *handle*. Returns the termination code of the process that was run in the pipe.

**pfsockopen.** `resource pfsockopen(string` *host*`, int` *port*`[, int` *error*`[, string` *message* `[, float` *timeout*`]]])`

Opens a persistent TCP or UDP connection to a remote *host* on a specific *port*. By default, TCP is used; to connect via UDP, *host* must begin with `udp://`. If specified, *timeout* indicates the length of time in seconds to wait before timing out.

If the connection is successful, the function returns a virtual file pointer that can be used with functions such as `fgets()` and `fputs()`. If the connection fails, it returns `false`. If *error* and *message* are supplied, they are set to the error number and error string, respectively.

Unlike `fsockopen()`, the socket opened by this function does not close automatically after completing a read or write operation on it; you must close it explicitly with a call to `fsclose()`.

**php_ini_loaded_file.** `string php_ini_loaded_file()`

Returns the path of the current *php.ini* file if there is one, or `false` otherwise.

**php_ini_scanned_files.** `string php_ini_scanned_files()`

Returns a string containing the names of the configuration files parsed when PHP started up. The files are returned in a comma-separated list. If the compile-time configuration option `--with-config-file-scan-dir` was not set, `false` is returned instead.

**php_logo_guid.** `string php_logo_guid()`

Returns an ID that you can use to link to the PHP logo. For example:

```
<?php $current = basename($PHP_SELF); ?>
<img src="<?= "$current?=" . php_logo_guid(); ?>" border="0" />
```

**php_sapi_name.** `string php_sapi_name()`

Returns a string describing the server API under which PHP is running—for example, `"cgi"` or `"apache"`.

**php_strip_whitespace.** `string php_strip_whitespace(string` *path*`)`

Returns a string containing the source from the file *path* with whitespace and comment tokens stripped.

**php_uname.** `string php_uname(string` *mode*`)`

Returns a string describing the operating system under which PHP is running. The *mode* parameter is a single character used to control what is returned. The possible values are:

| a (default) | All modes included (s, n, r, v, m) |
|---|---|
| s | Name of the operating system |
| n | The hostname |
| r | Release name |

| | |
|---|---|
| v | Version information |
| m | Machine type |

**phpcredits.** `bool phpcredits([int` *what*`])`

Outputs information about PHP and its developers; the information that is displayed is based on the value of *what*. To use more than one option, OR the values together. The possible values of *what* are:

| | |
|---|---|
| CREDITS_ALL (default) | All credits except CREDITS_SAPI |
| CREDITS_GENERAL | General credits about PHP |
| CREDITS_GROUP | A list of the core PHP developers |
| CREDITS_DOCS | Information about the documentation team |
| CREDITS_MODULES | A list of the extension modules currently loaded and the authors for each |
| CREDITS_SAPI | A list of the server API modules and the authors for each |
| CREDITS_FULLPAGE | Indicates that the credits should be returned as a full HTML page, rather than just a fragment of HTML code. Must be used in conjunction with one or more other options—for example, phpcredits(CREDITS_MODULES \| CREDITS_FULLPAGE) |

**phpinfo.** `bool phpinfo([int` *what*`])`

Outputs a great deal of information about the state of the current PHP environment, including loaded extensions, compilation options, version, server information, and so on. If specified, *what* can limit the output to specific pieces of information; *what* may contain several options ORed together. The possible values of *what* are:

| | |
|---|---|
| INFO_ALL (default) | All information |
| INFO_GENERAL | General information about PHP |
| INFO_CREDITS | Credits for PHP, including the authors |
| INFO_CONFIGURATION | Configuration and compilation options |
| INFO_MODULES | Currently loaded extensions |
| INFO_ENVIRONMENT | Information about the PHP environment |
| INFO_VARIABLES | A list of the current variables and their values |
| INFO_LICENSE | The PHP license |

**phpversion.** `string phpversion(string` *extension*`)`

Returns the version of the currently running PHP parser. If the *extension* option is used, by naming a particular extension, the version information about that extension is all that is returned.

**pi.** `float pi()`

Returns an approximate value of pi (3.14159265359).

**popen.** `resource popen(string` *command*`, string` *mode*`)`

Opens a pipe to a process executed by running *command* on the shell.

The parameter *mode* specifies the permissions to open the file with, which can only be unidirectional (that is, for reading or writing only). *mode* must be one of the following:

| | |
|---|---|
| `r` | Open file for reading; file pointer will be at beginning of file |
| `w` | Open file for writing. If the file exists, it will be truncated to zero length; if the file doesn't already exist, it will be created |

If any error occurs while attempting to open the pipe, `false` is returned. If not, the resource handle for the pipe is returned.

**pow.** `number pow(number` *base*`, number` *exponent*`)`

Returns *base* raised to the *exponent* power. When possible, the return value is an integer; if not, it is a float.

**prev.** `mixed prev(array` *array*`)`

Moves the internal pointer to the element before its current location and returns the value of the element to which the internal pointer is now set. If the internal pointer is already set to the first element in the array, returns `false`. Be careful when iterating over an array using this function—if an array has an empty element or an element with a key value of `0`, a value equivalent to `false` is returned, causing the loop to end. If an array might contain empty elements or an element with a key of `0`, use the `each()` function instead of a loop with `prev()`.

**print_r.** `mixed print_r(mixed` *value*`[, bool` *return*`])`

Outputs *value* in a human-readable manner. If *value* is a string, integer, or float, the value itself is output; if it is an array, the keys and elements are shown; and if it is an object, the keys and values for the object are displayed. This function returns `true`. If *return* is set to `true`, then the output is returned rather than displayed.

**printf.** `int printf(string` *format*`[, mixed` *arg1* `...])`

Outputs a string created by using *format* and the given arguments. The arguments are placed into the string in various places denoted by special markers in the *format* string.

Each marker starts with a percent sign (`%`) and consists of the following elements, in order. Except for the type specifier, the specifiers are all optional. To include a percent sign in the string, use `%%`.

1. An optional sign specifier that forces a sign (– or +) to be used on a number. By default, only the – sign is used on a number if it's negative. Additionally, this specifier forces positive numbers to have the + sign attached.
2. A padding specifier denoting the character to use to pad the results to the appropriate string size (given below). Either `0`, a space, or any character prefixed with a single quote may be specified; padding with spaces is the default.
3. An alignment specifier. By default, the string is padded to make it right-justified. To make it left-justified, specify a dash (`-`) here.
4. The minimum number of characters this element should contain. If the result would be less than this number of characters, the preceding specifiers determine the behavior to pad to the appropriate width.
5. For floating-point numbers, a precision specifier consisting of a period and a number; this dictates how many decimal digits will be displayed. For types other than float, this specifier is ignored.
6. Finally, a type specifier. This specifier tells `printf()` what type of data is being handed to the function for this marker. There are eight possible types:

| | |
|---|---|
| `b` | The argument is an integer and is displayed as a binary number |
| `c` | The argument is an integer and is displayed as the character with that value |
| `d` | The argument is an integer and is displayed as a decimal number |
| `f` | The argument is a float and is displayed as a floating-point number |
| `o` | The argument is an integer and is displayed as an octal (base-8) number |
| `s` | The argument is treated and displayed as a string |
| `x` | The argument is an integer and is displayed as a hexadecimal (base-16) number; lowercase letters are used |
| `X` | Same as `x`, except uppercase letters are used |

**proc_close.** `int proc_close(resource` *handle*`)`

Closes the process referenced by *handle* and previously opened by `proc_open()`. Returns the termination code of the process.

**proc_get_status.** `array proc_get_status(resource` *handle*`)`

Returns an associative array containing information about the process *handle*, previously opened by `proc_open()`. The array contains the following values:

| | |
|---|---|
| `command` | The command string this process was opened with |
| `pid` | The process ID |
| `running` | `true` if the process is currently running, and `false` otherwise |
| `signaled` | `true` if the process has been terminated by an uncaught signal, and `false` otherwise |
| `stopped` | `true` if the process has been stopped by a signal, and `false` otherwise |
| `exitcode` | If the process has terminated, the exit code from the process, and `-1` otherwise |
| `termsig` | The signal that caused the process to be terminated if `signaled` is `true`, and `undefined` otherwise |
| `stopsig` | The signal that caused the process to be stopped if `stopped` is `true`, and `undefined` otherwise |

**proc_nice.** `bool proc_nice(int` *increment*`)`

Changes the priority of the process executing the current script by *increment*. A negative value raises the priority of the process, while a positive value lowers the priority of the process. Returns `true` if the operation was successful, and `false` otherwise.

**proc_open.** `resource proc_open(string` *command*`, array` *descriptors*`, array` *pipes*`[, string` *dir*`[, array` *env*`[, array` *options*`]]])`

Opens a pipe to a process executed by running *command* on the shell, with a variety of options. The descriptors parameter must be an array with three elements—in order, they describe the `stdin`, `stdout`, and `stderr` descriptors. For each, specify either an array containing two elements or a stream resource. In the first case, if the first element is `"pipe"`, the second element is either `"r"` to read from the pipe or `"w"` to write to the pipe. If the first is `"file"`, the second must be a filename. The *pipes* array is filled with an array of file pointers corresponding to the processes' descriptors. If *dir* is specified, the process has its current working directory set to that path. If *env* is specified, the process has its environment set up with the values from that array. Finally, *options* contains an associative array with additional options. The following options can be set in the array:

| | |
|---|---|
| `suppress_errors` | If set and `true`, suppresses errors generated by the process (Windows only) |
| `bypass_shell` | If set and `true`, bypasses *cmd.exe* when running the process |
| `context` | If set, specifies the stream context when opening files |

If any error occurs while attempting to open the process, `false` is returned. If not, the resource handle for the process is returned.

**proc_terminate.** `bool proc_terminate(resource` *handle*`[, int` *signal*`])`

Signals to the process referenced by *handle* and previously opened by `proc_open()` that it should terminate. If *signal* is supplied, the process is sent that signal. The call returns immediately, which may be prior to the process finishing termination. To poll

for a process's status, use `proc_get_status()`. Returns `true` if the operation was successful, and `false` otherwise.

**property_exists.** `bool property_exists(mixed` *class*`, string` *name*`)`

Returns `true` if the object or *class* has a data member named *name* defined on it, and `false` if it does not.

**putenv.** `bool putenv(string` *setting*`)`

Sets an environment variable using *setting*, which is typically in the form *name* = *value*. Returns `true` if successful and `false` if not.

**quoted_printable_decode.** `string quoted_printable_decode(string` *string*`)`

Decodes *string*, which is data encoded using the quoted printable encoding, and returns the resulting string.

**quoted_printable_encode.** `string quoted_printable_encode(string` *string*`)`

Returns *string* formatted in quoted printable encoding. See RFC 2045 for a description of the encoding format.

**quotemeta.** `string quotemeta(string` *string*`)`

Escapes instances of certain characters in *string* by appending a backslash (`\`) to them and returns the resulting string. The following characters are escaped: period (`.`), backslash (`\`), plus sign (`+`), asterisk (`*`), question mark (`?`), brackets (`[` and `]`), caret (`^`), parentheses (`(` and `)`), and dollar sign (`$`).

**rad2deg.** `float rad2deg(float` *number*`)`

Converts *number* from radians to degrees and returns the result.

**rand.** `int rand([int` *min*`, int` *max*`])`

Returns a random number from *min* to *max*, inclusive. If the *min* and *max* parameters are not provided, returns a random number from 0 to the value returned by the `getrandmax()` function.

**random_bytes.** `string random_bytes(int` *length*`)`

Generates an arbitrary *length* string of cryptographic random bytes that are suitable for cryptographic use, such as when generating salts, keys, or initialization vectors.

**random_int.** `int random_int(int` *min*, `int` *max*`)`

Generates cryptographic random integers that can be used where unbiased results are mandatory, such as when mixing "balls" for Bingo. *Min* sets the lowest value range to be returned (must be `PHP_INT_MIN` or greater), *max* sets the highest (must be `PHP_INT_MAX` or lower).

**range.** `array range(mixed` *first*`, mixed` *second*`[, number` *step*`])`

Creates and returns an array containing integers or characters from *first* to *second*, inclusive. If *second* is smaller than *first*, the sequence is returned in reverse order. If *step* is provided, then the created array will have the specified step gaps in it.

**rawurldecode.** `string rawurldecode(string` *url*`)`

Returns a string created from decoding the URI-encoded *url*. Sequences of characters beginning with a `%` followed by a hexadecimal number are replaced with the literal the sequence represents.

**rawurlencode.** `string rawurlencode(string` *url*`)`

Returns a string created by URI encoding *url*. Certain characters are replaced by sequences of characters beginning with a `%` followed by a hexadecimal number; for example, spaces are replaced with `%20`.

**readdir.** `string readdir([resource` *handle*`])`

Returns the name of the next file in the directory referenced by *handle*. If not specified, *handle* defaults to the last directory handle resource returned by `opendir()`. The order in which files in a directory are returned by calls to `readdir()` is undefined. If there are no more files in the directory to return, `readdir()` returns `false`.

**readfile.** `int readfile(string` *path*`[, bool` *include*`[, resource` *context*`]])`

Reads the file at *path*, in the streams context *context* if provided, and outputs the contents. If *include* is specified and is `true`, the include path is searched for the file. If *path* begins with `http://`, an HTTP connection is opened and the file is read from it. If *path* begins with `ftp://`, an FTP connection is opened and the file is read from it; the remote server must support passive FTP.

This function returns the number of bytes output.

**readlink.** `string readlink(string` *path*`)`

Returns the path contained in the symbolic link file *path*. If *path* does not exist or is not a symbolic link file, or if any other error occurs, the function returns `false`.

**realpath.** `string realpath(string` *path*`)`

Expands all symbolic links, resolves references to `/./` and `/../`, removes extra `/` characters in *path*, and returns the result.

**realpath_cache_get.** `array realpath_cache_get()`

Returns the contents of the `realpath` cache as an associative array. The key for each item is the path name, and the value for each item is an associative array containing values that have been cached for the path. The possible values include:

| | |
|---|---|
| `expires` | The time when this cache entry will expire |
| `is_dir` | Whether this path represents a directory or not |
| `key` | A unique ID for the cache entry |
| `realpath` | The resolved path for the path |

**realpath_cache_size.** `int realpath_cache_size()`

Returns the size in bytes the `realpath` cache currently occupies in memory.

**register_shutdown_function.** `void register_shutdown_function(callable` *function*`[, mixed` *arg1* `[, mixed` *arg2* `[, ... mixed` *argN*`]]])`

Registers a shutdown function. The function is called when the page completes processing with the given arguments. You can register multiple shutdown functions, and they will be called in the order in which they were registered. If a shutdown function contains an exit command, functions registered after that function will not be called.

Because the shutdown function is called after the page has completely processed, you cannot add data to the page with `print()`, `echo()`, or similar functions or commands.

**register_tick_function.** `bool register_tick_function(callable` *function*`[, mixed` *arg1* `[, mixed` *arg2* `[, ... mixed` *argN*`]]])`

Registers the function *name* to be called on each tick. The function is called with the given arguments. Obviously, registering a tick function can have a serious impact on the performance of your script. Returns `true` if the operation was successful, and `false` otherwise.

**rename.** `bool rename(string` *old*`, string` *new*`[, resource` *context*`]))`

Renames the file *old*, using the streams context *context* if provided, to *new*; returns `true` if the renaming was successful and `false` if not.

**reset.** `mixed reset(array` *array*`)`

Resets the *array*'s internal pointer to the first element and returns the value of that element.

**restore_error_handler.** `bool restore_error_handler()`

Reverts to the error handler in place prior to the most recent call to `set_error_handler()` and returns `true`.

**restore_exception_handler.** `bool restore_exception_handler()`

Reverts to the exception handler in place prior to the most recent call to `set_exception_handler()` and returns `true`.

**rewind.** `int rewind(resource` *handle*`)`

Sets the file pointer for *handle* to the beginning of the file. Returns `true` if the operation was successful and `false` if not.

**rewinddir.** `void rewinddir([resource` *handle*`])`

Sets the file pointer for *handle* to the beginning of the list of files in the directory. If not specified, *handle* defaults to the last directory handle resource returned by `opendir()`.

**rmdir.** `int rmdir(string` *path*`[, resource` *context*`])`

Removes the directory *path*, using the streams context *context* if provided. If the directory is not empty, or the PHP process does not have appropriate permissions, or if any other error occurs, `false` is returned. If the directory is successfully deleted, `true` is returned.

**round.** `float round(float` *number*`[, int` *precision*`[, int` *mode*`]])`

Returns the integer value nearest to *number* at the *precision* number of decimal places. The default for precision is `0` (integer rounding). The *mode* parameter dictates the method of rounding used:

| PHP_ROUND_HALF_UP (default) | Round up |
|---|---|
| PHP_ROUND_HALF_DOWN | Round down |
| PHP_ROUND_HALF_EVEN | Round up if the significant digits are even |
| PHP_ROUND_HALF_ODD | Round down if the significant digits are odd |

**rsort.** `void rsort(array array[, int flags])`

Sorts an array in reverse order by value. The optional second parameter contains additional sorting flags. See Chapter 5 and `unserialize()` for more information on using this function.

**rtrim.** `string rtrim(string string[, string characters])`

Returns *string* with all characters in *characters* stripped from the end. If *characters* is not specified, the characters stripped are `\n`, `\r`, `\t`, `\v`, `\0`, and spaces.

**scandir.** `array scandir(string path [, int sort_order [, resource context]])`

Returns an array of filenames existing at *path*, in the streams context *context* if provided, or `false` if an error occurred. The filenames are sorted according to the *sort_order* parameter, which is one of the following types:

| | |
|---|---|
| `SCANDIR_SORT_ASCENDING` (default) | Sort ascending |
| `SCANDIR_SORT_DESCENDING` | Sort descending |
| `SCANDIR_SORT_NONE` | Perform no sorting (the resulting order is undefined) |

**serialize.** `string serialize(mixed value)`

Returns a string containing a binary data representation of *value*. This string can be used to store the data in a database or file, for example, and later restored using `unserialize()`. Except for resources, any kind of value can be serialized.

**set_error_handler.** `string set_error_handler(string function)`

Sets the named function as the current error handler, or unsets the current error handler if *function* is `NULL`. The error-handler function is called whenever an error occurs; the function can do whatever it wants, but typically will print an error message and clean up after a critical error happens.

The user-defined function is called with two parameters, an error code and a string describing the error. Three additional parameters may also be supplied—the filename in which the error occurred, the line number at which the error occurred, and the context in which the error occurred (which is an array pointing to the active symbol table).

`set_error_handler()` returns the name of the previously installed error-handler function, or `false` if an error occurred while setting the error handler (e.g., when *function* doesn't exist).

**set_exception_handler.** `callable set_exception_handler(callable` *function*`)`

Sets the named function as the current exception handler. The exception handler is called whenever an exception is thrown in a `try...catch` block, but is not caught; the function can do whatever it wants, but typically will print an error message and clean up after a critical error happens.

The user-defined function is called with one parameter—the exception object that was thrown.

`set_exception_handler()` returns the previously installed exception-handler function, an empty string if no previous handler was set, or `false` if an error occurred while setting the error handler (e.g., when *function* doesn't exist).

**set_include_path.** `string set_include_path(string` *path*`)`

Sets the include path configuration option; it lasts until the end of the script's execution, or until a call to `restore_include_path` in the script. Returns the value of the previous include path.

**set_time_limit.** `void set_time_limit(int` *timeout*`)`

Sets the timeout for the current script to *timeout* seconds and restarts the timeout timer. By default, the timeout is set to 30 seconds or the value for `max_execution_time` set in the current configuration file. If a script does not finish executing within the time provided, a fatal error is generated and the script is killed. If *timeout* is `0`, the script will never time out.

**setcookie.** `void setcookie(string` *name*`[, string` *value*`[, int` *expiration*`[, string` *path* `[, string` *domain*`[, bool` *is_secure*`]]]]])`

Generates a cookie and passes it along with the rest of the header information. Because cookies are set in the HTTP header, `setcookie()` must be called before any output is generated.

If only *name* is specified, the cookie with that name is deleted from the client. The *value* argument specifies a value for the cookie to take, *expiration* is a Unix timestamp value defining a time the cookie should expire, and the *path* and *domain* parameters define a domain for the cookie to be associated with. If *is_secure* is `true`, the cookie will be transmitted only over a secure HTTP connection.

**setlocale.** `string setlocale(mixed` *category*`, string` *locale*`[, string` *locale*`, ...]) string setlocale(mixed` *category*`, array` *locale*`)`

Sets the locale for *category* functions to *locale*. Returns the current locale after being set, or `false` if the locale cannot be set. Any number of options for *category* can be added (or `OR`ed) together. The following options are available:

| | |
|---|---|
| `LC_ALL` (default) | All of the following categories |
| `LC_COLLATE` | String comparisons |
| `LC_CTYPE` | Character classification and conversion |
| `LC_MONETARY` | Monetary functions |
| `LC_NUMERIC` | Numeric functions |
| `LC_TIME` | Time and date formatting |

If *locale* is `0` or the empty string, the current locale is unaffected.

**setrawcookie.** `void setrawcookie(string` *name*`[, string` *value*`[, int` *expiration*`[, string` *path* `[, string` *domain*`[, bool` *is_secure*`]]]]])`

Generates a cookie and passes it along with the rest of the header information. Because cookies are set in the HTTP header, `setcookie()` must be called before any output is generated.

If only *name* is specified, the cookie with that name is deleted from the client. The *value* argument specifies a value for the cookie to take—unlike `setcookie()`, the value specified here is not URL-encoded before being sent, *expiration* is a Unix timestamp value defining a time the cookie should expire, and the *path* and *domain* parameters define a domain for the cookie to be associated with. If *is_secure* is `true`, the cookie will be transmitted only over a secure HTTP connection.

**settype.** `bool settype(mixed` *value*`, string` *type*`)`

Converts *value* to the given *type*. Possible types are `"boolean"`, `"integer"`, `"float"`, `"string"`, `"array"`, and `"object"`. Returns `true` if the operation was successful and `false` if not. Using this function is the same as typecasting *value* to the appropriate type.

**sha1.** `string sha1(string` *string*`[, bool` *binary*`])`

Calculates the `sha1` encryption hash of *string* and returns it. If *binary* is set and is `true`, the raw binary is returned instead of a hex string.

**sha1_file.** `string sha1_file(string` *path*`[, bool` *binary*`])`

Calculates and returns the `sha1` encryption hash for the file at *path*. A `sha1` hash is a 40-character hexadecimal value that can be used to checksum a file's data. If *binary* is supplied and is `true`, the result is sent as a 20-bit binary value instead.

**shell_exec.** `string shell_exec(string` *command*`)`

Executes *command* via the shell and returns the output from the command's result. This function is called when you use the backtick operator (`` ` ``).

**shuffle.** `void shuffle(array` *array*`)`

Rearranges the values in *array* into a random order. Keys for the values are lost.

**similar_text.** `int similar_text(string` *one*`, string` *two*`[, float` *percent*`])`

Calculates the similarity between the strings *one* and *two*. If passed by reference, *percent* gets the percent by which the two strings differ.

**sin.** `float sin(float` *value*`)`

Returns the sine of *value* in radians.

**sinh.** `float sinh(float` *value*`)`

Returns the hyperbolic sine of *value* in radians.

**sleep.** `int sleep(int` *time*`)`

Pauses execution of the current script for *time* seconds. Returns `0` if the operation was successful, or `false` otherwise.

**sort.** `bool sort(array` *array*`[, int` *flags*`])`

Sorts the values in the given *array* in ascending order. For more control over the behavior of the sort, provide the second parameter, which is one of the following values:

| | |
|---|---|
| `SORT_REGULAR` (default) | Compare the items normally |
| `SORT_NUMERIC` | Compare the items numerically |
| `SORT_STRING` | Compare the items as strings |
| `SORT_LOCALE_STRING` | Compare the items as strings using the current locale sorting rules |
| `SORT_NATURAL` | Compare the items as strings using "natural ordering" |
| `SORT_FLAG_CASE` | Combine with `SORT_STRING` or `SORT_NATURAL` using a bitwise OR operation to sort using case-insensitive comparison |

Returns `true` if the operation was successful, and `false` otherwise. See Chapter 5 for more information on using this function.

**soundex.** `string soundex(string` *string*`)`

Calculates and returns the soundex key of *string*. Words that are pronounced similarly (and begin with the same letter) have the same soundex key.

**sprintf.** `string sprintf(string` *format*`[, mixed` *value1*`[, ... mixed` *valueN*`]])`

Returns a string created by filling *format* with the given arguments. See `printf()` for more information on using this function.

**sqrt.** `float sqrt(float` *number*`)`

Returns the square root of *number*.

**srand.** `void srand([int` *seed*`])`

Seeds the standard pseudorandom number generator with *seed*, or with a random seed if none is provided.

**sscanf.** `mixed sscanf(string` *string*`, string` *format*`[, mixed` *variableN* `...])`

Parses *string* for values of types given in *format*; the values found are either returned in an array or, if *variable1* through *variableN* (which must be variables passed by reference) are given, in those variables.

The *format* string is the same as that used in `sprintf()`. For example:

```
$name = sscanf("Name: k.tatroe", "Name: %s"); // $name has "k.tatroe"
list($month, $day, $year) = sscanf("June 30, 2001", "%s %d, %d");
$count = sscanf("June 30, 2001", "%s %d, %d", &$month, &$day, &$year);
```

**stat.** `array stat(string` *path*`)`

Returns an associative array of information about the file *path*. If *path* is a symbolic link, information about the file *path* references is returned. See `fstat` for a list of the values returned and their meanings.

**str_getcsv.** `array str_getcsv(string` *input*`[, string` *delimiter*`[, string` *enclosure* `[, string` *escape*`]]]])`

Parses a string as a comma-separated values (CSV) list and returns it as an array of values. If supplied, *delimiter* is used to delimit the values for the line instead of commas. If supplied, *enclosure* is a single character that is used to enclose values (by default, the double-quote character, `"`). *escape* sets the escape character to use; the default is a backslash, `\`.

**str_ireplace.** `mixed str_ireplace(mixed` *search*`, mixed` *replace*`, mixed` *string*`[, int &`*count*`])`

Performs a case-insensitive search for all occurrences of *search* in *string* and replaces them with *replace*. If all three parameters are strings, a string is returned. If *string* is an array, the replacement is performed for every element in the array and an array of results is returned. If *search* and *replace* are both arrays, elements in *search* are replaced with the elements in *replace* with the same numeric indices. Finally, if *search* is an array and *replace* is a string, any occurrence of any element in *search* is changed to *replace*. If supplied, *count* is filled with the number of instances replaced.

**str_pad.** `string str_pad(string` *string*`, string` *length*`[, string` *pad*`[, int` *type*`]])`

Pads *string* using *pad* until it is at least *length* characters and returns the resulting string. By specifying *type*, you can control where the padding occurs. The following values for *type* are accepted:

| | |
|---|---|
| `STR_PAD_RIGHT` (default) | Pad to the right of *string* |
| `STR_PAD_LEFT` | Pad to the left of *string* |
| `STR_PAD_BOTH` | Pad on either side of *string* |

**str_repeat.** `string str_repeat(string` *string*`, int` *count*`)`

Returns a string consisting of *count* copies of *string* appended to each other. If *count* is not greater than zero, an empty string is returned.

**str_replace.** `mixed str_replace(mixed` *search*`, mixed` *replace*`, mixed` *string*`[, int &`*count*`])`

Searches for all occurrences of *search* in *string* and replaces them with *replace*. If all three parameters are strings, a string is returned. If *string* is an array, the replacement is performed for every element in the array and an array of results is returned. If *search* and *replace* are both arrays, elements in *search* are replaced with the elements in *replace* with the same numeric indices. Finally, if *search* is an array and *replace* is a string, any occurrence of any element in *search* is changed to *replace*. If supplied, *count* is filled with the number of instances replaced.

**str_rot13.** `string str_rot13(string` *string*`)`

Converts *string* to its `rot13` version and returns the resulting string.

**str_shuffle.** `string str_shuffle(string` *string*`)`

Rearranges the characters in *string* into a random order and returns the resulting string.

**str_split.** `array str_split(string` *string*`[, int` *length*`])`

Splits *string* into an array of characters, each containing *length* characters; if *length* is not specified, it defaults to `1`.

**str_word_count.** `mixed str_word_count(string` *string*`[, int` *format*`[, string` *characters*`]])`

Counts the number of words in *string* using locale-specific rules. The value of *format* dictates the returned value:

| | |
|---|---|
| `0` (default) | The number of words found in *string* |
| `1` | An array of all words found in *string* |
| `2` | An associative array, with keys being the positions and values being the words found at those positions in *string* |

If *characters* is specified, it provides additional characters that are considered to be inside words (that is, are not word boundaries).

**strcasecmp.** `int strcasecmp(string` *one*`, string` *two*`)`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-insensitive—that is, "Alphabet" and "alphabet" are considered equal.

**strcmp.** `int strcmp(string` *one*`, string` *two*`)`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-sensitive—that is, "Alphabet" and "alphabet" are not considered equal.

**strcoll.** `int strcoll(string` *one*`, string` *two*`)`

Compares two strings using the rules of the current locale; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-sensitive—that is, "Alphabet" and "alphabet" are not considered equal.

**strcspn.** `int strcspn(string` *string*`, string` *characters*`[, int` *offset*`[, int` *length*`]])`

Returns the length of the subset of *string* starting at *offset*, examining a maximum of *length* characters, to the first instance of a character from *characters*.

**strftime.** `string strftime(string` *format*`[, int` *timestamp*`])`

Formats a time and date according to the *format* string provided in the first parameter and the current locale. If the second parameter is not specified, the current time and date is used. The following characters are recognized in the *format* string:

| | |
|---|---|
| `%a` | Name of the day of the week as a three-letter abbreviation (e.g., Mon) |
| `%A` | Name of the day of the week (e.g., Monday) |
| `%b` | Name of the month as a three-letter abbreviation (e.g., Aug) |
| `%B` | Name of the month (e.g., August) |
| `%c` | Date and time in the preferred format for the current locale |
| `%C` | The last two digits of the century |
| `%d` | Day of the month as two digits, including a leading zero if necessary (e.g., 01 through 31) |
| `%D` | Same as `%m/%d/%y` |
| `%e` | Day of the month as two digits, including a leading space if necessary (e.g., 1 through 31) |
| `%h` | Same as `%b` |
| `%H` | Hour in 24-hour format, including a leading zero if necessary (e.g., 00 through 23) |
| `%I` | Hour in 12-hour format (e.g., 1 through 12) |
| `%j` | Day of the year, including leading zeros as necessary (e.g., 001 through 366) |
| `%m` | Month, including a leading zero if necessary (e.g., 01 through 12) |
| `%M` | Minutes |
| `%n` | The newline character (`\n`) |
| `%p` | am or pm |
| `%r` | Same as `%I:%M:%S %p` |
| `%R` | Same as `%H:%M:%S` |
| `%S` | Seconds |
| `%t` | The tab character (`\t`) |
| `%T` | Same as `%H:%M:%S` |
| `%u` | Numeric day of the week, starting with 1 for Monday |
| `%U` | Numeric week of the year, starting with the first Sunday |
| `%V` | ISO 8601:1998 numeric week of the year—Week 1 starts on the Monday of the first week that has at least four days |
| `%W` | Numeric week of the year, starting with the first Monday |
| `%w` | Numeric day of the week, starting with 0 for Sunday |
| `%x` | The preferred date format for the current locale |
| `%X` | The preferred time format for the current locale |

| | |
|---|---|
| `%y` | Year with two digits (e.g., 98) |
| `%Y` | Year with four digits (e.g., 1998) |
| `%Z` | Time zone or name or abbreviation |
| `%%` | The percent sign (%) |

**stripcslashes.** `string stripcslashes(string` *string*`, string` *characters*`)`

Converts instances of *characters* after a backslash in *string* by removing the backslash before them. You can specify ranges of characters by separating them by two periods; for example, to unescape characters between `a` and `q`, use `"a..q"`. Multiple characters and ranges can be specified in *characters*. The `stripcslashes()` function is the inverse of `addcslashes()`.

**stripslashes.** `string stripslashes(string` *string*`)`

Converts instances of escape sequences that have special meaning in SQL queries in *string* by removing the backslash before them. Single quotes (`'`), double quotes (`"`), backslashes (`\`), and the NUL-byte (`"\0"`) are escaped. This function is the inverse of `addslashes()`.

**strip_tags.** `string strip_tags(string` *string*`[, string` *allowed*`])`

Removes PHP and HTML tags from *string* and returns the result. The *allowed* parameter can be specified to not remove certain tags. The string should be a comma-separated list of the tags to ignore; for example, `"<b>,<i>"` will leave bold and italic tags.

**stripos.** `int stripos(string` *string*`, string` *value*`[, int` *offset*`])`

Returns the position of the first occurrence of *value* in *string* using case-insensitive comparison. If specified, the function begins its search at position *offset*. Returns `false` if *value* is not found.

**stristr.** `string stristr(string` *string*`, string` *search*`[, int` *before*`])`

Returns the portion of *string* from the first occurrence of *search* using case-insensitive comparison until the end of *string*, or from the first occurrence of *search* until the beginning of *string* if *before* is specified and `true`. If *search* is not found, the function returns `false`. If *search* contains more than one character, only the first is used.

**strlen.** `int strlen(string` *string*`)`

Returns the number of characters in *string*.

**strnatcasecmp.** `int strnatcasecmp(string` *one*`, string` *two*`)`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-insensitive—that is, "Alphabet" and "alphabet" are considered equal. The function uses a "natural order" algorithm—numbers in the strings are compared more naturally than computers normally do. For example, the values `"1"`, `"10"`, and `"2"` are sorted in that order by `strcmp()`, but `strnatcasecmp()` orders them `"1"`, `"2"`, and `"10"`. This function is a case-insensitive version of `strnatcmp()`.

**strnatcmp.** `int strnatcmp(string` *one*`, string` *two*`)`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-sensitive—that is, "Alphabet" and "alphabet" are not considered equal. The `strnatcmp()` function uses a "natural order" algorithm—numbers in the strings are compared more naturally than computers normally do. For example, the values `"1"`, `"10"`, and `"2"` are sorted in that order by `strcmp()`, but `strnatcmp()` orders them `"1"`, `"2"`, and `"10"`.

**strncasecmp.** `int strncasecmp(string` *one*`, string` *two*`, int` *length*`)`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-insensitive—that is, "Alphabet" and "alphabet" are considered equal. This function is a case-insensitive version of `strcmp()`. If either string is shorter than *length* characters, the length of that string determines how many characters are compared.

**strncmp.** `int strncmp(string` *one*`, string` *two*`[, int` *length*`])`

Compares two strings; returns a number less than zero if *one* is less than *two*, `0` if the two strings are equal, and a number greater than zero if *one* is greater than *two*. The comparison is case-sensitive—that is, "Alphabet" and "alphabet" are not considered equal. If specified, no more than *length* characters are compared. If either string is shorter than *length* characters, the length of that string determines how many characters are compared.

**strpbrk.** `string strpbrk(string` *string*`, string` *characters*`)`

Returns a string consisting of the substring of *string*, starting from the position of the first instance of a character from *characters* in *string* to the end of the string, or `false` if none of the characters in *characters* is found in *string*.

**strpos.** `int strpos(string` *string*`, string` *value*`[, int` *offset*`])`

Returns the position of the first occurrence of *value* in *string*. If specified, the function begins its search at position *offset*. Returns `false` if *value* is not found.

**strptime.** `array strptime(string` *date*`, string` *format*`)`

Parses a time and date according to the *format* string and the current locale. The format uses the same format as `strftime()`. Returns an associative array with information about the parsed time containing the following elements:

| | |
|---|---|
| `tm_sec` | Seconds |
| `tm_min` | Minutes |
| `tm_hour` | Hours |
| `tm_mday` | Day of the month |
| `tm_wday` | Numeric day of the week (Sunday is 0) |
| `tm_mon` | Month |
| `tm_year` | Year |
| `tm_yday` | Day of the year |
| `unparsed` | The portion of *date* that was not parsed according to the given format |

**strrchr.** `string strrchr(string` *string*`, string` *character*`)`

Returns the portion of *string* from the last occurrence of *character* until the end of *string*. If *character* is not found, the function returns `false`. If *character* contains more than one character, only the first is used.

**strrev.** `string strrev(string` *string*`)`

Returns a string containing the characters of *string* in reverse order.

**strripos.** `int strripos(string` *string*`, string` *search*`[, int` *offset*`])`

Returns the position of the last occurrence of *search* in *string* using a case-insensitive search, or `false` if *search* is not found. If specified and positive, the search begins *offset* characters from the start of *string*. If specified and negative, the search begins *offset* characters from the end of *string*. This function is a case-insensitive version of `strrpos()`.

**strrpos.** `int strrpos(string` *string*`, string` *search*`[, int` *offset*`])`

Returns the position of the last occurrence of *search* in *string*, or `false` if *search* is not found. If specified and positive, the search begins *offset* characters from the start of

*string*. If specified and negative, the search begins *offset* characters from the end of *string*.

**strspn.** `int strspn(string` *string*`, string` *characters*`[, int` *offset*`[, int` *length*`]])`

Returns the length of the substring in *string* that consists solely of characters in *characters*. If *offset* is positive, the search starts at that character; if it is negative, the substring starts at the character *offset* characters from the string's end. If *length* is given and is positive, that many characters from the start of the substring are checked. If *length* is given and is negative, the check ends *length* characters from the end of *string*.

**strstr.** `string strstr(string` *string*`, string` *character*`[, bool` *before*`])`

Returns the portion of *string* from the first occurrence of *character* until the end of *string*, or from the first occurrence of *character* until the beginning of *string* if *before* is specified and `true`. If *character* is not found, the function returns `false`. If *character* contains more than one character, only the first is used.

**strtok.** `string strtok(string` *string*`, string` *token*`)` `string strtok(string` *token*`)`

Breaks *string* into tokens separated by any of the characters in *token* and returns the next token found. The first time you call `strtok()` on a string, use the first function prototype; afterward, use the second, providing only the tokens. The function contains an internal pointer for each string it is called with. For example:

```
$string = "This is the time for all good men to come to the aid of their
country."
$current = strtok($string, " .;,\"'");
while(!($current === false)) {
 print($current . "<br />";
}
```

**strtolower.** `string strtolower(string` *string*`)`

Returns *string* with all alphabetic characters converted to lowercase. The table used for converting characters is locale-specific.

**strtotime.** `int strtotime(string` *time*`[, int` *timestamp*`])`

Converts an English description of a time and date into a Unix timestamp value. Optionally, a *timestamp* can be given that the function uses as the "now" value; if this value is omitted, the current date and time is used. Returns `false` if the value could not be converted into a valid timestamp.

The descriptive string can be in a number of formats. For example, all of the following will work:

```
echo strtotime("now");
echo strtotime("+1 week");
echo strtotime("-1 week 2 days 4 seconds");
echo strtotime("2 January 1972");
```

**strtoupper.** `string strtoupper(string` *string*`)`

Returns *string* with all alphabetic characters converted to uppercase. The table used for converting characters is locale-specific.

**strtr.** `string strtr(string` *string*`, string` *from*`, string` *to*`) string strtr(string` *string*`, array` *replacements*`)`

When given three arguments, returns a string created by translating in *string* every occurrence of a character in *from* to the character in *to* with the same position. When given two arguments, returns a string created by translating occurrences of the keys in *replacements* in *string* with the corresponding values in *replacements*.

**strval.** `string strval(mixed` *value*`)`

Returns the string equivalent for *value*. If value is an object and that object implements the `__toString()` method, it returns the value of that method. Otherwise, if *value* is an object that doesn't implement `__toString()` or is an array, the function returns an empty string.

**substr.** `string substr(string` *string*`, int` *offset*`[, int` *length*`])`

Returns the substring of *string*. If *offset* is positive, the substring starts at that character; if it is negative, the substring starts at the character *offset* characters from the string's end. If *length* is given and is positive, that many characters from the start of the substring are returned. If *length* is given and is negative, the substring ends *length* characters from the end of *string*. If *length* is not given, the substring contains all characters to the end of *string*.

**substr_compare.** `int substr_compare(string` *first*`, string` *second*`, string` *offset*`[, int` *length*`[, bool` *case_insensitivity*`]])`

Compares *first*, starting at the position *offset*, to *second*. If *length* is specified, a maximum of that many characters are compared. Finally, if *case_insensitivity* is specified and `true`, the comparison is case-insensitive. Returns a number less than zero if the substring of *first* is less than *second*, `0` if they are equal, and a number greater than zero if the substring of *first* is greater than *second*.

**substr_count.** `int substr_count(string` *string*`, string` *search*`[, int` *offset*`[, int` *length*`]])`

Returns the number of times *search* appears in *string*. If *offset* is provided, the search begins at that character offset for at most *length* characters, or until the end of the string if *length* is not provided.

**substr_replace.** `string substr_replace(mixed` *string*`, mixed` *replace*`, mixed` *offset*`[, mixed` *length*`])`

Replaces a substring in *string* with *replace*. The substring replaced is selected using the same rules as those of `substr()`. If string is an array, replacements take place on each string within the array. In this case, *replace*, *offset*, and *length* can either be scalar values, which are used for all strings in *string*, or arrays of values to be used for each corresponding value in *string*.

**symlink.** `bool symlink(string` *path*`, string` *new*`)`

Creates a symbolic link to *path* at the path *new*. Returns `true` if the link was successfully created and `false` if not.

**syslog.** `bool syslog(int` *priority*`, string` *message*`)`

Sends an error message to the system logging facility. On Unix systems, this is `syslog(3)`; on Windows NT, the messages are logged in the NT Event Log. The message is logged with the given *priority*, which is one of the following (listed in decreasing order of priority):

| | |
|---|---|
| `LOG_EMERG` | Error has caused the system to be unstable |
| `LOG_ALERT` | Error notes a situation that requires immediate action |
| `LOG_CRIT` | Error is a critical condition |
| `LOG_ERR` | Error is a general error condition |
| `LOG_WARNING` | Error message is a warning |
| `LOG_NOTICE` | Error message is a normal, but significant, condition |
| `LOG_INFO` | Error is an informational message that requires no action |
| `LOG_DEBUG` | Error is for debugging only |

If *message* contains the characters `%m`, they are replaced with the current error message, if any is set. Returns `true` if the logging succeeded and `false` if a failure occurred.

**system.** `string system(string` *command*`[, int &`*return*`])`

Executes *command* via the shell and returns the last line of output from the command's result. If *return* is specified, it is set to the return status of the command.

**sys_getloadavg.** `array sys_getloadavg()`

Returns an array containing the load average of the machine running the current script, sampled over the last 1, 5, and 15 minutes.

**sys_get_temp_dir.** `string sys_get_temp_dir()`

Returns the path of the directory where temporary files, such as those created by `tmpfile()` and `tempname()`, are created.

**tan.** `float tan(float` *value*`)`

Returns the tangent of *value* in radians.

**tanh.** `float tanh(float` *value*`)`

Returns the hyperbolic tangent of *value* in radians.

**tempnam.** `string tempnam(string` *path*`, string` *prefix*`)`

Generates and returns a unique filename in the directory *path*. If *path* does not exist, the resulting temporary file may be located in the system's temporary directory. The filename is prefixed with *prefix*. Returns `false` if the operation could not be performed.

**time.** `int time()`

Returns the number of seconds since the Unix epoch (January 1, 1970, 00:00:00 GMT).

**time_nanosleep.** `bool time_nanosleep(int` *seconds*`, int` *nanoseconds*`)`

Pauses execution of the current script for *seconds* seconds and *nanoseconds* nanoseconds. Returns `true` on success and `false` on a failure; if the delay was interrupted by a signal, an associative array containing the following values is returned instead:

| | |
|---|---|
| `seconds` | Number of seconds remaining |
| `nanoseconds` | Number of nanoseconds remaining |

**time_sleep_until.** `bool time_sleep_until(float` *timestamp*`)`

Pauses execution of the current script until the time *timestamp* passes. Returns `true` on success and `false` on a failure.

**timezone_name_from_abbr.** `string timezone_name_from_abbr(string` *name*`[, int` *gmtOffset*`[, int` *dst*`]])`

Returns the name of a time zone given in *name*, or `false` if no appropriate time zone could be found. If given, *gmtOffset* is an integer offset from GMT used as a hint to find the appropriate time zone. If given, *dst* indicates whether the time zone has Daylight Saving Time or not as a hint to find the appropriate time zone.

**timezone_version_get.** `string timezone_version_get()`

Returns the version of the current time zone database.

**tmpfile.** `int tmpfile()`

Creates a temporary file with a unique name, opens it with read-write privileges, and returns a resource to the file, or `false` if an error occurred. The file is automatically deleted when closed with `fclose()` or at the end of the current script.

**token_get_all.** `array token_get_all(string` *source*`)`

Parses the PHP code *source* into PHP language tokens and returns them as an array. Each element in the array contains a single character token or a three-element array containing, in order, the token index, the source string representing the token, and the line number the *source* appeared in source.

**token_name.** `string token_name(int` *token*`)`

Returns the symbolic name of the PHP language token identified by *token*.

**touch.** `bool touch(string` *path*`[, int` *touch_time*`[, int` *access_time*`]])`

Sets the modification date of *path* to *touch_time* (a Unix timestamp value) and the access time of *path* to *access_time*. If not specified, *touch_time* defaults to the current time, while *access_time* defaults to *touch_time* (or the current time if that value is also not supplied). If the file does not exist, it is created. Returns `true` if the function completed without error and `false` if an error occurred.

**trait_exists.** `bool trait_exists(string name[, bool autoload])`

Returns `true` if a trait with the same name as the string has been defined; if not, it returns `false`. The comparison for trait names is case-insensitive. If `autoload` is set and is `true`, the autoloader attempts to load the trait before checking its existence.

**trigger_error.** `void trigger_error(string error[, int type])`

Triggers an error condition; if the type is not given, it defaults to `E_USER_NOTICE`. The following types are valid:

| | |
|---|---|
| `E_USER_ERROR` | User-generated error |
| `E_USER_WARNING` | User-generated warning |
| `E_USER_NOTICE` (default) | User-generated notice |
| `E_USER_DEPRECATED` | User-generated deprecated call warning |

If longer than 1,024 characters, *error* is truncated to 1,024 characters.

**trim.** `string trim(string string[, string characters])`

Returns *string* with every whitespace character in *characters* stripped from the beginning and end of the string. You can specify a range of characters to strip using `..` within the string. For example, `"a..z"` would strip each lowercase alphabetical character. If *characters* is not supplied, `\n`, `\r`, `\t`, `\x0B`, `\0`, and spaces are stripped.

**uasort.** `bool uasort(array array, callable function)`

Sorts an array using a user-defined function, maintaining the keys for the values. See Chapter 5 and `usort()` for more information on using this function. Returns `true` if the array was successfully sorted, and `false` otherwise.

**ucfirst.** `string ucfirst(string string)`

Returns *string* with the first character, if alphabetic, converted to uppercase. The table used for converting characters is locale-specific.

**ucwords.** `string ucwords(string string)`

Returns *string* with the first character of each word, if alphabetic, converted to uppercase. The table used for converting characters is locale-specific.

**uksort.** `bool uksort(array` *array*`, callable` *function*`)`

Sorts an array by keys using a user-defined function, maintaining the keys for the values. See Chapter 5 and `usort()` for more information on using this function. Returns `true` if the array was successfully sorted, and `false` otherwise.

**umask.** `int umask([int` *mask*`])`

Sets PHP's default permissions to the value *mask* `& 0777` and returns the previous mask if successful, or `false` if an error occurred. The previous default permissions are restored at the end of the current script. If *mask* is not supplied, the current permissions are returned.

When running on a multithreaded web server (e.g., Apache), use `chmod()` after creating a file to change its permissions, rather than this function.

**uniqid.** `string uniqid([string` *prefix*`[, bool` *more_entropy*`]])`

Returns a unique identifier, prefixed with *prefix*, based on the current time in microseconds. If *more_entropy* is specified and is `true`, additional random characters are added to the end of the string. The resulting string is either 13 characters (if *more_entropy* is unspecified or `false`) or 23 characters (if *more_entropy* is `true`) long.

**unlink.** `int unlink(string` *path*`[, resource` *context*`])`

Deletes the file *path*, using the streams context *context* if provided. Returns `true` if the operation was successful and `false` if not.

**unpack.** `array unpack(string` *format*`, string` *data*`)`

Returns an array of values retrieved from the binary string *data*, which was previously packed using the `pack()` function and the format *format*. See `pack()` for a listing of the format codes to use within *format*.

**unregister_tick_function.** `void unregister_tick_function(string` *name*`)`

Removes the function *name*, previously set using `register_tick_function()`, as a tick function. It will no longer be called during each tick.

**unserialize.** `mixed unserialize(string` *data*`)`

Returns the value stored in *data*, which must be a value previously serialized using `serialize()`. If the value is an object and that object has a `__wakeup()` method, that method is called on the object immediately after reconstructing the object.

**unset.** `void unset(mixed` *var*`[, mixed` *var2*`[, ... mixed` *varN*`]])`

Destroys the given variables. A global variable called within function scope only `unset`s the local copy of that variable; to destroy a global variable, you must call `unset` on the value in the `$GLOBALS` array instead. A variable in function scope passed by reference destroys only the local copy of that variable.

**urldecode.** `string urldecode(string` *url*`)`

Returns a string created from decoding the URI-encoded *url*. Sequences of characters beginning with a `%` followed by a hexadecimal number are replaced with the literal the sequence represents. In addition, plus signs (+) are replaced with spaces. See also `raw urlencode()`, which is identical except for its handling of spaces.

**urlencode.** `string urlencode(string` *url*`)`

Returns a string created by URI encoding *url*. All nonalphanumeric characters except dash (`-`), underscore (`_`), and period (`.`) characters in *url* are replaced by a sequence of characters beginning with a `%` followed by a hexadecimal number; for example, slashes (`/`) are replaced with `%2F`. In addition, any spaces in *url* are replaced by plus signs (`+`). See also `rawurlencode()`, which is identical except for its handling of spaces.

**usleep.** `void usleep(int` *time*`)`

Pauses execution of the current script for *time* microseconds.

**usort.** `bool usort(array` *array*`, callable` *function*`)`

Sorts an array using a user-defined function. The supplied function is called with two parameters. It should return an integer less than zero if the first argument is less than the second, `0` if the first and second arguments are equal, and an integer greater than zero if the first argument is greater than the second. The sort order of two elements that compare equal is undefined. See Chapter 5 for more information on using this function.

Returns `true` if the function was successful in sorting the array, and `false` otherwise.

**var_dump.** `void var_dump(mixed` *name*`[, mixed` *name2*`[, ... mixed` *nameN*`]])`

Outputs information about *name*, *name2*, and so on. Information output includes the variable's type, value, and, if an object, all public, private, and protected properties of the object. Arrays' and objects' contents are output in a recursive fashion.

**var_export.** `mixed var_export(mixed` *expression*`[, bool` *variable_representation*`])`

Returns the PHP code representation of *expression*. If *variable_representation* is set and is `true`, *expression*'s actual value is returned.

**version_compare.** `mixed version_compare(string` *one*`, string` *two*`[, string` *operator*`])`

Compares two version strings and returns `-1` if *one* is less than *two*, `0` if they are equal, and `1` if *one* is greater than *two*. The version strings are split into each numeric or string part, then compared as *string_value* `< "dev" < "alpha"` or `"a" < "beta"` or `"b" < "rc" <` *numeric_value* `< "pl"` or `"p"`.

If *operator* is specified, the operator is used to make a comparison between the version strings, and the value of the comparison using that operator is returned. The possible operators are `<` or `lt`; `<=` or `le`; `>` or `gt`; `>=` or `ge`; `==`, `=`, or `eq`; and `!=`, `<>`, and `ne`.

**vfprintf.** `int vfprintf(resource` *stream*`, string` *format*`, array` *values*`)`

Writes a string created by filling *format* with the arguments given in the array *values* to the stream *stream* and returns the length of the string sent. See `printf()` for more information on using this function.

**vprintf.** `void vprintf(string` *format*`, array` *values*`)`

Prints a string created by filling *format* with the arguments given in the array *values*. See `printf()` for more information on using this function.

**vsprintf.** `string vsprintf(string` *format*`, array` *values*`)`

Creates and returns a string created by filling *format* with the arguments given in the array *values*. See `printf()` for more information on using this function.

**wordwrap.** `string wordwrap(string` *string*`[, int` *length*`[, string` *postfix*`[, bool` *force*`]]])`

Inserts *postfix* into *string* every *length* characters and at the end of the string and returns the resulting string. While inserting breaks, the function attempts to not break in the middle of a word. If not specified, *postfix* defaults to `\n` and *size* defaults to `75`. If *force* is given and is `true`, the string is always wrapped to the given length (this makes the function behave the same as `chunk_split()`).

**zend_thread_id.** `int zend_thread_id()`

Returns a unique identifier for the thread of the currently running PHP process.

**zend_version.** `string zend_version()`

Returns the version of the Zend engine in the currently running PHP process.

# Index

## Symbols


! (exclamation point), logical negation operator, 45
!= (exclamation point, equals sign), inequality operator, 41
!== (exclamation point, double equals signs), not identical operator, 42
# (hash mark), preceding comments, 17
$ (dollar sign)
  preceding variable names, 20, 30
  in regular expressions, 107
$$ (dollar signs, double), preceding variables containing variable names, 30
$GLOBALS array, 32
$this variable, 160
$_COOKIE array, 189, 209, 211
$_FILES array, 189
$_GET array, 11, 189, 191
$_POST array, 11, 189, 191
$_REQUEST array, 189
$_SERVER array, 189-191, 216, 378
$_SESSION array, 212
% (percent sign)
  in format string, 86
  modulus operator, 39
%= (percent sign, equals sign), modulus-equals operator, 48
& (ampersand)
  bitwise AND operator, 43
  indicating value returned by reference, 71, 76
&& (ampersands, double), logical AND operator, 44
&= (ampersand, equals sign), bitwise-AND-equals operator, 48
(…) (parentheses)
  enclosing subpatterns, 110
  forcing operator precedence, 37
* (asterisk), multiplication operator, 39
** (exponentiation), 39
*= (asterisk, equals sign), multiply-equals operator, 47
+ (plus sign)
  addition operator, 38
  assertion operator, 39
++ (plus signs, double), auto-increment operator, 40
+= (plus sign, equals sign), plus-equals operator, 47
- (hyphen), in regular expressions, 108, 116
-> (hyphen, right angle bracket), 28, 157
. (period)
  in regular expressions, 107, 111
  string concatenation operator, 39
.= (period, equals sign), concatenate-equals operator, 48
/ (slash)
  division operator, 39
  in regular expressions, 110
/*…*/ (slash, asterisk), enclosing comments, 19
// (slashes, double), preceding comments, 18
/= (slash, equals sign), divide-equals operator, 47
: (colon), following labels, 59
:: (colons, double)
  preceding static method calls, 158
  preceding static properties, 163

; (semicolon), separating statements, 16
< (left angle bracket)
  HTML entity for, 93
  less than operator, 42, 97
<!DOCTYPE...> tag, in XML document, 289
<< (left angle brackets, double), left shift operator, 44
<<< (left angle brackets, triple), preceding here documents, 84
<= (left angle bracket, equals sign), less than or equal to operator, 42, 97
<=> (spaceship, aka "Darth Vadar's TIE Fighter"), 42
<> (angle brackets), inequality operator, 41
<?php…?> tag, enclosing PHP code, 8, 16, 62, 293
<?xml…?> tag, preceding XML document, 289
<?…?> SGML short tags, 63
= (equals sign), assignment operator, 47
== (equals signs, double), equal to operator, 25, 41, 97
=== (equals signs, triple), identity operator, 41, 97
=> (equals sign, right angle bracket), in array() construct, 127
> (right angle bracket)
  greater than operator, 42, 97
  HTML entity for, 93
>= (right angle bracket, equals sign), greater than or equal to operator, 42, 97
>> (right angle brackets, double), right shift operator, 44
? (question mark)
  preceding conditional expressions, 118
  preceding query string in GET request, 191
? : (question mark, colon)
  preceding noncapturing groups, 114
  ternary conditional operator, 48, 51
?! (question mark, exclamation point), in regular expressions, 116
?<! (question mark, left angle bracket, exclamation point), in regular expressions, 116
?<= (question mark, left angle bracket, equals sign), in regular expressions, 116
?= (question mark, equals sign), in regular expressions, 116
?? (null coalescing operator), 42
@ (at sign)
  error suppression operator, 48, 367
  silence operator, 60
[. … .] (square bracket, period), enclosing character classes, 111
[:…:] (square bracket, colon), enclosing character classes, 111
[=…=] (square bracket, equals sign), enclosing equivalence classes, 112
[…] (square brackets)
  appending array values using, 128
  enclosing array keys, 126
  enclosing character classes, 108
\ (backslash)
  preceding C-string escape sequences, 96
  preceding regular expression escape sequences, 107
  preceding SQL escape sequences, 96
  in regular expressions, 117
^ (caret)
  bitwise XOR operator, 44
  in regular expressions, 107, 108
^= (caret, equals sign), bitwise-XOR-equals operator, 48
__ (underscores, two), reserved for methods in PHP, 159
`…` (backticks), execution operator, 48, 332
{…} (curly braces)
  enclosing code blocks, 16
  enclosing multidimensional arrays, 130
  enclosing variables to be interpolated, 82
| (vertical bar)
  bitwise OR operator, 43
  in regular expressions, 109
|= (vertical bar, equals sign), bitwise-OR-equals operator, 48
|| (vertical bars, double), logical OR operator, 45
~ (tilde), bitwise negation operator, 43
– (minus sign)
  negation operator, 39
  subtraction operator, 39
– – (minus signs, double), auto-decrement operator, 40
–= (minus sign, equals sign), minus-equals operator, 47
'…' (single quotes)
  enclosing array keys, 126
  enclosing string literals, 24, 82
  HTML entity for, 93
"…" (double quotes)

enclosing array keys, 126
enclosing string literals, 24, 83
HTML entity for, 93

## A

ab benchmarking utility, 343
abs function, 396
abstract methods, 168-169
Accept header, 188
acos function, 396
acosh function, 396
addcslashes function, 96, 396
AddFont method, FPDF, 277
addition operator (+), 38
addLink method, FPDF, 281
AddPage method, FPDF, 273
addslashes function, 96, 396
aliases for variables, 30
AliasNbPages method, FPDF, 278, 280
allow_url_fopen option, php.ini file, 60, 332
alpha channel, 249, 265
ampersand (&)
bitwise AND operator, 43
HTML entity for, 93
indicating value returned by reference, 71, 76
ampersand, equals sign (&=), bitwise-AND-equals operator, 48
ampersands, double (&&), logical AND operator, 44
anchors, in regular expressions, 107, 112
AND operator
bitwise (&), 43
logical (&&, and), 44
AND-equals operator, bitwise (&=), 48
angle brackets (<>), inequality operator, 41
anonymous classes, 171
anonymous functions, 77-78
antialiasing, 249, 256
application techniques, 335-349
benchmarking, 343-344
code libraries, 335
execution time, optimizing, 346
load balancing, 347
memory requirements, optimizing, 346
MySQL replication, 348
output buffering, 339-341
output compression, 341
performance tuning for, 342-349
profiling, 344-346
redirection, 347
reverse proxy caches, 347
templating systems for, 336-339
approximate equality, string comparisons, 99-100
arithmetic operators, 38
array keyword, 74
(array) operator, 45
array value type, JSON, 310
array() construct, 26, 127
arrays, 125-153
acting on entire, 147-148
appending values to, 128
assigning a range of values to, 128
associative arrays, 125
basics, 26-27
calculating difference between, 147
calculating sum of, 147
casting operators, 46
converting between arrays and variables, 134-135
creating, 26, 127-128
data types, implementing, 149
differences between, 147
elements, checking existence of, 132
empty, 128
filtering elements of, 148
filtering with regular expression, 123
for loop, using, 138
foreach construct, 136
functions for, 136-137, 385-386
identifying elements of, 126
indexed arrays, 125
inserting elements in, 133-134
keys of, 126, 129, 132
merging, 147
multidimensional arrays, 129
multiple values of, extracting, 130-134
padding with identical values, 129
randomizing order of, 146
reducing, 139
removing elements in, 133-134
reversing order of, 145
searching for values in, 140-142
sets implemented using, 149
size of, determining, 129
slicing, 131
sorting, 27, 142-146

splicing, 133-134
splitting into chunks, 131
stacks implemented using, 149-151
storing data in, 127-129
traversing, 27, 135-142
values of, 130-134, 140-142
array_change_key_case function, 396
array_chunk function, 131, 396
array_combine function, 397
array_count_values function, 397
array_diff function, 147, 397
array_diff_assoc function, 397
array_diff_key function, 397
array_diff_uassoc function, 397
array_diff_ukey function, 398
array_fill function, 398
array_fill_keys function, 398
array_filter function, 148, 398
array_flip function, 93, 398
array_intersect function, 149, 399
array_intersect_assoc function, 399
array_intersect_key function, 399
array_intersect_uassoc function, 399
array_intersect_ukey function, 399
array_keys function, 132, 400
array_key_exists function, 132, 399
array_map function, 400
array_merge function, 149, 400
array_merge_recursive function, 400
array_multisort function, 145, 400
array_pad function, 129, 401
array_pop function, 149, 401
array_product function, 401
array_push function, 149, 401
array_rand function, 402
array_reduce function, 139, 402
array_replace function, 402
array_replace_recursive function, 402
array_reverse function, 145, 402
array_search function, 402
array_shift function, 150, 403
array_slice function, 131, 403, 403
array_splice function, 133-134
array_sum function, 147, 403
array_udiff function, 403
array_udiff_assoc function, 404
array_udiff_uassoc function, 404
array_uintersect function, 404
array_uintersect_assoc function, 404
array_uintersect_uassoc function, 405
array_unique function, 149, 405
array_unshift function, 150, 405
array_values function, 132, 405
array_walk function, 138, 405
array_walk_recursive function, 406
arsort function, 142, 406
as keyword, 168
asin function, 66, 406
asinh function, 406
asort function, 27, 142, 406
assert function, 406
assertion operator (+), 39
assert_options function, 406
assignment operator (=), 47
assignment operators, 46-48
associative arrays, 125
associative index, defined, 26
associativity of operators, 37
asterisk (*), multiplication operator, 39
asterisk, equals sign (*=), multiply-equals operator, 47
asXml method, SimpleXML, 305
at sign (@)
error suppression operator, 48, 367
silence operator, 60
atan function, 407
atan2 function, 407
atanh function, 407
attributes method, SimpleXML, 305
authentication, HTTP, 206
AUTH_TYPE element, $_SERVER array, 190
auto-decrement operator (– –), 40
auto-increment operator (++), 40


## B


backreferences, regular expressions, 114
backslash (\)
preceding C-string escape sequences, 96
preceding regular expression escape sequences, 107
preceding SQL escape sequences, 96
in regular expressions, 117
backticks (`...`), execution operator, 48, 332
base64_decode function, 407
base64_encode function, 407
basename function, 325, 407
base_convert function, 407
Benchmark class, PEAR, 345

benchmarking, 343-344
bin2hex function, 408
binary numbers, 23
bindec function, 43, 408
bitwise operators, 42-44
  bitwise AND (&), 43
  bitwise negation (~), 43
  bitwise OR (|), 43
  bitwise XOR (^), 44
  bitwise-AND-equals (&=), 48
  bitwise-OR-equals (|=), 48
  bitwise-XOR-equals (^=), 48
  left shift (<<), 44
  right shift (>>), 44
block, in if statement, 49
(bool) operator, 45
(boolean) operator, 45
boolean value type, JSON, 310
Booleans, 25
bound parameters, for SQL protection, 324
break keyword, 51, 53
buildTable method, FPDF, 284
buttons, creating dynamic, 13-14, 258-262


## C


C comment style, 18-20
C++ comment style, 18
C-strings, encoding and decoding, 96
Cache-Control header, 347
caching
  for dynamically generated buttons, 259-262
  web caching, 205, 347
callable keyword, 74
callbacks, 29
call_user_func function, 29, 408
call_user_func_array function, 408
caret (^)
  bitwise XOR operator, 44
  in regular expressions, 107, 108
caret, equals sign (^=), bitwise-XOR-equals operator, 48
case folding option, XML parser, 296
case of strings, changing, 91
case sensitivity
  basics, 15
  of class names, 159
  of regular expressions, 107
casting operators, 45-46
casting, implicit, 37
CDATA (character data), XML, 292
ceil function, 408
cell method, FPDF, 273, 274, 276, 277
character classes, in regular expressions, 108-109
character data, XML (see CDATA (character data), XML)
character encoding option, XML parser, 296
chdir function, 408
checkdate function, 408
checkdnsrr function, 408
chgrp function, 409
children method, SimpleXML, 305
chmod function, 409
chown function, 409
chr function, 409
chroot function, 409
chunk_split function, 409
class keyword, 27, 159
classes, 159-176
  anonymous, 171
  case insensitivity of, 15
  case sensitivity of, 159
  constants in, 163
  constructors for, 169
  declaring or defining, 159-170
  defining or declaring, 27
  destructors for, 170
  examining, 171-172
  functions for, 386
  inheritance of, 156, 164
  interfaces for, 165
  introspection of, 171-176
  methods of (see methods)
  names of, 21
  properties of (see properties)
  static methods called on, 158
  traits shared by, 165-168
class_alias function, 410
class_exists function, 171, 410
class_implements function, 410
class_parents function, 410
clearstatcache function, 411
__clone method, 158
clone operator, 158
close function, 422
closedir function, 411
closelog function, 411
collating sequences, regular expressions, 111

colon (:), following labels, 59
colons, double (::)
  preceding static method calls, 158
  preceding static properties, 163
color palette, 248, 264
colors
  images, 248, 264-269
  text in PDF files, 277
COM, interfacing with, 381-384
command-line scripting, 1
comments, 17-20
commit method, database, 223
community server, 223
compact function, 134, 411
comparison operators, 25, 41-42, 97
concatenate-equals operator (.=), 48
conditional (ternary) operator (? :), 48, 51
conditional expressions, in regular expressions, 118
conditional statements, 49-53
configuration (see php.ini file)
connect function, 28
connection_aborted function, 411
connection_status function, 411
const keyword, 163
constant function, 411
constants, 21, 163
__construct method, 170
constructors, 169
content-escaping rules, 319
Content-Type header, 188, 205, 251
continue statement, 54
convert_cyr_string function, 412
convert_uudecode function, 412
convert_uuencode function, 412
cookies, 208-212, 215
coordinates, for PDF file, 273-276
copy function, 412
copy-on-write, 33
cos function, 412
cosh function, 412
count function, 129, 412
count_chars function, 413
crc32 function, 413
CREATE command, SQL, 226
create_function function, 29, 413
cross-site scripting (XSS), 322
crypt function, 413
curl extension, 354-356
curly braces {...}
  enclosing code blocks, 16
  enclosing multidimensional arrays, 130
  enclosing variables to be interpolated, 82
curl_setopt function, 356
current function, 136, 413
current method, 151
cut (once-only subpattern), in regular expressions, 118

# D

Data Definition Language (see DDL)
data filtering (see filtering input)
Data Manipulation Language (see DML)
data types, 22-29, 37, 75, 149
databases, 217-245
  accessing, 217
  as alternative to files, 329
  connecting to, 220
  debugging statements, 223
  file manipulation as alternative to, 229-237
  interacting with, 221
  MariaDB, 224
  MongoDB, 237-245
  MySQLi interface, 223-225
  PDF files, adding data to, 283-285
  PDO library for, 217, 219-223
  prepared statements for, 221
  protecting, 318-319, 323
  querying, 11-13
  RDBMS, 218-223
  as resources, 28
  retrieving data for display, 225
  SQL commands for, 218-219
  SQLite, 226-229
  supported by PHP, 2, 11-13, 217
  transactions for, 222
date function, 181, 413
DateInterval class, 183
dates and times, 181-185, 387
DateTime class, 182-183
DateTimeZone class, 182-185
date_default_timezone_get function, 414
date_default_timezone_set function, 414
date_format function, 182
date_parse function, 415
date_parse_from_format function, 415
date_sunrise function, 416
date_sunset function, 416

date_sun_info function, 415
DDE (Dynamic Data Exchange), 382
DDL (Data Definition Language), 218
debugDumpParams method, database, 223
debugging, 361-376
  development environment, 361-362
  error handling, 365-371
  error logs, 373
  IDEs, 374-375
  manual debugging, 371-373
  PDO statements, 223
  php.ini settings, 363-365
  production environment, 363
  staging environment, 362
debug_backtrace function, 416
debug_print_backtrace function, 417
decbin function, 43, 417
dechex function, 417
declare statement, 58
decoct function, 43, 417
decomposing a string, 102-104
default keyword, 51
default parameters, 72
Defense in Depth principle, 323
define function, 21, 163, 417
defined function, 417
define_syslog_variables function, 417
deflate_add function, 418
deflate_init function, 418
deg2rad function, 418
DELETE command, SQL, 218
DELETE verb, REST, 352, 356
delimiters, in regular expressions, 110
__destruct method, 170
destructors, 170
die function, 66
diff method, 183
dir function, 418
directives (see execution directives)
directories, functions for, 387
dirname function, 418
disable_functions option, php.ini file, 332
disk_free_space function, 418
disk_total_space function, 419
displayClasses function, 172
display_errors option, php.ini file, 363, 365
divide-equals operator (/=), 47
division operator (/), 39
DML (Data Manipulation Language), 218
do/while statement, 54
<!DOCTYPE...> tag, in XML document, 289
document type definition (see DTD)
dollar sign ($)
  preceding variable names, 20, 30
  in regular expressions, 107
dollar signs, double ($$), preceding variables containing variable names, 30
DOM parser, for XML, 304
(double) operator, 45
double quotes ("…")
  enclosing array keys, 126
  enclosing string literals, 24, 83
  HTML entity for, 93
DTD (document type definition), 288
Dynamic Data Exchange (DDE), 382
dynamic web content, 1

## E

each function, 137, 419
echo construct, 8, 85, 346, 371, 419
EGPCS (environment, GET, POST, cookies, server), 188, 327
else keyword, 49
elseif statement, 50
email, sending, 379
empty function, 419
encapsulation, 156, 158
enclosing scope of anonymous function, 78
encoding directive, 58
encryption of data, 333
end function, 137, 419
end-of-file handling, 380
end-of-line handling, 380
endfor keyword, 56
endwhile keyword, 53
entities
  $_ENV array, 189
  HTML, 92-94
  XML, 293-295
equal to operator (==), 25, 41, 97
equals sign (=), assignment operator, 47
equals sign, right angle bracket (=>), in array() construct, 127
equals signs, double (==), equal to operator, 25, 41, 97
equals signs, triple (===), identity operator, 41, 97
equivalence classes, regular expressions, 111

error handling, 365-371
  defining error handlers, 368-371
  exceptions, 366
  functions for, 388
  reporting errors, 365
  suppression of errors, 367
  triggering errors, 367
  try…catch statement, 57
error suppression operator (@), 48
error_clear_last function, 419
error_get_last function, 419
error_log function, 369-370, 420
error_log option, php.ini file, 364, 365
error_reporting function, 365, 366, 368, 420
error_reporting option, php.ini file, 363, 365-366
escape sequences for C-strings, 96
  for SQL, 96
  for strings, 82
escapeshellarg function, 332, 421
escapeshellcmd function, 421
escaping output data, 318-319
Essential PHP Security (O'Reilly), 333
eval function, 293, 331-332
exact string comparisons, 97-98
exclamation point (!), logical negation operator, 45
exclamation point, double equals signs (!==), not identical operator, 42
exclamation point, equals sign (!=), inequality operator, 41
exec function, 332, 421
execute method, database, 221
execution directives, 58
execution operator (`…`), 48
execution time, optimizing, 346
exit statement, 58
exp function, 421
expiration of web documents, 205
Expires header, 205
explode function, 102, 421
expm1 function, 421
exponentiation (**), 39
expressions, 35
  (see also regular expressions)
  number of operands in, 36
  operator precedence in, 36
extends keyword, 164, 165
Extensible Markup Language (see XML)
Extensible Stylesheet Language Transformations (see XSLT)
extensions (libraries)
  concealing, 330
  creating, 335
  platform-specific, 381
extension_loaded function, 421
external entities, XML, 294
external links, PDF files, 281
extract function, 134, 422

## F

fall-through, switch statement, 52
false keyword, 26
  (see also Booleans)
fclose function, 229
feof function, 381, 422
fflush function, 422
fgetc function, 422
fgetcsv function, 423
fgets function, 423
fgetss function, 423
file function, 236, 423
file uploads, 200-201, 327-328
fileatime function, 424
filectime function, 424
filegroup function, 424
fileinode function, 424
filemtime function, 424
filenames
  pathname differences, handling, 378
  security vulnerability, 324-326
fileowner function, 424
fileperms function, 424
files
  database as alternative to, 329
  as alternative to database, 229-237
  end-of-file handling, 380
  functions for, 229, 388-389
  including, 59-61
  permissions for, 329
  session files, protecting, 330
filesize function, 229, 259, 425
filetype function, 425
file_exists function, 229, 423
file_get_contents function, 234, 424
file_put_contents function, 425
fillTemplate function, 338
filtering input, 315-318, 387

filter_has_var function, 425
filter_id function, 425
filter_input function, 426
filter_input_array function, 426
filter_list function, 426
filter_var function, 426
filter_var_array function, 426
final keyword, 160
findone method, database, 241
flags, in regular expressions, 114
(float) operator, 45
floating-point numbers, 23-24
floatval function, 427, 427
flock function, 229, 234
floor function, 427
flow-control statements, 49-59
flush function, 340, 427
fmod function, 427
fnmatch function, 427
font attributes, for PDF file, 276-278
fonts for graphics, 255-259
footer method, FPDF, 278, 280
footers, in PDF files, 278-279
fopen function, 229, 233, 325, 329, 428
for statement, 55-56, 138
foreach statement, 27, 57, 136, 151-153
form tag, method attribute, 201
format method, 182, 183
format string, 86-88
forms, 191-204
    creating, 10
    file uploads in, 200-201
    methods for, 191
    multivalued parameters for, 197-200
    parameters, 192-193
    self-processing, 194-195
    sticky forms, 196, 199-200
    validating, 202-204
forward_static_call function, 429
forward_static_call_array function, 429
fpassthru function, 429
FPDF constructor, 273
FPDF library, 271, 278
fprintf function, 429
fputcsv function, 429
Francia, Steve (author)
    MongoDB and PHP (O'Reilly), 237
fread function, 229, 237, 429
fscanf function, 430
fseek function, 430
fsockopen function, 430
fstat function, 430
ftell function, 431
ftruncate function, 431
functions, 65-79
    anonymous, 77-78
    arrays, 136-137, 385
    callbacks, 29
    calling, 65-66, 138-139
    case insensitivity of, 15
    classes and objects, 386
    data filtering, 387
    dates and times, 182-185, 387
    defining, 67-68
    directories, 387
    errors and logging, 388
    file management, 229
    filesystem, 388-389
    filtering an array with a regular expression, 123
    HTML in, 67
    iterator, 136-137
    lib, 395
    mail, 389
    matching of, 119-120
    math, 389-390
    miscellaneous, 101, 390
    names of, 21
    nesting, 68
    network, 390
    output buffering, 391
    parameters, 65, 67, 69, 71-75
    PHP options, 391-392
    program execution, 392
    quoting for regular expressions, 123
    in regular expressions, 119-123
    replacing, 121-122
    return value of, 67, 75
    scope of parameters in, 33
    scope of variables in, 69-71
    session handling, 392
    splitting, 122
    streams, 393
    strings, 101, 393-395
    tokenizer, 391
    type hinting, 74, 161
    URLs, 395
    variables containing name of, 76

for variables, 395
function_exists function, 431
func_get_arg function, 431
func_get_args function, 431
func_num_args function, 431
fwrite function, 229, 234, 431

## G

garbage collection, 33-34
GATEWAY_INTERFACE element, $_SERVER array, 190
gc_collect_cycles function, 432
gc_disable function, 432
gc_enable function, 432
gc_enabled function, 432
GD extension, 13-14, 247
Genghis (for MongoDB), 238
__get method, 163
GET method, HTTP, 187, 191
GET verb, REST, 352, 354
getcwd function, 436
getdate function, 436
getenv function, 437
gethostbyaddr function, 437
gethostbyname function, 437
gethostbynamel function, 437
gethostname function, 437
getlastmod function, 437
getLocation method, 184
getmxrr function, 437
getmygid function, 435
getmyinode function, 437
getmypid function, 437
getmyuid function, 436
getopt function, 438
getprotobyname function, 438
getprotobynumber function, 438
getrandmax function, 438
getrusage function, 438
getservbyname function, 438
getservbyport function, 439
gettimeofday function, 439
gettype function, 439
get_browser function, 432
get_called_class function, 432
get_cfg_var function, 432
get_class function, 173, 432
get_class_methods function, 172, 433
get_class_vars function, 172, 173, 433
get_current_user function, 433
get_declared_classes function, 171, 433
get_declared_interfaces function, 433
get_declared_traits function, 433
get_defined_constants function, 433
get_defined_functions function, 433
get_defined_vars function, 434
get_extension_funcs function, 434
get_headers function, 434
get_html_translation_table function, 93, 434
get_included_files function, 61, 435
get_include_path function, 435
get_ini function, 184
get_loaded_extensions function, 435
get_meta_tags function, 94, 435
get_object_vars function, 173, 436
get_parent_class function, 174, 436
get_resource_type function, 436
glob function, 439
global keyword, 32, 69
global scope, variable, 32
global variables, 69, 78
gmdate function, 440
gmmktime function, 440
gmstrftime function, 440
goto statement, 59
graphics, 247-269
    alpha channel, 249, 265
    antialiasing for, 249, 256
    basics, 248
    for buttons, dynamic, 258-262
    color handling, 264-269
    creating and drawing, 13-14, 249-254
    embedding in a page, 247-248
    file formats for, 249, 251-252
    in PDF files, 280-282
    reading existing graphics files, 252
    scaling, 262
    structure of a program, 250
    text in, 254-259
    text representation of, 268
    transparency of, 249
    true color indexes for, 249, 267-268
greater than operator (>), 42, 97
greater than or equal to operator (>=), 42, 97
greed, of regular expressions, 113
Gutmans, Andi (developer of PHP), 5

## H

handles, 28
Harold, Elliotte Rusty (author)
    XML in a Nutshell (O'Reilly), 289
hash function, 440
hash mark (#), preceding comments, 17
hash_algos function, 440
hash_file function, 440
header function, 204, 289, 441
header method, FPDF, 278
headers, HTTP
    request headers, 187, 191
    response headers, 188, 204-207
headers, in PDF files, 278-279
headers_list function, 441
headers_sent function, 441
header_remove function, 441
hebrev function, 441
here documents (heredocs), 81, 83-85
hex2bin function, 441
hexadecimal numbers, 23
hexdec function, 441
highlight_file function, 441
highlight_string function, 442
history of PHP, 2-7
hrtime function, 442
HTML
    converting special characters to entities in, 92-94
    echoing PHP content in, 63
    embedding PHP code in, 8, 61-64
    including in functions, 67
    loading from another module, 59-61
    meta tags, finding in strings, 94
    removing tags from strings, 94
HTML & XHTML: The Definitive Guide (O'Reilly), xx
htmlentities function, 92, 318, 442
htmlspecialchars function, 93, 443
htmlspecialchars_decode function, 444
html_entity_decode function, 443
HTTP (HyperText Transfer Protocol)
    authentication, 206
    basics, 187-188
    cookies with, 208-212, 215
    maintaining state, 207-215
    methods, 187, 187, 191
    server information, 189-191, 247
    server response headers, 204-207
    sessions with, 212-215
    status codes for, 353
    variables for, 188
HTTP Pocket Reference (O'Reilly), 187
HTTPS, 216
http_build_query function, 444
HTTP_REFERER element, $_SERVER array, 191
HTTP_USER_AGENT element, $_SERVER array, 191
Hypertext Transfer Protocol (see HTTP)
hyphen (-), in regular expressions, 108, 116
hyphen, right angle bracket (->), accessing object members, 28, 157
hypot function, 444

## I

idate function, 444
IDE (Integrated Development Environment), 374-375
idempotence, 192
identifiers, 20-21
identity operator (===), 41, 97
if statement, 49-51
if tag, 5
ignore_repeated_errors option, php.ini file, 364
ignore_user_abort function, 445
image method, FPDF, 280, 281
imagearc function, 253
imagecolorallocate function, 250, 264
imagecolorallocatealpha function, 264, 267
imagecolorat function, 266, 268
imagecolorresolvealpha function, 267
imagecolorsforindex function, 266
imagecopyresampled function, 262
imagecopyresized function, 262
imagecreate function, 250, 264
imagecreatefromjpeg function, 252
imagecreatefrompng function, 252
imagecreatetruecolor function, 264
imagedashedline function, 253
imagefill function, 253
imagefilledpolygon function, 253
imagefilledrectangle function, 251, 253, 264, 266
imagefilltoborder function, 253
imagegif function, 251
imagejpg function, 251
imageline function, 253

imageloadfont function, 255
imagepng function, 251
imagepolygon function, 253
imagerectangle function, 253
imagerotate function, 253
images (see graphics)
imagesetpixel function, 253
imagestring function, 255
imagetruecolortopalette function, 265
imagettftext function, 256, 257
imagetypes function, 252, 252
imagewbmp function, 251
implements keyword, 165
implicit casting, 37
implode function, 103, 445
include function, 325
include keyword, 59-61
include_once function, 335, 346
include_once keyword, 60
indexed arrays, 125
inequality operator (!= or <> ), 41
inet_ntop function, 445
inet_pton function, 445
inheritance, 156, 164
ini_get function, 446
ini_get_all function, 446
ini_restore function, 446
ini_set function, 185, 365, 446
inline options, regular expressions, 116
INSERT command, SQL, 218
insert method, database, 239
installation, 7
instanceof operator, 49
insteadof keyword, 167
(int) operator, 45
intdiv function, 446
(integer) operator, 45
integers, 22-23
Integrated Development Environment (IDE), 374-375
interfaces, defining for classes, 165
interface_exists function, 446
internal links, PDF files, 281
interpolation of variables, 81
introspection, for classes, 171-176
intval function, 446
in_array function, 140, 445
ip2long function, 447
isset function, 34, 133, 449
is_a function, 447
is_array function, 27, 447
is_bool function, 26, 447
is_callable function, 447
is_countable function, 447
is_dir function, 447
is_executable function, 447
is_file function, 448
is_finite function, 448
is_float function, 448
is_infinite function, 448
is_int function, 23, 448
is_integer function, 23
is_iterable function, 448
is_link function, 448
is_nan function, 448
is_null function, 29, 448
is_numeric function, 448
is_object function, 28, 173, 448
is_readable function, 237, 449
is_resource function, 29, 449
is_scalar function, 449
is_string function, 25, 449
is_subclass_of function, 449
is_uploaded_file function, 328, 449
is_writable function, 237, 449
iterator functions, 136-137
Iterator interface, 151-153

## J

join function, 103
JSON (JavaScript Object Notation), 309-313
JsonSerializable interface, 310-313
JSON_BIGINT_AS_STRING option, 312
json_decode function, 309, 312, 450
json_encode function, 309, 310, 450
JSON_FORCE_OBJECT option, 312
JSON_INVALID_UTF8_IGNORE option, 313
JSON_INVALID_UTF8_SUBSTITUTE option, 313
JSON_NUMERIC_CHECK option, 312
JSON_OBJECT_AS_ARRAY option, 312
JSON_PRETTY_PRINT option, 313
JSON_THROW_ON_ERROR option, 313

## K

Kennedy, Bill (author)
    HTML & XHTML: The Definitive Guide (O'Reilly), xx

key function, 137, 450
key method, 151
keywords, 21-22
Kline, Keven (author)
  SQL in a Nutshell (O'Reilly), 219
krsort function, 142, 450
ksort function, 143, 450

## L

labels, goto statement, 59
Lane, David (author)
  Web Database Applications with PHP and MySQL, 2nd Edition (O'Reilly), 217
lcfirst function, 450
lcg_value function, 450
lchgrp function, 450
lchown function, 451
Learning XML (O'Reilly), 289
left angle bracket (<)
  HTML entity for, 93
  less than operator, 42, 97
left angle bracket, equals sign (<=), less than or equal to operator, 42, 97
left angle bracket, equals, right angle bracket (<=>), spaceship, aka “Darth Vader’s TIE Fighter”, 42
left angle brackets, triple (<<<), preceding here documents, 84
left shift operator (<<), 44
Lerdorf, Rasmus (developer of PHP), 2-7
less than operator (<), 42, 97
less than or equal to operator (<=), 42, 97
Levenshtein algorithm, 99
levenshtein function, 99, 451
lexical structure of PHP, 15-21
libraries (see extensions)
Libxslt library, 306
line spaces, 16
link function, 451
linkinfo function, 451
list function, 130, 451
literals
  basics, 20
  floating-point numbers, 23-24
  integer, 23
  strings, 81-85
ln method, FPDF, 274, 276
load balancing, 347
local scope, variable, 31, 33
localeconv function, 451
localhost environment, 362
localtime function, 452
Location header, 205
log function, 452
log10 function, 453
log1p function, 453
logical operators, 44
  logical AND (&&, and), 44
  logical negation (!), 45
  logical OR (||, or), 45
  logical XOR (xor), 45
long2ip function, 453
lookahead and lookbehind, regular expressions, 116-117
loop statements, 53-57
  for statement, 138
  foreach statement, 27, 136, 151-153
lstat function, 453
ltrim function, 90, 453

## M

mail
  functions for, 379, 389, 453
  sending, 379
MariaDB, 224
match behavior, regular expressions, 111, 119-120
math
  arithmetic operators, 38
  functions for, 389-390
max function, 453
MAX_FILE_SIZE parameter, 200
mb_strlen function, 317
md5 function, 453
md5_file function, 454
Means, W. Scott (author)
  XML in a Nutshell (O'Reilly), 289
memory management, 33-34
memory requirements, optimizing, 346
memory_get_peak_usage function, 454
memory_get_usage function, 454
meta tags, finding in strings, 94
Metaphone algorithm, 99
metaphone function, 99, 454
method attribute, form tag, 201
methods
  abstract, 168-169
  accessing, 27, 157-158

callbacks, 29
constructors, 169
declaring or defining, 159-161
defined, 156
destructors, 170
introspection of, 171
preventing from overriding, 160
protected, 160
public or private, 158, 160
static, 158, 160
methods, HTTP
GET method, 187, 191
POST method, 188, 191, 192
method_exists function, 173, 454
microtime function, 344, 454
min function, 454
minimal matching, regular expression searching, 113
minus sign (–)
negation operator, 39
subtraction operator, 39
minus sign, equals sign (–=), minus-equals operator, 47
minus signs, double (– –), auto-decrement operator, 40
minus-equals operator (–=), 47
missing parameters, 74
mkdir function, 229, 231, 455
mktime function, 455
modulus operator (%), 39
modulus-equals operator (%=), 48
MongoDB and PHP (O'Reilly), 237
MongoDB database, 237-245
move_uploaded_file function, 201, 328, 455
mt_getrandmax function, 455
mt_rand function, 455
mt_srand function, 455
multidimensional arrays, 129
multiple arrays, sorting, 145
multiplication operator (*), 39
multiply-equals operator (*=), 47
multivalued parameters, forms, 197-200
multi_query method, database, 225, 228
Musciano, Chuck (author)
HTML & XHTML: The Definitive Guide (O'Reilly), xx
MySQL replication, 348
MySQLi object interface, 223-225
mysqli_real_escape_string function, 319

## N

namespaces, in XML, 289
natcasesort function, 144, 455
natsort function, 144, 456
natural-order sorting, arrays, 144
negation operator (–), 39
negation operator, bitwise (~), 43
negation operator, logical (!), 45
network functions, 390
(see also HTTP)
new keyword, 28, 157
next function, 137, 456
next method, 151
nl2br function, 456
nl_langinfo function, 456
noncapturing groups, regular expressions, 114
NoSQL databases (MongoDB), 237-245
not identical operator (!==), 42
null coalescing operator (??), 42
NULL data type, 29, 30
NULL keyword, 29, 75
null value type, JSON, 310
number value type, JSON, 310
numbers
integers, 22-23
sorting strings containing, 144
strings used as, 38
number_format function, 456

## O

Object Linking and Embedding (OLE), 382
(object) operator, 45
object value type, JSON, 310
object-oriented programming (see OOP)
objects, 155-179
basics, 27
classes (see classes)
cloning, 158
creating, 157
examining, 173
functions for, 386
instantiating, 28
iterating over, 151-153
methods of (see methods)
properties of (see properties)
serializing, 177-179
terminology in PHP, 156
testing whether value is, 28
ob_clean function, 340, 457

ob_end_clean function, 340, 457
ob_end_flush function, 340, 457
ob_flush function, 340, 457
ob_get_clean function, 457
ob_get_contents function, 340, 457
ob_get_flush function, 457
ob_get_length function, 340, 457
ob_get_level function, 457
ob_get_status function, 458
ob_gzhandler function, 341, 458
ob_implicit_flush function, 458
ob_implicit_handlers function, 458
ob_start function, 340, 458
octal numbers, 23
octdec function, 43, 458
OLE (Object Linking and Embedding), 382
OOP (object-oriented programming), 27, 155, 224, 226
Open Web Application Security Project, 319, 333
opendir function, 459
openlog function, 459
open_basedir option, php.ini file, 328
operating systems, 377-384
    API specifications, 384
    COM, interfacing with, 381-384
    determining platform, 378
    end-of-file handling, 380
    end-of-line handling, 380
    extensions common to, 381
    external commands, using, 381
    mail, sending, 379
    navigating server environment, 378
    pathname differences, handling, 378
    supported by PHP, 1, 377
operators, 35-49
    arithmetic operators, 38
    assignment operators, 46-48
    associativity of, 37
    bitwise, 42-44
    casting operators, 45-46
    comparison operators, 41-42, 97
    implicit casting used with, 38
    list of, 35
    logical operators, 44
    number of operands used by, 36
    precedence of, 36
optimization (see performance tuning)
OR operator, bitwise (|), 43
OR operator, logical (||, or), 45
OR-equals operator, bitwise (|=), 48
ord function, 459
output buffering, 339-341
    in error handlers, 370
    functions for, 391
output compression, 341
output formats, 2
output method, FPDF, 273
output_add_rewrite_var function, 460
output_reset_rewrite_vars function, 460

## P

pack function, 460
page layout, for PDF file, 273-276
parameters, 65, 71-75
    default, 72
    defined in function, 67, 69
    form, 192-193
    missing, 74
    passing by reference, 71
    passing by value, 71
    scope of, 33
    type-hinted, 75
    variable number of, 73-74
parentheses (...)
    enclosing subpatterns, 110
    forcing operator precedence, 37
parse_ini_file function, 461
parse_ini_string function, 461
parse_str function, 262, 461
parse_url function, 106, 461
passthru function, 332, 462
pathinfo function, 462
pathname differences, handling, 378
PATH_INFO element, $_SERVER array, 190
PATH_TRANSLATED element, $_SERVER array, 190
patterns, matching (see regular expressions)
pclose function, 462
PDF files, 271-285
    creating, 271-273
    data from database in, 283-285
    graphics in, 280-282
    links in, 280-282
    PHP extensions for, 271
    tables in, 283-285
    text in, 273-279
        color of, 277

coordinates for, 273-276
font attributes of, 276-278
headers and footers, 278-279
PDO (PHP Data Objects) library, 217, 219-223
PEAR (PHP Extension and Application Repository), 2, 345
percent sign (%)
in format string, 86
modulus operator, 39
percent sign, equals sign (%=), modulus-equals operator, 48
performance tuning, 342-349
benchmarking, 343-344
execution time, optimizing, 346
load balancing for, 347
memory requirements, optimizing, 346
MySQLi replication for, 348
profiling for, 344-346
redirection for, 347
reverse proxy caches, 347
period (.)
in regular expressions, 107, 111
string concatenation operator, 39
period, equals sign (.=), concatenate-equals operator, 48
Perl-compatible regular expressions, 124
(see also regular expressions)
pfsockopen function, 462
PHP, xix-xxii, 1-14
configuring (see php.ini file)
data types, 22-29
databases supported, 2, 11-13
debugging, 361-376
embedding in web pages, 61-64
expressions, 35
flow-control statements, 49-59
forms, 10
graphics, 13-14
history of, 2-7
informational functions for, 391-392
installing, 7
lexical structure, 15-21
loading from another module, 59-61
operating systems supported by, 1, 377
operators, 35-49
output formats supported, 2
portable code for Windows and Unix, 377-381
security issues for code, 331-332
usage of, 7
variables, 30-34
versions of, 7
web servers supported by, 1
PHP Data Object library (see PDO (PHP Data Objects) library)
PHP Extension and Application Repository (see PEAR)
<?php…?> tag, enclosing PHP code, 8, 16, 62, 293
php.ini file, 7, 363-365
allow_url_fopen option, 60, 332
assert.exception option, 364
changing settings, 364
disable_functions option, 332
displaying information about, 9
display_errors option, 363, 365
error_log option, 364, 365
error_reporting option , 363, 365
ignore_repeated_errors option, 364
open_basedir option, 328
post_max_size option, 327
register_globals option, 333
request_order option, 364, 365
sendmail_path option, 379
session.cookie_lifetime option, 214
session.save_path option, 214
session.serialize_handler option, 215
track_errors option, 366
upload_max_filesize option, 200
upload_tmp_dir option, 201
variables_order option, 364, 365
zend.assertions option, 364
phpcredits function, 464
phpinfo function, 9, 464
PHPSESSID cookie, 212
phpversion function, 464
PHP_EOL constant, 380
php_ini_loaded_file function, 463
php_ini_scanned_files function, 463
php_logo_guid function, 463
PHP_OS constant, 378
php_sapi_name function, 463
PHP_SELF element, $_SERVER array, 189
php_strip_whitespace function, 463
php_uname function, 378, 463
pi function, 464
platforms (see operating systems)
plus sign (+)

addition operator, 38
assertion operator, 39
plus sign, equals sign (+=), plus-equals operator, 47
plus signs, double (++), auto-increment operator, 40
plus-equals operator (+=), 47
popen function, 332, 465
portable code for Windows and Unix, 377-381
POST method, HTTP, 192
POST verb, REST, 352, 356
post_max_size option, php.ini file, 327
pow function, 465
precedence of operators, 36
preg_grep function, 123
preg_match function, 114, 119
preg_match_all function, 119
preg_quote function, 123
preg_replace function, 121-122, 332, 346
preg_replace_callback function, 122
preg_split function, 122
prepare method, database, 221
prev function, 137, 465
print function, 86
printf function, 86-88, 346, 465
print_r function, 88-89, 465
private keyword, 158, 160, 162, 164
proc_close function, 466
proc_get_status function, 466
proc_nice function, 467
proc_open function, 467
proc_terminate function, 467
profiling, 344-346
program execution
functions for, 392
security issues, 331-332
Programming Web Services in XML-RPC (O'Reilly), 360
Programming Web Services with SOAP (O'Reilly), 360
properties
accessing, 27, 157-158
declaring or defining, 162-163
defined, 156
introspection of, 171
protected, 162
public or private, 158, 162
static, 162
property_exists function, 468
protected keyword, 162, 164
public keyword, 158, 160, 162, 163
public properties, 162
PUT verb, REST, 352, 355
putenv function, 468

## Q

Qmail, 379
quantifiers, in regular expressions, 109, 113
query method, database, 221, 225
query string, 95, 191
queryExec method, database, 228
QUERY_STRING element, $_SERVER array, 190
question mark (?)
preceding conditional expressions, 118
preceding query string in GET request, 191
question mark, colon (? :)
preceding noncapturing groups, 114
ternary conditional operator, 48, 51
question mark, double (??) null coalescing operator, 42
question mark, equals sign (?=), in regular expressions, 116
question mark, exclamation point (?!), in regular expressions, 116
question mark, left angle bracket, equals sign (?<=), in regular expressions, 116
question mark, left angle bracket, exclamation point (?<!), in regular expressions, 116
quotation marks (see double quotes; single quotes)
quoted_printable_decode function, 468
quoted_printable_encode function, 468
quotemeta function, 468

## R

rad2deg function, 468
rand function, 468
randomizing order of arrays, 146
Random_bytes function, 468
Random_int function, 469
range function, 128, 469
rawurldecode function, 95, 469
rawurlencode function, 95, 469
Ray, Erik (author)
Learning XML (O'Reilly), 289
RDBMS (Relational Database Management System), 218-223

readdir function, 469
readfile function, 259, 469
readlink function, 469
real numbers (see floating-point numbers)
(real) operator, 45
realpath function, 325, 470
realpath_cache_get function, 470
realpath_cache_size function, 470
redirection, 205, 347
reference counting, 33-34
reference, passing parameters by, 71
register_globals option, php.ini file, 333
register_shutdown_function function, 470
register_tick_function function, 58, 470
regular expressions, 106-124
- alternatives in, 109
- anchors in, 107, 112
- backreferences in, 114
- case sensitivity of, 107
- character classes in, 108-109
- compared to Perl regular expressions, 124
- conditional expressions in, 118
- cut (once-only subpattern) in, 118
- delimiters in, 110
- filtering an array with, 123
- functions in, 119-123
- greed of, 113
- inline options in, 116
- lookahead and lookbehind in, 116-117
- match behavior in, 111, 119-120
- noncapturing groups in, 114
- quantifiers in, 109, 113
- repeating sequences in, 109
- subpatterns in, 110, 114, 118
- trailing options (flags) in, 114

Relational Database Management System (see RDBMS)
remote procedure call (see RPC)
remote procedure calls, XML in (see XML-RPC)
REMOTE_ADDR element, $_SERVER array, 190
REMOTE_HOST element, $_SERVER array, 190
REMOTE_USER element, $_SERVER array, 191
rename function, 470
repeating sequences, in regular expressions, 109
REQUEST_METHOD element, $_SERVER array, 190, 192
request_order option, php.ini file, 364, 365
require function, 272, 325
require keyword, 59-61
require_once function, 335, 346
require_once keyword, 60
reserved words (see keywords)
reset function, 136, 471
resources, 28
- database (see databases)
- information (see books and publications; website resources)

response, HTTP and web server, 187, 204-207
RESTful web service, 351-356
- resources
  - creating, 356
  - deleting, 356
  - retrieving, 354
  - updating, 355
- responses from, 353
- verbs for, 352

restore_error_handler function, 368, 471
restore_exception_handler function, 471
return keyword, 75
return statement, 58, 67, 75
reverse proxy caches, 347
rewind function, 471
rewind method, 151
rewinddir function, 471
RFC 3986 encoding, 95
RGB value, defined, 249
Rich Site Summary (see RSS)
right angle bracket (>)
- greater than operator, 42, 97
- HTML entity for, 93

right angle bracket, equals sign (>=), greater than or equal to operator, 42, 97
right shift operator (>>), 44
rmdir function, 471
rollback method, database, 223
round function, 471
RPC (remote procedure call), 287, 381-384
rsort function, 142, 472
RSS (Rich Site Summary), 289
rtrim function, 90, 472

## S

save method, 239

scandir function, 472
schema, XML, 288
scope of variables, 31-33, 69-71, 78
script style of embedding PHP, 61
<script> tag, 61
scripting, 1, 5
SCRIPT_NAME element, $_SERVER array, 190
searching
  arrays for values, 140-142
  strings, 104-106
Secure Sockets Layer (SSL), 216
security, 315-334
  data encryption considerations, 333
  escaping output data, 318-319
  file upload traps, 327-328
  filename vulnerabilities, 324-326
  filtering input, 315-318
  PHP code evaluation, 331-332
  session fixation, 326
  shell commands, 332
  SQL injection, 323
  unauthorized file access, 328-331
  XSS (cross-site scripting), 322
SELECT command, SQL, 218, 225
self keyword, 163
self-processing forms, 194
semicolon (;), separating statements, 16
sendmail functions, 379
sendmail_path option, php.ini file, 379
serialization, 177-179, 310-313
serialize function, 177, 472
Server header, 188
server-side scripting, 1
SERVER_NAME element, $_SERVER array, 190
SERVER_PORT element, $_SERVER array, 190
SERVER_PROTOCOL element, $_SERVER array, 190
SERVER_SOFTWARE element, $_SERVER array, 189
session files, protecting, 330
session fixation, 326
session tracking (see state, maintaining)
session.cookie_lifetime option, php.ini file, 214
session.save_path option, php.ini file, 214
session.serialize_handler option, php.ini file, 215
sessions, 212-215, 392
session_destroy function, 213
session_id function, 212
session_regenerate_id function, 326
session_start function, 212
set cookie function, 208
__set method, 163
setcookie function, 473
SetFont method, FPDF, 273, 276
setLink method, FPDF, 281
setlocale function, 474
setrawcookie function, 474
sets, implementing with arrays, 149
SetTextColor method, FPDF, 277
settype function, 474
set_error_handler function, 368, 472
set_exception_handler function, 473
set_include_path function, 473
set_time_limit function, 473
SGML style of embedding PHP, 63
sha1 function, 474
sha1_file function, 475
shell commands
  platform differences in, 381
  security issues, 332
shell-style comments, 17
shell_exec function, 475
Shiflett, Chris (author)
  Essential PHP Security (O'Reilly), 333
short tags, 63
shuffle function, 146, 475
silence operator (@), 60
similar_text function, 99, 475
SimpleXML, 305
sin function, 66, 475
single quotes ('…')
  enclosing array keys, 126
  enclosing string literals, 24, 82
  HTML entity for, 93
sinh function, 475
sizeof function, 129
slash (/)
  division operator, 39
  in regular expressions, 110
slash, asterisk (/*…*/), enclosing comments, 19
slash, equals sign (/=), divide-equals operator, 47
slashes, double (//), preceding comments, 18
sleep function, 475
__sleep method, 177

Smarty templating system, 339
Snell, James (author)
  Programming Web Services with SOAP (O'Reilly), 360
SOAP, 357, 360
sort function, 27, 142, 475
sorting arrays, 27, 142-146
Soundex algorithm, 99
soundex function, 99, 476
spaces (see whitespace)
spaceship (<=>), aka “Darth Vader’s TIE Fighter”, 42
special characters
  C-string escape sequences for, 96
  converting to HTML entities, 92-94
  regular expression escape sequences for, 107
  SQL escape sequences for, 96
  string escape sequences for, 82
speed testing of code, 343
sprintf function, 88, 476
SQL (Structured Query Language)
  commands, 218-219
  escaping special characters in, 96
SQL in a Nutshell (O'Reilly), 219
SQL injection, 323
SQLite database, 226-229
sqrt function, 476
square bracket, colon ([:…:]), enclosing character classes, 111
square bracket, equals sign ([=…=]), enclosing equivalence classes, 112
square bracket, period ([. … .]), enclosing character classes, 111
square brackets ([…])
  appending array values using, 128
  enclosing array keys, 126
  enclosing character classes, 108
Squid Guard, 347
Squid proxy cache, 347
srand function, 476
sscanf function, 104, 476
SSL (Secure Sockets Layer), 216
St. Laurent, Simon (author)
  Programming Web Services in XML-RPC (O'Reilly), 360
stacks, implementing with arrays, 149-151
stat function, 476
state, maintaining between server and HTTP, 207-215
statements
  basics, 16
  conditional statements, 49-53
  flow-control statements, 49-59
  loop statements, 53-57
static keyword, 33, 70, 160
static methods, 158, 160
static variables, 33, 70
sticky forms, 196, 199-200
strcasecmp function, 98, 478
strchr function, 105
strcmp function, 41, 98, 478
strcoll function, 478
strcspn function, 106, 479
streams, 393
strftime function, 479
strict_types directive, 58
string concatenation operator (.), 39
(string) operator, 45
string value type, JSON, 310
strings, 81-124
  accessing individual characters of, 89
  changing case of, 91
  cleaning, 90-91
  comparing, 97-100
  concatenating, 39
  decomposing, 102-104
  encoding and escaping, 91-97
  escape sequences for, 82
  functions for, 393-395
  literals, 81-85
  padding with another string, 102
  printing to web pages, 85-89
  quoting string constants, 81-85
  removing HTML tags in, 94
  repeating, 102
  reversing, 101
  searching, 104-106
    (see also regular expressions)
  special characters in (see special characters)
  substrings of, 100-101
  used as numbers, 38
stripcslashes function, 96, 480
stripslashes function, 96, 480
strip_tags function, 94, 480
stristr function, 105, 480
strlen function, 66, 89, 480
strnatcasecmp function, 481
strnatcmp function, 98, 481

strncasecmp function, 98, 481
strncmp function, 98, 481
strpbrk function, 481
strpos function, 105, 480, 482
strptime function, 482
strrchr function, 105, 482
strrev function, 101, 482
strripos function, 482
strrpos function, 105, 482
strspn function, 105, 203, 483
strstr function, 105, 483
strtok function, 103, 483
strtolower function, 91, 483
strtotime function, 483
strtoupper function, 91, 484
strtr function, 93, 484
Structured Query Language (see SQL)
strval function, 484
str_getcsv function, 476
str_ireplace function, 477
str_pad function, 102, 477
str_repeat function, 102, 477
str_replace function, 346, 477
str_rot13 function, 477
str_shuffle function, 478
str_split function, 478
str_word_count function, 478
subclass, defined, 156
subpatterns, in regular expressions, 110, 114, 118
substr function, 100, 484
substrings, 100-101
substr_compare function, 484
substr_count function, 100, 485
substr_replace function, 101, 485
subtraction operator (–), 39
superclass, defined, 156
Suraski, Zeev (developer of PHP), 5
switch statement, 51-53
symbol table, 33
symlink function, 485
syslog function, 485
system function, 332, 486
sys_getloadavg function, 486
sys_get_temp_dir function, 486

## T

tables, in PDF files, 283-285
tabs (see whitespace)
tan function, 486
tanh function, 486
TCPDF library, 271
templating systems, 336-339
tempnam function, 486
ternary conditional operator (? :), 48, 51
text
    adding to images, 254-258
    in PDF files, 273-279
        color of, 277
        coordinates for, 273
        font attributes, 276-278
        headers and footers, 278-279
text representation of image, 268
ticks directive, 58
Tidwell, Doug (author)
    XSLT (O'Reilly), 308
tilde (~), bitwise negation operator, 43
time function, 486
time zone management, 181-185
times (see dates and times)
timezone_name_from_abbr function, 487
timezone_version_get function, 487
time_nanosleep function, 486
time_sleep_until function, 487
tmpfile function, 487
tokenizer
    for PHP code, 391
    for strings, 103
token_get_all function, 487
token_name function, 487
touch function, 487
track_errors option, php.ini file, 366
trailing options, regular expressions, 114
trait keyword, 165
traits, shared by classes, 165-168
trait_exists function, 488
transactions, database, 222
transformToDoc method, XSLT, 306
transformToUri method, XSLT, 306
transformToXml method, XSLT, 306
transparency of graphics, 249
trigger_error function, 488
trim function, 90, 488
true color indexes, 249, 267-268
true keyword, 26
    (see also Booleans)
TrueType fonts for graphics, 256
try…catch statement, 57, 222

type (instanceof) operator, 49
type hinting, functions, 74, 161
type juggling (see implicit casting)

## U

uasort function, 142, 488
ucfirst function, 91, 488
ucwords function, 91, 488
uksort function, 142, 489
umask function, 329, 489
unauthorized file access, preventing, 328-331
underscores, two (__), reserved for methods in PHP, 159
uniqid function, 489
Unix-based operating systems, 377-384
unlink function, 66, 489
unpack function, 489
unparsed entities, XML, 294, 295
unregister_tick_function function, 489
unserialize function, 177, 489
unset function, 34, 490
(unset) operator, 45
UPDATE command, SQL, 218, 221
update method, database, 242
upload_max_filesize option, php.ini file, 200
upload_tmp_dir option, php.ini file, 201
urldecode function, 95, 490
urlencode function, 95, 259, 490
URLs
  decomposing (parsing), 106
  encoding and decoding, 94-95
  functions for, 395
  in PDF files, 281
  redirecting, 205
  session IDs in, 212
use keyword, 78, 166
User-Agent header, 191
user-defined functions, 67-68
usleep function, 490
usort function, 77, 142, 490

## V

valid method, 151
value, passing parameters by, 71
variable functions, 76
variable parameters, 73-74
variable references, 30
variable variables, 30
variables, 30-34
  aliases for, 30
  case sensitivity of, 15
  containing function names, 76
  containing variable names, 30
  converting between arrays and, 134-135
  creating, 30
  creating arrays from, 135
  creating from array elements, 135
  functions for, 395
  garbage collection for, 33-34
  global, 69, 78
  for HTTP, 188
  interpolation of, 81
  names of, 20
  scope of, 31-33, 69-71, 78
  static, 33, 70
  symbol table for, 33
  testing if set, 34
  unsetting, 34
variables_order option, php.ini file, 364, 365
var_dump function, 88-89, 372, 490
var_export function, 491
version_compare function, 491
vertical bar (|)
  bitwise OR operator, 43
  in regular expressions, 109
vertical bar, equals sign (|=), bitwise-OR-equals operator, 48
vertical bars, double (||), logical OR operator, 45
vfprintf function, 491
vprintf function, 491
vsprintf function, 491

## W

__wakeup method, 177
web caching, 205, 347
Web Caching (O'Reilly), 206, 347
Web Database Applications with PHP and MySQL 2nd Edition (O'Reilly), 217
web forms (see forms)
web pages
  dynamic, server-side scripting for, 1
  embedding graphics in, 247-248
  embedding PHP in, 61-64
  expiration of, 205
  printing strings to, 85-89
  templating systems for, 336-339
web server

HTTP requests sent to, 187
HTTP response sent from, 188, 204-207
information from ($_SERVER), 189-191
maintaining state, 207-215
supported by PHP, 1
variables for, 188
web services, 351-360
RESTful, 351-356
SOAP, 357, 360
XML-RPC, 357-360
website resources, 7
ColorPic, 249
FPDF library, 271
Genghis (for MongoDB), 238
installation instructions, 7
MariaDB, 224
MongoDB, 238, 242, 245
MySQL versions, 223
Open Web Application Security Project, 319, 333
PDO library, 220
php.ini error constants, 364
php.ini runtime-changeable settings, 365
security, 333
Smarty templating system, 339
Squid Guard, 347
Squid proxy cache, 347
TCPDF library, 271
time zone names, 184
xmlrpc-epi distribution, 359
Zend Server CE, 238
Wessels, Duane (author)
Web Caching (O'Reilly), 206, 347
while statement, 53-55
whitespace
basics, 16
HTML entity for, 90
removing from strings, 90
Williams, Hugh (author)
Web Database Applications with PHP and MySQL, 2nd Edition (O'Reilly), 217
Windows operating system, 377-384
Wong, Clinton (author)
HTTP Pocket Reference (O'Reilly), 187
word boundaries, regular expressions, 113
wordwrap function, 491
write method, FPDF, 281
writeHTML method, TCPDF, 271
writeHTMLCell method, TCPDF, 271
WWW-Authenticate header, 206

## X

XML (Extensible Markup Language), 287-308
DTD for, 288
embedding PHP code in, 61-64
generating, 289
nesting XML tags, 288
parsing, 291-305
character data handler, 292
default handler, 295
DOM parser for, 304
element handlers, 291
entity handlers, 293-295
errors from, 298
event-based library for, 291-302
methods as handlers for, 299
options, 296
processing instructions, 293
sample application, 300-302
SimpleXML for, 305
using the parser, 297
PHP code in, 293
in remote procedure calls, 287
syntax rules for, 287-289
transforming with XSLT, 306-308
XML in a Nutshell (O'Reilly), 289
XML style of embedding PHP, 62-63
<?xml…?> tag, preceding XML document, 289
XML-RPC, 287, 357-360
xmlrpc extension, 357
xmlrpc-epi distribution, 359
xmlrpc_server_call_method function, 358
xmlrpc_server_create function, 358
xmlrpc_server_register_method function, 358
xml_error_string function, 299
xml_get_error_code function, 298
XML_OPTION_CASE_FOLDING option, 297
XML_OPTION_SKIP_TAGSTART option, 297
XML_Option_SKIP_WHITE option, 297
XML_OPTION_TARGET_ENCODING, 296
xml_parse function, 297
xml_parser_create function, 297
xml_parser_free function, 297
xml_parser_get_option function, 296
xml_parser_set_option function, 296
xml_set_character_data_handler function, 292
xml_set_default_handler function, 295
xml_set_element_handler function, 291

xml_set_external_entity_ref_handler function, 294
xml_set_notation_decl_handler function, 295
xml_set_object function, 299
xml_set_processing_instruction_handler function, 293
xml_set_unparsed_entity_decl_handler function, 295
XOR operator, bitwise (^), 44
XOR operator, logical (xor), 45
XOR-equals operator, bitwise (^=), 48
XSLT (Extensible Stylesheet Language Transformations), 306-308
XSLT (O'Reilly), 308
XSS (cross-site scripting), 322
xu_rpc_http_concise function, 360

## Z

Zend Server CE, 238, 377
Zend Studio for Eclipse, 374
zend_thread_id function, 491
zend_version function, 492
zlib, functions for, 395

## About the Authors

**Kevin Tatroe** has been an Apple Platforms and web stack engineer for almost 30 years, developing websites and mobile, desktop, and TV apps both small and enormous. He's attracted to technologies that allow for rapid iteration, experimentation, and highly opinionated architecture. Kevin, his wife Jenn, his son Hadden, and their two cats are recent transplants to Los Angeles, trading the quiet farmland of Colorado for the bustle of Hollywood. Their house remains filled with LEGO creations, board games, books, and numerous other distractions.

**Peter B. MacIntyre** has over 30 years of experience in the information technology industry, primarily in the area of PHP and web technologies. He has contributed writing material for many IT industry publications: Author of *PHP: The Good Parts* (O'Reilly), and coauthor of *Pro PHP Programming* (APress), *Using Visual Objects*, *Using PowerBuilder 5*, *ASP.NET Bible*, and *Zend Studio for Eclipse Developer's Guide*.

Peter is a cofounder and past cochair for the Northeast PHP Developer's Conference held in Boston, MA and Charlottetown, PE, Canada for 6 years. Peter has also spoken several times at North American and International computer conferences including PHPDay 2019 in Verona, Italy; PHPCE 2017 in Warsaw, Poland; PHP[World] 2016 in Washington, DC; ZendCon 2016 in Las Vegas; NortheastPHP 2017 & 2016 (Charlottetown, PE, Canada); Prairie Dev Con 2016 in Winnipeg, MB, Canada; CA-World in New Orleans, USA; CA-TechniCon in Cologne, Germany; and CA-Expo in Melbourne, Australia.

Peter lives in Prince Edward Island, Canada with his wonderful wife Dawn and their cat, Campbell. He is a Zend Certified Engineer in both PHP 5.3, PHP 4.0 and Nomad PHP Level 1 (PHP 7.0) certified.

## Colophon

The animal on the cover of *Programming PHP* is the great spotted cuckoo (*Clamator glandarius*). The great spotted cuckoo can be found throughout Africa and Southern Europe.

This cream-breasted brown bird has a grey "cap" on the top of its head, as well as distinctive white spots on its wings. Its tail feathers have white tips as well. Although there are regional variations in this bird's size, adult spotted cuckoos are generally larger than the common cuckoo. The calls of this bird are loud, harsh, and varied.

The great spotted cuckoo primarily eats insects. Their food of choice is hairy or spiny caterpillars, though they will also eat termites, grasshoppers, moths, and some small lizards. They hop along the ground with their tails raised in search of their food, occasionally making long, fluttering bounds in pursuit of fast prey.

A parasitic egg layer, the great spotted cuckoo will often lay its eggs in the nest of the pied crow and open and hole nesting starlings. The female great spotted cuckoo will search out a suitable host nest, often removing or damaging one of the host's eggs before laying its own. The male spotted cuckoo will sometimes distract the host bird while the female lays as many as 13 eggs in the host nest. The young spotted cuckoo, after hatching, will be cared for by the host bird for up to 18 days before leaving the nest.

While this bird's conservation status is currently classified as of Least Concern, 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 *Lydekker's Royal Natural History*. 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®

# There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

©2019 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175