

Pro PHP: Patterns, Frameworks, Testing and More

Copyright © 2008 by Kevin McArthur

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

ISBN-13 (pbk): 978-1-59059-819-1

ISBN-10 (pbk): 1-59059-819-9

ISBN-13 (electronic): 978-1-4302-0279-0

ISBN-10 (electronic): 1-4302-0279-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editors: Jason Gilmore, Tom Welsh

Technical Reviewer: Jeffrey Sambells

Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell,
Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann,
Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Beth Christmas

Copy Editor: Marilyn Smith

Associate Production Director: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Susan Glinert

Proofreader: Lisa Hamilton

Indexer: Broccoli Information Management

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.

Contents

About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii

PART 1 ■■■ OOP and Patterns

■ CHAPTER 1	Abstract Classes, Interfaces, and Programming by Contract	3
	Abstract Classes	3
	Interfaces	6
	The instanceof Operator	8
	Programming by Contract	9
	Just the Facts	10
■ CHAPTER 2	Static Variables, Members, and Methods	11
	Static Variables	11
	Static Usage in Classes	12
	Static Members	12
	Paamayim Nekudotayim	13
	Static Methods	16
	The Static Debate	18
	Just the Facts	18
■ CHAPTER 3	Singleton and Factory Patterns	21
	Responsibility and the Singleton Pattern	21
	The Factory Pattern	23
	The Image Factory	24
	The Portable Database	27
	Just the Facts	29

CHAPTER 4	Exceptions	31
	Implementing Exceptions	31
	Exception Elements	31
	Extending Exceptions	34
	Logging Exceptions	35
	Logging Custom Exceptions	35
	Defining an Uncaught Exception Handler	36
	Exception Overhead	37
	Error Coding	37
	Type Hinting and Exceptions	38
	Rethrowing Exceptions	39
	Just the Facts	40
CHAPTER 5	What's New in PHP 6	41
	PHP Installation	41
	Unicode in PHP 6	44
	Unicode Semantics	44
	Unicode Collations	46
	Namespaces	47
	Late Static Binding	48
	Dynamic Static Methods	50
	Ternary Assignment Shorthand (ifsetor)	50
	XMLWriter Class	50
	Just the Facts	52

PART 2 ■■■ Testing and Documentation

CHAPTER 6	Documentation and Coding Conventions	55
	Coding Conventions	55
	PHP Comments and Lexing	57
	Types of Comments	57
	More About Doccomments	57
	Lexing	58
	Metadata	58
	PHPDoc	59

DocBook	62
Creating an XML File for DocBook	62
Parsing a DocBook File	63
Using DocBook Elements	67
Just the Facts	71
CHAPTER 7 Reflection API	73
Introducing the Reflection API	73
Retrieving User-Declared Classes	74
Understanding the Reflection Plug-in Architecture	76
Parsing Reflection-Based Documentation Data	81
Installing the Docblock Tokenizer	81
Accessing Doccomment Data	82
Tokenizing Doccomment Data	83
Parsing the Tokens	84
Extending the Reflection API	86
Integrating the Parser with the Reflection API	86
Extending Reflection Classes	88
Updating the Parser to Handle In-Line Tags	96
Adding Attributes	99
Just the Facts	102
CHAPTER 8 Testing, Deployment, and Continuous Integration	105
Subversion for Version Control	105
Installing Subversion	106
Setting Up Subversion	106
Committing Changes and Resolving Conflicts	108
Enabling Subversion Access	110
PHPUnit for Unit Testing	110
Installing PHPUnit	110
Creating Your First Unit Test	111
Understanding PHPUnit	112
Phing for Deployment	115
Installing Phing	115
Writing a Phing Deployment Script	116
Xinc, the Continuous Integration Server	118
Installing Xinc	118
Creating the Xinc Configuration File	119
Starting Xinc	120

Xdebug for Debugging	120
Installing Xdebug	120
Tracing with Xdebug	121
Profiling with Xdebug	123
Checking Code Coverage with Xdebug	123
Remote Debugging with Xdebug	124
Just the Facts	124

PART 3 ■■■ The Standard PHP Library (SPL)

■ CHAPTER 9 Introduction to SPL	127
SPL Fundamentals	127
Iterators	128
Iterator Interface	128
Iterator Helper Functions	129
Array Overloading	130
ArrayAccess Interface	130
Counting and ArrayAccess	131
The Observer Pattern	131
Serialization	135
SPL Autoloading	137
Object Identification	140
Just the Facts	141
■ CHAPTER 10 SPL Iterators	143
Iterator Interfaces and Iterators	143
Iterator Interfaces	143
Iterators	146
Real-World Iterator Implementations	158
Parsing XML with SimpleXML	158
Accessing Flat-File Databases with DBA	159
Just the Facts	161
■ CHAPTER 11 SPL File and Directory Handling	163
File and Directory Information	163
Iteration of Directories	166
Listing Files and Directories	166
Finding Files	168
Creating Custom File Filter Iterators	169

SPL File Object Operations	171
File Iteration.....	172
CSV Operation	172
Searching Files	176
Just the Facts	177
CHAPTER 12 SPL Array Overloading	179
Introducing ArrayAccess	179
Introducing ArrayObject	180
Building an SPL Shopping Cart	182
Using Objects As Keys	184
Just the Facts	188
CHAPTER 13 SPL Exceptions	189
Logic Exceptions	189
Runtime Exceptions	191
Bad Function and Method Call Exceptions	192
Domain Exceptions	192
Range Exceptions	193
Invalid Argument Exceptions	194
Length Exceptions	194
Overflow Exceptions	195
Underflow Exceptions	196
Just the Facts	198

PART 4 ■■■ The Model-View-Controller (MVC) Pattern

CHAPTER 14 MVC Architecture	201
Why Use MVC?	201
MVC Application Layout	203
From the Web Server	203
Actions and Controllers	203
Models	203
Views	203

Criteria for Choosing an MVC Framework	204
Architecture of the MVC Framework.....	204
MVC Framework Documentation	204
MVC Framework Community.....	205
MVC Framework Support.....	205
MVC Framework Flexibility	205
Roll Your Own MVC Framework	205
Setting Up a Virtual Host	206
Creating an MVC Framework.....	207
Just the Facts	213
CHAPTER 15 Introduction to the Zend Framework	215
Setting Up the Zend Framework	215
Installing the Zend Framework	215
Creating a Virtual Host	216
Bootstrapping	217
Creating Controllers, Views, and Models	219
Adding an Index Controller.....	219
Adding a View	220
Defining Models	221
Adding Functionality	224
Using the Request and Response Objects.....	224
Using Built-in Action Helpers.....	226
Using Built-in View Helpers	227
Validating Input	229
Just the Facts	233
CHAPTER 16 Advanced Zend Framework	235
Managing Configuration Files	235
The Array Approach	235
The INI Approach	236
The XML Approach	237
Setting Site-Wide View Variables	237
Sharing Objects	238
Error Handling	238
Application Logging	239

Caching	241
Caching Security Considerations	241
Caching Techniques	242
Authorizing Users	245
Using JSON with PHP	248
Customizing Routes	249
Managing Sessions	251
Sending Mail	252
Creating PDF Files	253
Creating New PDF Pages	254
Drawing on PDF Pages	254
Integrating with Web Services	256
Just the Facts	257
CHAPTER 17 The Zend Framework Applied	259
Module and Model Setup	259
Conventional Modular Directory Structure	259
Model Libraries and Zend_Loader	260
The Request Cycle	261
Creating Plug-ins	262
Creating Helpers	263
Writing Action Helpers	263
Writing View Helpers	264
Implementing Access Control	265
Using a Two-Step View	267
Creating a Master Layout	267
Using Placeholders	268
Just the Facts	270
PART 5 ■■■ Web 2.0	
CHAPTER 18 Ajax and JSON	273
JSON and PHP	273
The JSON Extension	274
JSON in the Zend Framework	275
JSON and JavaScript	276
The XMLHttpRequest Object	278

Some Ajax Projects	280
GET Requests	280
POST Requests	281
Just the Facts	284
CHAPTER 19 Introduction to Web Services with SOAP	285
Introduction to the PHP Web Services Architecture	285
Introduction to WSDL	286
WSDL Terminology	286
A WSDL File	287
Introduction to SOAP	289
Using the PHP SOAP Extension	290
SoapClient Class Methods and Options	291
SoapServer Class Methods and Options	294
A Real-World Example	295
Just the Facts	297
CHAPTER 20 Advanced Web Services	299
Complex Types	299
A Complex Type Example	299
Class Mapping	304
Authentication	305
HTTP Authentication	305
Communicated-Key Authentication	306
Client-Certificate Authentication	306
Sessions	306
Objects and Persistence	308
Binary Data Transmission	309
Just the Facts	311
CHAPTER 21 Certificate Authentication	313
Public Key Infrastructure Security	313
Certificate Authority	313
Web Server Certificate	314
Client Certificate	314
Root CA Certificate	314

Setting Up Client Certificate Authentication	315
Creating Your Own Certificate Authority.....	315
Create a Self-Signed Web Server Certificate.....	317
Configuring Apache for SSL.....	319
Creating the Client-Side Certificates	320
Permitting Only Certificate Authentication.....	323
Testing the Certificate	324
PHP Authentication Control	325
Binding PHP to a Certificate.....	325
Setting Up Web Service Authentication	325
Just the Facts	327
INDEX	329

About the Author

■ **KEVIN MCARTHUR** is an open source developer, residing in British Columbia, Canada. He is a self-taught entrepreneur and has been running a very successful PHP application development studio for more than eight years. His company, StormTide Digital Studios, has worked with industry in the United States and Canada to provide scaling solutions for web statistics, VoIP, and print automation. An avid IRC user, Kevin helps to administer one of the largest PHP support organizations, PHP EFnet.

Kevin's contributions to open source projects, including the Zend Framework, have made him a well-known authority in the industry. He has written several articles for PHPRiot.com on topics such as reflection, the Standard PHP Library, object-oriented programming, and PostgreSQL.

Introduction

Over the past decade, PHP has transformed itself from a set of simple tools for web site development to a full-fledged object-oriented programming (OOP) language. PHP now rivals mainstream languages like Java and C# for web application development, with more and more enterprises turning to it to power their web sites. The reasons for this are clear: PHP has found the right combination of an easy-to-learn language and powerful features.

In this book, you will learn how to make the most of your PHP programming, from a detailed understanding of OOP theory to frameworks and advanced system interoperability.

Who Should Read This Book

This is an advanced book. I have needed to choose carefully which information to include and what readers should be expected to know already. Readers should have a solid understanding of HTTP and PHP—that is, you should understand how to make web pages and build forms, and you should understand key concepts like the HTTP request cycle.

If this doesn't sound like you, I recommend reading *Beginning PHP and PostgreSQL 8* by Jason Gilmore and Robert Treat (Apress, 2006; 1-59059-547-3). It is an excellent introduction to PHP programming and a definite must-read for any would-be developer.

If you are comfortable at the intermediate to advanced level, then this book is for you.

How This Book Is Organized

Each chapter builds on lessons learned in previous chapters, but recognizes that readers will have a wide variety of skill levels. If you think you already know the content covered in a chapter, I encourage you to skip ahead, but before you do, be sure to read the “Just the Facts” section at the end of each chapter. This section provides a terse summary of what was covered in the chapter. But note that even the most seasoned programmers are likely to find something worth learning in each chapter.

The book is organized into five parts:

Part 1, OOP and Patterns: This part provides a foundation for advanced OOP concepts. It dives right in and tells you all you need to know about abstract classes, interfaces, static methods, and patterns like the singleton and factory, as well as exceptions. The part concludes with an introduction to the new features in PHP 6.

Part 2, Testing and Documentation: This part covers all those interesting “peripheral” concepts, like test-driven development and automated deployment. It teaches you about writing great documentation and includes introductions to several documentation standards, including PHPDoc and DocBook. You will find information about the reflection API and learn how to extract metadata from your programs. Finally, you’ll learn about continuous integration and how to use tools like Phing and Xinc to improve your development workflow.

Part 3, The Standard PHP Library (SPL): The SPL contains some of the most advanced PHP code ever written. It offers language support for advanced OOP concepts like indexers and iterators, and also provides structures for exceptions and patterns like observer/reporter. The information in this part will allow you to create much more elegant and well-formed classes than would normally be possible.

Part 4, The Model-View-Controller (MVC) Pattern: MVC is probably the most useful development pattern for PHP developers. It allows you to structure your applications and work in teams using the best resources to get the job done. A strong understanding of this pattern is probably the single most important job qualification for any PHP developer, so this book makes a special effort to fully explain it. This part of the book also introduces you to the Zend Framework, an MVC-based framework embraced by thousands of PHP companies. It starts with a complete walk-through of how to get a framework application up and running, and then presents the core concepts and advanced features of the Zend Framework.

Part 5, Web 2.0: This part covers all the things you need to know about Web 2.0. You will find information about Ajax and JSON, SOAP web services, and SSL client authentication. This part includes a lot of really useful tutorials, based on personal experience.

Contacting the Author

Please feel free to contact the author at Kevin.McArthur@StormTide.ca. You can find the latest information about this book at <http://www.stormtide.ca/pro-php-book> or on the Apress web site, <http://www.apress.com/book/view/9781590598191>. Last but not least, you can chat with the author via IRC by visiting #PHP EFnet.



Introduction to SPL

The Standard PHP Library (SPL) is where PHP 5's object-oriented capabilities truly shine. It improves the language in five key ways: iterators, exceptions, array overloading, XML, and file and data handling. It also provides a few other useful items, such as the observer pattern, counting, helper functions for object identification, and iterator processing. Additionally, it offers advanced functionality for autoloading classes and interfaces. This chapter introduces you to this very important library, and in the following chapters, you will learn more about some of the advanced SPL classes.

SPL is enabled by default and is available on most PHP 5 systems. However, because SPL's features were dramatically expanded with the PHP 5.2 release, I recommend using that version or newer if you want to truly take advantage of this library.

SPL Fundamentals

The SPL is a series of Zend Engine 2 additions, internal classes, and a set of PHP examples. At the engine level, the SPL implements a set of six classes and interfaces that provide all the magic. These interfaces and the `Exception` class are special in that they are not really like a traditional interface. They have extra powers and allow the engine to hook into your code in a specific and special way. Here are brief descriptions of these elements:

ArrayAccess: The `ArrayAccess` interface allows you to create classes that can be treated as arrays. This ability is commonly provided by *indexers* in other languages.

Exception: The `Exception` class was introduced in Chapter 4. The SPL extension contains a series of enhancements and classifications for this built-in class.

Iterator: The `Iterator` interface makes your objects work with looping structures like `foreach`. This interface requires you to implement a series of methods that define which entries exist and the order in which they should be retrieved.

IteratorAggregate: The `IteratorAggregate` interface takes the `Iterator` concept a bit further and allows you to offload the methods required by the `Iterator` interface to another class. This lets you use some of the other SPL built-in iterator classes so you can gain iterator functionality without needing to implement `Iterator`'s methods in your class directly.

Serializable: The `Serializable` interface hooks into the `Serialize` and `Unserialize` functions, as well as any other functionality, like sessions, that may automatically serialize your classes. Using this interface, you can ensure that your classes can be persisted and restored properly. Without it, storing object data in sessions can cause problems, especially where resource type variables are used.

Traversable: The `Traversable` interface is used by the `Iterator` and `IteratorAggregate` interfaces to determine if the class can be iterated with `foreach`. This is an internal interface and cannot be implemented by users; instead, you implement `Iterator` or `IteratorAggregate`.

In the rest of this chapter, we'll take a closer look at some of the SPL features, beginning with iterators.

Iterators

Iterators are classes that implement the `Iterator` interface. By implementing this interface, the class may be used in looping structures and can provide some advanced data-access patterns.

Iterator Interface

The `Iterator` interface is defined internally in C code, but if it were represented in PHP, it would look something like Listing 9-1.

Listing 9-1. *The Iterator Interface*

```
interface Iterator {
    public function current();
    public function key();
    public function next();
    public function rewind();
    public function valid();
}
```

Note You do not need to declare `Iterator` or any other SPL interface yourself. These interfaces are automatically provided by PHP.

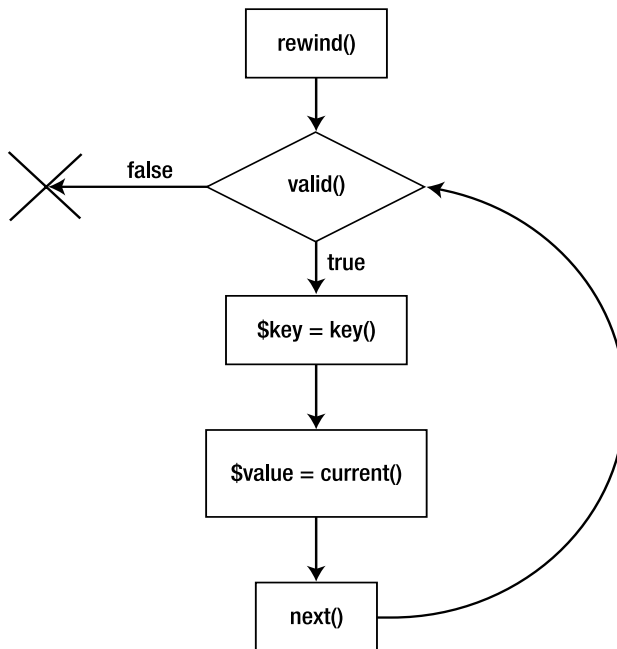
All iterable objects are responsible for keeping track of their current state. In a normal array, this would be called the array pointer. In your object, any type of variable could be used to track the current element. It is very important to remember the position of elements, as iteration requires stepping through elements one at a time.

Table 9-1 lists the methods in the `Iterator` interface.

Figure 9-1 shows the flow of the `Iterator` interface methods in a `foreach` loop.

Table 9-1. *Iterator Interface Methods*

Method	Description
current()	Returns the value of the current element
key()	Returns the current key name or index
next()	Advances the array pointer forward one element
rewind()	Moves the pointer to the beginning of the array
valid()	Determines if there is a current element; called after calls to next() and rewind()

**Figure 9-1.** *foreach Iterator method flow*

Uses for iterators range from looping over objects to looping over database result sets, and even looping around files. In the next chapter, you will learn about the types of built-in iterator classes and their uses.

Iterator Helper Functions

Several useful convenience functions can be used with iterators:

`iterator_to_array($iterator)`: This function can take any iterator and return an array containing all the data in the iterator. It can save you some verbose iteration and array-building loops when working with iterators.

`iterator_count($iterator)`: This function returns exactly how many elements are in the iterator, thereby exercising the iterator.

Caution The `iterator_to_array($iterator)` and `iterator_count($iterator)` functions can cause some spooky action if you call them on an iterator that does not have a defined ending point. This is because they require an internal exercise of the entire iterator. So if your iterator's `valid()` method will never return `false`, do not use these functions, or you will create an infinite loop.

`iterator_apply(iterator, callback, [user data])`: This function is used to apply a function to every element of an iterator, in the same way `array_walk()` is used on arrays. Listing 9-2 shows a simple `iterator_apply()` application.

Listing 9-2. *Using `iterator_apply`*

```
function print_entry($iterator) {
    print( $iterator->current() );
    return true;
}

$array = array(1,2,3);
$iterator = new ArrayIterator($array);
iterator_apply($iterator, 'print_entry', array($iterator));
```

This code outputs the following:

```
123
```

While the callback function returns `true`, the loop will continue executing. Once `false` is returned, the loop is exited.

Array Overloading

Array overloading is the process of using an object as an array. This means allowing data access through the `[]` array syntax. The `ArrayAccess` interface is at the core of this process and provides the required hooks to the Zend Engine.

ArrayAccess Interface

The `ArrayAccess` interface is described in Listing 9-3.

Listing 9-3. *The ArrayAccess Interface*

```
interface ArrayAccess {
    public function offsetExists($offset);
    public function offsetSet($offset, $value);
    public function offsetGet($offset);
    public function offsetUnset($offset);
}
```

Table 9-2 lists the methods in the ArrayAccess interface.

Table 9-2. *ArrayAccess Interface Methods*

Method	Description
offsetExists	Determines if a given offset exists in the array
offsetSet	Sets or replaces the data at a given offset
offsetGet	Returns the data at a given offset
offsetUnset	Nullifies data at a given offset

In the following chapters, you will be introduced to some of the advanced uses for array overloading.

Counting and ArrayAccess

When working with objects acting as arrays, it is often advantageous to allow them to be used exactly as an array would be used. However, by itself, an ArrayAccess implementer does not define a counting function and cannot be used with the `count()` function. This is because not all ArrayAccess objects are of a finite length.

Fortunately, there is a solution: the Countable interface. This interface is provided for just this purpose and defines a single method, as shown in Listing 9-4.

Listing 9-4. *The Countable Interface*

```
interface Countable {
    public function count();
}
```

When implemented, the Countable interface's `count()` method must return the valid number of elements in the Array object. Once Countable is implemented, the PHP `count()` function may be used as normal.

The Observer Pattern

The observer pattern is a very simple event system that includes two or more interacting classes. This pattern allows classes to observe the state of another class and to be notified when the observed class's state has changed.

In the observer pattern, the class that is being observed is called a *subject*, and the classes that are doing the observing are called *observers*. To represent these, SPL provides the `SplSubject` and `SplObserver` interfaces, as shown in Listings 9-5 and 9-6.

Listing 9-5. *The SplSubject Interface*

```
interface SplSubject {
    public function attach(SplObserver $observer);
    public function detach(SplObserver $observer);
    public function notify();
}
```

Listing 9-6. *The SplObserver Interface*

```
interface SplObserver {
    public function update(SplSubject $subject);
}
```

The idea is that the `SplSubject` class maintains a certain state, and when that state is changed, it calls `notify()`. When `notify()` is called, any `SplObserver` instances that were previously registered with `attach()` will have their `update()` methods invoked.

Listing 9-7 shows an example of using `SplSubject` and `SplObserver`.

Listing 9-7. *The Observer Pattern*

```
class DemoSubject implements SplSubject {

    private $observers, $value;

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

    public function attach(SplObserver $observer) {
        $this->observers[] = $observer;
    }

    public function detach(SplObserver $observer) {
        if($idx = array_search($observer,$this->observers,true)) {
            unset($this->observers[$idx]);
        }
    }

    public function notify() {
        foreach($this->observers as $observer) {
            $observer->update($this);
        }
    }
}
```

```
public function setValue($value) {
    $this->value = $value;
    $this->notify();
}

public function getValue() {
    return $this->value;
}

}

class DemoObserver implements SplObserver {

    public function update(SplSubject $subject) {
        echo 'The new value is '. $subject->getValue();
    }

}

$subject = new DemoSubject();
$observer = new DemoObserver();
$subject->attach($observer);
$subject->setValue(5);
```

Listing 9-7 generates the following output:

The new value is 5

The benefits of the observer pattern are that there may be many or no observers attached to the subscriber and you don't need to have prior knowledge of which classes will consume events from your subject class.

PHP 6 introduces the `SplObjectStorage` class, which improves the verbosity of this pattern. This class is similar to an array, except that it can store only unique objects and store only a reference to those objects. It offers a few benefits. One is that you cannot attach a class twice, as you can with the example in Listing 9-7, and because of this, you can prevent multiple `update()` calls to the same object. You can also remove objects from the collection without iterating/searching the collection, and this improves efficiency.

Since `SplObjectStorage` supports the `Iterator` interface, you can use it in `foreach` loops, just as a normal array can be used. Listing 9-8 shows the PHP 6 pattern using `SplObjectStorage`.

Listing 9-8. *SplObjectStorage and the Observer Pattern*

```
class DemoSubject implements SplSubject {

    private $observers, $value;
```

```

public function __construct() {
    $this->observers = new SplObjectStorage();
}

public function attach(SplObserver $observer) {
    $this->observers->attach($observer);
}

public function detach(SplObserver $observer) {
    $this->observers->detach($observer);
}

public function notify() {
    foreach($this->observers as $observer) {
        $observer->update($this);
    }
}

public function setValue($value) {
    $this->value = $value;
    $this->notify();
}

public function getValue() {
    return $this->value;
}

}

class DemoObserver implements SplObserver {

    public function update(SplSubject $subject) {
        echo 'The new value is '. $subject->getValue();
    }

}

$subject = new DemoSubject();
$observer = new DemoObserver();
$subject->attach($observer);
$subject->setValue(5);

```

Listing 9-8 generates the following output:

```
The new value is 5
```

Serialization

The SPL's `Serializable` interface provides for some advanced serialization scenarios. The non-SPL serialization magic method's `__sleep` and `__wakeup` have a couple of issues that are addressed by the SPL interface.

The magic methods cannot serialize private variables from a base class. The `__sleep` function you implement must return an array of variable names to include in the serialized output. Because of where the serialization occurs, private members of the base class are restricted. `Serializable` lifts this restriction by allowing you to call `serialize()` on the parent class, returning the serialized private members of that class.

Listing 9-9 demonstrates a scenario that magic methods cannot handle.

Listing 9-9. Magic Method Serialization

```
error_reporting(E_ALL); //Ensure notices show

class Base {
    private $baseVar;

    public function __construct() {
        $this->baseVar = 'foo';
    }
}

class Extender extends Base {
    private $extenderVar;

    public function __construct() {
        parent::__construct();
        $this->extenderVar = 'bar';
    }

    public function __sleep() {
        return array('extenderVar', 'baseVar');
    }
}

$instance = new Extender();
$serialized = serialize($instance);
echo $serialized . "\n";
$restored = unserialize($serialized);
```

Running the code in Listing 9-9 results in the following notice:

```
Notice: serialize(): "baseVar" returned as member variable from
__sleep() but does not exist ...
```

```
O:8:"Extender":2:{s:21:"ExtenderextenderVar";s:3:"bar";
s:7:"baseVar";N;}
```

To solve this problem and properly serialize the `baseVar` member, you need to use SPL's `Serializable` interface. The interface is simple, as shown in Listing 9-10.

Listing 9-10. *The Serializable Interface*

```
interface Serializable {
    public function serialize();
    public function unserialize( $serialized );
}
```

The `serialize()` method, when you implement it, requires that you return the serialized string representing the object; this is usually provided by using the `serialize()` function.

The `unserialize()` function will allow you to reconstruct the object. It takes the serialized string as an input.

Listing 9-11 shows the serialization of a private member of a base class.

Listing 9-11. *Serializing a Private Member in a Base Class*

```
error_reporting(E_ALL);

class Base implements Serializable {
    private $baseVar;

    public function __construct() {
        $this->baseVar = 'foo';
    }

    public function serialize() {
        return serialize($this->baseVar);
    }

    public function unserialize($serialized) {
        $this->baseVar = unserialize($serialized);
    }

    public function printMe() {
        echo $this->baseVar . "\n";
    }
}
```

```

class Extender extends Base {
    private $extenderVar;

    public function __construct() {
        parent::__construct();
        $this->extenderVar = 'bar';
    }

    public function serialize() {
        $baseSerialized = parent::serialize();
        return serialize(array($this->extenderVar, $baseSerialized));
    }

    public function unserialize( $serialized ) {
        $temp = unserialize($serialized);
        $this->extenderVar = $temp[0];
        parent::unserialize($temp[1]);
    }
}

$instance = new Extender();
$serialized = serialize($instance);
echo $serialized . "\n";
$restored = unserialize($serialized);
$restored->printMe();

```

Listing 9-11 has the following output:

```

C:8:"Extender":42:{a:2:{i:0;s:3:"bar";i:1;s:10:"s:3:"foo";";}}
foo

```

As you can see, the `foo` value of the base class was properly remembered and restored. The code in Listing 9-11 is very simple, but you can combine the code with functions like `get_object_vars()` to serialize every member of an object.

The `Serializable` interface offers a few other benefits. Unlike with the `__wakeup` magic method, which is called after the object is constructed, the `unserialize()` method is a constructor of sorts and will give you the opportunity to properly construct the object by storing construction input in the serialized data. This is distinct from `__wakeup`, which is called after the class is constructed and does not take any inputs.

The `Serializable` interface offers a lot of advanced serialization functionality and has the ability to create more robust serialization scenarios than the magic method approach.

SPL Autoloading

The `__autoload($classname)` magic function, if defined, allows you to dynamically load classes on their first use. This lets you retire your `require_once` statements. When declared, this function

is called every time an undefined class or interface is called. Listing 9-12 demonstrates the `__autoload($classname)` method.

Listing 9-12. *The `__autoload` Magic Method*

```
function __autoload($class) {
    require_once($class . '.inc');
}
$test = new SomeClass(); //Calls autoload to find SomeClass
```

Now, this isn't SPL. However, SPL does take this concept to the next level, introducing the ability to declare multiple autoload functions.

If you have a large application consisting of several different smaller applications or libraries, each application may wish to declare an `__autoload()` function to find its files. The problem is that you cannot simply declare two `__autoload()` functions globally without getting redeclaration errors. Fortunately, the solution is simple.

The `spl_autoload_register()` function, provided by the SPL extension, gets rid of the magic abilities of `__autoload()`, replacing them with its own type of magic. Instead of automatically calling `__autoload()` once `spl_autoload_register()` has been called, calls to undefined classes will end up calling, in order, all the functions registered with `spl_autoload_register()`.

The `spl_autoload_register()` function takes two arguments: a function to add to the autoload stack and whether to throw an exception if the loader cannot find the class. The first argument is optional and will default to the `spl_autoload()` function, which automatically searches the path for the lowercased class name, using either the `.php` or `.inc` extension, or any other extensions registered with the `spl_autoload_extensions()` function. You can also register a custom function to load the missing class.

Listing 9-13 shows the registration of the default methods, the configuration of file extensions for the default `spl_autoload()` function, and the registration of a custom loader.

Listing 9-13. *SPL Autoload*

```
spl_autoload_register(null,false);
spl_autoload_extensions('.php,.inc,.class,.interface');
function myLoader1($class) {
    //Do something to try to load the $class
}
function myLoader2($class) {
    //Maybe load the class from another path
}
spl_autoload_register('myLoader1',false);
spl_autoload_register('myLoader2',false);
$test = new SomeClass();
```

In Listing 9-13, the `spl_autoload()` function will search the include path for `someclass.php`, `someclass.inc`, `someclass.class`, and `someclass.interface`. After it does not find the definition in the path, it will invoke the `myLoader()` method to try to locate the class. If the class is not defined after `myLoader()` is called, an exception about the class not being properly declared will be thrown.

It is critical to remember that as soon as `spl_autoload_register()` is called, `__autoload()` functions elsewhere in the application may fail to be called. If this is not desired, a safer initial call to `spl_autoload_register()` would look like Listing 9-14.

Listing 9-14. *Safe `spl_autoload_register` Call*

```
if(false === spl_autoload_functions()) {
    if(function_exists('__autoload')) {
        spl_autoload_register('__autoload',false);
    }
}
//Continue to register autoload functions
```

The initialization in Listing 9-14 first calls the `spl_autoload_functions()` function, which returns either an array of registered functions or if, as in this case, the SPL autoload stack has not been initialized, the Boolean value `false`. Then you check to see if a function called `__autoload()` exists; if so, you register that function as the first function in the autoload stack and preserve its abilities. After that, you are free to continue registering autoload functions, as shown in Listing 9-13.

You can also call `spl_autoload_register()` to register a callback instead of providing a string name for the function. For example, providing an array like `array('class', 'method')` would allow you to use a method of an object.

Next, you can manually invoke the loader without actually attempting to utilize the class, by calling the `spl_autoload_call('className')` function. This function could be combined with the function `class_exists('className', false)` to attempt to load a class and gracefully fail if none of the autoloaders can find the class.

Note The second parameter to `class_exists()` controls whether or not it attempts to invoke the autoloading mechanism. The function `spl_autoload_call()` is already integrated with `class_exists()` when used in autoloading mode.

Listing 9-15 shows an example of a clean-failure load attempt using both `spl_autoload_call()` and `class_exists()` in non-autoloading mode.

Listing 9-15. *Clean Loading*

```
//Try to load className.php
if(spl_autoload_call('className')
    && class_exists('className',false)
) {

    echo 'className was loaded';
```

```
//Safe to instantiate className
$instance = new className();

} else {

//Not safe to instantiate className
echo 'className was not found';

}
```

Object Identification

Sometimes it is advantageous to have a unique code for each instance of a class. For this purpose, SPL provides the `spl_object_hash()` function. Listing 9-16 shows its invocation.

Listing 9-16. *Invoking `spl_object_hash`*

```
class a {}
$instance = new a();
echo spl_object_hash($instance);
```

This code generates the following output:

```
c5e62b9f928ed0ca74013d3e85bbf0e9
```

Each hash is guaranteed to be unique for every object within the context of a single call. Repeated execution will likely result in the same hashes being generated but is not guaranteed to produce duplicate hashes. References to the same object in the same call are guaranteed to be identical, as shown in Listing 9-17.

Listing 9-17. *`spl_object_hash` and References*

```
class a {}
$instance = new a();
$reference = $instance;
echo spl_object_hash($instance) . "\n";
echo spl_object_hash($reference) . "\n";
```

Listing 9-17 generates the following output:

```
c5e62b9f928ed0ca74013d3e85bbf0e9
c5e62b9f928ed0ca74013d3e85bbf0e9
```

This data is similar to the comparison `===` operator; however, some uses may benefit from a hash code approach. For example, when registering objects in an array, the hash code may be used as the key for easier access.

Just the Facts

In this chapter, you were introduced to the SPL. The following chapters will build on this introduction.

Iterators can be used in looping structures. SPL provides the `Iterator` interface, along with some iterator helper functions, including `iterator_to_array()`, `iterator_count()`, and `iterator_apply()`. Array overloading allows you to treat objects as arrays.

SPL includes the `Countable` interface. You can use it to hook into the global `count()` function for your custom array-like objects.

Using the SPL observer pattern and the PHP 6-specific `SplObjectStorage` class, you can make certain objects monitor other objects for changes.

SPL autoloading is provided by the `spl_autoload()`, `spl_autoload_register()`, `spl_autoload_functions()`, `spl_autoload_extensions()`, and `spl_autoload_call()` functions.

Object identification is provided by the `spl_object_hash()` function. References to the same object in the same call are guaranteed to be identical.

In the following chapter, you will be introduced to some of the more advanced iterator patterns, so be sure to keep the lessons learned about iterators and their helper functions in mind.