Advanced PHP Programming
Advanced PHP Programming

A practical guide to developing large-scale Web sites and applications with PHP 5

George Schlossnagle

Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240 USA
Contents at a Glance

Introduction

I Implementation and Development Methodologies
1 Coding Styles
2 Object-Oriented Programming Through Design Patterns
3 Error Handling
4 Implementing with PHP: Templates and the Web
5 Implementing with PHP: Standalone Scripts
6 Unit Testing
7 Managing the Development Environment
8 Designing a Good API

II Caching
9 External Performance Tunings
10 Data Component Caching
11 Computational Reuse

III Distributed Applications
12 Interacting with Databases
13 User Authentication and Session Security
14 Session Handling
15 Building a Distributed Environment
16 RPC: Interacting with Remote Services
IV Performance

17 Application Benchmarks: Testing an Entire Application

18 Profiling

19 Synthetic Benchmarks: Evaluating Code Blocks and Functions

V Extensibility

20 PHP and Zend Engine Internals

21 Extending PHP: Part I

22 Extending PHP: Part II

23 Writing SAPIs and Extending the Zend Engine

Index
# Table of Contents

**Introduction** 1

I Implementation and Development Methodologies

1 Coding Styles 9
   Choosing a Style That Is Right for You 10
   Code Formatting and Layout 10
   Indentation 10
   Line Length 13
   Using Whitespace 13
   SQL Guidelines 14
   Control Flow Constructs 14

Naming Symbols 19
   Constants and Truly Global Variables 21
   Long-Lived Variables 22
   Temporary Variables 23
   Multiword Names 24
   Function Names 24
   Class Names 25
   Method Names 25
   Naming Consistency 25
   Matching Variable Names to Schema Names 26

Avoiding Confusing Code 27
   Avoiding Using Open Tags 27
   Avoiding Using echo to Construct HTML 27
   Using Parentheses Judiciously 28

Documentation 29
   Inline Comments 29
   API Documentation 30

Further Reading 35
2 Object-Oriented Programming Through Design Patterns 37

Introduction to OO Programming 38
Inheritance 40
Encapsulation 41
Static (or Class) Attributes and Methods 41
Special Methods 42
A Brief Introduction to Design Patterns 44
The Adaptor Pattern 44
The Template Pattern 49
Polymorphism 50
Interfaces and Type Hints 52
The Factory Pattern 54
The Singleton Pattern 56
Overloading 58
SPL 63
__call() 68
__autoloader() 70

Further Reading 71

3 Error Handling 73

Handling Errors 75
Displaying Errors 76
Logging Errors 77
Ignoring Errors 78
Acting On Errors 79
Handling External Errors 80
Exceptions 83
Using Exception Hierarchies 86
A Typed Exceptions Example 88
Cascading Exceptions 94
Handling Constructor Failure 97
Installing a Top-Level Exception Handler 98
Data Validation 100

When to Use Exceptions 104
Further Reading 105
4 Implementing with PHP: Templates and the Web 107
Smarty 108
   Installing Smarty 109
   Your First Smarty Template: Hello World! 110
   Compiled Templates Under the Hood 111
   Smarty Control Structures 111
   Smarty Functions and More 114
   Caching with Smarty 117
   Advanced Smarty Features 118
Writing Your Own Template Solution 120
Further Reading 121

5 Implementing with PHP: Standalone Scripts 123
Introduction to the PHP Command-Line Interface (CLI) 125
Handling Input/Output (I/O) 125
Parsing Command-Line Arguments 128
Creating and Managing Child Processes 130
   Closing Shared Resources 131
   Sharing Variables 132
   Cleaning Up After Children 132
   Signals 134
Writing Daemons 138
   Changing the Working Directory 140
   Giving Up Privileges 140
   Guaranteeing Exclusivity 141
Combining What You’ve Learned: Monitoring Services 141
Further Reading 150

6 Unit Testing 153
An Introduction to Unit Testing 154
   Writing Unit Tests for Automated Unit Testing 155
   Writing Your First Unit Test 155
   Adding Multiple Tests 156
7 Managing the Development Environment 183

Change Control 184
  CVS Basics 185
  Modifying Files 188
  Examining Differences Between Files 189
  Helping Multiple Developers Work on the Same Project 191
  Symbolic Tags 193
  Branches 194
  Maintaining Development and Production Environments 195

Managing Packaging 199
  Packaging and Pushing Code 201
  Packaging Binaries 203
  Packaging Apache 204
  Packaging PHP 205

Further Reading 206
8 Designing a Good API  207

Design for Refactoring and Extensibility  208
Encapsulating Logic in Functions  208
Keeping Classes and Functions Simple  210
Namespace  210
Reducing Coupling  212
Defensive Coding  213
Establishing Standard Conventions  214
Using Sanitization Techniques  214
Further Reading  216

II Caching

9 External Performance Tunings  219

Language-Level Tunings  219
Compiler Caches  219
Optimizers  222
HTTP Accelerators  223
Reverse Proxies  225
Operating System Tuning for High Performance  228
Proxy Caches  229
Cache-Friendly PHP Applications  231
Content Compression  235
Further Reading  236
RFCs  236
Compiler Caches  236
Proxy Caches  236
Content Compression  237

10 Data Component Caching  239

Caching Issues  239
Recognizing Cacheable Data Components  241
Choosing the Right Strategy: Hand-Made or Prefab Classes  241
Output Buffering  242
In-Memory Caching  244
Flat-File Caches 244
Cache Size Maintenance 244
Cache Concurrency and Coherency 245
DBM-Based Caching 251
Cache Concurrency and Coherency 253
Cache Invalidation and Management 253
Shared Memory Caching 257
Cookie-Based Caching 258
Cache Size Maintenance 263
Cache Concurrency and Coherency 263
Integrating Caching into Application Code 264
Caching Home Pages 266
Using Apache’s mod_rewrite for Smarter Caching 273
Caching Part of a Page 277
Implementing a Query Cache 280
Further Reading 281

11 Computational Reuse 283
Introduction by Example: Fibonacci Sequences 283
Caching Reused Data Inside a Request 289
Caching Reused Data Between Requests 292
Computational Reuse Inside PHP 295
PCREs 295
Array Counts and Lengths 296
Further Reading 296

III Distributed Applications

12 Interacting with Databases 299
Understanding How Databases and Queries Work 300
Query Introspection with EXPLAIN 303
Finding Queries to Profile 305
Database Access Patterns 306
Ad Hoc Queries 307
The Active Record Pattern 307
The Mapper Pattern 310
The Integrated Mapper Pattern 315
Tuning Database Access 317
Limiting the Result Set 317
Lazy Initialization 319
Further Reading 322

13 User Authentication and Session Security 323
Simple Authentication Schemes 324
HTTP Basic Authentication 325
Query String Munging 325
Cookies 326
Registering Users 327
Protecting Passwords 327
Protecting Passwords Against Social Engineering 330
Maintaining Authentication: Ensuring That You Are Still Talking to the Same Person 331
Checking That $_SERVER[REMOTE_IP] Stays the Same 331
Ensuring That $_SERVER['USER_AGENT'] Stays the Same 331
Using Unencrypted Cookies 332
Things You Should Do 332
A Sample Authentication Implementation 334
Single Signon 339
A Single Signon Implementation 341
Further Reading 346

14 Session Handling 349
Client-Side Sessions 350
Implementing Sessions via Cookies 351
Building a Slightly Better Mousetrap 353
Server-Side Sessions 354
Tracking the Session ID 356
A Brief Introduction to PHP Sessions 357
15 Building a Distributed Environment 367
What Is a Cluster? 367
Clustering Design Essentials 370
Planning to Fail 371
Working and Playing Well with Others 371
Distributing Content to Your Cluster 373
Scaling Horizontally 374
Specialized Clusters 375
Caching in a Distributed Environment 375
Centralized Caches 378
Fully Decentralized Caches Using Spread 380
Scaling Databases 384
Writing Applications to Use Master/Slave Setups 387
Alternatives to Replication 389
Alternatives to RDBMS Systems 390
Further Reading 391

16 RPC: Interacting with Remote Services 393
XML-RPC 394
Building a Server: Implementing the MetaWeblog API 396
Auto-Discovery of XML-RPC Services 401
SOAP 403
WSDL 405
Rewriting system.load as a SOAP Service 408
Amazon Web Services and Complex Types 410
Generating Proxy Code 412
SOAP and XML-RPC Compared 413
Further Reading 414
SOAP 414
XML-RPC 414
Web Logging 415
Publicly Available Web Services 415

IV Performance

17 Application Benchmarks: Testing an Entire Application 419
   Passive Identification of Bottlenecks 420
   Load Generators 422
      ab 422
      httpft 424
      Daiquiri 426
   Further Reading 427

18 Profiling 429
   What Is Needed in a PHP Profiler 430
   A Smorgasbord of Profilers 430
   Installing and Using APD 431
   A Tracing Example 433
   Profiling a Larger Application 435
   Spotting General Inefficiencies 440
   Removing Superfluous Functionality 442
   Further Reading 447

19 Synthetic Benchmarks: Evaluating Code Blocks and Functions 449
   Benchmarking Basics 450
   Building a Benchmarking Harness 451
      PEAR's Benchmarking Suite 451
   Building a Testing Harness 454
   Adding Data Randomization on Every Iteration 455
   Removing Harness Overhead 456
   Adding Custom Timer Information 458
   Writing Inline Benchmarks 462
V Extensibility

20 PHP and Zend Engine Internals  475
   How the Zend Engine Works: Opcodes and Op Arrays  476
   Variables  482
   Functions  486
   Classes  487
      The Object Handlers  489
      Object Creation  490
      Other Important Structures  490
   The PHP Request Life Cycle  492
      The SAPI Layer  494
      The PHP Core  496
      The PHP Extension API  497
      The Zend Extension API  498
      How All the Pieces Fit Together  500
   Further Reading  502

21 Extending PHP: Part I  503
   Extension Basics  504
      Creating an Extension Stub  504
      Building and Enabling Extensions  507
      Using Functions  508
      Managing Types and Memory  511
      Parsing Strings  514
      Manipulating Types  516
      Type Testing Conversions and Accessors  520
      Using Resources  524
      Returning Errors  529
      Using Module Hooks  529
For Pei, my number one.
About the Author

George Schlossnagle is a principal at OmniTI Computer Consulting, a Maryland-based tech company that specializes in high-volume Web and email systems. Before joining OmniTI, he led technical operations at several high-profile community Web sites, where he developed experience managing PHP in very large enterprise environments. He is a frequent contributor to the PHP community and his work can be found in the PHP core, as well as in the PEAR and PECL extension repositories.

Before entering the information technology field, George trained to be a mathematician and served a two-year stint as a teacher in the Peace Corps. His experience has taught him to value an interdisciplinary approach to problem solving that favors root-cause analysis of problems over simply addressing symptoms.

Acknowledgments

Writing this book has been an incredible learning experience for me, and I would like to thank all the people who made it possible. To all the PHP developers: Thank you for your hard work at making such a fine product. Without your constant efforts, this book would have had no subject.

To Shelley Johnston, Damon Jordan, Sheila Schroeder, Kitty Jarrett, and the rest of the Sams Publishing staff: Thank you for believing in both me and this book. Without you, this would all still just be an unrealized ambition floating around in my head.

To my tech editors, Brian France, Zak Greant, and Sterling Hughes: Thank you for the time and effort you spent reading and commenting on the chapter drafts. Without your efforts, I have no doubts this book would be both incomplete and chock full of errors.

To my brother Theo: Thank you for being a constant technical sounding board and source for inspiration as well as for picking up the slack at work while I worked on finishing this book.

To my parents: Thank you for raising me to be the person I am today, and specifically to my mother, Sherry, for graciously looking at every chapter of this book. I hope to make you both proud.

Most importantly, to my wife, Pei: Thank you for your unwavering support and for selflessly sacrificing a year of nights and weekends to this project. You have my undying gratitude for your love, patience, and support.
We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

You can email or write me directly to let me know what you did or didn’t like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book’s title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: opensource@samspublishing.com
Mail: Mark Taber
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

For more information about this book or others from Sams Publishing, visit our Web site at www.samspublishing.com. Type the ISBN (excluding hyphens) or the title of the book in the Search box to find the book you’re looking for.
Foreword

I have been working my way through the various William Gibson books lately and in *All Tomorrow’s Parties* came across this:

*That which is over-designed, too highly specific, anticipates outcome; the anticipation of outcome guarantees, if not failure, the absence of grace.*

Gibson rather elegantly summed up the failure of many projects of all sizes. Drawing multicolored boxes on whiteboards is fine, but this addiction to complexity that many people have can be a huge liability. When you design something, solve the problem at hand. Don’t try to anticipate what the problem might look like years from now with a large complex architecture, and if you are building a general-purpose tool for something, don’t get too specific by locking people into a single way to use your tool.

PHP itself is a balancing act between the specificity of solving the Web problem and avoiding the temptation to lock people into a specific paradigm for solving that problem. Few would call PHP graceful. As a scripting language it has plenty of battle scars from years of service on the front lines of the Web. What is graceful is the simplicity of the approach PHP takes.

Every developer goes through phases of how they approach problem solving. Initially the simple solution dominates because you are not yet advanced enough to understand the more complex principles required for anything else. As you learn more, the solutions you come up with get increasingly complex and the breadth of problems you can solve grows. At this point it is easy to get trapped in the routine of complexity.

Given enough time and resources every problem can be solved with just about any tool. The tool’s job is to not get in the way. PHP makes an effort to not get in your way. It doesn’t impose any particular programming paradigm, leaving you to pick your own, and it tries hard to minimize the number of layers between you and the problem you are trying to solve. This means that everything is in place for you to find the simple and graceful solution to a problem with PHP instead of getting lost in a sea of layers and interfaces diagrammed on whiteboards strewn across eight conference rooms.

Having all the tools in place to help you not build a monstrosity of course doesn’t guarantee that you won’t. This is where George and this book come in. George takes you on a journey through PHP which closely resembles his own journey not just with PHP, but with development and problem solving in general. In a couple of days of reading you get to learn what he has learned over his many years of working in the field. Not a bad deal, so stop reading this useless preface and turn to Chapter 1 and start your journey.

Rasmus Lerdorf
This page intentionally left blank
This book strives to make you an expert PHP programmer. Being an expert programmer does not mean being fully versed in the syntax and features of a language (although that helps); instead, it means that you can effectively use the language to solve problems. When you have finished reading this book, you should have a solid understanding of PHP’s strengths and weaknesses, as well as the best ways to use it to tackle problems both inside and outside the Web domain.

This book aims to be idea focused, describing general problems and using specific examples to illustrate—as opposed to a cookbook method, where both the problems and solutions are usually highly specific. As the proverb says: “Give a man a fish, he eats for a day. Teach him how to fish and he eats for a lifetime.” The goal is to give you the tools to solve any problem and the understanding to identify the right tool for the job.

In my opinion, it is easiest to learn by example, and this book is chock full of practical examples that implement all the ideas it discusses. Examples are not very useful without context, so all the code in this book is real code that accomplishes real tasks. You will not find examples in this book with class names such as foo and bar; where possible, examples have been taken from live open-source projects so that you can see ideas in real implementations.

PHP in the Enterprise

When I started programming PHP professionally in 1999, PHP was just starting its emergence as more than a niche scripting language for hobbyists. That was the time of PHP 4, and the first Zend Engine had made PHP faster and more stable. PHP deployment was also increasing exponentially, but it was still a hard sell to use PHP for large commercial Web sites. This difficulty originated mainly from two sources:

- Perl/ColdFusion/other-scripting-language developers who refused to update their understanding of PHP’s capabilities from when it was still a nascent language.
- Java developers who wanted large and complete frameworks, robust object-oriented support, static typing, and other “enterprise” features.

Neither of those arguments holds water any longer. PHP is no longer a glue-language used by small-time enthusiasts; it has become a powerful scripting language whose design makes it ideal for tackling problems in the Web domain.
A programming language needs to meet the following six criteria to be usable in business-critical applications:

- Fast prototyping and implementation
- Support for modern programming paradigms
- Scalability
- Performance
- Interoperability
- Extensibility

The first criterion—fast prototyping—has been a strength of PHP since its inception. A critical difference between Web development and shrink-wrapped software development is that in the Web there is almost no cost to shipping a product. In shipped software products, however, even a minor error means that you have burned thousands of CDs with buggy code. Fixing that error involves communicating with all the users that a bug fix exists and then getting them to download and apply the fix. In the Web, when you fix an error, as soon as a user reloads the page, his or her experience is fixed. This allows Web applications to be developed using a highly agile, release-often engineering methodology.

Scripting languages in general are great for agile products because they allow you to quickly develop and test new ideas without having to go through the whole compile, link, test, debug cycle. PHP is particularly good for this because it has such a low learning curve that it is easy to bring new developers on with minimal previous experience.

PHP 5 has fully embraced the rest of these ideas as well. As you will see in this book, PHP’s new object model provides robust and standard object-oriented support. PHP is fast and scalable, both through programming strategies you can apply in PHP and because it is simple to reimplement critical portions of business logic in low-level languages. PHP provides a vast number of extensions for interoperating with other services—from database servers to SOAP. Finally, PHP possesses the most critical hallmark of a language: It is easily extensible. If the language does not provide a feature or facility you need, you can add that support.

This Book’s Structure and Organization

This book is organized into five parts that more or less stand independently from one another. Although the book was designed so that an interested reader can easily skip ahead to a particular chapter, it is recommended that the book be read front to back because many examples are built incrementally throughout the book.

This book is structured in a natural progression—first discussing how to write good PHP, and then specific techniques, and then performance tuning, and finally language extension. This format is based on my belief that the most important responsibility of a professional programmer is to write maintainable code and that it is easier to make well-written code run fast than to improve poorly written code that runs fast already.
Part I, “Implementation and Development Methodologies”

Chapter 1, “Coding Styles”
Chapter 1 introduces the conventions used in the book by developing a coding style around them. The importance of writing consistent, well-documented code is discussed.

Chapter 2, “Object-Oriented Programming Through Design Patterns”
Chapter 2 details PHP 5’s object-oriented programming (OOP) features. The capabilities are showcased in the context of exploring a number of common design patterns. With a complete overview of both the new OOP features in PHP 5 and the ideas behind the OOP paradigm, this chapter is aimed at both OOP neophytes and experienced programmers.

Chapter 3, “Error Handling”
Encountering errors is a fact of life. Chapter 3 covers both procedural and OOP error-handling methods in PHP, focusing especially on PHP 5’s new exception-based error-handling capabilities.

Chapter 4, “Implementing with PHP: Templates and the Web”
Chapter 4 looks at template systems—toolsets that make bifurcating display and application easy. The benefits and drawbacks of complete template systems (Smarty is used as the example) and ad hoc template systems are compared.

Chapter 5, “Implementing with PHP: Standalone Scripts”
Very few Web applications these days have no back-end component. The ability to reuse existing PHP code to write batch jobs, shell scripts, and non-Web-processing routines is critical to making the language useful in an enterprise environment. Chapter 5 discusses the basics of writing standalone scripts and daemons in PHP.

Chapter 6, “Unit Testing”
Unit testing is a way of validating that your code does what you intend it to do. Chapter 6 looks at unit testing strategies and shows how to implement flexible unit testing suites with PHPUnit.

Chapter 7, “Managing the Development Environment”
Managing code is not the most exciting task for most developers, but it is nonetheless critical. Chapter 7 looks at managing code in large projects and contains a comprehensive introduction to using Concurrent Versioning System (CVS) to manage PHP projects.

Chapter 8, “Designing a Good API”
Chapter 8 provides guidelines on creating a code base that is manageable, flexible, and easy to merge with other projects.
Part II, “Caching”

Chapter 9, “External Performance Tunings”
Using caching strategies is easily the most effective way to increase the performance and scalability of an application. Chapter 9 probes caching strategies external to PHP and covers compiler and proxy caches.

Chapter 10, “Data Component Caching”
Chapter 10 discusses ways that you can incorporate caching strategies into PHP code itself. How and when to integrate caching into an application is discussed, and a fully functional caching system is developed, with multiple storage back ends.

Chapter 11, “Computational Reuse”
Chapter 11 covers making individual algorithms and processes more efficient by having them cache intermediate data. In this chapter, the general theory behind computational reuse is developed and is applied to practical examples.

Part III, “Distributed Applications”

Chapter 12, “Interacting with Databases”
Databases are a central component of almost every dynamic Web site. Chapter 12 focuses on effective strategies for bridging PHP and database systems.

Chapter 13, “User Authentication and Session Security”
Chapter 13 examines methods for managing user authentication and securing client/server communications. This chapter’s focuses include storing encrypted session information in cookies and the full implementation of a single signon system.

Chapter 14, “Session Handling”
Chapter 14 continues the discussion of user sessions by discussing the PHP session extension and writing custom session handlers.

Chapter 15, “Building a Distributed Environment”
Chapter 15 discusses how to build scalable applications that grow beyond a single machine. This chapter examines the details of building and managing a cluster of machines to efficiently and effectively manage caching and database systems.

Chapter 16, “RPC: Interacting with Remote Services”
Web services is a buzzword for services that allow for easy machine-to-machine communication over the Web. This chapter looks at the two most common Web services protocols: XML-RPC and SOAP.
Part IV, “Performance”

Chapter 17, “Application Benchmarks: Testing an Entire Application”
Application benchmarking is necessary to ensure that an application can stand up to the traffic it was designed to process and to identify components that are potential bottlenecks. Chapter 17 looks at various application benchmarking suites that allow you to measure the performance and stability of an application.

Chapter 18, “Profiling”
After you have used benchmarking techniques to identify large-scale potential bottlenecks in an application, you can use profiling tools to isolate specific problem areas in the code. Chapter 18 discusses the hows and whys of profiling and provides an in-depth tutorial for using the Advanced PHP Debugger (APD) profiler to inspect code.

Chapter 19, “Synthetic Benchmarks: Evaluating Code Blocks and Functions”
It’s impossible to compare two pieces of code if you can’t quantitatively measure their differences. Chapter 19 looks at benchmarking methodologies and walks through implementing and evaluating custom benchmarking suites.

Part V, “Extensibility”

Chapter 20, “PHP and Zend Engine Internals”
Knowing how PHP works “under the hood” helps you make intelligent design choices that target PHP’s strengths and avoid its weaknesses. Chapter 20 takes a technical look at how PHP works internally, how applications such as Web servers communicate with PHP, how scripts are parsed into intermediate code, and how script execution occurs in the Zend Engine.

Chapter 21, “Extending PHP: Part I”
Chapter 21 is a comprehensive introduction to writing PHP extensions in C. It covers porting existing PHP code to C and writing extensions to provide PHP access to third-party C libraries.

Chapter 22, “Extending PHP: Part II”
Chapter 22 continues the discussion from Chapter 21, looking at advanced topics such as creating classes in extension code and using streams and session facilities.

Chapter 23, “Writing SAPIs and Extending the Zend Engine”
Chapter 23 looks at embedding PHP in applications and extending the Zend Engine to alter the base behavior of the language.
Platforms and Versions

This book targets PHP 5, but with the exception of about 10% of the material (the new object-oriented features in Chapters 2 and 22 and the SOAP coverage in Chapter 16), nothing in this book is PHP 5 specific. This book is about ideas and strategies to make your code faster, smarter, and better designed. Hopefully you can apply at least 50% of this book to improving code written in any language.

Everything in this book was written and tested on Linux and should run without alteration on Solaris, OS X, FreeBSD, or any other Unix clone. Most of the scripts should run with minimal modifications in Windows, although some of the utilities used (notably the `pcntl` utilities covered in Chapter 5) may not be completely portable.
Errors are a fact of life. Mr. Murphy has an entire collection of laws detailing the prevalence and inescapability of errors. In programming, errors come in two basic flavors:

- **External errors**—These are errors in which the code takes an unanticipated path due to a part of the program not acting as anticipated. For example, a database connection failing to be established when the code requires it to be established successfully is an external error.

- **Code logic errors**—These errors, commonly referred to as bugs, are errors in which the code design is fundamentally flawed due to either faulty logic (“it just doesn’t work that way”) or something as simple as a typo.

These two categories of errors differ significantly in several ways:

- External errors will always occur, regardless of how “bug free” code is. They are not bugs in and of themselves because they are external to the program.

- External errors that aren’t accounted for in the code logic can be bugs. For example, blindly assuming that a database connection will always succeed is a bug because the application will almost certainly not respond correctly in that case.

- Code logic errors are much more difficult to track down than external errors because by definition their location is not known. You can implement data consistency checks to expose them, however.

PHP has built-in support for error handling, as well as a built-in severity system that allows you to see only errors that are serious enough to concern you. PHP has three severity levels of errors:

- `E_NOTICE`
- `E_WARNING`
- `E_ERROR`
E_NOTICE errors are minor, nonfatal errors designed to help you identify possible bugs in your code. In general, an E_NOTICE error is something that works but may not do what you intended. An example might be using a variable in a non-assignment expression before it has been assigned to, as in this case:

```php
<?php
    $variable++;
?>
```

This example will increment $variable to 1 (because variables are instantiated as 0/false/empty string), but it will generate an E_NOTICE error. Instead you should use this:

```php
<?php
    $variable = 0;
    $variable++;
?>
```

This check is designed to prevent errors due to typos in variable names. For example, this code block will work fine:

```php
<?php
    $variable = 0;
    $variabel++;
?>
```

However, $variable will not be incremented, and $variabel will be. E_NOTICE warnings help catch this sort of error; they are similar to running a Perl program with use warnings and use strict or compiling a C program with -Wall.

In PHP, E_NOTICE errors are turned off by default because they can produce rather large and repetitive logs. In my applications, I prefer to turn on E_NOTICE warnings in development to assist in code cleanup and then disable them on production machines.

E_WARNING errors are nonfatal runtime errors. They do not halt or change the control flow of the script, but they indicate that something bad happened. Many external errors generate E_WARNING errors. An example is getting an error on a call to fopen() to mysql_connect().

E_ERROR errors are unrecoverable errors that halt the execution of the running script. Examples include attempting to instantiate a non-existent class and failing a type hint in a function. (Ironically, passing the incorrect number of arguments to a function is only an E_WARNING error.)

PHP supplies the trigger_error() function, which allows a user to generate his or her own errors inside a script. There are three types of errors that can be triggered by the user, and they have identical semantics to the errors just discussed:

- E_USER_NOTICE
- E_USER_WARNING
- E_USER_ERROR
You can trigger these errors as follows:

```php
while(!feof($fp)) {
    $line = fgets($fp);
    if(!parse_line($line)) {
        trigger_error('Incomprehensible data encountered', E_USER_NOTICE);
    }
}
```

If no error level is specified, `E_USER_NOTICE` is used.

In addition to these errors, there are five other categories that are encountered somewhat less frequently:

- **E_PARSE**—The script has a syntactic error and could not be parsed. This is a fatal error.
- **E_COMPILE_ERROR**—A fatal error occurred in the engine while compiling the script.
- **E_COMPILE_WARNING**—A nonfatal error occurred in the engine while parsing the script.
- **E_CORE_ERROR**—A fatal runtime error occurred in the engine.
- **E_CORE_WARNING**—A nonfatal runtime error occurred in the engine.

In addition, PHP uses the `E_ALL` error category for all error reporting levels. You can control the level of errors that are percolated up to your script by using the `php.ini` setting `error_reporting`. `error_reporting` is a bit-field test set that uses defined constants, such as the following for all errors:

```
error_reporting = E_ALL
```

`error_reporting` uses the following for all errors except for `E_NOTICE`, which can be set by XOR'ing `E_ALL` and `E_NOTICE`:

```
error_reporting = E_ALL ~ E_NOTICE
```

Similarly, `error_reporting` uses the following for only fatal errors (bitwise OR of the two error types):

```
error_reporting = E_ERROR | E_USER_ERROR
```

Note that removing `E_ERROR` from the `error_reporting` level does not allow you to ignore fatal errors; it only prevents an error handler from being called for it.

**Handling Errors**

Now that you’ve seen what sort of errors PHP will generate, you need to develop a plan for dealing with them when they happen. PHP provides four choices for handling errors that fall within the `error_reporting` threshold:
- Display them.
- Log them.
- Ignore them.
- Act on them.

None of these options supersedes the others in importance or functionality; each has an important place in a robust error-handling system. Displaying errors is extremely beneficial in a development environment, and logging them is usually more appropriate in a production environment. Some errors can be safely ignored, and others demand reaction. The exact mix of error-handling techniques you employ depends on your personal needs.

**Displaying Errors**

When you opt to display errors, an error is sent to the standard output stream, which in the case of a Web page means that it is sent to the browser. You toggle this setting on and off via this `php.ini` setting:

```plaintext
display_errors = On
```

display_errors is very helpful for development because it enables you to get instant feedback on what went wrong with a script without having to tail a logfile or do anything but simply visit the Web page you are building.

What’s good for a developer to see, however, is often bad for an end user to see. Displaying PHP errors to an end user is usually undesirable for three reasons:

- It looks ugly.
- It conveys a sense that the site is buggy.
- It can disclose details of the script internals that a user might be able to use for nefarious purposes.

The third point cannot be emphasized enough. If you are looking to have security holes in your code found and exploited, there is no faster way than to run in production with `display_errors` on. I once saw a single incident where a bad INI file got pushed out for a couple errors on a particularly high-traffic site. As soon as it was noticed, the corrected file was copied out to the Web servers, and we all figured the damage was mainly to our pride. A year and a half later, we tracked down and caught a cracker who had been maliciously defacing other members’ pages. In return for our not trying to prosecute him, he agreed to disclose all the vulnerabilities he had found. In addition to the standard bag of JavaScript exploits (it was a site that allowed for a lot of user-developed content), there were a couple particularly clever application hacks that he had developed from perusing the code that had appeared on the Web for mere hours the year before.

We were lucky in that case: The main exploits he had were on unvalidated user input and nondefaulted variables (this was in the days before register_global). All our
database connection information was held in libraries and not on the pages. Many a site has been seriously violated due to a chain of security holes like these:

- Leaving display_errors on.
- Putting database connection details (mysql_connect()) in the pages.
- Allowing nonlocal connections to MySQL.

These three mistakes together put your database at the mercy of anyone who sees an error page on your site. You would (hopefully) be shocked at how often this occurs.

I like to leave display_errors on during development, but I never turn it on in production.

**Production Display of Errors**

How to notify users of errors is often a political issue. All the large clients I have worked for have had strict rules regarding what to do when a user incurs an error. Business rules have ranged from display of a customized or themed error page to complex logic regarding display of some sort of cached version of the content they were looking for. From a business perspective, this makes complete sense: Your Web presence is your link to your customers, and any bugs in it can color their perceptions of your whole business.

Regardless of the exact content that needs to be returned to a user in case of an unexpected error, the last thing I usually want to show them is a mess of debugging information. Depending on the amount of information in your error messages, that could be a considerable disclosure of information.

One of the most common techniques is to return a 500 error code from the page and set a custom error handler to take the user to a custom error page. A 500 error code in HTTP signifies an internal server error. To return one from PHP, you can send this:

```php
header("HTTP/1.0 500 Internal Server Error");
```

Then in your Apache configuration you can set this:

```apache
ErrorDocument 500 /custom-error.php
```

This will cause any page returning a status code of 500 to be redirected (internally—meaning transparently to the user) to /custom-error.php.

In the section “Installing a Top-Level Exception Handler,” later in this chapter, you will see an alternative, exception-based method for handling this.

**Logging Errors**

PHP internally supports both logging to a file and logging via syslog via two settings in the php.ini file. This setting sets errors to be logged:

```ini
log_errors = On
```
And these two settings set logging to go to a file or to syslog, respectively:

```php
error_log = /path/to/filename
```

```php
error_log = syslog
```

Logging provides an auditable trace of any errors that transpire on your site. When diagnosing a problem, I often place debugging lines around the area in question. In addition to the errors logged from system errors or via trigger_error(), you can manually generate an error log message with this:

```php
error_log("This is a user defined error");
```

Alternatively, you can send an email message or manually specify the file. See the PHP manual for details. `error_log` logs the passed message, regardless of the `error_reporting` level that is set; `error_log` and `error_reporting` are two completely different entries to the error logging facilities.

If you have only a single server, you should log directly to a file. `syslog` logging is quite slow, and if any amount of logging is generated on every script execution (which is probably a bad idea in any case), the logging overhead can be quite noticeable.

If you are running multiple servers, though, `syslog`'s centralized logging abilities provide a convenient way to consolidate logs in real-time from multiple machines in a single location for analysis and archival. You should avoid excessive logging if you plan on using `syslog`.

**Ignoring Errors**

PHP allows you to selectively suppress error reporting when you think it might occur with the @ syntax. Thus, if you want to open a file that may not exist and suppress any errors that arise, you can use this:

```php
$fp = @fopen($file, $mode);
```

Because (as we will discuss in just a minute) PHP’s error facilities do not provide any flow control capabilities, you might want to simply suppress errors that you know will occur but don’t care about.

Consider a function that gets the contents of a file that might not exist:

```php
$content = file_get_content($sometimes_valid);
```

If the file does not exist, you get an E_WARNING error. If you know that this is an expected possible outcome, you should suppress this warning; because it was expected, it’s not really an error. You do this by using the @ operator, which suppresses warnings on individual calls:

```php
$content = @file_get_content($sometimes_valid);
```
Handling Errors

In addition, if you set the `php.ini` setting `track_errors = On`, the last error message encountered will be stored in `$php_errormsg`. This is true regardless of whether you have used the `@` syntax for error suppression.

Acting On Errors

PHP allows for the setting of custom error handlers via the `set_error_handler()` function. To set a custom error handler, you define a function like this:

```php
<?php
require "DB/Mysql.inc";
function user_error_handler($severity, $msg, $filename, $linenum) {
    $dbh = new DB_Mysql_Prod;
    $query = "INSERT INTO errorlog
        (severity, message, filename, linenum, time)
        VALUES(?,?,?,?, NOW())";
    $sth = $dbh->prepare($query);
    switch($severity) {
    case E_USER_NOTICE:
        $sth->execute('NOTICE', $msg, $filename, $linenum);
        break;
    case E_USER_WARNING:
        $sth->execute('WARNING', $msg, $filename, $linenum);
        break;
    case E_USER_ERROR:
        $sth->execute('FATAL', $msg, $filename, $linenum);
        print 'FATAL error $msg at $filename:$linenum<br>';
        break;
    default:
        print 'Unknown error at $filename:$linenum<br>';
        break;
    }
}
?>
```

You set a function with this:

```php
set_error_handler("user_error_handler");
```

Now when an error is detected, instead of being displayed or printed to the error log, it will be inserted into a database table of errors and, if it is a fatal error, a message will be printed to the screen. Keep in mind that error handlers provide no flow control. In the case of a nonfatal error, when processing is complete, the script is resumed at the point where the error occurred; in the case of a fatal error, the script exits after the handler is done.
Mailing Oneself

It might seem like a good idea to set up a custom error handler that uses the \texttt{mail()} function to send an email to a developer or a systems administrator whenever an error occurs. In general, this is a very bad idea.

Errors have a way of clumping up together. It would be great if you could guarantee that the error would only be triggered at most once per hour (or any specified time period), but what happens more often is that when an unexpected error occurs due to a coding bug, many requests are affected by it. This means that your nifty mailing \texttt{error_handler()} function might send 20,000 mails to your account before you are able to get in and turn it off. Not a good thing.

If you need this sort of reactive functionality in your error-handling system, I recommend writing a script that parses your error logs and applies intelligent limiting to the number of mails it sends.

Handling External Errors

Although we have called what we have done so far in this chapter \textit{error handling}, we really haven't done much handling at all. We have accepted and processed the warning messages that our scripts have generated, but we have not been able to use those techniques to alter the flow control in our scripts, meaning that, for all intents and purposes, we have not really handled our errors at all. Adaptively handling errors largely involves being aware of where code can fail and deciding how to handle the case when it does.

External failures mainly involve connecting to or extracting data from external processes.

Consider the following function, which is designed to return the \texttt{passwd} file details (home directory, shell, gecos information, and so on) for a given user:

```php
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(";", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

As it stands, this code has two bugs in it: One is a pure code logic bug, and the second is a failure to account for a possible external error. When you run this example, you get an array with elements like this:

```php
<?php
print_r(get_passwd_info('www'));
?>
```
Array
{
    [0] => www:*:70:70:World Wide Web Server:/Library/WebServer:/noshell
}

This is because the first bug is that the field separator in the passwd file is ;, not ,. So this:

```
$fields = explode(";", $line);
```

needs to be this:

```
$fields = explode(":", $line);
```

The second bug is subtler. If you fail to open the passwd file, you will generate an E_WARNING error, but program flow will proceed unabated. If a user is not in the passwd file, the function returns false. However, if the fopen fails, the function also ends up returning false, which is rather confusing.

This simple example demonstrates one of the core difficulties of error handling in procedural languages (or at least languages without exceptions): How do you propagate an error up to the caller that is prepared to interpret it?

If you are utilizing the data locally, you can often make local decisions on how to handle the error. For example, you could change the password function to format an error on return:

```php
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return "Error opening file";
    }
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(":", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

Alternatively, you could set a special value that is not a normally valid return value:

```php
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return -1;
```
while(!feof($fp)) {
    $line = fgets($fp);
    $fields = explode(':', $line);
    if($user == $fields[0]) {
        return $fields;
    }
}
return false;
?>

You can use this sort of logic to bubble up errors to higher callers:

<?php
function is_shelled_user($user) {
    $passwd_info = get_passwd_info($user);
    if(is_array($passwd_info) && $passwd_info[7] != '/bin/false') {
        return 1;
    } else if($passwd_info == -1) {
        return -1;
    } else {
        return 0;
    }
}
?>

When this logic is used, you have to detect all the possible errors:

<?php
$v = is_shelled_user('www');
if($v === 1) {
    print "Your Web server user probably shouldn't be shelled.\n";
} else if($v === 0) {
    print "Great!\n";
} else {
    print "An error occurred checking the user\n";
}
?>

If this seems nasty and confusing, it's because it is. The hassle of manually bubbling up errors through multiple callers is one of the prime reasons for the implementation of exceptions in programming languages, and now in PHP5 you can use exceptions in PHP as well. You can somewhat make this particular example work, but what if the
function in question could validly return any number? How could you pass the error up in a clear fashion then? The worst part of the whole mess is that any convoluted error-handling scheme you devise is not localized to the functions that implement it but needs to be understood and handled by anyone in its call hierarchy as well.

**Exceptions**

The methods covered to this point are all that was available before PHP5, and you can see that this poses some critical problems, especially when you are writing larger applications. The primary flaw is in returning errors to a user of a library. Consider the error checking that you just implemented in the `passwd` file reading function.

When you were building that example, you had two basic choices on how to handle a connection error:

- Handle the error locally and return invalid data (such as `false`) back to the caller.
- Propagate and preserve the error and return it to the caller instead of returning the result set.

In the `passwd` file reading function example, you did not select the first option because it would have been presumptuous for a library to know how the application wants it to handle the error. For example, if you are writing a database-testing suite, you might want to propagate the error in high granularity back to the top-level caller; on the other hand, in a Web application, you might want to return the user to an error page.

The preceding example uses the second method, but it is not much better than the first option. The problem with it is that it takes a significant amount of foresight and planning to make sure errors can always be correctly propagated through an application. If the result of a database query is a string, for example, how do you differentiate between that and an error string?

Further, propagation needs to be done manually: At every step, the error must be manually bubbled up to the caller, recognized as an error, and either passed along or handled. You saw in the last section just how difficult it is to handle this.

Exceptions are designed to handle this sort of situation. An *exception* is a flow-control structure that allows you to stop the current path of execution of a script and unwind the stack to a prescribed point. The error that you experienced is represented by an object that is set as the exception.

Exceptions are objects. To help with basic exceptions, PHP has a built-in `Exception` class that is designed specifically for exceptions. Although it is not necessary for exceptions to be instances of the `Exception` class, there are some benefits of having any class that you want to throw exceptions derive from `Exception`, which we’ll discuss in a moment. To create a new exception, you instantiate an instance of the `Exception` class you want and you throw it.

When an exception is thrown, the `Exception` object is saved, and execution in the current block of code halts immediately. If there is an exception-handler block set in the
current scope, the code jumps to that location and executes the handler. If there is no
handler set in the current scope, the execution stack is popped, and the caller’s scope is
checked for an exception-handler block. This repeats until a handler is found or the
main, or top, scope is reached.

Running this code:

```php
<?php
    throw new Exception;
?>
```

returns the following:

```bash
> php uncaught-exception.php
Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

An uncaught exception is a fatal error. Thus, exceptions introduce their own mainte-
nance requirements. If exceptions are used as warnings or possibly nonfatal errors in a
script, every caller of that block of code must know that an exception may be thrown
and must be prepared to handle it.

Exception handling consists of a block of statements you want to try and a second
block that you want to enter if and when you trigger any errors there. Here is a simple
example that shows an exception being thrown and caught:

```php
try {
    throw new Exception;
    print "This code is unreached\n";
}
catch (Exception $e) {
    print "Exception caught\n";
}
```

In this case you throw an exception, but it is in a try block, so execution is halted and
you jump ahead to the catch block. catch catches an Exception class (which is the
class being thrown), so that block is entered. catch is normally used to perform any
cleanup that might be necessary from the failure that occurred.

I mentioned earlier that it is not necessary to throw an instance of the Exception
class. Here is an example that throws something other than an Exception class:

```php
<?php

class AltException {}

try {
    throw new AltException;
}
catch (Exception $e) {
```
print "Caught exception\n";
}
?>

Running this example returns the following:

> php failed_catch.php
Fatal error: Uncaught exception 'altexception'! in Unknown on line 0

This example failed to catch the exception because it threw an object of class AltException but was only looking to catch an object of class Exception.

Here is a less trivial example of how you might use a simple exception to facilitate error handling in your old favorite, the factorial function. The simple factorial function is valid only for natural numbers (integers > 0). You can incorporate this input checking into the application by throwing an exception if incorrect data is passed:

<?php
// factorial.inc
// A simple Factorial Function
function factorial($n) {
    if(!preg_match('/^\d+$/', $n) || $n < 0 ) {
        throw new Exception;
    } else if ($n == 0 || $n == 1) {
        return $n;
    } else {
        return $n * factorial($n - 1);
    }
}
?>

Incorporating sound input checking on functions is a key tenant of defensive programming.

Why the regex?
It might seem strange to choose to evaluate whether $n is an integer by using a regular expression instead of the is_int function. The is_int function, however, does not do what you want. It only evaluates whether $n has been typed as a string or as integer, not whether the value of $n is an integer. This is a nuance that will catch you if you use is_int to validate form data (among other things). We will explore dynamic typing in PHP in Chapter 20, "PHP and Zend Engine Internals."

When you call factorial, you need to make sure that you execute it in a try block if you do not want to risk having the application die if bad data is passed in:

<html>
<form method="POST">
Compute the factorial of
<input type="text" name="input" value="<?=$_POST['input'] ?>"><br>
<?php
include 'factorial.inc';
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$_POST[input]! = $output";
    }
    catch (Exception $e) {
        echo 'Only natural numbers can have their factorial computed.';
    }
}
?>
<br>
<input type=submit name=posted value="Submit">
</form>

Using Exception Hierarchies
You can have try use multiple catch blocks if you want to handle different errors differently. For example, we can modify the factorial example to also handle the case where $n is too large for PHP’s math facilities:

class OverflowException {}
class NaNException {}
function factorial($n)
{
    if(!preg_match('/^\d+$/', $n) || $n < 0 ) {
        throw new NaNException;
    }
    else if ($n == 0 || $n == 1) {
        return $n;
    }
    else if ($n > 170 ) {
        throw new OverflowException;
    }
    else {
        return $n * factorial($n - 1);
    }
}

Now you handle each error case differently:

<?php
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$input! = $output";
    }
    catch (Exception $e) {
        echo 'Only natural numbers can have their factorial computed.';
    }
}
?>
As it stands, you now have to enumerate each of the possible cases separately. This is both cumbersome to write and potentially dangerous because, as the libraries grow, the set of possible exceptions will grow as well, making it ever easier to accidentally omit one.

To handle this, you can group the exceptions together in families and create an inheritance tree to associate them:

```php
class MathException extends Exception {}
class NaNException extends MathException {}
class OverflowException extends MathException {}
```

You could now restructure the catch blocks as follows:

```php
if($_POST['input']) {  
    try {  
        $input = $_POST['input'];  
        $output = factorial($input);  
        print "$_POST[input]! = $output";  
    }  
    catch (OverflowException $e) {  
        print "The requested value is too large.";  
    }  
    catch (NaNException $e) {  
        print "Only natural numbers can have their factorial computed.";  
    }  
    }  
?>
```

In this case, if an OverflowException error is thrown, it will be caught by the first catch block. If any other descendant of MathException (for example, NaNException) is thrown, it will be caught by the second catch block. Finally, any descendant of Exception not covered by any of the previous cases will be caught.
This is the benefit of having all exceptions inherit from `Exception`: It is possible to write a generic `catch` block that will handle all exceptions without having to enumerate them individually. Catchall exception handlers are important because they allow you to recover from even the errors you didn’t anticipate.

### A Typed Exceptions Example

So far in this chapter, all the exceptions have been (to our knowledge, at least) attribute free. If you only need to identify the type of exception thrown and if you have been careful in setting up our hierarchy, this will satisfy most of your needs. Of course, if the only information you would ever be interested in passing up in an exception were strings, exceptions would have been implemented using strings instead of full objects. However, you would like to be able to include arbitrary information that might be useful to the caller that will catch the exception.

The base exception class itself is actually deeper than indicated thus far. It is a *built-in class*, meaning that it is implemented in C instead of PHP. It basically looks like this:

```php
class Exception {
    Public function __construct($message=false, $code=false) {
        $this->file = __FILE__;
        $this->line = __LINE__;
        $this->message = $message; // the error message as a string
        $this->code = $code;  // a place to stick a numeric error code
    }

    public function getFile() {
        return $this->file;
    }

    public function getLine() {
        return $this->line;
    }

    public function getMessage() {
        return $this->message;
    }

    public function getCode() {
        return $this->code;
    }
}
```

Tracking `__FILE__` and `__LINE__` for the last caller is often useless information. Imagine that you decide to throw an exception if you have a problem with a query in the `DB_Mysql` wrapper library:

```php
class DB_Mysql {
    // ...

    public function execute($query) {
        if(!$this->dbh) {
            $this->connect();
```
Exceptions

$ret = mysql_query($query, $this->dbh);
if(!is_resource($ret)) {
    throw new Exception;
}
return new MysqlStatement($ret);

Now if you trigger this exception in the code by executing a syntactically invalid query, like this:

```php
<?php
require_once "DB.inc";
try {
    $dbh = new DB_Mysql_Test;
    // ... execute a number of queries on our database connection
    $rows = $dbh->execute('SELECT * FROM')->fetchall_assoc();
} catch (Exception $e) {
    print_r($e);
}
?>
```

you get this:

```exception Object
{
    [file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
    [line] => 42
}
```

Line 42 of DB.inc is the execute() statement itself! If you executed a number of queries within the try block, you would have no insight yet into which one of them caused the error. It gets worse, though: If you use your own exception class and manually set $file and $line (or call parent::__construct to run Exception's constructor), you would actually end up with the first callers __FILE__ and __LINE__ being the constructor itself! What you want instead is a full backtrace from the moment the problem occurred.

You can now start to convert the DB wrapper libraries to use exceptions. In addition to populating the backtrace data, you can also make a best-effort attempt to set the message and code attributes with the MySQL error information:

class MysqlException extends Exception {
    public $backtrace;
    public function __construct($message=false, $code=false) {
        if(!$message) {
            $this->message = mysql_error();
        }
    }
}
Chapter 3  Error Handling

```php
if (!$code)
    $this->code = mysql_errno();
...
```

If you now change the library to use this exception type:

```php
class DB_Mysql
{
    public function execute($query) {
        if (!$this->dbh)
            $this->connect();
        $ret = mysql_query($query, $this->dbh);
        if (!is_resource($ret))
            throw new MysqlException;
        return new MysqlStatement($ret);
    }
}
```

and repeat the test:

```php
<?php
require_once "DB.inc";
try {
    $dbh = new DB_Mysql_Test;
    // ... execute a number of queries on our database connection
    $rows = $dbh->execute("SELECT * FROM...")->fetchall_assoc();
} catch (Exception $e) {
    print_r($e);
}
?>
```

you get this:

```php
mysqlexception Object
(
    [backtrace] => Array
    (0) => Array
        ([file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
         [line] => 45
         [function] => _ _construct
         [class] => mysqlexception
    )
)
Compared with the previous exception, this one contains a cornucopia of information:

- Where the error occurred
- How the application got to that point
- The MySQL details for the error

You can now convert the entire library to use this new exception:

class MysqlException extends Exception {
    public $backtrace;
    public function __construct($message=false, $code=false) {
        if(!$message) {
            $this->message = mysql_error();
        }
        if(!$code) {
            $this->code = mysql_errno();
        }
        $this->backtrace = debug_backtrace();
    }
}
class DB_Mysql {
    protected $user;
    protected $pass;
    protected $dbhost;
Chapter 3  Error Handling

protected $dbname;
protected $dbh;

public function __construct($user, $pass, $dbhost, $dbname) {
    $this->user = $user;
    $this->pass = $pass;
    $this->dbhost = $dbhost;
    $this->dbname = $dbname;
}

protected function connect() {
    $this->dbh = mysql_pconnect($this->dbhost, $this->user, $this->pass);
    if(!is_resource($this->dbh)) {
        throw new MysqlException;
    }
    if(!mysql_select_db($this->dbname, $this->dbh)) {
        throw new MysqlException;
    }
}

public function execute($query) {
    if(!$this->dbh) {
        $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
        throw new MysqlException;
    } else if(!is_resource($ret)) {
        return TRUE;
    } else {
        return new DB_MysqlStatement($ret);
    }
}

public function prepare($query) {
    if(!$this->dbh) {
        $this->connect();
    }
    return new DB_MysqlStatement($this->dbh, $query);
}

class DB_MysqlStatement {
    protected $result;
    protected $binds;
    public $query;
    protected $dbh;
public function __construct($dbh, $query) {
    $this->query = $query;
    $this->dbh = $dbh;
    if(!is_resource($dbh)) {
        throw new MysqliException("Not a valid database connection");
    }
}
public function bind_param($ph, $pv) {
    $this->binds[$ph] = $pv;
}
public function execute() {
    $binds = func_get_args();
    foreach($binds as $index => $name) {
        $this->binds[$index + 1] = $name;
    }
    $cnt = count($binds);
    $query = $this->query;
    foreach($this->binds as $ph => $pv) {
        $query = str_replace("$ph", $this->dbh); mysql_escape_string($pv));
    }
    $this->result = mysql_query($query, $this->dbh);
    if(!$this->result) {
        throw new MysqliException;
    }
}
public function fetch_row() {
    if(!$this->result) {
        throw new MysqliException("Query not executed");
    }
    return mysql_fetch_row($this->result);
}
public function fetch_assoc() {
    return mysql_fetch_assoc($this->result);
}
public function fetchall_assoc() {
    $retval = array();
    while($row = $this->fetch_assoc()) {
        $retval[] = $row;
    }
    return $retval;
}
Cascading Exceptions

Sometimes you might want to handle an error but still pass it along to further error handlers. You can do this by throwing a new exception in the catch block:

```php
<?php
try {
    throw new Exception;
}
catch (Exception $e) {
    print "Exception caught, and rethrown\n";
    throw new Exception;
}
?>
```

The catch block catches the exception, prints its message, and then throws a new exception. In the preceding example, there is no catch block to handle this new exception, so it goes uncaught. Observe what happens as you run the code:

```shell
> php re-throw.php
Exception caught, and rethrown
Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

In fact, creating a new exception is not necessary. If you want, you can rethrow the current Exception object, with identical results:

```php
<?php
try {
    throw new Exception;
}
catch (Exception $e) {
    print "Exception caught, and rethrown\n";
    throw $e;
}
?>
```

Being able to rethrow an exception is important because you might not be certain that you want to handle an exception when you catch it. For example, say you want to track referrals on your Web site. To do this, you have a table:

```sql
CREATE TABLE track_referrers (  
    url varchar2(128) not null primary key,  
    counter int  
);
```

The first time a URL is referred from, you need to execute this:

```sql
INSERT INTO track_referrers VALUES('http://some.url/', 1)
```
On subsequent requests, you need to execute this:

```sql
UPDATE track_referrers SET counter=counter+1 where url = 'http://some.url/
```

You could first select from the table to determine whether the URL's row exists and choose the appropriate query based on that. This logic contains a race condition though: If two referrals from the same URL are processed by two different processes simultaneously, it is possible for one of the inserts to fail.

A cleaner solution is to blindly perform the insert and call `update` if the insert failed and produced a unique key violation. You can then catch all `MysqlException` errors and perform the update where indicated:

```php
<?php
include "DB.inc";

function track_referrer($url) {
  $insertq = "INSERT INTO referrers (url, count) VALUES(:1, :2)";
  $updateq = "UPDATE referrers SET count=count+1 WHERE url = :1";
  $dbh = new DB_Mysql_Test;
  try {
    $sth = $dbh->prepare($insertq);
    $sth->execute($url, 1);
  } catch (MysqlException $e) {
    if($e->getCode == 1062) {
      $dbh->prepare($updateq)->execute($url);
    } else {
      throw $e;
    }
  }
}
?>
```

Alternatively, you can use a purely typed exception solution where `execute` itself throws different exceptions based on the errors it incurs:

```php
class Mysql_Dup_Val_On_Index extends MysqlException {} //...
class DB_Mysql {
  //...
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      if(mysql_errno() == 1062) {
```

throw new Mysql_Dup_Val_On_Index;
else {
    throw new MysqlException;
}

else if(!is_resource($ret)) {
    return TRUE;
} else {
    return new MysqlStatement($ret);
}
}

Then you can perform your checking, as follows:

function track_referrer($url) {
    $insertq = "INSERT INTO referrers (url, count) VALUES('$url', 1)";
    $updateq = "UPDATE referrers SET count=count+1 WHERE url = '$url'";
    $dbh = new DB_Mysql_Test;
    try {
        $sth = $dbh->execute($insertq);
    }
    catch (Mysql_Dup_Val_On_Index $e) {
        $dbh->execute($updateq);
    }
}

Both methods are valid; it's largely a matter of taste and style. If you go the path of typed
exceptions, you can gain some flexibility by using a factory pattern to generate your
errors, as in this example:

class MysqlException {
    // ...
    static function createError($message=false, $code=false) {
        if(!$code) {
            $code = mysql_errno();
        }
        if(!$message) {
            $message = mysql_error();
        }
        switch($code) {
        case 1062:
            return new Mysql_Dup_Val_On_Index($message, $code);
            break;
        default:
            return new MysqlException($message, $code);
            break;
        }
There is the additional benefit of increased readability. Instead of a cryptic constant being thrown, you get a suggestive class name. The value of readability aids should not be underestimated. Now instead of throwing specific errors in your code, you just call this:

```php
throw MysqliException::createError();
```

### Handling Constructor Failure

Handling constructor failure in an object is a difficult business. A class constructor in PHP **must** return an instance of that class, so the options are limited:

- You can use an initialized attribute in the object to mark it as correctly initialized.
- You can perform no initialization in the constructor.
- You can throw an exception in the constructor.

The first option is very inelegant, and we won’t even consider it seriously. The second option is a pretty common way of handling constructors that might fail. In fact, in PHP4, it is the preferable way of handling this.

To implement that, you would do something like this:

```php
class ResourceClass {
    protected $resource;
    public function __construct() {
        // set username, password, etc
    }
    public function init() {
        if (($this->resource = resource_connect()) == false) {
            return false;
        }
        return true;
    }
}
```

When the user creates a new `ResourceClass` object, there are no actions taken, which can mean the code fails. To actually initialize any sort of potentially faulty code, you call the `init()` method. This can fail without any issues.

The third option is usually the best available, and it is reinforced by the fact that it is the standard method of handling constructor failure in more traditional object-oriented languages such as C++. In C++ the cleanup done in a `catch` block around a constructor call is a little more important than in PHP because memory management might need to be performed. Fortunately, in PHP memory management is handled for you, as in this example:
class Stillborn {
    public function __construct() {
        throw new Exception;
    }
    public function __destruct() {
        print "destructing\n";
    }
}
try {
    $sb = new Stillborn;
} catch(Stillborn $e) {}

Running this generates no output at all:
>php stillborn.php
>
The Stillborn class demonstrates that the object’s destructors are not called if an exception is thrown inside the constructor. This is because the object does not really exist until the constructor is returned from.

Installing a Top-Level Exception Handler

An interesting feature in PHP is the ability to install a default exception handler that will be called if an exception reaches the top scope and still has not been caught. This handler is different from a normal catch block in that it is a single function that will handle any uncaught exception, regardless of type (including exceptions that do not inherit from Exception).

The default exception handler is particularly useful in Web applications, where you want to prevent a user from being returned an error or a partial page in the event of an uncaught exception. If you use PHP’s output buffering to delay sending content until the page is fully generated, you gracefully back out of any error and return the user to an appropriate page.

To set a default exception handler, you define a function that takes a single parameter:

function default_exception_handler($exception) {}

You set this function like so:

$old_handler = set_exception_handler('default_exception_handler');

The previously defined default exception handler (if one exists) is returned.

User-defined exception handlers are held in a stack, so you can restore the old handler either by pushing another copy of the old handler onto the stack, like this:

set_exception_handler($old_handler);

or by popping the stack with this:

restore_exception_handler();
An example of the flexibility this gives you has to do with setting up error redirects for
errors incurred for generation during a page. Instead of wrapping every questionable
statement in an individual try block, you can set up a default handler that handles the
redirection. Because an error can occur after partial output has been generated, you need
to make sure to set output buffering on in the script, either by calling this at the top of
each script:

```
ob_start();
```

or by setting the php.ini directive:

```
output_buffering = On
```

The advantage of the former is that it allows you to more easily toggle the behavior on
and off in individual scripts, and it allows for more portable code (in that the behavior is
dictated by the content of the script and does not require any nondefault .ini settings).
The advantage of the latter is that it allows for output buffering to be enabled in every
script via a single setting, and it does not require adding output buffering code to every
script. In general, if I am writing code that I know will be executed only in my local
environment, I prefer to go with .ini settings that make my life easier. If I am author-
ing a software product that people will be running on their own servers, I try to go with
a maximally portable solution. Usually it is pretty clear at the beginning of a project
which direction the project is destined to take.

The following is an example of a default exception handler that will automatically
generate an error page on any uncaught exception:

```php
<?php
function redirect_on_error($e) {
    ob_end_clean();
    include("error.html");
}
set_exception_handler("redirect_on_error");
ob_start();
// ... arbitrary page code goes here
?>
```

This handler relies on output buffering being on so that when an uncaught exception is
bubbled to the top calling scope, the handler can discard all content that has been gener-
ated up to this point and return an HTML error page instead.

You can further enhance this handler by adding the ability to handle certain error
conditions differently. For example, if you raise an AuthException exception, you can
redirect the person to the login page instead of displaying the error page:

```php
<?php
function redirect_on_error($e) {
    ob_end_clean();
    if(is_a($e, "AuthException")) {
        header("Location: /login.php");
    }
    ob_start();
    include("error.html");
set_exception_handler("redirect_on_error");
ob_start();
<?
```
Data Validation

A major source of bugs in Web programming is a lack of validation for client-provided data. *Data validation* involves verification that the data you receive from a client is in fact in the form you planned on receiving. Unvalidated data causes two major problems in code:

- Trash data
- Maliciously altered data

Trash data is information that simply does not match the specification of what it should be. Consider a user registration form where users can enter their geographic information. If a user can enter his or her state free form, then you have exposed yourself to getting states like

- New York ( typo)
- Lalalala (intentionally obscured)

A common tactic used to address this is to use drop-down option boxes to provide users a choice of state. This only solves half the problem, though: You’ve prevented people from accidentally entering an incorrect state, but it offers no protection from someone maliciously altering their `POST` data to pass in a non-existent option.

To protect against this, you should always validate user data in the script as well. You can do this by manually validating user input before doing anything with it:

```php
<?php
$STATES = array('al' => 'Alabama',
                'wy' => 'Wyoming');
function is_valid_state($state) {
    global $STATES;
    return array_key_exists($STATES, $state);
}
?>
```
I often like to add a validation method to classes to help encapsulate my efforts and ensure that I don’t miss validating any attributes. Here’s an example of this:

```php
class User {
    public id;
    public name;
    public city;
    public state;
    public zipcode;
    public function __construct($attr = false) {
        if($attr) {
            $this->name = $attr['name'];
            $this->email = $attr['email'];
            $this->city = $attr['city'];
            $this->state = $attr['state'];
            $this->zipcode = $attr['zipcode'];
        }
    }
    public function validate() {
        if(strlen($this->name) > 100) {
            throw new DataException;
        }
        if(strlen($this->city) > 100) {
            throw new DataException;
        }
        if(!is_valid_state($this->state)) {
            throw new DataException;
        }
        if(!is_valid_zipcode($this->zipcode)) {
            throw new DataException;
        }
    }
}
```

The `validate()` method fully validates all the attributes of the `User` object, including the following:

- Compliance with the lengths of database fields
- Handling foreign key data constraints (for example, the user’s U.S. state being valid)
- Handling data form constraints (for example, the zip code being valid)

To use the `validate()` method, you could simply instantiate a new `User` object with untrusted user data:
Chapter 3  Error Handling

```php
$user = new User($_POST);
and then call validate on it
try {
    $user->validate();
} catch (DataException $e) {
    /* Do whatever we should do if the users data is invalid */
}
```

Again, the benefit of using an exception here instead of simply having `validate()` return `true` or `false` is that you might not want to have a `try` block here at all; you might prefer to allow the exception to percolate up a few callers before you decide to handle it.

Malicious data goes well beyond passing in nonexistent state names, of course. The most famous category of bad data validation attacks are referred to as cross-site scripting attacks. Cross-site scripting attacks involve putting malicious HTML (usually client-side scripting tags such as JavaScript tags) in user-submitted forms.

The following case is a simple example. If you allow users of a site to list a link to their home page on the site and display it as follows:

```html
<a href='<?= $url ?>'>Click on my home page</a>
```

where `url` is arbitrary data that a user can submit, they could submit something like this:

```php
$url = 'http://example.foo/* onClick=bad_javascript_func foo=';
```

When the page is rendered, this results in the following being displayed to the user:

```html
<a href='http://example.foo/* onClick=bad_javascript_func foo='>
  Click on my home page
</a>
```

This will cause the user to execute `bad_javascript_func` when he or she clicks the link. What's more, because it is being served from your Web page, the JavaScript has full access to the user's cookies for your domain. This is, of course, really bad because it allows malicious users to manipulate, steal, or otherwise exploit other users' data.

Needless to say, proper data validation for any user data that is to be rendered on a Web page is essential to your site's security. The tags that you should filter are of course regulated by your business rules. I prefer to take a pretty draconian approach to this filtering, declining any text that even appears to be JavaScript. Here's an example:

```php
<?php
$UNSAFE_HTML[] = '!javascript\s*:\s*is';
$UNSAFE_HTML[] = '!vbscri?pt\s*:\s*is';
$UNSAFE_HTML[] = '<\s*embed.*swf\s*:\s*is';
$UNSAFE_HTML[] = '<[^>]*[^a-z]onabort\s*:\s*is';
```
Exceptions

```php
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onblur\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onchange\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onfocus\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onmouseout\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onmouseover\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onload\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onreset\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onselect\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onsubmit\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onunload\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onerror\s*=!is";
$UNSAFE_HTML[] = "!*[^>]*[^a-z]onclick\s*=!is";

function unsafe_html($html) {
    global $UNSAFE_HTML;
    $html = html_entities($html, ENT_COMPAT, ISO-8859-1_)
    foreach ( $UNSAFE_HTML as $match ) {
        if( preg_match($match, $html, $matches) ) {
            return $match;
        }
    }
    return false;
}

If you plan on allowing text to be directly integrated into tags (as in the preceding example), you might want to go so far as to ban any text that looks at all like client-side scripting tags, as in this example:

```php
$UNSAFE_HTML[] = "!*onabort\s*=!is";
$UNSAFE_HTML[] = "!*onblur\s*=!is";
$UNSAFE_HTML[] = "!*onchange\s*=!is";
$UNSAFE_HTML[] = "!*onfocus\s*=!is";
$UNSAFE_HTML[] = "!*onmouseover\s*=!is";
$UNSAFE_HTML[] = "!*onload\s*=!is";
$UNSAFE_HTML[] = "!*onreset\s*=!is";
$UNSAFE_HTML[] = "!*onselect\s*=!is";
$UNSAFE_HTML[] = "!*onsubmit\s*=!is";
$UNSAFE_HTML[] = "!*onunload\s*=!is";
$UNSAFE_HTML[] = "!*onerror\s*=!is";
$UNSAFE_HTML[] = "!*onclick\s*=!is";

It is often tempting to turn on magic_quotes_gpc in your php.ini file. magic_quotes automatically adds quotes to any incoming data. I do not care for magic_quotes. For one, it can be a crutch that makes you feel safe, although it is
simple to craft examples such as the preceding ones that are exploitable even with `magic_quotes` on.

With data validation (especially with data used for display purposes), there is often the option of performing filtering and conversion inbound (when the data is submitted) or outbound (when the data is displayed). In general, filtering data when it comes in is more efficient and safer. Inbound filtering needs to be performed only once, and you minimize the risk of forgetting to do it somewhere if the data is displayed in multiple places. The following are two reasons you might want to perform outbound filtering:

- You need highly customizable filters (for example, multilingual profanity filters).
- Your content filters change rapidly.

In the latter case, it is probably best to filter known malicious content on the way in and add a second filtering step on the way out.

Further Data Validation

Web page display is not the only place that unvalidated data can be exploited. Any and all data that is received from a user should be checked and cleaned before usage. In database queries, for instance, proper quoting of all data for insert should be performed. There are convenience functions to help perform these conversion operations.

A high-profile example of this are the so-called SQL injection attacks. A SQL injection attack works something like this: Suppose you have a query like this:

```perl
$query = "SELECT * FROM users where userid = $userid";
```

If `$userid` is passed in, unvalidated, from the end user, a malicious user could pass in this:

```perl
$userid = "10; DELETE FROM users;";
```

Because MySQL (like many other RDBMS systems) supports multiple queries inline, if this value is passed in unchecked, you will have lost your user's table. This is just one of a number of variations on this sort of attack. The moral of the story is that you should always validate any data in queries.

When to Use Exceptions

There are a number of views regarding when and how exceptions should be used. Some programmers feel that exceptions should represent fatal or should-be-potentially-fatal errors only. Other programmers use exceptions as basic components of logical flow control. The Python programming language is a good representative of this latter style: In Python exceptions are commonly used for basic flow control.

This is largely a matter of style, and I am inherently distrustful of any language that tries to mandate a specific style. In deciding where and when to use exceptions in your own code, you might reflect on this list of caveats:
- Exceptions are a flow-control syntax, just like `if{ }, else{ }, while{ }, and foreach{ }.
- Using exceptions for nonlocal flow control (for example, effectively long-jumping out of a block of code into another scope) results in non-intuitive code.
- Exceptions are bit slower than traditional flow-control syntaxes.
- Exceptions expose the possibility of leaking memory.

**Further Reading**


Because exceptions are rather new creatures in PHP, the best references regarding their use and best practices are probably Java and Python books. The syntax in PHP is very similar to that in Java and Python (although subtly different—especially from Python), but the basic ideas are the same.
This page intentionally left blank
This page intentionally left blank
Symbols

__autoload() function, 70-71
{} braces
  control flow constructs, 15-16
  function names, 24
__call() callback, 68-70
__destruct() class, 42
== (equal operator), 485
! parameter modifier, zend_parse_parameter() method, 515
/ parameter modifier, zend_parse_parameter() method, 515
| parameter modifier, zend_parse_parameter() method, 515
() parentheses, clarifying code, 28-29
$_SERVER[‘USER_AGENT’] setting, 331
$_SERVER[REMOTE_IP] setting, 331
_ (underscore)
  class names, 25
  function names, 24
  word breaks, 24

Numbers

404 errors, 276
500 error codes, 77

A

ab (ApacheBench) contrived load generator, 422-424
absolute pathnames, 158
abstract classes, 53-54
abstract stream data type, 571
abstraction layers, computational reuse between requests, 293
access
databases
  tuning, 317-322
  wrapper classes, 197
objects, Adapter patterns, 44-48
properties, overloading, 60
streams-compatible protocols, 568
access handlers, class objects, 490
access libraries, client-side sessions, 353-354
accessors
  functions, 22
  INI setting, 534
  zvals, 522-523
accounts, locking, 329
accumulator arrays, 287
activation, CGI SAPI, 584
Active Record pattern, 307-310
ad hoc, 245, 307
Adapter pattern, 44-48
addresses (email), unique identifiers, 327
addTestSuite() method, 161
add_assoc_zval() method, 517
Advanced PHP Debugger (APD) profiler
caching tables, 446-447
counters, 432
culling, 442-446
inefficiencies, 440-442
installing, 431-433
large applications, 435-440
trace files, 431-434
advisory file locks, 247-250
Ahmdahl’s Law, 471
algorithms
cryptography, 332
sorting algorithms, 286
speed, 285-286
allocated objects
classes, 490
destroying, 560
Amazon free SOAP interface Web site, 415
Amazon.com Web services API, 410-412
analyze method, 176
analyzers, lexical (lexers), 476
Apache
404 errors, 276
mod, mod_rewrite, 273-277
cache integration, 273-277
modules, 327
packaging, 204-205
Web site, 237
ApacheBench (ab) contrived load
generator, 422-424
APC
compiler cache, 220
Zend Engine, 606
APD (Advanced PHP Debugger) profiler
caching tables, 446-447
counters, 432
culling, 442-446
inefficiencies, 440-442
installing, 431-433
large applications, 435-440
trace files, 431-434
Zend Engine, 605-606
apd_execute() method, 605
apd_execute_internal() method, 605
apd_get_active_function_name() method, 606
APIs (application programming interfaces), 29-31
Amazon.com Web services, 410-412
Blogger Web site, 415
designs
bottom-up, 207
coupling, 212-213
data sanitization, 215-216
data validation, 216
defensive coding, 213-214
method logic, 208-209
namespaces, 210-212
security, 214-215
simple methods, 210
top-down, 208
hook structures, session handlers, 564
MetaWeblog
implementing, 396-400
Web site, 415
MovableType Web site, 415
phpDocumentor project, 31-35
PHP extension, 493, 497-498
streams, 579
C streams-compatible protocols,
accessing, 568
custom stream implementation, 570
I/O operations, 570
memory-mapped files, 571-578
storing, 570
Zend extension, 493, 498-500
application benchmarking, 450
bottlenecks, passively identifying,
420-422
load generators
contrived, 422-424
Daiquiri, 426-427
httperf, 424-426
realistic, 422
application layers, 492-496
application management
change control, CVS ( Concurrent
Versioning System), 184-188
binary files, 187-188
branches, 186-187, 194-195
development/production environments,
195-199
diffs, 189-191
file versions, 189-191
log messages, 186
modifying files, 188-189
repositories, 185-186
single tree limitations, 195
symbolic tags, 193-194, 199
updating files, 191-193
packaging, 199
Apache, 204-205
binaries, 203-204
pack(), 200-201
PHP, 205-206
pushing code, 201-203
application programming interfaces.
See APIs
application servers, database scaling, 390-391
applications
APD (Advanced PHP Debugger) profiler, 435-440
PHP lifecycle
PHP core, 493, 496
PHP extension API, 493, 497-498
SAPI (Server Abstraction API layer), 492-496
Zend extension API, 493, 498-500
Web, default exception handlers, 98
architecture, Web servers, 228
arguments
command-line, parsing, 128-130
input filters, 596
mmap_open() method, 577
types, functions, 483
write() method, 571
ZEND_FETCH_RESOURCE() macro, 528
arrays
accumulator, 287
associative
algorithm speed, 285
macros, 464
computational reuse, 296
creating, 516
data extraction, 519-520
indexed, 518-519
op (operations), Zend Engine, 476-482
persistent associative, creating, 61
types, adding, 516-517
zvals, adding, 517
associative arrays
algorithm speed, 285
macros, 464
attacks
cross-site scripting, 330
dictionary, 327-329
security, remote command injection, 214
social engineering, 330
attributes. See properties
authentication
dictionary attacks, 327-329
exceptions, 336
handlers, 327
implementing, 334-339
maintaining state
encrypted cookies, 332
expiration logic, 332-333
log outs, 333
$_SERVER[‘USER_AGENT’] setting, 331
$_SERVER[REMOTE_IP] setting, 331
unencrypted cookies, 332
user identity information, 333
versioning information, 333
passwords, storing, 339
schemes, 324
cookies, 326-327
HTTP Basic Authentication, 325
query string munging, 325-326
single signon, 339-340
implementing, 341-346
Microsoft Passport, 339
user registration
password protection, 327-330
unique identifiers, 327
Web unit testing, 179-182
AuthException exception, 336

How can we make this index more useful? Email us at indexes@samspublishing.com
AuthorRequest object, 412
automatic query dispatching, 387-389
avoiding confusing code
() parentheses, 28-29
echo, 27-28
open tags, 27

backups, bandwidth, 385
bandwidth, 384-385
Basic Authentication (HTTP), 325
Benchmark libraries, installing, 452
benchmarking
applications, 450
bottlenecks, passively identifying, 420-422
load generators, 422-427
inline, writing, 462
interpolation versus concatenation, 470-471
macro expansions, 464-470
overview, 450-451
strings, matching characters, 463-464
synthetic, 449-450
Whetstone, 450
benchmarking harness
custom timer information, adding, 458-461
features, 451
inline benchmarks, writing, 462
iterations, data randomization, 455-456
overhead, removing, 456-458
PEAR suite (Benchmark_Iterate), 451-454
testing harnesses, creating, 454-455
Benchmark_Iterate (PEAR benchmarking suite), 451-454
binaries, packaging, 203-204
binary data, strings, 296
binary files, CVS (Concurrent Versioning System), 187-188
bind SQL, 47

binding nodes, WSDL, 407
BitKeeper versioning system, 185
blocking network connections, 225
blocks
catch, 84, 94
try, 84
Blogger API Web site, 415
blogid() method, MetaWeblog API, 397
bottlenecks
ADP (Advanced PHP Debugger), culling, 442-446
database optimization, 300
network latency, 223-225
passively identifying, 420-422
bottom-up culling, 443
bottom-up designs, 207
braces {}
ccontrol flow constructs, 15-16
function names, 24
branches, CVS (Concurrent Versioning System), 186-187, 194-195
break loops, 18-19
BSD methodology, 257
BSD style, braces {}, 15
bubblesort sorting algorithm, 286
buffers, cache integration, 265-266. See also output buffering
bug reports, TDD (test-driven development), 177-179
build system macros, 507
built-in classes, 88
built-in functions, 452-453
buses (messaging), Spread toolkit, 380-384

C

C strings, 296
C++-style comments, 29
C-style comments, 29
Cache-Control HTTP header, 232
cache-friendly applications, external performance tuning, 231-235
Cache_File module, 379
caching, 375-376
centralized, 378-380
coherency, 240
compiler caches, 219-221, 236
computational reuse
arrays, 296
between requests, 292-295
Fibonacci Sequences, 283-289
inside requests, 289-292
PCREs (Perl Compatible Regular Expressions), 295
concurrency, 240
cookie-based
coherency, 263-264
concurrency, 263-264
personalized navigation bar, 258-263
scalability, 263
size maintenance, 263
user identities, 258-263
user profile information, 258-263
DBM-based, 251-252
concurrency, 253
expiration, 254-255
garbage collection, 257
invalidation, 253-254
keys, 257
loopkups, 255-256
maintenance, 257
storage, 255-256
updates, 253-254
decentralized, Spread toolkit, 380-384
features, 239-241
file locks, 247-259
coherency, file swaps, 250-251
concurrency, 245-251
flat-file caches, 244
size maintenance, 244-245
files, poisoning, 383
handlers, Smarty, 120
hierarchical, 240
HTTP caching, 229
in-memory, 244-251
integrated caching, 230
integration
file swaps, 264-265
home pages, 266-273
mod_rewrite, 273-277
output buffering, 265-266
partial pages, 277-280
query cache, 280-281
invalidation, 240
locality, 241
output buffering, 242-244
overview, 239
PEAR classes, 241-242
poisoning, 240
pre-fetching, 240
proxies, 229-230, 236-237
recognizable data components, 241
removal policies, 245
session data, 377
shared memory,
BSD methodology, 257
maintenance, 258
System V methodology, 257-258
size maintenance, 239
Smarty, 109, 117-118
stale, 240
caching logic, factory classes, 292
caching tables, APD (Advanced PHP Debugger), 446-447
calculations, algorithm speed, 285-286
callback methods, registering, 396
callbacks
__call, 68-70
statement handler, Zend Engine, 607
Zend Engine extension, 606-609
calling functions, 479-480
calling methods, speed, 210
camel caps, word breaks, 24
canonical pathnames, 159
capacity, clusters, 368
cascading exceptions, 94-97
catch block, 84, 94
catching exceptions, 84-85
CBC (Cypher Block Chaining) mode, 337
cdb libraries, 252
centralized cache, 378-380
CFB (Cypher Feedback) mode, 337
CGI SAPI
  activation/deactivation, 584
  cookies, 587
  data writing callback, 584
  deactivation code, 584
  environment variables lookup interface, 585
  flush helper method, 584
  header handlers, 586
  logging, 588
  main() routine, 588, 591
  method pointers, 583
  POST data, reading, 587
  SAPI name, 583
  sapi_module_struct structure, 582-584
  sending headers, 586
  server variables, 588
  shell environment variables, 588
  shutdown, 584
  stat() override, 585
  stream buffers, flushing, 585
  userspace error callback, 586
change control, CVS (Concurrent Versioning System), 184
  binary files, 187-188
  branches, 186-187, 194-195
  development/production environments, 195-199
  diffs, 189-191
  file versions, 189-191
  log messages, 186
  modifying files, 188-189
  repositories, 185-186
  single tree limitations, 195
  symbolic tags, 193-194, 199
  updating files, 191-193
characters (matching), strings, 463-464
check_credentials function, 329
child processes, 130
  creating, 131
  reaping, 132-134
  resources, sharing, 131
  signals
    SIGALRM, 134, 137-138
    SIGCHILD, 134-135, 137
    SIGHUP, 134, 138
    SIGINT, 134
    SIGKILL, 134
    SIGUSR1, 134
    SIGUSR2, 134
  variables, sharing, 132
chroot() method, 140
CISC (Complex Instruction Set Computer), 476
classes
  abstract, 53-54
  built-in, 88
  constructors, adding, 557-558
  creating, 550-551
  custom objects, 559-562
  DB_Result, 58-60
  __destruct(), 42
  documenting, 32
  Exception, 83, 558
  exceptions, throwing, 558-559
  factory, 292
  factory methods, 562
  implementing, 549-550
  inheritance, 554
  interfaces, defining/implementing, 562-564
  methods, adding, 555-557
  naming, 25
  Net_Telnet, 69
  PEAR, caching, 241-242
  PEAR XML-RPC, 395
  private properties, 554
  private variables, 559
  properties, adding, 551-553
ServiceCheckRunner, 144-146  
Spread_LOGGER, 547  
TestCase, 156-157  
Word, 169-177  
wrapper, database access, 197  
Zend Engine, 487  
   components, 488  
   global data structures, 490-492  
   object handlers, 489-490  
   object model, 488-489  
   objects, 490  
cleaning user-submitted data, 351  
CLI (command-line interface), scripts  
   arguments, 128-130  
   executing, 125  
   I/O, handling, 125-128  
CLI SAPI (Server Abstraction API layer), 494  
client-side caching, cookie-based  
   coherency, 263-264  
   concurrency, 263-264  
   personalized navigation bar, 258-263  
   scalability, 263  
   size maintenance, 263  
   user identities, 258-263  
   user profile information, 258-263  
client-side sessions, 349-350  
   access libraries, 353-354  
   benefits, 352-353  
   implementing via cookies, 351-353  
   limitations, 353  
   session data encryptions, 351-352  
   versus server-side sessions, 366  
clients  
   Spread, 382  
   XML-RPC, 395  
clone() method, 560  
close() method, streams, 570  
clusters  
   cache, 375-377  
      centralized, 378-380  
      decentralized, Spread toolkit, 380-384  
   capacity, 368  
content distribution, 373-374  
database scaling, 386  
   application servers, 390-391  
   partitioning, 384, 389-390  
   RDBMS systems, 390  
   replication, 385-389  
design, 370-373  
   overview, 367-370  
   redundancy, 367  
   scaling, 368-369, 374  
   specialized, 374-375  
code  
   authentication implementation, 334-339  
   confusing, avoiding, 27-29  
   coupling, 212-213  
   defensive coding, 213-216  
   fast, include files, 212  
   intermediate, 220, 476-479  
   method logic, 208-209  
   modular, include files, 212  
   namespaces, 210, 212  
   production, pushing, 412-413  
   simple methods, 210  
   testing, 153-154  
code logic errors, 73  
code optimizers, 223  
coding styles  
   choosing, 10  
   confusing code, avoiding  
      () parentheses, 28-29  
      echo, 27-28  
      open tags, 27  
documentation  
   API (application programming interface), 29-35  
   classes, 32  
   functions, 32  
   inline comments, 29-30  
design, 370-373  
documentation  
   API (application programming interface), 29-35  
   classes, 32  
   functions, 32  
   inline comments, 29-30  
format/layout  
   control flow constructs, 14-19  
   indentation, 10-12  
   line length, 13
coding styles

SQL, 14
whitespace, 13
naming symbols, 19-20
class names, 25
consistency, 25
constants, 21-22
function names, 24
long-lived variables, 22
method names, 25
multiword variable names, 24
temporary variables, 23
truly global variables, 21-22
variable names, matching, 26-27

coherency
cookie-based caching, 263-264
DBM-based caching, 253
in-memory caching
file locks, 247-250
file swaps, 250-251

command-line arguments, parsing, 128-130
command-line interface. See CLI

comments
inline, 29-30
magic, 12

commodity hardware, 371
common Log Format Apache logs, 426
compiled templates, 111
compiler caches, 219-221, 236
compilerGlobals struct, 490-492
Complex Instruction Set Computer (CISC), 476
components (data), recognizing for cache, 241
compression, external performance tuning, 235-237
computational reuse
arrays, 296
between requests, 292-295
Fibonacci Sequences, 283-289
inside requests, 289-292
PCREs (Perl Compatible Regular Expressions), 295

concatenation, 470-471
concurrency
loading, 240
cookie-based caching, 263-264
DBM-based caching, 253
home pages, caching, 272
in-memory caching, 245-246
file locks, 247-250
file swaps, 250-251

Concurrent Versioning System (CVS), 184
binary files, 187-188
branches, 186-187, 194-195
development/production environment, 195-199
diffs, 189-191
file versions, differentiating, 189-191
files
 modifying, 188-189
 updating, 191-193
log messages, 186
pushing production code, 203
repositories, creating, 185-186
single tree limitations, 195
symbolic tags, 193-194, 199

conditionals, 14-16
conditions, adding unit testing, 164-165
config.m4 files, 506
configuration files, monitoring engine script, 148-149

confusing code, avoiding
() parentheses, 28-29
echo, 27-28
open tags, 27

connect() method, Spread client wrapper, 539-541

connections
networks
 blocking, 225
FIN packets, 229
nonpersistent, 539
persistent, 539
Spread client wrapper, 539-541
consecutive_failures parameter, ServiceCheck object, 143
Console_Getopt package (PEAR), 128-129
constant-folding, optimizers, 222
constants, 21-22, 530-531
constructors, 38-39
adding to classes, 557-558
failing, exceptions, 97-98
constructs, control flow
braces {}, 15-16
conditionals, 14
loops, 14
break, 18-19
continue, 18-19
controlling flow, 18-19
deeply nested, 19
for, 16-18
foreach, 16-18
while, 16-18
content compression, external performance tuning, 235-237
content distribution, clusters, 373-374
continue loops, 18-19
ccontrived load generators, 422-424
control flow constructs
braces {}, 15-16
conditionals, 14
loops, 14
break, 18-19
continue, 18-19
controlling flow, 18-19
deeply nested, 19
for, 16-18
foreach, 16-18
while, 16-18
conversion, 104
cookie-based caching
coherency, 263-264
concurrency, 263-264
personalized navigation bar, 258-263
scalability, 263
size maintenance, 263
user identities, 258-263
user profile information, 258-263
cookies, 326-327
APD (Advanced PHP Debugger) profiler, inefficiencies, 440-442
CGI SAPI, 587
client-side sessions, implementing, 351-353
encrypted, 332
JavaScript, 330
session IDs, tracking, 356-357
unencrypted, 332
count() function, 296
counters, 432
counts, arrays, 296
coupling, 212-213
create_object() method, 560
create_table() function, 116
CREDITS file, 507
cross-site scripting, 102, 330
culling, APD (Advanced PHP Debugger), 442-446
curl extension, 179
current_status parameter, ServiceCheck object, 143
custom error handlers, 79-80
custom objects, creating, 559-562
custom timer information, adding, 458-461
CVS (Concurrent Versioning System), 184
binary files, 187-188
branches, 186-187, 194-195
development/production environment, 195-199
diffs, 189-191
file versions, differentiating, 189-191
files
modifying, 188-189
updating, 191-193
log messages, 186
pushing production code, 203
repositories, creating, 185-186
single tree limitations, 195
symbolic tags, 193-194, 199

How can we make this index more useful? Email us at indexes@samspublishing.com
Cypher Block Chaining (CBC) mode, 337
Cypher Feedback (CFB) mode, 337

D

daemons
  exclusivity, 141
  privileges, 140-141
  working directories, 140
  writing, 138-139
Daiquiri load generator, 426-427
data
  binary, strings, 296
  computational reuse
    arrays, 296
    between requests, 292-295
    Fibonacci Sequences, 283-289
    inside requests, 289-292
    PCREs (Perl Compatible Regular Expressions), 295
  displaying, Smarty, 112
  maliciously altered, cross-site scripting attacks, 102
  trash data, 100-102
data components, recognizing for cache, 241
data extraction, arrays, 519-520
data randomization, iterations, 455-456
data sanitization, 215-216
data structures, global, classes, 490-492
data types, union, 484
data validation, 103, 216
  maliciously altered, 102
  SQL injection attacks, 104
  trash data, 100-102
database access patterns, 306
  Active Record pattern, 307-310
  ad hoc queries, 307
  Integrated Mapper pattern, 315-317
  Mapper pattern, 310-315
database management system (DBMS), 299
database objects, creating via factory methods, 55
databases
  accessing, wrapper classes, 197
  defined, 299
  introspection, EXPLAIN SQL syntax, 303-304
  multiple, development environments, 197-198
  profiles, 300-302, 305-306
  queries, bandwidth, 384
  RDBMSs (relational database management systems), 299
    database access patterns, 306-317
    indexes, 300-302
    queries, 300-303
  scaling
    application servers, 390-391
    partitioning, 384, 389-390
    RDBMS systems, 390
    replication, 385-389
  terminology, 299
tuning
  lazy initialization, 319-322
  limiting result sets, 317-319
Dave Winer XML-RPC Web site, 414
DBG profiler, 431
DBM, libraries or licenses, 252
DBM-based caching, 251-252
  concurrency, 253
  expiration, 254-255
  garbage collection, 257
  invalidation, 253-254
  keys, 257
  lookups, 255-256
  maintenance, 257
  storage, 255-256
  updates, 253-254
DBMS (database management system), 299
DB_Result class, 58-60
deactivation, CGI SAPI, 584
dead code elimination, optimizers, 222
debugging, ADP (Advanced PHP Debugger)
caching tables, 446-447
counters, 432
culling, 442-446
inefficiencies, 440-442
installing, 431-433
large applications, 435-440
trace files, 431-434
decentralized cache, Spread toolkit, 380-384
declaring methods, 509
deepl y nested loops, avoiding, 19
default exception handlers, installing, 98-100
defensive coding, 213
data sanitization, 215-216
data validation, 216
security, 214-215
standard conventions, 214
defining
constants, module initialization, 530-531
interfaces, 562-564
wrappers (streams API), 576
delegation, OOP (object-oriented programming), 50-52
description parameter, ServiceCheck object, 143
design patterns
Adapter pattern, 44-48
Factory pattern, 54-55
interfaces, 52-54
polymorphism, 50-52
Singleton pattern, 56-57
Template pattern, 49
type hinting, 52-54
designing clusters, 370
cohabitation, 371-373
commodity hardware, 371
designs
bottom-up, 207
defensive coding, 213
data sanitization, 215-216
data validation, 216
security, 214-215
standard conventions, 214
refactoring
coupling, 212-213
method logic, 208-209
namespaces, 210-212
simple methods, 210
top-down, 208
destroying
allocated objects, 560
session handlers, 567
destructing objects, 42-43
destructors, creating, 560
development environments
maintaining, CVS (Concurrent Versioning System), 195-199
multiple databases, 197-198
dictionary attacks, 327-329
diffs, 189-191
directives
ErrorDocument, 276
max-age, 232
must-revalidate, 232
no-cache, 232
private, 232
proxy-revalidate, 232
public, 232
s-maxage, 233
directories, 140, 246
disconnecting Spread client wrapper, 541-542
displaying data, Smarty, 112
displaying errors (error handling), 76-77
distributing content, clusters, 373-374
documentation
   API (application programming interface),
      29-35
   classes, 32
   functions, 32
   inline comments, 29-30
dynamic extensions, creating, 508
dynamic instance properties, 551
dynamically typed languages, 482-483

ingo, 27-28
efree() method, 512
email addresses, unique identifiers, 327
emalloc() method, 511-512
embed SAPI (Server Abstraction API layer), 494, 591-593
embedding HTML, 27
encapsulation
   OOP (object-oriented programming),
      39-41
   PPP (public, protected, private), 41
encrypted cookies, 332
encryption
   algorithms, 332
   session data, client-side sessions, 351-352
enterprise, 183
environment variables
   looking up, 585
   printing, 113
   shell, CGI SAPI, 588
equal operator (==), 485
erealloc() method, 512
error handling
   code logic errors, 73
   custom error handlers, setting, 79-80
   displaying errors, 76-77
   exceptions, 82
      cascading, 94-97
      catching, 84-85
      constructor failure, 97-98
      creating, 83
data validation, 100-104
default exception handlers, installing,
      98-100
   Exception class, 83
   hierarchies, 86-88
   Python programming, 104
   rethrowing, 94
   throwing, 83-85
   typed example, 88-93
   when to use, 104-105
   external, 80-83
   external errors, 73
   E_ERROR errors, 74
   E_NOTICE errors, 74-75
   E_USER_NOTICE errors, 75
   E_WARNING errors, 74
   ignoring errors, 78-79
   logging errors, 77-78
   severity levels, 73
error messages, informative (unit testing),
    163-164
ErrorDocument directive, 276
errors
   404, 276
   500 error code, 77
   handling, extensions, 529
   runtime, detecting, 52
error_reporting, 75
estrndup() method, 512
eval() function, 468
event-based architecture, Web servers, 228
example.php script, 507
Exception class, 83, 558
exceptions, 82
   authentication, 336
   AuthException, 336
   cascading, 94-97
   catching, 84-85
   constructor failure, 97-98
   creating, 83
   default exception handlers, installing,
      98-100
Exception class, 83
hierarchies, 86-88
Python programming, 104
rethrowing, 94
throwing, 83-85, 558-559
typed example, 88-93
validation
  maliciously altered data, 102
  SQL injection attacks, 104
  trash data, 100-102
warnings as (Zend Engine), 599-601
when to use, 104-105
exclusivity, daemons, 141
executor_globals struct, 490-492
expansions, macros (benchmarking),
  464-470
EXPERIMENTAL file, 507
expiration, DBM-based caching, 254-255
expiration logic, 332-333
Expires HTTP header, 231
EXPLAIN SQL syntax, 303-304
explode method, 176
extensions
  config.m4 file, 506
curl, 179
dynamic, creating, 508
errors, 529
files, mcrypt, 332
hex-encoding strings example, 511-512
hooks, 497
master C source file example, 505
memory management, 511-513
methods, Fibonacci Sequence example,
  508-510
module hooks, 529
  module shutdown, 535
module startup/shutdown, 530-535
phpinfo() registration, 536-537
request shutdown, 536
request startup, 535
request startup/shutdown, 535
mysqli, 387-388
PHP extension API, 493, 497-498
registering, 497
resources
  creating, 524
  finding, 526-528
  handling, 524
  nonpersistent, 524
  persistent, 524
  POSIX file handles as, 524
  registering, 525-526
socket, 390
Spread client wrapper example
  connecting, 539-541
  disconnecting, 541-542
  groups, joining, 542-543
  method registration, 546
  module initialization, 538
  module registration, 546
  module shutdown, 539
  receiving messages, 543-545
  sending messages, 545-546
  Spread library, 537
  Spread_LOGGER class, 547
Spread client wrapper Web site, 548
static, creating, 507
strings, parsing
  format characters, 514
  parameter modifiers, 514-515
  return macros, 515
stubs, creating, 504, 507
Zend Engine callbacks, 606-609
Zend extension API, 493, 498-500
zvals
  accessors, 522-523
  arrays. See arrays
  assignments, 516
  creating, 516
  hashtables, 519-520
  macros, 516
  separation, 522
  type conversions, 521-522
  variables, copying, 523
external errors, 73, 80-83
external performance tunings
  cache-friendly applications, HTTP headers, 231-235
  content compression, 235-237
language-level tunings
  compiler caches, 219-221, 236
  HTTP accelerators, 223-225
  operating systems, 228-229
  optimizers, 222-223
  proxy caches, 229-230, 236-237
  reverse proxies, 225-228
Extreme Programming, unit testing, 154
E_ERROR errors, 74
E_NOTICE errors, 74-75
E_USER_NOTICE errors, 75
E_WARNING errors, 74

factory classes, 292
factory methods, 562
  database objects, creating, 55
  singletons, creating, 56-57
Factory pattern, 54-55
failover solutions, clusters, 373-374
failure_time parameter, ServiceCheck object, 143
fast code, include files, 212
fastcgi SAPI (Server Abstraction API layer), 494
Fibonacci Sequences, 283-289
fibonacci() method, 509
FIFO (first in, first out), 245
file extensions, mcrypt, 332
file handles, 125-127
file systems, 245, 385
files
  Amazon WSDL, 410
  binary, CVS (Concurrent Versioning System), 187-188
  cache, poisoning, 383
  config.m4, 506
  CREDITS, 507
  EXPERIMENTAL, 507
  include, modular versus fast code, 212
  individual, pushing, 199-200
  locking, 247-250
  master C source, 505
  modifying, CVS (Concurrent Versioning System), 188-189
  network shares, centralized cache, 378
  PHP, moving, 201-202
  php_example.h, 507
  sharing, networks, centralized cache, 378
  swapping, 250-251, 264-265
  trace, APD (Advanced PHP Debugger) profiler, 431-434
  updating, CVS (Concurrent Versioning System), 191-193
  versions, CVS (Concurrent Versioning System), 189-191
files session handler, 361-366
filtering, 104
  output, Smarty, 119
  postfilters, Smarty, 119
  prefilters, Smarty, 119
  SAPI input, 593
    input_filter, 594-598
    post_reader, 598
    treat_data, 598
FINAL packets, 229
first in, first out (FIFO), 245
flags, is_ref, 484
flat-file caches, 244
Flesh score calculator, 169
flock() function, 248
flow
  control (Smarty), 111-114
  loops, controlling, 18-19
flush() method, streams, 570
focused tuning, 471
for loops, 16-18
foreach loops, 16-18
format characters, strings, 514
formats
  coding styles
    control flow constructs, 10-19
    indentation, 10-12
    magic comments, 12
    tabs, 11-14

formatting
  coding styles, indentation, 10-12
  tabs, 12

frequency parameter, ServiceCheck object, 143

full descriptive names, clusters, 373

function calls, 479-480

function pointers, Zend Engine, 498-500

function-based indexes, 301

functions
  accessor, 22
  APD (Advanced PHP Debugger) profiler,
    inefficiencies, 441
  argument types, 483
  __autoload, 70-71
  built-in, 452-453
  check_credentials, 329
  documenting, 32
  flock(), 248
  invariant, loops, 440
  iterations, 455-456
  login(), 69
  mail(), 80
  namespacing, clusters, 372-373
  naming, 20, 24
  PCREs (Perl Compatible Regular
    Expressions), 295
  recursive, 283-289
  rename(), 251
  set_error_handler, 79
  shmop, 258
  trigger_error(), 74
  userspace, 452-453
  Zend Engine, 486-487

garbage collection
  DBM-based caching, 257
  server-side sessions, 358-359, 364-366
  session handlers, 568
  Smarty, 118
gdbm libraries, 252
generateProxyCode() method, 413
generate_plugins() function, 437
get() method, 455
getusage() function, resource values, 458
getSOAPServiceDescription() method, 408
getSOAPServiceName() method, 408
getSOAPServiceNamesapce() method, 408
getThis() method, 555
getTypeName() method, 411
getTypeNameSpace() method, 411
getWSDLURI() method, 408
global data structures, classes, 490-492
global keyword, 21
global variables, 20
  accessor functions, 22
  module initialization, 531-532
  truly, 21-22
GNU style, braces {}, 15
Google free SOAP interface Web site, 415
graphical interfaces, unit testing, 167-168
groups, joining, 542-543
gzip output handler, 235

“Hacking the PHP Source” Web site, 548

handlers
  access, class objects, 490
  authentication, 327
  cache, Smarty, 120
  files, 363-365
    methods 361-362, 366
    native, implementing, 366
  objects (classes), 489-490
  PHP_MINIT_FUNCTION(), 525
session
  API hook structures, 564
  closing, 565
  destroying, 567
  garbage collection, 568
  methods, 360-365
  opening, 564
  reading data, 566
  writing data, 566
signals
  child processes. See signals, child processes
  monitoring engine script, 146
handling
  I/O (input/output), 125-128
  resources, 524
handling errors, 74-75
  custom error handlers, setting, 79-80
  displaying errors, 76-77
exceptions, 82
  cascading, 94-97
  catching, 84-85
  constructor failure, 97-98
  creating, 83
  data validation, 100-104
  default exception handlers, installing, 98-100
  Exception class, 83
  hierarchies, 86-88
  Python programming, 104
  rethrowing, 94
  throwing, 83-85
  typed example, 88-93
  when to use, 104-105
extensions, 529
  external, 80-83
  ignoring errors, 78-79
  logging errors, 77-78
  severity levels, 73
hard tabs, indentation, 11-12
hardware, commodity, 371

harness
  benchmarking
    custom timer information, adding, 458-461
    features, 451
    inline benchmarks, writing, 462
    iterations, 455-456
    data randomization, 455
    overhead, removing, 456-458
    PEAR suite (Benchmark_Iterate), 451-454
  testing harnesses, creating, 454-455
hashtables, zvals, 519-520
HASH_OF() macro, 519
HEAD branches, CVS ( Concurrent Versioning System), 187
headers, HTTP
  cache-friendly applications, 231-235
  output buffering, 243-244
heavyweight sessions. See client-side sessions
Hello World! Smarty template, 110-111
hexdecode() method, 512
hexencode() method, 511
hierarchical caching, 240
hierarchies, exceptions, 86-88
home pages, caching, 266
  concurrency, 272
  templatization, 267-273
hooks, extensions, 497
horizontal scalability, 374
HTML (Hypertext Markup Language), embedding, 27
HTTP (Hypertext Transfer Protocol)
  accelerators, 223-225
  caching, 229
  headers
    cache-friendly applications, 231-235
    output buffering, 243-244
HTTP Basic Authentication, 325
httpperf load generator, 424-426
Hypertext Markup Language (HTML), embedding, 27
Hypertext Transfer Protocol. See HTTP

I

I/O (Input/Output)
    handling, 125-128
    operations, 570
identification, passively identifying bottlenecks, 420-422
identifiers, unique, 327
identities (users), cookie-based caching, 258-263
IDs, session, 356-357, 360-361
ignoring errors (error handling), 78-79
implementing
    classes, 549-550
    custom streams, 570
    interfaces, 562-564, 571
MetaWeblog API, 396
    blogid() method, 397
    callback, 399
    entries, posting, 398
    item_struct() method, 397
    publish() method, 397
    RSS, 397
    Unix timestamp, 400
    single signon, 341-346
in-memory caching
    coherency, 245-246
    file locks, 247-250
    file swaps, 250-251
    concurrency, 245-246
    file locks, 247-250
    file swaps, 250-251
    flat-file caches, 244
    size maintenance, 244-245
inbound conversion, 104
inbound filtering, 104
include files, modular versus fast code, 212
include function, Smarty, 114
indentation, 10-12

Index Organized Table (IOT), 301
indexed arrays, 518-519
indexes
    function-based, 301
    RDBMSs (relational database management systems), 300-302
    readability indexes, 169
    unique, 300
informative error messages, unit testing, 163-164
inheritance
    classes, 554
    exceptions, 86-88
    OOP (object-oriented programming), 39-40
INI entries
    accessors, 534
    declaring, 532
    parsing, module initialization, 532-535
    registering, 534
    storing, 533
init() method, 97
initialization
    lazy initialization, 319-322
    modules, Spread client wrapper, 538
inline benchmarks, writing, 462
inline comments, 29-30
inline unit testing, 157-159
input filters, SAPI, 593
    input_filter, 594-598
    post_reader, 598
    treat_data, 598
Input/Output (I/O)
    handling, 125-128
    operations, 570
input_filter input filter, 594-598
installations
    APD (Advanced PHP Debugger) profiler, 431-433
    Benchmark libraries, 452
    default exception handlers, 98-100
    PEAR XML-RPC libraries, 382
Smarty, 109-110
Spread wrapper, 382
instantiation, OOP (object-oriented programming), 38-39
integrated caching, 230
Integrated Mapper pattern, 315-317
integrating cache
  file swaps, 264-265
  home pages, 266
    concurrency, 272
    templatization, 267-273
  mode_rewrite, 273-277
  output buffering, 265-266
  partial pages, 277-280
  query cache, 280-281
interfaces
  defining, 562-564
  design patterns, 52-54
  graphical, unit testing, 167-168
  implementing, 562-564, 571
  registering (streams API), 575
  runtime error detection, 52
  SchemaTypeInfo, 411
  ServiceLogger, 143
  write, 571
intermediate code, 220, 476-479
International Organization for Standardization (ISO), 302
interpolation versus concatenation (benchmarking), 470-471
interpreters, running, 496
invalidation
  caching, 240
    DBM-based caching, 253-254
invariant functions, loops, 440
ionAccelerator Web site, 236
ionCube Accelerator compiler cache, 220
IOT (Index Organized Table), 301
ISO (International Organization for Standardization), 302
is_cached() method, MetaWeblog API, 397
iterations, 455-456
J-L
JavaScript, cookies, 330
Jim Winstead “Hacking the PHP Source” Web site, 548
K&R brace styling, 16, 24
keys
  DBM-based caching, 257
  primary, 300
keywords
  global, 21
  parent, 42
  self, 42
  static, 41
language-level tunings
  compiler caches, 219-221, 236
  HTTP accelerators, 223-225
  operating systems, 228-229
  optimizers, 222-223
  proxy caches, 229-230, 236-237
  reverse proxies, 225-228
languages, programming, 482-483
Last-Modified HTTP header, 231
latency, networks, 223-225
layers
  abstraction, computational reuse between layers, 293
  applications, 492-496
layout
  coding styles
    control flow constructs, 14-19
    indentation, 10-12
    magic comments, 12
    tabs, 11-14
lazy initialization, 319-322
least recently used (LRU) cache removal policy, 245
lengths, arrays or strings, 296
lexers (lexical analyzers), 476
libraries
access, client-side sessions, 353-354
Benchmark, installing, 452
DBM, 252
licenses, DBM, 252
lifecycles, PHP and Zend Engine
PHP core, 493, 496
PHP extension API, 493, 497-498
SAPI (Server Abstraction API layer), 492-496
Zend extension API, 493, 498-500
LIMIT syntax, 319
line breaks, 13-14
line length, code, 13
listeners, adding unit testing, 166-167
load balancing, 368
clusters, content distribution, 373-374
session stickiness, 354-355
load generators
contrived, 422-424
Daiquiri, 426-427
httperf, 424-426
realistic, 422
locking accounts, 329
locking files, 247-250
log messages, CVS (Concurrent Versioning System), 186
log outs, authentication, 333
log-based generator, 425
logging
CGI SAPIs, 588
errors (error handling), 77-78
logic
caching, factory classes, 292
templates, 114
login() function, 69
logs
slow queries, 305
Web, profiling, 435
long options, 129
long tags, 27
long-lived variables, 21-22
lookup tables, 319-320
lookups, DBM-based caching, 255-256
loop() method, ServiceCheckRunner class, 146
loops, 14
break, 18-19
continue, 18-19
deeply nested, avoiding, 19
flow, controlling, 18-19
for, 16-18
foreach, 16-18
invariant functions, 440
while, 16-18
LRU (last recently used) cache removal policy, 245
macro expansions (benchmarking), 464-470
macro substitution routines, 464-468
macros
associative arrays, 464
build system, 507
defining constants, 530
HASH_OF(), 519
PS_MOD(), 564
return, 515
SEPARATE_ZVAL(), 522
SEPARATE_ZVAL_IF_NOT_REF(), 522
ZEND_BEGIN_MODULE_GLOBALS, 531
ZEND_END_MODULE_GLOBALS, 531
ZEND_FETCH_RESOURCE(), 528
zval type conversion, 522
zvals, 516
Z_TYPE_P(), 521
magic comments, 12
magic_quotes, 103
mail() function, 80
mailto function, Smarty, 115
main() routine (CGI), 588, 591
maintaining state (authentication)
encrypted cookies, 332
expiration logic, 332-333
log outs, 333
$_SERVER['USER_AGENT'] setting, 331
$_SERVER[REMOTE_IP] setting, 331
user identity information, 333
versioning information, 333
maliciously altered data, cross-site scripting attacks, 102
managing applications
change control, CVS (Concurrent Versioning System), 184-188
binary files, 187-188
branches, 186-187, 194-195
development/production environments, 195-199
diffs, 189-191
file versions, 189-191
log messages, 186
modifying files, 188-189
repositories, 185-186
single tree limitations, 195
symbolic tags, 193-194, 199
updating files, 191-193
packaging, 199
Apache, 204-205
binaries, 203-204
pack(), 200-201
PHP, 205-206
pushing code, 201-203
managing packaging, 199
Apache, 204-205
binaries, 203-204
pack(), 200-201
PHP, 205-206
pushing code, 201-203
mandatory file locks, 247
Mapper pattern, 310-315
master C source files, extensions, 505
master/master replication, 385-386
master/slave replication, 386-389
matching characters, strings (benchmarking), 463-464
max() function, 452
max-age directives, 232
mcrypt file extension, 332
mcrypt wrappers, 341
mean, iterations, 455
memory, shared memory caching, 257-258
memory-management methods, 511-513
memory-mapped files streams API
example
abstract stream data type, 571
data, flushing, 572
fsync() interpretation, 572
interface implementation, 571
interface registration, 575
mmap_open() method, defining, 577-578
number of bytes written, returning, 572
seek functionality, 573-574
streams, 574
wrappers, 576
merging branches, CVS (Concurrent Versioning System), 195
message nodes, WSDL, 407
messages
receiving, Spread client wrapper, 543-545
sending
Spread client wrapper, 545-546
XML-RPC, 395
messaging buses, Spread toolkit, 380-384
MetaWeblog API
implementing, 396-400
blogid() method, 397
callback, 399
entries, posting, 398
item_struct() method, 397
publish() method, 397
RSS, 397
Unix timestamp, 400
Web site, 415
metaWeblog_newPost() method, 398
method pointers, Zend Engine, 598
methodologies
BSD, 257
System V, 257-258
methods
adding to classes, 555-557
addTestSuite(), 161
add_assoc_zval(), 517
analyze, 176
apd_execute(), 605
apd_execute_internal(), 605
apd_get_active_function_name(), 606
blogid(), MetaWeblog API, 397
callback, registering, 396
calling, speed, 210
chroot(), 140
close(), streams, 570
connect(), Spread client wrapper, 539-541
create_object, 560
create_table(), 116
declaring, 509
efree(), 512
emalloc(), 511-512
erealloc(), 512
estrndup(), 512
explosion, 176
factory, 562
database objects, creating, 55
singletons, creating, 56-57
Fibonacci Sequence example, 508-510
fibonacci(), 509
flush(), streams, 570
generateProxyCode(), 413
getSOAPServiceDescription(), 408
getSOAPServiceName(), 408
getSOAPServiceNamespace(), 408
getThis(), 555
getTypeName(), 411
getTypeNamesapce(), 411
getWSDLURI(), 408
hexdecode(), 512
hexencode(), 511
include, Smarty, 114
init(), 97
is_cached(), 117
item_struct(), MetaWeblog API, 397
loop(), ServiceCheckRunner class, 146
mailto, Smarty, 115
memory-management, 511-513
metaWeblog_newPost(), 398
mmap_flush(), 572
mmap_open(), 575-578
mmap_read(), 572
mmap_seek(), 573-574
mmap_write(), 572
mysql_escape_string(), 216
naming, 25
open(), streams, 570
pcntl_fork(), 130
pcntl_wait(), 132
pcntl_waitpid(), 132
pfopen(), resources, 526-527
php, Smarty, 115
phpinfo(), 536-537, 583
php_info_print_table_row(), 537
PHP_MINFO_FUNCTION(), 536
php_module_shutdown(), 584
php_module_startup(), 583
PHP_RINIT_FUNCTION(), 535
PHP_RSHUTDOWN_FUNCTION(), 536
php_sapi_name(), 583
posix_kill(), 137
posix_setuid(), 140
posix_setgid(), 140
post_run(), 143
PS_CLOSE_FUNC(), 565
PS_DESTROY_FUNC(), 567
PS_GC_FUNC(), 568
PS_OPEN_FUNC(), 564
PS_READ_FUNC(), 566
PS_WRITE_FUNC(), 566
publish(), MetaWeblog API, 397

How can we make this index more useful? Email us at indexes@samspublishing.com
read(), streams, 570
refactoring, 41
registering, 115, 546
register_block(), 118
register_function(), 115
register_modifier, 117
register_outputfilter(), 120
register_postfilter(), 119
register_prefilter(), 119
sapi_cgidbin_flush(), 585
sapi_cgidbin_getenv(), 586
seek(), streams, 570
send_headers(), 586
serendipity_fetchEntry(), 397
serendipity_updateEntry(), 397
session handlers, 360
  files, 361-366
  mm, 361, 366
  MySession, 366
  user, 361-362
session_destroy(), 358
session_start(), 357-358
session_write_close(), 358
setUp(), 165
showConversion(), 254
sig_alarm(), 137
simple, 210
Smarty, 114-117
special, OOP (object-oriented programming), 39, 42-44
spread_connect(), Spread client wrapper, 541
spread_disconnect(), Spread client wrapper, 541-542
spread_join(), Spread client wrapper, 542
spread_multicast(), Spread client wrapper, 545-546
spread_receive(), Spread client wrapper, 543-545
SP_disconnect(), 542
sp_join(), Spread client wrapper, 543
SP_multicast(), 545
SP_multigroup_multicast() method, 545
SP_receive(), 543
stat(), overriding, 585
static
  function namespacing, 372
  OOP (object-oriented programming), 41-42
system.listMethods(), 401
system.methodHelp(), 401
system.methodSignature(), 401
system_load(), 396
tearDown(), 165
trace_function_entry(), 606
trace_function_exit(), 606
urlencode(), 117
validate(), 101, 336
variables, extracting, 510
write(), 570-571
XML_RPC_decode(), 395
zend_declare_property(), 556
zend_hash_get_current_key(), 520
zend_hash_internal_pointer_reset(), 520
zend_object_store_get_object(), 561
zend_parse_parameters()  
  format strings, 514
  parameter modifiers, 514-515
  variable extraction, 510
zend_read_property(), 555
zend_register_list_destructors_ex(), 524
zend_update_property(), 555
zval_copy_ctor(), 523
Microsoft Passport, single signon, 339
microtime() timers, 459
mm session handler, 361, 366
mmap_flush() method, 572
mmap_open() method, 575-578
mmap_read() method, 572
mmap_seek() method, 573-574
mmap_write() method, 572
Model-View-Controller (MVC), 107
models, object, 488-489
modifiers, variable, 116-117
modular code, include files, 212

module hooks, 529
  module shutdown, 535
  module startup/shutdown
    constants, defining, 530-531
    globals, 531-532
    INI entries, parsing, 532-535
  phpinfo() registration, 536-537
  request startup/shutdown, 535-536

modules
  Apache, 327
  Cache_File, 379
  initializing, Spread client wrapper, 538
  registering, Spread client wrapper, 546
  shutdown
    module hooks, 535
    Spread client wrapper, 539
  startup/shutdown
    constants, defining, 530-531
    globals, 531-532
    INI entries, parsing, 532-535

mod_accel proxy server, 225
mod_backhand proxy server, 225
mod_php5 SAPI (Server Abstraction API layer), 494
mod_proxy proxy server, 225-227
mod_rewrite, cache integration, 273-277

monitoring engines, writing, 150
  abstract class implementation, 141-143
  architecture, 146
  configuration file, 148-149
  options, 149
  ServiceCheck object, 143
  ServiceCheckRunner class, 144-146
  ServiceLogger interface, 143
  ServiceLogger process, 147-148
  signals, 146

monolithic packages, 204

MovableType API Web site, 415

multiple databases, development environments, 197-198

multiple tests (unit testing), 156-157, 161-162

multitasking support. See child processes

multiword variable names, 24

must-revalidate directives, 232

MVC (Model-View-Controller), 107

MySession session handler, 366

mysqli extension, 387-388

mysql_escape_string() method, 216

my_max() function, 452

N

Nagios, 151

name-munging, function namespacing, 372

namespaces, 210, 212
  functions, clusters, 372-373
  SOAP, 405
  system resources, 373

naming
  classes, 25
  functions, 20, 24
  methods, 25
  schema, variable names, matching to, 26-27
  variables, 20, 24-27

naming symbols, 19-20
  class names, 25
  consistency, 25
  constants, 21-22
  function names, 24
  long-lived variables, 22
  method names, 25
  multiword variable names, 24
  temporary variables, 23
  truly global variables, 21-22
  variable names, matching, 26-27

native session handlers, implementing, 366

navigation bars
  cache integration, 277
  cookie-based caching, 258-263

ndbm libraries, 252

nesting deeply nested loops, avoiding, 19

network connections, FIN packets, 229

network file shares, centralized cache, 378

How can we make this index more useful? Email us at indexes@samspublishing.com
network latency, 223-225
network partitions, decentralized cache, 381
networked file systems, bandwidth, 385
networks, blocking connections, 225
Net_Telnet class, 69
next_attempt parameter, ServiceCheck object, 143
NFS (Network File System)
   network file shares, centralized cache, 378-380
   pushing production code, 203
no-cache directives, 232
nodes, WSDL (Web Services Description Language), 407
nonpersistent connections, 539
nonpersistent resources, 524

O

object handlers (classes), 489-490
object models, 488-489
object-oriented programming (OOP)
   constructors, 38-39
delegation, 50-52
design patterns
   Adapter pattern, 44-48
   Factory pattern, 54-55
   interfaces, 52-54
   polymorphism, 50-52
   Singleton pattern, 56-57
   Template pattern, 49
type hinting, 52-54
encapsulation, 39-41
inheritance, 39-40
instantiation, 38-39
overloading, 58-62
   __autoload() function, 70-71
   __call() callback, 68-70
   SPL (Standard PHP Library), 63-68
overview, 37-40
polymorphism, 40
special methods, 39, 42-44

static methods, 41-42
static properties, 41

objects. See also exceptions
   access handlers (classes), 490
   accessing, Adapter patterns, 44-48
   allocated, destroying, 560
   allocating, classes, 490
   AuthorRequest, 412
copying, 43
   creating, classes, 490
custom
   clone method, 560
   create_object() method, 560
   creating, 559-562
destructors, 560
   object store extraction, 561
database, creating via factory methods, 55
   destructing, 42-43
   ServiceCheck, 143
   SOAP_Client, 407
   Template, 120
   XML_RPC_Client, 395
   XML_RPC_Message, 395

ob_end_clean(), 243
ob_end_flush(), 243
ob_get_contents(), 243

OOF (Output Feedback) mode, 337

OOP (object-oriented programming)
   constructors, 38-39
delegation, 50-52
design patterns
   Adapter pattern, 44-48
   Factory pattern, 54-55
   interfaces, 52-54
   polymorphism, 50-52
   Singleton pattern, 56-57
   Template pattern, 49
type hinting, 52-54
encapsulation, 39-41
inheritance, 39-40
instantiation, 38-39
overloading, 58-61
patterns

__autoload() function, 70-71
__call() callback, 68-70
SPL (Standard PHP Library), 63-68
overview, 37-40
polymorphism, 40
special methods, 39, 42-44
static methods, 41-42
static properties, 41
op (operations) arrays, Zend Engine, 476-482
opcode dumper, 601, 604-605
opcodes, Zend Engine, 476-482
open tags, 27
open() method, streams, 570
operating systems (OSs), external performance tuning, 228-229
operations arrays (op arrays), 476-482
operator precedence, () parentheses, 28
operators, equal (==), 485
optimizers, 222-223
op_dumper tool, 477
OSs (operating systems), external performance tuning, 228-229
out-of-line unit testing, writing, 157-160
outbound conversion, 104
outbound filtering, 104
output buffering, 99, 242
  cache integration, 265-266
  HTTP headers, 243-244
Output Feedback (OFB) mode, 337
output filters, Smarty, 119
parent keyword, 42
parentheses (), clarifying code, 28-29
parsing
  command-line arguments, 128-130
  INI entries, module initialization, 532-535
  script execution, compiler caches, 221
  strings, 514-515
partitions
  database scaling, 384, 389-390
  network, decentralized cache, 381
Passport (Microsoft), single signon, 339
password generators, 328
passwords
  protecting, 327-330
  storing, 339
pathnames, 158-159
patterns
  Adapter, 44-48
  database access patterns, 306
    Active Record pattern, 307-310
    ad hoc queries, 307
pack(), 200-201
packages
  Console_Getopt (PEAR), 128-129
  monolithic, 204
packaging management, 199
  Apache, 204-205
  binaries, 203-204
  pack(), 200-201
  PHP, 205-206
  pushing code, 201-203
pages
  home, caching, 266-273
  concurrency, 272
  templatization, 267-273
  Web, partial pages, 277-280
parameters
  cookie-based session support, 357
  mmap seek() method, 573
  modifiers, strings, 514-515
  query string session support, 357
  ServiceCheck object, 143
  WNOHANG, pcntl_wait()/pcntl_waitpid() methods, 132
  WUNTRACED, pcntl_wait()/pcntl_waitpid() methods, 132
parent keyword, 42
parentheses (), clarifying code, 28-29
How can we make this index more useful? Email us at indexes@samspublishing.com
Integrated Mapper pattern, 315-317
Mapper pattern, 310-315
design
Adapter pattern, 44-48
Factory pattern, 54-55
interfaces, 52-54
polymorphism, 50-52
Singleton pattern, 56-57
Template pattern, 49
type hinting, 52-54
Factory, 54-55
Singleton, 56-57
Template, 49
pcntl_fork() method, 130
pcntl_wait() method, 132
pcntl_waitpid() method, 132
PCREs (Perl Compatible Regular Expressions), 295
pcre_compile() function, 295
pcre_exe() function, 295
PEAR (PHP Extension and Application Repository), 20, 69
classes, caching, 241-242
Console_Getopt package, 128-129
installer, APD (Advanced PHP Debugger)
profiler, 431
package format, 203
Web site, 122
XML-RPC classes, 395
PEAR benchmarking suite
(Benchmark_Iterate), 451-454
PEAR Extension Code Library (PECL), 220
PEAR XML-RPC libraries, installing, 382
PECL (PEAR Extension Code Library), 220
peephole optimizations, 223
performance tunings, external
cache-friendly applications, 231-235
content compression, 235-237
language-level tunings
compiler caches, 219-221, 236
HTTP accelerators, 223-225
operating systems, 228-229
optimizers, 222-223
proxy caches, 229-230, 236-237
reverse proxies, 225-228
Perl Compatible Regular Expressions (PCREs), 295
persistent associative arrays, creating, 61
persistent connections, 539
persistent hash, creating, 61
persistent resources, 524
personalized navigation bar
    cache integration, 277
    cookie-based caching, 258-263
pfopen() method, resources, finding, 526-527
PHP Extension and Application Repository. See PEAR
php function, Smarty, 115
php|architect, 151
PHP-GTK, 151
phpDocumentor project, 31-35
phpinfo() method, 536-537, 583
php_example.h file, 507
php_info_print_table_row() method, 537
PHP_MININFO_FUNCTION() method, 536
PHP_MINIT_FUNCTION() handler, 525
php_module_shutdown() method, 584
php_module_startup() method, 583
PHP_RINIT_FUNCTION() method, 535
PHP_RSHUTDOWN_FUNCTION() method, 536
php_sapi_name() method, 583
pointers
    functions, Zend Engine, 498-500
    method, Zend Engine, 598
    zval, 555
poisoning caches, 240, 383
polymorphism, 40, 50-52
portType nodes, WSDL, 407
POSIX file handles, as resources, 524
posix_kill() method, 137
posix_setgid() method, 140
posix_setuid() method, 140
postfilters, Smarty, 119
post_reader input filter, 598
post_run() method, 143
PPP (public, protected, private), 41
Pragmatic Programmer: From Journeyman to Master, 124
Pragma: no-cache HTTP header, 231
pre-fetching cache, 240
pre-fork architecture, Web servers, 228
prefilters, Smarty, 119
preg_match() function, 295
preg_replace() function, 295
previous_status parameter, ServiceCheck object, 143
primary keys, 300
printing environment variables, 113
private directives, 232
private properties, classes, 554
private variables, classes, 559
privileges, daemons, 140-141
procedural programming, 37-38
processes
  child, 130
    creating, 131
    reaping, 132-134
    resources, sharing, 131
    signals. See signals, child processes variables, sharing, 132
daemons
  exclusivity, 141
  privileges, 140-141
  working directories, 140
  writing, 138-139
ServiceLogger, 147-148
production code, pushing, 201-203
production environments, CVS (Concurrent Versioning System), 195-199
profiling, 419
  APD (Advanced PHP Debugger)
    caching tables, 446-447
    counters, 432
culling, 442-446
inefficiencies, 440-442
installing, 431-433
large applications, 435-440
trace files, 431-434
DBG profiler, 431
queries, databases, 305-306
requirements, 430
user information, cookie-based caching, 258-263
userspace profilers, 430
Web logs, 435
Xdebug profiler, 431
programming. See also OOP (object-oriented programming)
  Extreme Programming, unit testing, 154
  languages, 482-483
  procedural, 37-38
  Python, exceptions, 104
properties
  access, overloading, 60
  classes, 551-553
  dynamic instance, 551
  private, classes, 554
  static, OOP (object-oriented programming), 41
protocols, SOAP (Simple Object Application Project), 280-281
proxies
  caches, 229-230, 236-237
  code, generating, SOAP, 412-413
  reverse, 225-228
proxy-revalidate directives, 232
PS_CLOSE_FUNC() method, 565
PS_DESTROY_FUNC() method, 567
PS_GC_FUNC() method, 568
PS_MOD() macro, 564
PS_OPEN_FUNC() method, 564
PS_READ_FUNC() method, 566
PS_WRITE_FUNC() method, 566
public directives, 232
public, protected, private (PPP), 41
How can we make this index more useful? Email us at indexes@samspublishing.com
publish() method, MetaWeblog API, 397
pushing individual files, 199-200
pushing production code, 201-203
Python, exceptions, 104

Q-R

queries
ad hoc, 307
automatic query dispatching, 387-389
databases, 300-302
  bandwidth, 384
  introspection, EXPLAIN SQL syntax, 303-304
  profiles, 305-306
slow query logs, 305
troubleshooting, 305
query cache, implementing, 280-281
query string munging, 325-326, 356-357
random data, iterations, 455-456
RCS (Revision Control System), 184
RDBMSs (relational database management systems), 299
  database access patterns, 306
  Active Record pattern, 307-310
  ad hoc queries, 307
  Integrated Mapper pattern, 315-317
  Mapper pattern, 310-315
database scaling, 390
indexes, 300-302
network file shares, centralized cache, 380
queries, 300-302
  introspection, EXPLAIN SQL syntax, 303-304
  profiles, 305-306
tables, 300-302
read() method, streams, 570
readability indexes, 169
Real Time counter (wall-clock time), 432
realistic data generator, 425-426
realistic load generators, 422
Really Simple Syndication (RSS)
  MetaWeblog API, 397
  Web site, 415
reaping child processes, 132-134
receiving messages, Spread client wrapper, 543-545
recursive diffs, 191
recursive functions
  computational reuse, 283-289
tree, 286
Reduced Instruction Set Computer (RISC), 476
redundancy, clusters, 367
refactoring, 153-154, 312
  code
    coupling, 212-213
    method logic, 208-209
    namespaces, 210-212
    simple methods, 210
methods, 41
refcount (reference counter), 484
reference counting, variables, 42
registering
  callback methods, 396
  INI entries, 534
  interfaces (streams API), 575
  methods, 115, 546
  modules, Spread client wrapper, 546
  phpinfo() method, 536-537
  resources, 525-526
  SOAP services, 409
  streams (streams API), 574
  users (authentication), 327-330
  variable modifiers, 117
  wrappers (streams API), 576
  Zend Engine extension callbacks, 608
register_block() method, 118
register_function() method, 115
registermodifier() method, 117
register_outputfilter() method, 120
register_postfilter() method, 119
register_prefilter() method, 119
relational database management systems (RDBMSs), 299
  database access patterns, 306
  Active Record pattern, 307-310
  ad hoc queries, 307
  Integrated Mapper pattern, 315-317
  Mapper pattern, 310-315
  indexes, 300-302
  queries, 300-302
  introspection, EXPLAIN SQL syntax, 303-304
  profiles, 305-306
  tables, 300-302
relative pathnames, 158
remote command injection, 214
remote procedure calls (RPCs). See RPCs
removal policies, cache, 245
rename() function, 251
replication, database scaling
  master/master, 385-386
  master/slave, 386-389
repositories, CVS (Concurrent Versioning System), 185-186
Request for Comment (RFC), 236
requests
  shutdown, 536
  startup, 535
  startup/shutdown, 535
XML-RPC, 396
resources
  balancing, session stickiness, 355
  creating, 524
  finding, 526-528
  handling, 524
  nonpersistent, 524
  persistent, 524
  POSIX file handles as, 524
  registering, 525-526
  sharing, child processes, 131
result buffers, allocating, 511
result sets, databases, 317-319
rethrowing exceptions, 94
return macros, 515
  RETURN_BOOL() macro, 515
  RETURN_DOUBLE() macro, 515
  RETURN_EMPTY_STRING() macro, 515
  RETURN_FALSE() macro, 515
  RETURN_LONG() macro, 515
  RETURN_NULL() macro, 515
  RETURN_STRING() macro, 515
  RETURN_STRINGL() macro, 515
  RETURN_TRUE() macro, 515
reverse proxies, 225-228
Revision Control System (RCS), 184
RFC (Request for Comment), 236
RISC (Reduced Instruction Set Computer), 476
routines, macro substitution routines, 464-468
rows RDBMSs (relational database management systems), 300
RPCs (remote procedure calls), 393
  __call() callback, 68
  SOAP
    Amazon author search example, 410-412
    envelopes, 403-404
    namespaces, 405
    proxy code, 412-413
    registering services, 409
    Schema, 404
    user-defined types, 410-412
    writing services, 408-410
    WSDL, 405-408
    XML-RPC, compared, 413-414
  speed, 394
  XML-RPC, 394
    auto-discovery, 401-403
    callback methods registration, 396
    clients, 395
    Dave Winer Web site, 414
    messages, sending, 395
    MetaWeblog API implementation, 396-400
s-maxage directives, 233
sandboxing, 492
sanitizing data, 215-216
SAPIs, 581

CGI (command line interface)
  activation/deactivation, 584
  cookies, 587
  data writing callback, 584
  deactivation code, 584
  environment variables lookup interface, 585
  flush helper method, 584
  header handlers, 586
  logging, 588
  main() routine, 588, 591
  method pointers, 583
  POST data, reading, 587
  SAPI name, 583
  sapi_module_struct structure, 582-584
  sending headers, 586
  server variables, 588
  shell environment variables, 588
  shutdown, 584
  stat() override, 585
  stream buffers, flushing, 585
  userspace error callback, 586
  embed, 591-593
  input filters, 593
  input_filter, 594-598
  post_reader, 598
  treat_data, 598

sapi_cginbin_flush() method, 585
sapi_cginbin_getenv() method, 586
sapi_module_struct structure, 582-584

scaling, 368-369
  client-side sessions (scalability), 353
  cookie-based caching (scalability), 263
  databases
    application servers, 390-391
    partitioning, 384, 389-390
    RDBMS systems, 390
    replication, 385-389
  horizontally, 374

schema, 26-27, 404

SchemaTypeInfo interface, 411

scope, 21

scripts
  CLI (command line interface)
    arguments, 128-130
    executing, 125
    I/O, handling, 125-128
  example.php, 507
  monitoring engine, 150
    abstract class implementation, 141-143
    architecture, 146
    configuration file, 148-149
    options, 149
    ServiceCheck object, 143
    ServiceCheckRunner class, 144-146
    ServiceLogger, 143, 147-148
    signals, 146

SCSS (Source Code Control System), 184

security
  attacks, remote command injection, 214
  authentication
    dictionary attacks, 327-329
    exception, 336
    handlers, 327
    implementing, 334-339
    maintaining state, 331-333
    passwords, storing, 339
    schemes, 324-327
    single signon, 339-346
    user registration, 327-330
defensive coding, 214-215
Smarty, 119
user-submitted data, cleaning, 351
seek() method, streams, 570
SELECT statement, 318
self keyword, 42
semaphores, 257
sending messages
  Spread client wrapper, 545-546
  XML-RPC, 395
send_headers() method, 586
SEPARATE_ZVAL() macro, 522
SEPARATE_ZVAL_IF_NOT_REF() macro, 522
separation, zvals, 522
Serendipity software, 435
Serendipity Web logging system, 397-398
Serendipity Web logging system Web site, 415
serendipity_drawcalendar() function, 439
serendipity_fetchEntry() method, 397
serendipity_updertEntry() method, 397
serialization, 295
serialize() function, 292-293
server variables, CGI SAPI, 588
server-side sessions, 349, 354-355
  ending, 358
  garbage collection, 358-359, 364-366
  overview, 357-359
  session handler methods
    files, 361-366
    mm, 361, 366
    MySession, 366
    user, 361-362
  session IDs, 356-357, 360-361
  versus client-side, 366
servers
  application, database scaling, 390-391
  reverse proxies, 225-228
  Web, architecture, 228
service nodes, WSDL, 407
ServiceCheck object, 143
ServiceCheckRunner class, 144-146
ServiceLogger, 143, 147-148
services
  SOAP (Simple Object Access Protocol), 408-410
  Web, Web site, 415
session data, caching, 377
session handlers
  API hook structures, 564
  closing, 565
  destroying, 567
  garbage collection, 568
  methods, 360
    files, 361-366
    mm, 361, 366
    MySession, 366
    user, 361-362
    native, implementing, 366
  opening, 564
  reading data, 566
  writing data, 566
session IDs, 356-357, 360-361
session simulator, 425
session stickiness, 354-355
sessions
  client-side, 349-350
    access libraries, 353-354
    benefits, 352-353
    implementing via cookies, 351-353
    limitations, 353
    session data encryption, 351-352
    versus server-side, 366
  server-side, 349, 354-355
    ending, 358
    garbage collection, 358-359, 364-366
    overview, 357-359
    session handler methods
      files, 361-366
      mm, 361, 366
      MySession, 366
      user, 361-362
    session IDs, 356-357, 360-361
    versus client-side, 366
session_destroy() method, 358
session_start() method, 357-358

How can we make this index more useful? Email us at indexes@samspublishing.com
session_write_close() method, 358

setMaker() method, 454, 459-460

setUp() method, 165

set_error_handler() function, 79

Shane Caraveo Web services talks Web site, 414

shared memory caching, 257-258

sharing, child processes, 131-132

shell environment variables, CGI SAPI, 588

Shell/Perl-style comments, 29

shmop functions, 258

short options, 129

short tags, 27

showConversion() method, 254

shutting down

clusters, content distribution, 373

modules

    constants, defining, 530-531
    globals, 531-532
    INI entries, parsing, 532-535
    module hooks, 535
    Spread client wrapper, 539

requests, 535-536

SIGALRM signal, 134, 137-138

SIGCHILD signal, 134-137

SIGHUP signal, 134, 138

SIGINT signal, 134

SIGKILL signal, 134

signals

child processes

    SIGALRM, 134, 137-138
    SIGCHILD, 134-137
    SIGHUP, 134, 138
    SIGINT, 134
    SIGKILL, 134
    SIGUSR1, 134
    SIGUSR2, 134

    monitoring engine script, 146

SIGUSR1 signal, 134

SIGUSR2 signal, 134

sig_alarm() method, 137

simple methods, 210

Simple Object Access Protocol. See SOAP

Simple Object Application Project

(SOAP), 280-281

single signons, 340

    implementing, 341-346
    Microsoft Passport, 339

Singleton pattern, 56-57

singletons, creating via factory methods, 56-57

sites. See Web sites

size, cookie-based caching, 263

slaves, master/slave replication, 386-389

Sleepycat libraries, 252

slow query logs, 305

SmartTemplate Web site, 122

Smarty, 108

    block handling, 118
    cache handlers, 120
    caching, 109, 117-118
    compiled templates, 111
    custom tags, 120
    data, displaying, 112
    environment variables, printing, 113
    flow control, 111-114
    garbage collection, 118
    Hello World! template, 110-111
    installing, 109-110
    logic, 114
    manual Web site, 117
    methods, 114-117
    output filters, 119
    postfilters, 119
    prefilters, 119
    security, 119
    tables, creating, 111
    variable modifiers, 116
    Web site, 109, 121

smarty ($ before) variable, 113

SOAP (Simple Object Access Protocol)

    Amazon author search example, 410-412
    Amazon free interface Web site, 415
stale cache, 240
Standard PHP Library (SPL), 63-68
starting
modules
  constants, defining, 530-531
  globals, 531-532
  INI entries, parsing, 532-535
requests, 535
stat() method, overriding, 585
state
  cookies, 326
  maintaining (authentication)
    encrypted cookies, 332
    expiration logic, 332-333
log outs, 333
$_SERVER[‘USER_AGENT’] setting, 331
$_SERVER[REMOTE_IP] setting, 331
  unencrypted cookies, 332
  user identity information, 333
  versioning information, 333
statement handler callback, Zend Engine, 607
static extensions, creating, 507
static keyword, 41
static methods
  function namespace, 372
  OOP (object-oriented programming), 41-42
static properties, OOP (object-oriented programming), 41
statically typed languages, 482-483
status_time parameter, ServiceCheck object, 143
stderr file handle, 126-127
stdin file handle, 125-126
stdout file handle, 125
storage
  DBM-based caching, 255-256
  INI entries, 533
  passwords, 339
  streams, 570
stream buffers, flushing, 585
streams API, 579
  C streams-compatible protocols, accessing, 568
  custom stream implementation, 570
  I/O operations, 570
memory-mapped files
  abstract stream data type, 571
  data flushing, 572
  fsync() interpretation, 572
  interface implementation, 571
  interface registration, 575
  mmap_open() method, defining, 577-578
  number of bytes written, returning, 572
  opening streams, 574
  registering streams, 574
  seek functionality, 573-574
  wrappers, 576
  opening streams, 574
  registering streams, 574
  storing, 570
streams-compatible protocols, accessing, 568
strings
  binary data, 296
  C, 296
  hex-encoding, 511-512
  matching characters (benchmarking), 463-464
  parsing
    format characters, 514
    parameter modifiers, 514-515
    return macros, 515
  query string munging, 325-326, 356-357
strlen() function, 296
strncmp function, 463-464
strongly typed languages, 482-483
structs, 490-492
str_replace function, 468-469
stubs, extensions, 504, 507
symmetric cipher, 337
syntax. See code
synthetic benchmarks, 449-450
system resource namespace, 373
System Time counter, 432
System V interprocess communication (IPC), 257-258
system.listMethods() method, 401
system.methodHelp() method, 401
system.methodSignature() method, 401
system_load() method, 396

T

tables
caching, APD (Advanced PHP Debugger), 446-447
creating, Smarty, 111
defined, 299
IOT (Index Organized Table), 301
lookup, 319-320
primary keys, 300
RDBMSs (relational database management systems), 300-302
symbol, 19
tabs, indentation, 11-12
tags
long, 27
open, 27
phpDocumentor, 31
Smarty, 120
symbolic, CVS ( Concurrent Versioning System), 193-194, 199
tar, pushing production code, 202
TDD (test-driven development)
benefits, 168
bug reports, 177-179
Flesch score calculator, 169
Word class, 169-177
tearDown() method, 165
Template object, 120
Template pattern, 49
templates
home pages, caching, 267-273
Smarty, 108
block handling, 118
cache handlers, 120
caching, 109, 117-118
compiled templates, 111
custom tags, 120
data, displaying, 112
environment variables, printing, 113
flow control, 111-114
garbage collection, 118
Hello World! template, 110-111
installing, 109-110
logic, 114
manual Web site, 117
methods, 114-117
output filters, 119
postfilters, 119
prefilters, 119
security, 119
tables, creating, 111

How can we make this index more useful? Email us at indexes@samspublishing.com
variable modifiers, 116
Web site, 109, 121
writing, 120-121
TemplateTamer Web site, 122
temporary variables, 21-23
test cases, unit testing, 155
test-driven development (TDD)
   benefits, 168
   bug reports, 177-179
   Flesch score calculator, 169
   Word class, 169-177
TestCase class, 156-157
testing
code, 153-154
benefits, 168
bug reports, 177-179
Flesch score calculator, 169
unit testing, 153-154, 162-163, 168
Word class, 169-182
writing
   conditions, adding, 164-165
   Extreme Programming, 154
   graphical interfaces, 167-168
   informative error messages, 163-164
   inline, 157-159
   listeners, adding, 166-167
   multiple tests, 156-157, 161-162
   out-of-line, writing, 157-160
   overview, 154-155
   setUp() method, 165
   tearDown() method, 165
   test cases, 155
   writing, 155-156
testing harnesses, benchmarks, 454-455
tests/001.php test unit test, 507
throwing exceptions, 83-85
threaded process architectures, Web servers, 228
time-based diffs, 191
timeElapsed() method, 455
timeElapsed() method, 455
timeout parameter, ServiceCheck object, 143										
timers, custom information, 458-461
tools, op_dumper, 477
top-down culling, 443
top-down designs, 208
trace files, APD (Advanced PHP Debugger) profiler, 431-434
trace_function_entry() method, 606
trace_function_exit() method, 606
tracking session IDs
   cookies, 356-357
   query string munging, 356-357
trash data, 100-102
treat_data input filter, 598
type hinting, design patterns, 52-54
typed exceptions, example, 88-93
types
   adding to arrays, 516-517
   zvals, converting, 521-522
typing strategies
   argument types, functions, 483
data types, union, 484
dynamically typed, 482-483
functions, Zend Engine, 487
statically typed, 482-483
strongly typed, 482-483
variables, Zend Engine, 482-485
weakly typed, 483
zval, 484
U

underscore (_)

- class names, 25
- function names, 24
- word breaks, 24

unencrypted cookies, 332

unified diffs, 189

union data type, 484

unique identifiers, 327

unique indexes, 300

unit testing, 153

- automated, writing, 155
- conditions, adding, 164-165
- Extreme Programming, 154
- graphical interfaces, 167-168
- informative error messages, 163-164
- inline, 157-159
- listeners, adding, 166-167
- multiple tests, 156-157, 161-162
- out-of-line, writing, 157-160
- overview, 154-155
- setUp() method, 165
- TDD (test-driven development)
  - benefits, 168
  - bug reports, 177-179
  - Flesch score calculator, 169
  - Word class, 169-177
- tearDown() method, 165
- test cases, 155
- tests/001.php, 507
- Web, 179-182
- writing, 155-156

Unix multitasking support. See child processes

Unix timestamp, MetaWeblog API, 400 updates

- DBM-based caching, 253-254
- files, CVS (Concurrent Versioning System), 191-193

urlencode() function, 117

user authentication, Web unit testing, 179-182

user registration (authentication), 327-330

user session handlers, 361-362

User Time counter, 432

user-defined functions (Zend Engine), 486

user-defined types (SOAP), 410-412

userspace functions, 452-453

userspace profilers, 430

V

validate() method, 101, 336

validation, data validation, 100-104, 216

variable modifiers, 116-117

variables

- copying, 523
- environment
  - looking up, 585
  - printing, 113
  - shell, 588
- global, 20
  - accessor functions, 22
  - module initialization, 531-532
  - truly, 21-22
- interpolation, versus concatenation
  - (benchmarking), 470-471
- long-lived, 21-22
- methods, extracting, 510
- multiword names, 24
- names, matching to schema names, 26-27
- naming, 20
- private, classes, 559
- reference counting, 42
- scope, 21
- server, CGI SAPI, 588
- sharing, child processes, 132
- $smarty, 113
- temporary, 21-23
- Zend Engine
  - typing strategies, 482-485
  - zval, 483-485
  - zvals, 516
- arrays. See arrays
- assignments, 516
variables

creating, 516
hashtables, 519-520
macros, 516

vendor branches, CVS ( Concurrent Versioning System), 186
version tags, 333
VM (Virtual Machine). See Zend Engine

W

warnings, as exceptions (Zend Engine), 599-601
weakly typed languages, 483
Web
applications, default exception handlers, 98
logs, profiling, 435
pages, partial, cache integration, 277-280
servers, architecture, 228
services, Web site, 415
traffic, bandwidth, 385
unit testing, 179-182
Web Services Description Language (WSDL), 405-410
Web sites
Amazon free SOAP interface, 415
Amazon.com, 410
Apache, 237
Blogger API, 415
Dave Winer XML-RPC, 414
Google free SOAP interface, 415
“Hacking the PHP Source”, 548
home pages, caching, 266-273
ionAccelerator, 236
MetaWeblog API, 415
Movable Type API, 415
Nagios, 151
PEAR (PHP Extension and Application Repository), 69, 122
phplarchitect, 151
PHP-GTK, 151
RSS, 397, 415
Schema, 404

Serendipity Web logging system, 415
Shane Caraveo Web services talks, 414
Smarty, 109, 121
Smarty manual, 117
SOAP, 414
Spread client wrapper extension, 548
Squid, 236
TemplateTamer, 122
Web services, 415
Wez Furlong streams API talk, 579
XML-RPC, 414
Zend Accelerator, 236

Wez Furlong streams API talk Web site, 579
Whetstone benchmark, 450
while loops, 16-18
whitespace, 13-14
WNOHANG parameter, pcntl_wait()/pcntl_waitpid() methods, 132
word breaks, 24
Word class, 169-177
working directories, daemons, 140
wrapper classes, database access, 197
wrappers
mcrypt, 341
Spread, installing, 382
streams API, 576
write interface, 571
write() method, 570-571
writing
automated unit testing, 155
daemons, 138-141
inline unit testing, 157-159
methods, Fibonacci Sequence example, 508-510
monitoring engines, 150
abstract class implementation, 141-143
architecture, 146
configuration file, 148-149
options, 149
ServiceCheck object, 143
ServiceCheckRunner class, 144-146
ServiceLogger, 143, 147-148
  signals, 146
out-of-line unit testing, 157-160
  session handlers
  API hook structures, 564
  closing, 565
  destroying, 567
  garbage collection, 568
  opening, 564
  reading data, 566
  writing data, 566
SOAP services, 408-410
templates, 120-121
unit testing, 155-156
WSDL (Web Services Description Language), 405-410
WUNTRACED parameter, pcntl_wait()/pcntil_waitpid() methods, 132

X

Xdebug profiler, 431
XML-RPC, 394
  auto-discovery, 401-403
  callback methods registration, 396
  clients, 395
  Dave Winer Web site, 414
  messages, sending, 395
MetaWeblog API implementation, 396
  blogid() method, 397
  callback, 399
  entries, posting, 398
  item_struct() method, 397
  publish() method, 397
  RSS, 397
  Unix timestamp, 400
requests, 396
SOAP, compared, 413-414
Web site, 414
XML-RPC libraries, PEAR, 382
XML_RPC_Client object, 395
XML_RPC_decode() method, 395
XML_RPC_Message object, 395

Y-Z

Zend Accelerator compiler cache, 220
Zend Accelerator Web site, 236
Zend Engine
  APC, 606
  APD, 605-606
  classes, 487
    components, 488
    global data structures, 490-492
    objects, 488-490
  executing scripts, 220
  extension callbacks, 606-609
  functions, 486-487, 498-500
  intermediate code, 476-479
  method pointers, 598
  op (operations) arrays, 476-482
  opcodes, 476-482, 601, 604-605
PHP lifecycle
  PHP core, 493, 496
  PHP extension API, 493, 497-498
  SAPI (Server Abstraction API layer), 492-496
  Zend extension API, 493, 498-500
  script execution, 476-477
  variables, 484-485
  warnings as exceptions, 599-601
  zend_compile function, 477
  zend_execute function, 477
ZEND_BEGIN_MODULE_GLOBALS macro, 531
zend_compile function, 477
zend_compile_file() method pointer, 598
zend_declare_property() method, 556
ZEND_END_MODULE_GLOBALS macro, 531
zend_error_cb() method pointer, 599
zend_execute function, 477
zend_execute() method pointer, 599

How can we make this index more useful? Email us at indexes@samspublishing.com
ZEND_FETCH_RESOURCE() macro, 528
zend_fopen() method pointer, 599
zend_hash_get_current_key() method, 520
zend_hash_internal_pointer_reset() method, 520
zend_object_store_get_object() method, 561
zend_parse_parameters() method
  format strings, 514
  parameter modifiers, 514-515
  variable extraction, 510
zend_read_property() method, 555
zend_register_list_destructors_ex() method, 524
zend_update_property() method, 555
Zeus Web server, 228
zval pointer, 555
zvals
  accessors, 522-523
  adding to arrays, 517
arrays
  creating, 516
  data extraction, 519-520
  indexed, 518-519
  types, adding, 516-517
  zvals, adding, 517
assignments, 516
creating, 516
hashtables, 519-520
macros, 516
separation, 522
type conversions, 521-522
variables, 483-485, 523
zval_copy_ctor() method, 523
Z.ARRVAL macro, 522
Z.BVAL macro, 522
Z.LVAL macro, 522
Z.RESVAL macro, 522
Z.STRENUM macro, 522
Z.STRLEN macro, 522
Z.TYPE_P() macro, 521