PHP Advanced and Object-Oriented Programming

LARRY ULLMAN
Dedication
To my good friend Michael K. and his family: I cannot thank you all enough for your continuing friendship, generosity, and kindness over these many years.
My utmost thanks to…

Jessica, the love of my life, for just about everything.

Zoe and Sam, for making my world a better place.

Everyone at Peachpit Press for their support, for their dedication to putting out quality books, and for everything else they do to make all this happen.

The most excellent editor, Rebecca Gulick, for so many reasons.

Liz Welch, for her spot-on copyediting and attention to detail.

The production coordinator, Myrna Vladic, the compositor, Danielle Foster, the proofreader, Patricia Pane, and the indexer, Valerie Haynes Perry, who turn my mess of files into an actual book.

Alan Solis, for his very, very helpful technical review.

Thomas Larsson, for his input on the design patterns chapter. Always helpful to get even one more extra set of eyes!

Tjobbe Andrews (http://tawd.co.uk), for volunteering to create a new HTML5 design for the example chapter. And for doing so on such short notice!

Sara, for entertaining the kids so that I can get some work done, even if I'd rather not.

The readers, the readers, the readers!
# Table of Contents

## Introduction
- ix

## Chapter 1: Advanced PHP Techniques
- Advanced PHP Techniques: 1
  - Multidimensional Arrays: 2
  - Advanced Function Definitions: 17
  - The Heredoc Syntax: 31
  - Using `printf()` and `sprintf()`: 37
  - Review and Pursue: 42

## Chapter 2: Developing Web Applications
- Developing Web Applications: 43
  - Modularizing a Web Site: 44
  - Improved SEO with `mod_rewrite`: 67
  - Affecting the Browser Cache: 75
  - Review and Pursue: 80

## Chapter 3: Advanced Database Concepts
- Advanced Database Concepts: 81
  - Storing Sessions in a Database: 82
  - Working with U.S. Zip Codes: 96
  - Creating Stored Functions: 108
  - Displaying Results Horizontally: 112
  - Review and Pursue: 118

## Chapter 4: Basic Object-Oriented Programming
- Basic Object-Oriented Programming: 119
  - OOP Theory: 120
  - Defining a Class: 121
  - Creating an Object: 124
  - The `$this` Attribute: 127
  - Creating Constructors: 133
  - Creating Destructors: 136
  - Designing Classes with UML: 140
  - Better Documentation with phpDocumentor: 143
  - Review and Pursue: 148
<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>Advanced OOP</th>
<th>149</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Advanced Theories</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Inheriting Classes</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>Inheriting Constructors and Destructors</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>Overriding Methods</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>Access Control</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>Using the Scope Resolution Operator</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>Creating Static Members</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>182</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>More Advanced OOP</th>
<th>183</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract Classes and Methods</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>Interfaces</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>Traits</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>Type Hinting</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>Namespaces</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>212</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Design Patterns</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Understanding Design Patterns</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>The Singleton Pattern</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>The Factory Pattern</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>The Composite Pattern</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>The Strategy Pattern</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>242</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>Using Existing Classes</th>
<th>243</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Catching Exceptions</td>
<td>244</td>
</tr>
<tr>
<td></td>
<td>Extending the Exception Class</td>
<td>251</td>
</tr>
<tr>
<td></td>
<td>Using PDO</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>Using the Standard PHP Library</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>282</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>Example—CMS with OOP</th>
<th>283</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Identifying the Goals</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>Creating the Database</td>
<td>286</td>
</tr>
<tr>
<td></td>
<td>Making the Template</td>
<td>290</td>
</tr>
<tr>
<td></td>
<td>Writing a Utilities File</td>
<td>294</td>
</tr>
<tr>
<td></td>
<td>Creating the Error View File</td>
<td>297</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Pages</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>10</td>
<td>Networking with PHP</td>
<td>327</td>
</tr>
<tr>
<td></td>
<td>Accessing Other Web Sites</td>
<td>328</td>
</tr>
<tr>
<td></td>
<td>Working with Sockets</td>
<td>333</td>
</tr>
<tr>
<td></td>
<td>Performing IP Geolocation</td>
<td>339</td>
</tr>
<tr>
<td></td>
<td>Using <strong>cURL</strong></td>
<td>343</td>
</tr>
<tr>
<td></td>
<td>Creating Web Services</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>352</td>
</tr>
<tr>
<td>11</td>
<td>PHP and the Server</td>
<td>353</td>
</tr>
<tr>
<td></td>
<td>Compressing Files</td>
<td>354</td>
</tr>
<tr>
<td></td>
<td>Establishing a <strong>cron</strong></td>
<td>363</td>
</tr>
<tr>
<td></td>
<td>Using MCrypt</td>
<td>366</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>376</td>
</tr>
<tr>
<td>12</td>
<td>PHP's Command-Line Interface</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>Testing Your Installation</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>Executing Bits of Code</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td>Interactive PHP CLI</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>Creating a Command-Line Script</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td>Running a Command-Line Script</td>
<td>391</td>
</tr>
<tr>
<td></td>
<td>Working with Command-Line Arguments</td>
<td>395</td>
</tr>
<tr>
<td></td>
<td>Taking Input</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Built-In Server</td>
<td>405</td>
</tr>
<tr>
<td></td>
<td>Review and Pursue</td>
<td>408</td>
</tr>
<tr>
<td>13</td>
<td>XML and PHP</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>What Is XML?</td>
<td>410</td>
</tr>
<tr>
<td></td>
<td>XML Syntax</td>
<td>412</td>
</tr>
<tr>
<td></td>
<td>Attributes, Empty Elements, and Entities</td>
<td>415</td>
</tr>
<tr>
<td></td>
<td>Defining XML Schemas</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>Parsing XML</td>
<td>432</td>
</tr>
<tr>
<td>Chapter 14</td>
<td>Debugging, Testing, and Performance</td>
<td>453</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>Debugging Tools</td>
<td></td>
<td>454</td>
</tr>
<tr>
<td>Unit Testing</td>
<td></td>
<td>460</td>
</tr>
<tr>
<td>Profiling Scripts</td>
<td></td>
<td>471</td>
</tr>
<tr>
<td>Improving Performance</td>
<td></td>
<td>473</td>
</tr>
<tr>
<td>Review and Pursue</td>
<td></td>
<td>476</td>
</tr>
</tbody>
</table>

Index ............................................ 477
In this humble author’s (or not-so-humble author’s) opinion, “advanced PHP” is about continuing to learn: you already know how to use PHP, and presumably MySQL, for all the standard stuff, and now it's time to expand that knowledge. This new knowledge can range from how to do different things, how to improve on the basic things, and how other technologies intersect with PHP. In short, you know how to make a dynamic Web site with PHP, but you’d like to know how to make a better Web site, with every possible meaning of “better.”

This is the approach I’ve taken in writing this book. I haven’t set out to blow your mind discussing esoteric idiosyncrasies the language has; rewriting the PHP, MySQL, or Apache source code; or making theoretically interesting but practically useless code. In short, I present to you several hundred pages of beyond-the-norm but still absolutely necessary (and often cool) tips and techniques.

About This Book

Simply put, I’ve tried to make this book’s content accessible and useful for every PHP intermediate-level programmer out there. As I suggest in the introductory paragraphs, I believe that “advanced” PHP is mostly a matter of extended topics. You already possess all the basic knowledge—you retrieve database query results in your sleep—but want to go further. This may mean learning object-oriented programming (OOP), using PEAR (PHP Extension and Application Repository), invoking PHP on the command line, picking up eXtensible Markup Language (XML), or fine-tuning aspects of your existing skill set.

My definition of advanced PHP programming covers three loosely grouped areas:

- Doing what you already do better, faster, and more securely
- Learning OOP
- Doing standard things using PHP and other technologies (like networking, unit testing, or XML)
This book can be loosely divided into three sections. The first three chapters cover advanced PHP knowledge in general: programming techniques, Web applications, and databases. Those chapters all cover information that the average PHP programmer may not be familiar with but should be able to comprehend. In the process, you’ll pick up lots of useful code, too.

The next six chapters focus on object-oriented programming. This section constitutes about half of the book. OOP is explained starting with the fundamentals, then going into lots of advanced topics, and ending with plenty of real-world examples.

The final five chapters are all “PHP and...” chapters:
- Communicating with networked servers
- Communicating with the host server
- Using the command-line interface
- XML
- Debugging, testing, and performance

Most examples used in this book are intended to be applicable in the real world, omitting the frivolous code you might see in other books, tutorials, and manuals. I focus almost equally on the philosophies involved as on the coding itself so that, in the end, you will come away with not just how to do this or that but also how to apply the new skills and ideas to your own projects.

Unlike with most of my other books, I do not expect that you’ll necessarily read this book in sequential order, for the most part. Some chapters do assume that you’ve read others, like the object-oriented ones, which have a progression to them. Some later chapters also reference examples completed in earlier ones. If you read the later ones first, you’ll just need to skip back over to the earlier ones to generate whatever database or scripts the later chapter requires.

Finally, I’ll be using HTML5 in my scripts instead of HTML. I’ll also use some CSS, as warranted. I do not discuss either of these subjects in this book (and, to be frank, may not adhere to them perfectly). If you are not already familiar with the subjects, you should look at some online resources or good books (such as Elizabeth Castro’s excellent Visual QuickStart Guides) for more information.
What’s new in this edition

I had three goals in writing this new edition:

- Greatly expanding the coverage of OOP
- Introducing new, more current topics, such as unit testing and debugging
- Cutting content that is outdated or has since been better covered in my other books

In terms of additional new material, by far the biggest change has been the additional coverage of object-oriented programming, including a chapter on design patterns. There’s also a new example chapter that uses objects instead of procedural code.

Of course, all of the code and writing has been refreshed, edited, and improved as needed. This could mean just switching to HTML5 and better use of CSS, or my doing a better job of explaining complex ideas and examples.

How this book compares to my others

Those readers who have come to this book from my *PHP for the Web: Visual Quick-Start Guide* (Peachpit Press, 2011) may find themselves in a bit over their heads. This book does assume complete comfort with standard PHP programming, in particular debugging your own scripts. I’m not suggesting you put this book down, but if you find it goes too fast for you or assumes knowledge you don’t currently possess, you may want to check out my *PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide* (Peachpit Press, 2011) instead.

If you have read the *PHP and MySQL* book, or a previous edition of this one, I’m hoping that you’ll find this to be a wonderful addition to your library and skill set.

What You’ll Need

Just as this book assumes that you already possess the fundamental skills to program in PHP (and, more important, to debug it when things go awry), it also assumes that you already have everything you need to follow along with the material. For starters, this means a PHP-enabled server. As of this writing, the latest version of PHP was 5.4, and much of the book depends on your using at least PHP 5.3.

Along with PHP, you’ll often need a database application. I use MySQL for the examples, but you can use anything. And, for the scripts in some of the chapters to work—particularly the last five—your PHP installation will have to include support for the corresponding technology, and that technology’s library may need to be installed, too. Fortunately, PHP 5 comes with built-in support for many advanced features. If the scripts in a particular chapter require special extensions, that will be referenced in the chapter’s introduction. This includes the few times where I make use of a PEAR or PECL class. Nowhere in this book will I discuss installation of PHP, MySQL, and a Web server, though, as I expect you should already know or have accomplished that.

Should you have questions or problems, you can always search the Web or post a message in my support forums ([www.LarryUllman.com/forums/](http://www.LarryUllman.com/forums/)) for assistance.

Beyond PHP, you need the things you should already have: a text editor or IDE, an FTP application (if using a remote server), and a Web browser. All of the code in this book has been tested on both Windows XP and Mac OS X; you’ll see screen shots in both operating systems.
Support Web Site

I have developed a Web site to support this book, available at www.LarryUllman.com. This site:

- Has every script available for download
- Has the SQL commands available for download
- Has extra files, as necessary, available for download
- Lists errors that have been found in the book
- Features a support forum where you can get help or assist others
- Provides a way to contact me directly

I'll also post at the site articles that extend some of the information covered in this book.

When using this site, please make sure you've gone to the correct URL (the book's title and edition are plastered everywhere). Each book I've written has its own support area; if you go to the wrong one, the downloadable files won't match those in the book.
Although PHP is still not as strong in its OOP feature set as other languages, object-oriented programming in PHP has a lot going for it. And while it is possible to have a good career without learning and using OOP, you should familiarize yourself with the concept. At the very least, being able to use both OOP and procedural programming allows you to better choose the right approach for each individual project.

In this chapter, and the next (Chapter 5, “Advanced OOP”), I will explain not only the syntax of OOP in PHP 5 and later, but the key underlying OOP theories as well. In this chapter, I will use somewhat mundane examples, but in subsequent chapters, practical, real-world code will be used. Through multiple examples and plenty of explanation, I hope in this book to fully demonstrate not just how you do object-oriented programming in PHP but also when and why.

In This Chapter

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOP Theory</td>
<td>120</td>
</tr>
<tr>
<td>Defining a Class</td>
<td>121</td>
</tr>
<tr>
<td>Creating an Object</td>
<td>124</td>
</tr>
<tr>
<td>The $this Attribute</td>
<td>127</td>
</tr>
<tr>
<td>Creating Constructors</td>
<td>133</td>
</tr>
<tr>
<td>Creating Destructors</td>
<td>136</td>
</tr>
<tr>
<td>Designing Classes with UML</td>
<td>140</td>
</tr>
<tr>
<td>Better Documentation with phpDocumentor</td>
<td>143</td>
</tr>
<tr>
<td>Review and Pursue</td>
<td>148</td>
</tr>
</tbody>
</table>
OOP Theory

The first thing that you must understand about OOP is that it presents not just new syntax but a new way of thinking about a problem. By far the most common mistake beginning OOP programmers make is to inappropriately apply OOP theory. PHP will tell you when you make a syntactical mistake, but you’ll need to learn how to avoid theoretical mistakes as well. To explain...

All programming comes down to **taking actions with data**: a user enters data in an HTML form; the PHP code validates it, emails it, and stores it in a database; and so forth. These are simply verbs (actions) and nouns (data). With procedural programming, the focus is on the verbs: do this, then this, then this. In OOP, the focus is on the nouns: with what types of things will the application work? In both approaches, you need to identify both the nouns and the verbs required; the difference is in the focus of the application’s design.

The two most important terms for OOP are **class** and **object**. A class is a generalized definition of a thing. Think of classes as blueprints. An object is a specific implementation of that thing. Think of objects as the house built using the blueprint as a guide. To program using OOP, you design your classes and then implement them as objects in your programs when needed.

One of the tenets of OOP is **modularity**: breaking applications into specific subparts. Web sites do many, many things: interact with databases, handle forms, send emails, generate HTML, etc. Each of these things can be a module, which is to say a class. By separating unrelated (albeit interacting) elements, you can develop code independently, make maintenance and updates less messy, and simplify debugging.

Related to modularity is **abstraction**: classes should be defined broadly. This is a common and understandable beginner’s mistake. As an example, instead of designing a class for interacting with a MySQL database, you should make one that interacts with a nonspecific database. From there, using **inheritance** and **overriding**, you would define a more particular class for MySQL. This class would look and act like the general database class, but some of its functionality would be customized.

Another principle of OOP is **encapsulation**: separating out and hiding how something is accomplished. A properly designed class can do everything you need it to do without your ever knowing how it’s being done. Coupled with encapsulation is **access control** or visibility, which dictates how available components of the class are.

Those are the main concepts behind OOP. You’ll see how they play out in the many OOP examples in this book. But before getting into the code, I’ll talk about OOP’s dark side.

First of all, know that OOP is **not a better way to program**, just a different way. In some cases, it **may be** better and in some cases worse.

As for the technical negatives of OOP, use of objects can be less efficient than a procedural approach. The performance difference between using an object or not may be imperceptible in some cases, but you should be aware of this potential side effect.

A second issue that arises is what I have already pointed out: misuse and overuse of objects. Whereas bad procedural programming can be a hurdle to later fix, bad OOP can be a nightmare. However, the information taught over the next several chapters should prevent that from being the case for you.
Defining a Class

OOP programming begins with classes, a class being an abstract definition of a thing: what information must be stored and what functionality must be possible with that information? A User class would be able to store information such as the user’s name, ID, email address, and so forth. The functionality of a User could be login, logout, change password, and more.

Syntactically, a class definition begins with the word class, followed by the name of the class. The class name cannot be a reserved word and is often written in uppercase, as a convention. After the class name, the class definition is placed within curly braces:

```php
class ClassName {
}
```

Classes contain variables and functions, which are referred to as attributes (or properties) and methods, respectively (you’ll see other terms, too). Collectively, a class’s attributes and methods are called its members.

Functions are easy to add to classes:

```php
class ClassName {
    function functionName() {
        // Function code.
    }
}
```

The methods you define within a class are defined just like functions outside of a class. They can take arguments, have default values, return values, and so on.

Attributes within classes are a little different than variables outside of classes. First, all attributes must be prefixed with a keyword indicating the variable’s visibility. The options are public, private, and protected. Unfortunately, these values won’t mean anything to you until you understand inheritance (in Chapter 5), so until then, just use public:

```php
class ClassName {
    public $var1, $var2;
    function functionName() {
        // Function code.
    }
}
```

As shown here, a class’s attributes are listed before any method definitions.

The second distinction between attributes and normal variables is that if an attribute is initialized with a set value, that value must be a literal value and not the result of an expression:

```php
class GoodClass {
    public $var1 = 123;
    public $var2 = 'string';
    public $var3 = array(1, 2, 3);
}
```

```php
class BadClass {
    // These won't work!
    public $today = get_date();
    public $square = $num * $num;
}
```
Note that you don’t have to initialize the attributes with a value. And, aside from declaring variables, all of a class’s other code goes within its methods. You cannot execute statements outside of a class method:

```php
class BadClass {
    public $num = 2;
    public $square;
    $square = $num * $num; // No!
}
```

With all of this in mind, let’s create an easy, almost useless class just to make sure it’s all working fine and dandy. Naturally, I’ll use a *Hello, world!* example (it’s either that or *foo* and *bar*). To make it a little more interesting, this class will be able to say *Hello, world!* in different languages.

### To define a class:

1. Create a new PHP document in your text editor or IDE, to be named *HelloWorld.php* (Script 4.1):

   ```php
   <?php # Script 4.1 - HelloWorld.php
   /*
   * This page defines the HelloWorld class.
   */
   class HelloWorld {
   // This method prints a greeting.
   // It takes one argument: the language to use.
   // Default language is English.
   function sayHello($language = 'English') {
      // Put the greeting within P tags:
      echo '<p> ';
      // Print a message specific to a language:
      switch ($language) {
         case 'Dutch':
            echo 'Hallo, wereld!';
            break;
         case 'French':
            echo 'Bonjour, monde!';
            break;
         case 'German':
            echo 'Hallo, Welt!';
            break;
         case 'Italian':
            echo 'Ciao, mondo!';
            break;
         case 'Spanish':
            echo '¡Hola, mundo!';
            break;
         default:
            echo 'Hello, world!';
            break;
      }
      // End of switch.
      // Close the HTML paragraph:
      echo '</p> ';
   } // End of sayHello() method.
   } // End of HelloWorld class.
   ```

2. Begin defining the class:

   ```php
class HelloWorld {
   ```

   Using the syntax outlined earlier, start with the keyword `class`, followed by the name of the class, followed by the opening curly brace (which could go on the next line, if you prefer).

   For the class name, I use the “uppercase camel” capitalization: initial letters are capitalized, as are the first letters of new words. This is a pseudo-standardized convention in many OOP languages.

   ```php
   } // End of HelloWorld class.
   ```

   `Script 4.1` This simple class will allow you to say *Hello, world!* through the magic of objects! (Okay, so it’s completely unnecessary, but it’s a fine introductory demonstration.)
3. Begin defining the first (and only) method:

```php
function sayHello($language = 'English') {
    // This class currently contains no attributes (variables), as those would have been declared before the methods. This method is called sayHello(). It takes one argument: the language for the greeting.
    // For the methods, I normally use the "lowercase camel" convention: start with lowercase letters, separating words with an uppercase letter. This is another common convention, although not one as consistently followed as that for the class name itself.
}
```

4. Start the method's code:

```php
    echo '<p>
```

The method will print Hello, world! in one of several languages. The message will be wrapped within HTML paragraph tags, begun here.

5. Add the method's switch:

```php
switch ($language) {
    case 'Dutch':
        echo 'Hallo, wereld!';
        break;
    case 'French':
        echo 'Bonjour, monde!';
        break;
    case 'German':
        echo 'Hallo, Welt!';
        break;
    case 'Italian':
        echo 'Ciao, mondo!';
        break;
    case 'Spanish':
        echo '¡Hola, mundo!';
        break;
}
```

6. Complete the sayHello() method:

```php
    echo '</p>
```
Creating an Object

Using OOP is a two-step process. The first—defining a class—you just did when you wrote the HelloWorld class. The second step is to make use of that class by creating an object (or a class instance).

Going back to my User class analogy, an instance of this class may be for the user with a username of janedoe. The user’s attributes might be that username, a user ID of 2459, and an email address of jane@example.com. This is one instance of the User class. A second instance, john_doe, has that username, a user ID of 439, and an email address of john.doe@example.edu. These are separate objects derived from the same class. They are the same in general, but different in specificity.

Creating an object is remarkably easy in PHP once you’ve defined your class. It requires the keyword new:

```php
$object = new ClassName();
```

Now the variable $object exists and is of type ClassName (instead of type string or array). More technically put, $object is an instance of ClassName.

To call the methods of the class, you use this syntax:

```php
$object->methodName();
```

(The -> can be called the object operator.) If a method takes arguments, you provide those within parentheses, as in any function call:

```php
$object->methodName('value', 32, true);
```

To access an object’s properties, use

```php
$object->propertyName;
```

Note that you would not use the property variable’s dollar sign, which is a common cause of parse errors:

```php
$object->$propertyName; // Error!
```

(As you’ll also see in the next chapter, the ability to reference an object’s method or property in this manner depends upon the member’s visibility.)

Once you’ve finished with an object, you can delete it as you would any variable:

```php
unset($object);
```

Simple enough! Let’s go ahead and quickly make use of the HelloWorld class.

**To create an object:**

1. Create a new PHP document in your text editor or IDE, to be named hello_object.php, beginning with the standard HTML (Script 4.2):

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello, World!</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <?php # Script 4.2 - ➝ hello_object.php
    <!-- The class definition file itself contains no HTML, as it’s not meant to be used on its own. This PHP page will include all of the code necessary to make a valid HTML page. -->
  
  $object = new ClassName();

  // Call the methods of the class.
  $object->methodName();

  // Access an object’s properties.
  $object->propertyName;

  // To call the methods of the class, you use this syntax:
  $object->methodName();

  // If a method takes arguments, you provide those within parentheses, as in any function call:
  $object->methodName('value', 32, true);

  // To access an object’s properties, use
  $object->propertyName;

  // Note that you would not use the property variable’s dollar sign, which is a common cause of parse errors:
  $object->$propertyName; // Error!

  // (As you’ll also see in the next chapter, the ability to reference an object’s method or property in this manner depends upon the member’s visibility.)

  // Once you’ve finished with an object, you can delete it as you would any variable:
  unset($object);
```

Simple enough! Let’s go ahead and quickly make use of the HelloWorld class.
Script 4.2 In this page, PHP uses the defined class in order to say Hello, world! in several different languages.

1. **Include the class definition:**
   `require('HelloWorld.php');`

   In order to create an instance of a class, the PHP script must have access to that class definition. As the definition is stored in a separate file, that file must be included here. By using `require()` (as opposed to `include()`), the script will stop executing with a fatal error if the file could not be included (and there is no point in continuing without this file).

2. **Create the object:**
   `$obj = new HelloWorld();`

   This one line of code is all there is to it! You can give the object variable any valid name you’d like, of course.

3. **Invoke the `sayHello()` method:**
   `$obj->sayHello();`

   This line of code will call the `sayHello()` method, which is part of the `$obj` object. Since the method is not being given any arguments, the greeting will be in the default language of English.

4. **Say hello in a few more languages:**
   `unset($obj);`

   An object’s methods can be called multiple times, like any other function. Different arguments are provided to vary the result.

continues on next page

---

**Fatal error:** Class 'HelloWorld' not found in /Users/larryullman/Sites/phpvq3/hello_object.php on line 18

You’ll see an error like this if you go to create an object whose class definition cannot be found.
6. Delete the object and complete the page:

```php
unset($obj);
?>
</body>
</html>
```

You don’t technically have to delete the object—it will be deleted as soon as the script ends. Still, I think it’s better programming form to tidy up like this.

7. Save the file as `hello_object.php` and place it in your Web directory, along with `HelloWorld.php`.

You don’t have to place both documents in the same directory, but if they are stored separately, you will need to change the `require()` line accordingly.

8. Test `hello_object.php` by viewing it in your Web browser.

Note that you should run `hello_object.php`, not `HelloWorld.php`, in your Web browser.

---

**TIP** Class names are not case-sensitive. However, object names, like any variable in PHP, are case-sensitive.

**TIP** Because function names in PHP are not case-sensitive, the same is true for method names in classes.

---

### Analyzing the HelloWorld Example

As I state in the first section of this chapter, OOP is both syntax and theory. For this first example, the `HelloWorld` class, the emphasis is on the syntax. Hopefully you can already see that this isn’t great use of OOP. But why? Well, it’s both too specific and too simple. Having an object print one string is a very focused idea, whereas classes should be much more abstract. It also makes absolutely no sense to use all this code—and the extra memory required—for one `echo` statement. It’s nice that the object handles different languages, but still…

The `HelloWorld` class does succeed in a couple of ways, though. It does demonstrate some of the syntax. And it is reusable: if you have a project that needs to say `Hello, world!` dozens of times, this one object will do it. And if you need to change it to `Hello, World!` (with a capital “W”), edit just the one file and you’re golden. To that end, however, it’d be better for the method to return the string, rather than just print it, so the string could be used in more ways.

Finally, this class kind of reflects the notion of *encapsulation*: you can use the object to say `Hello, world!` in multiple languages without any knowledge of how the class does that.
The $this Attribute

The HelloWorld class actually does something, which is nice, but it’s a fairly minimal example. The class includes a method, but it does not contain any attributes (variables).

As I say in the section “Defining a Class,” attributes:

- Are variables
- Must be declared as public, private, or protected (I’ll use only public in this chapter)
- If initialized, must be given a static value (not the result of an expression)

Those are the rules for defining a class’s attributes, but using those attributes requires one more piece of information. As already explained, through the object, you can access attributes via the object notation operator (->):

$object->propertyName;

The issue is that within the class itself (i.e., within a class’s methods), you must use an alternative syntax to access the class’s attributes. You cannot do just this:

class BadClass {
    public $var;
    function do() {
        // This won't work:
        print $var;
    }
}

The do() method cannot access $var in that manner. The solution is a special variable called $this. The $this variable in a class always refers to the current instance (i.e., the object involved) of that class. Within a method, you can refer to the instance of a class and its attributes by using the $this->attributeName syntax.

Rather than over-explaining this concept, I’ll go right into another example that puts this new knowledge into action. This next, much more practical, example will define a class representing a rectangle.
To use the $this variable:

1. Create a new PHP document in your text editor or IDE, to be named Rectangle.php (Script 4.3):
   ```php
   <?php # Script 4.3 - Rectangle.php
   1./*
   2. This page defines the Rectangle class.
   3. * The class contains two attributes:
   4. * width and height.
   5. * The class contains four methods:
   6. * - setSize()
   7. * - getArea()
   8. * - getPerimeter()
   9. * - isSquare()
   10. */
   11.
   12. class Rectangle {
   13.
   14.     // Declare the attributes:
   15.     public $width = 0;
   16.     public $height = 0;
   17.     // Method to set the dimensions:
   18.     function setSize($w = 0, $h = 0) {
   19.         $this->width = $w;
   20.         $this->height = $h;
   21.     }
   22.
   23.     // Method to calculate and return the area.
   24.     function getArea() {
   25.         return ($this->width *
   26.             $this->height);
   27.     }
   28.
   29.     // Method to calculate and return the perimeter.
   30.     function getPerimeter() {
   31.         return ( ($this->width +
   32.             $this->height) * 2 );
   33.     }
   34.
   35.     // Method to determine if the rectangle is also a square.
   36.     function isSquare() {
   37.         if ($this->width ==
   38.             $this->height) {
   39.             return true; // Square
   40.         }
   ```

Script 4.3 This class is much more practical than the HelloWorld example. It contains two attributes—for storing the rectangle’s width and height—and four methods.

This class has two attributes: one for the rectangle’s width and another for its height. Both are initialized to 0.

4. Create a method for setting the rectangle’s dimensions:
   ```php
   function setSize($w = 0, $h = 0) {
       $this->width = $w;
       $this->height = $h;
   }
   ```

The setSize() method takes two arguments, corresponding to the width and height. Both have default values of 0, just to be safe.

Within the method, the class’s attributes are given values using the numbers to be provided when this method is called (assigned to $w and $h). Using $this->width and $this->height refers to this class’s $width and $height attributes.

script continues on next page
5. Create a method that calculates and returns the rectangle’s area:

```php
function getArea() {
    return ($this->width * $this->height);
}
```

This method doesn’t need to take any arguments, because it can access the class’s attributes via $this. Calculating the area of a rectangle is simple: multiply the width times the height. This value is then returned.

6. Create a method that calculates and returns the rectangle’s perimeter:

```php
function getPerimeter() {
    return (($this->width + $this->height) * 2);
}
```

This method is like `getArea()`, except it uses a different formula.

7. Create a method that indicates if the rectangle is also a square:

```php
function isSquare() {
    if ($this->width == $this->height) {
        return true;
    } else {
        return false;
    }
}
```

This method compares the rectangle’s dimensions. If they are the same, the Boolean `true` is returned, indicating the rectangle is a square. Otherwise, `false` is returned.

8. Complete the class:

```php
} // End of Rectangle class.
```

9. Save the file as `Rectangle.php`.

---
To use the Rectangle class:

1. Create a new PHP document in your text editor or IDE, to be named rectangle1.php, beginning with the standard HTML (Script 4.4):

   ```html
   <!doctype html>
   <html lang="en">
   <head>
     <meta charset="utf-8">
     <title>Rectangle</title>
     <link rel="stylesheet" href="style.css">
   </head>
   <body>
   <?php # Script 4.4 - rectangle1.php
   ```

2. Include the class definition:

   ```php
   require('Rectangle.php');
   ```

3. Define the necessary variables and print an introduction:

   ```php
   $width = 42;
   $height = 7;
   echo "<h2>With a width of $width ➝ and a height of $height...</h2>";
   ```

4. Create the object and assign the rectangle's dimensions:

   ```php
   $r = new Rectangle();
   $r->setSize($width, $height);
   ```

   The first line creates an object of type Rectangle. The second line assigns the values of the variables in this script—$width and $height—to the object's attributes. The values here are assigned to $w and $h in the setSize() method when it's called, which are then assigned to $this->width and $this->height within that method.

5. Print the rectangle’s area:

   ```php
   echo '<p>The area of the rectangle ➝ is ' . $r->getArea() . '</p>,'
   ```

   To print the rectangle's area, you only need to have the object tell you what that value is by calling its getArea() method. As this method returns the area (instead of printing it), it can be used in an echo statement like this.

6. Print the rectangle’s perimeter:

   ```php
   echo '<p>The perimeter ➝ of the rectangle is ' . $r->getPerimeter() . '</p>,'
   ```

   This is a variation on the code in Step 5.

7. Indicate whether or not this rectangle is also a square:

   ```php
   echo '<p>This rectangle is ';
   if ($r->isSquare()) {
     echo 'also';
   } else {
     echo 'not';
   }
   echo ' a square.</p>','
   ```

   Since the isSquare() method returns a Boolean value, I can invoke it as a condition. This code will print either This rectangle is also a square. or This rectangle is not a square.

8. Delete the object and complete the page:

   ```php
   unset($r);
   ?>
   </body>
   </html>
   ```

9. Save the file as rectangle1.php and place it in your Web directory, along with Rectangle.php.

   continues on page 132
The Rectangle class is used in this PHP script. The rectangle's dimensions are first assigned to the class's attributes by invoking the `setSize()` method, and then various properties of the rectangle are reported.
10. Test `rectangle1.php` by viewing it in your Web browser A.

11. Change the variables’ values in `rectangle1.php` and rerun it in your Web browser B.

**Tip** Having `get` and `set` methods in a class is a common convention. Methods starting with `set` are used to assign values to class attributes. Methods starting with `get` are used to return values: either attributes or the results of calculations.

**Tip** Methods can call each other, just as they would any other function, but you’ll need to use `$this` again. The following is unnecessary but valid:

```php
function getArea() {
    if ($this->isSquare()) {
        return ($this->width * $this->width);
    } else {
        return ($this->width * $this->height);
    }
}
```

Various attributes for a rectangle are revealed using the `Rectangle` class.

If the width and height are the same, the rectangle is also a square.

Analyzing the Rectangle Example

The `Rectangle` class as defined isn’t perfect, but it’s pretty good, if I do say so myself. It encapsulates all the things you might want to do with or know about a rectangle. The methods also only handle calculations and return values; no HTML is used within the class, which is a better way to design.

One criticism may be that the class is too specific. Logically, if you’ve created a site that performs a lot of geometry, the `Rectangle` class might be an inherited class from a broader `Shape`. You’ll learn about inheritance in the next chapter.

From the first two examples you can see the benefit of objects: the ability to create your own data type. Whereas a string is a variable type whose only power is to contain characters, the `Rectangle` is a new, powerful type with all sorts of features.
Creating Constructors

A constructor is a special kind of method that differs from standard ones in three ways:

- Its name is always `__construct()`.
- It is automatically and immediately called whenever an object of that class is created.
- It cannot have a `return` statement.

The syntax for defining a constructor is therefore

```php
class ClassName {
    public $var;
    function __construct() {
        // Function code.
    }
}
```

A constructor could be used to connect to a database, set cookies, or establish initial values. Basically, you’ll use constructors to do whatever should always be done—and done first—when an object of this class is made.

Because the constructor is still just another method, it can take arguments, and values for those arguments can be provided when the object is created:

```php
class User {
    function __construct($id) {
        // Function code.
    }
}
```

```php
$me = new User(2354);
```

The `Rectangle` class could benefit from having a constructor that assigns the rectangle’s dimensions when the rectangle is created.

To add and use a constructor:

1. Open `Rectangle.php` (Script 4.3) in your text editor or IDE.
2. After declaring the attributes and before defining the `setSize()` method, add the constructor (Script 4.5):

```php
function __construct($w = 0, $h = 0) {
    $this->width = $w;
    $this->height = $h;
}
```

Script 4.5 A constructor has been added to the `Rectangle` class. This makes it possible to assign the rectangle’s dimensions when the object is created.

```
1  <?php # Script 4.5 - Rectangle.php
2  /*
3  * This page defines the Rectangle class.
4  * The class contains two attributes: width and height.
5  * The class contains five methods:
6  * - __construct()
7  * - setSize()
8  * - getArea()
9  * - getPerimeter()
10  * - isSquare()
11  */
```

`script continues on next page`
This method is exactly like the `setSize()` method, albeit with a different name. Note that constructors are normally the first method defined in a class (but still defined after the attributes).

3. Save the file as `Rectangle.php`.

4. Open `rectangle1.php` (Script 4.4) in your text editor or IDE.

5. If you want, change the values of the `$width` and `$height` variables (Script 4.6):
   
   ```php
   $width = 160;
   $height = 75;
   ```

6. Change the way the object is created so that it reads

   ```php
   $r = new Rectangle($width, $height);
   ```

   The object can now be created and the rectangle assigned its dimensions in one step.

7. Delete the invocation of the `setSize()` method.

   This method is still part of the class, though, which makes sense. By keeping it in there, you ensure that a rectangle object’s size can be changed after the object is created.
Script 4.6  This new version of the script assigns the rectangle’s dimensions when the object is created (thanks to the constructor).

```php
<?php
# Script 4.6 - rectangle2.php
/* This page uses the revised Rectangle class.
 * This page shows a bunch of information
 */

// Include the class definition:
require('Rectangle.php');

// Define the necessary variables:
$width = 160;
$height = 75;

// Print a little introduction:
echo "<h2>With a width of $width and a height of $height... </h2>";

// Create a new object:
$r = new Rectangle($width, $height);

// Print the area.
echo 'The area of the rectangle is ' . $r->getArea() . '</p> ';

// Print the perimeter.
echo 'The perimeter of the rectangle is ' . $r->getPerimeter() . '</p> ';

// Is this a square?
echo 'This rectangle is ';
if ($r->isSquare()) {
    echo 'also';
} else {
    echo 'not';
}
echo ' a square.</p> ';

// Delete the object:
unset($r);
?>
```

8. Save the file as `rectangle2.php`, place it in your Web directory along with the new `Rectangle.php` (Script 4.5), and test in your Web browser A.

Tip A constructor like the one just added to the Rectangle class is called a default constructor, as it provides default values for its arguments. This means that a Rectangle object can be created using either of these techniques:

```php
$\text{x} = \text{new Rectangle}($\text{width}, \text{height});
$\text{x} = \text{new Rectangle}();
```

Tip You can directly call a constructor (although you will rarely need to):

```php
$\text{a} = \text{new SomeClass}();
$\text{a}->_\text{construct}();
```

With the Rectangle example, this would let you get rid of the `setSize()` method without losing the ability to resize a rectangle.

Tip In PHP 4 and in other programming languages (like C++), a constructor is declared by creating a method whose name is the same as the class itself.

Tip If PHP 5 cannot find a `__construct()` method in a class, it will then try to find a constructor whose name is the same as the class (the PHP 4 constructor naming scheme).

![Image of the resulting output](rectangle2.php)

**With a width of 160 and a height of 75...**

The area of the rectangle is 12000
The perimeter of the rectangle is 470
This rectangle is not a square.

A The resulting output is not affected by the incorporation of a constructor in the Rectangle class.
Creating Destructors

The corollary to the constructor is the destructor. Whereas a constructor is automatically invoked when an object is created, the destructor is called when the object is destroyed. This may occur when you overtly remove the object:

```
$obj = new ClassName();
unset($obj); // Calls destructor, too.
```

Or this may occur when a script ends (at which point PHP releases the memory used by variables).

Being the smart reader that you are, you have probably already assumed that the destructor is created like so:

```php
class ClassName {
    // Attributes and methods.
    function __destruct() {
        // Function code.
    }
}
```

Destructors do differ from constructors and other methods in that they cannot take any arguments.

The Rectangle class used in the last two examples doesn't lend itself to a logical destructor (there's nothing you need to do when you're done with a rectangle). And rather than do a potentially confusing but practical example, I'll run through a dummy example that shows how and when constructors and destructors are called.

Autoloading Classes

When you define a class in one script that is referenced in another script, you have to make sure that the second script includes the first, or there will be errors. To that end, PHP 5 supports a special function called `__autoload` (note that functions in PHP beginning with two underscores are special ones).

The `__autoload` function is invoked when code attempts to instantiate an object of a class that hasn’t yet been defined. The `__autoload` function's goal is to include the corresponding file. In simplest form, this might be

```php
function __autoload ($class) {
    require($class . '.php');
}
```

For each new object type created in the following code, the function will be invoked:

```
$obj = new Class();
$me = new Human();
$r = new Rectangle();
```

Thanks to the `__autoload` function, those three lines will automatically include `Class.php`, `Human.php` and `Rectangle.php` (within the current directory).

Notice that this `__autoload` function is defined outside of any class; instead, it is placed in a script that instantiates the objects.

The previous edition of this book demonstrated use of the `__autoload` function, but that approach has been deprecated in favor of using the Standard PHP Library (SPL). It will be discussed in Chapter 8, “Using Existing Classes.”
To create a destructor:

1. Create a new PHP document in your text editor or IDE, to be named demo.php, beginning with the standard HTML (Script 4.7):

```
<?php # Script 4.7 - demo.php
```

2. Begin defining the class:

   ```php
   class Demo {
   
   To make this example even simpler, I'll define and use the class in the same script.
   ```

3. Create the constructor:

   ```php
   function __construct() {
        echo '<p>In the constructor.</p> ' ;
   }
   ```

   The constructor doesn't do anything but print a message indicating that it has been invoked. This will allow you to trace when the class's automatic methods are called.

4. Create the destructor:

   ```php
   function __destruct() {
        echo '<p>In the destructor.</p> ' ;
   }
   ```

5. Complete the class:

   ```php
   }
   ```

   It's a very simple class!

---

**Script 4.7** This script doesn't do anything except best convey when constructors and destructors are called.
6. Define a simple function that also creates an object:

```php
function test() {
    echo '<p>In the function.
    Creating a new object...</p>;
    $f = new Demo();
    echo '<p>About to leave the
    function.</p>';
}
```

To best illuminate the life of objects, which affects when constructors and destructors are called, I’m adding this simple function. It prints messages and creates its own object, which will be a variable that’s local to this function.

7. Create an object of class `Demo`:

```php
echo '<p>Creating a new object...</p>;
$o = new Demo();
```

When this object is created, the constructor will be called. So this script first prints this line (Creating a new object...) and will then print *In the constructor*.

8. Call the `test()` function:

```php
echo '<p>Calling the function...</p>;
test();
```

After printing the status statement, the function is called. Consequently, the function is entered, wherein *In the function. Creating a new object...* will first be printed. Then, in that function, a new object is created (called `$f`). Therefore, the constructor will be called again, and the *In the constructor* message printed, as you’ll see in the final output.

---

**Script 4.7 continued**

```php
16 // Define the class:
17 class Demo {
18    // No attributes.
19    // Constructor:
20    function __construct() {
21        echo '<p>In the constructor.</p>;
22    }
23    // Destructor:
24    function __destruct() {
25        echo '<p>In the destructor.</p>;
26    }
27 }
28 // End of Demo class.
29
30 // Define a test() function:
31 function test() {
32    echo '<p>In the function. Creating a
    new object...</p>;
33    $f = new Demo();
34    echo '<p>About to leave the
    function.</p>';
35 }
36
37 // Call the test() function:
38 echo '<p>Calling the function...</p>;
39 test();
40 // Delete the object:
41 echo '<p>About to delete the object...
    </p>);
42 unset($o);
43
44 // Call the test() function:
45 echo '<p>Calling the function...</p>;
46 test();
47
48 // End of the script.</p>;
49 ?
50 </body>
51 </html>
```
After the object is created in the function, the *About to leave the function.* message is printed. Then the function is exited, at which point in time the object defined in the function—$f$—goes away, thus invoking the $f$ object’s destructor, printing *In the destructor.*

9. Delete the $s_o$ object:

   ```php
   echo ' <p>About to delete the object...</p>';
   unset($o);
   ```

   Once this object is deleted, its destructor is invoked.

10. Complete the page:

   ```php
   echo ' <p>End of the script.</p> ' ;
   ?>
   </body>
   </html>
   ```

11. Save the file as *demo.php* and place it in your Web directory. Then test by viewing it in your Web browser A.

12. Delete the `unset($o)` line, save the file, and rerun it in your Web browser B.

   Also check the HTML source code of this page C to really understand the flow.

   (Arguably, you could also delete the *About to delete the object...* line, although I did not for the two figures.)

   **Tip**: In C++ and C#, the destructor’s name for the class *ClassName* is *~ClassName*, the corollary of the constructor, which is *ClassName*. Java does not support destructors.
Designing Classes with UML

To this point, the chapter has discussed OOP in terms of both syntax and theory, but there are two other related topics worth exploring, both new additions to this edition. First up is an introduction to Unified Modeling Language (UML), a way to graphically represent your OOP designs. Entire books are written on the subject, but since this chapter covers the fundamentals of OOP, I’ll also introduce the fundamentals of UML.

A class at its core has three components:
- Its name
- Its attributes
- Its methods

UML graphically represents a class by creating a class diagram: a three-part box for each class, with the class name at the top. The next section of the box would identify the class attributes, and the third would list the methods.

For the attributes, the attribute type (e.g., string, array, etc.) is listed after the attribute’s name, as in:

userId: number
username: string

If the attribute had a default value, you could reflect that too:

width: number = 0

To define a method in a class diagram, you would start with the method name, placing its arguments and types within parentheses. This is normally followed by the type of value the method returns:

sayHello(language: string): void

The sayHello() method doesn’t return anything, so its return type is void.

Benefits of a Class Design

While making a formal UML class design may at first appear to be more of an exercise than anything, there are concrete benefits to creating one. First of all, if you sketch out the design before doing any coding, you improve your chances of getting the code correct from the start. In other words, if you put the effort into your visual design, and ponder whether the design fully reflects the application’s needs, you minimize the number of times you’ll need to update your class definitions down the road.

Second, a principle of OOP is encapsulation: separating out and hiding how something is accomplished. A UML, with its listing of attributes, methods, and arguments, can act as a user guide for those classes. Any code that requires classes that have been modeled should be able to use the classes, and its methods and attributes, without ever looking at the underlying code. In fact, you can distribute the UML along with your code as a service to your clients.
With this in mind, you can complete the class diagram for the **HelloWorld** class. In the next steps, you'll design the diagram that reflects the **Rectangle** class.

### To design a class using UML:

1. **Using paper or software, draw a three-part box.**
   
   If you like the feeling of designing with paper and pencil, feel free, but there are also plenty of software tools that can fulfill this role, too. Search online for an application that will run on your platform, or for a site that can serve the same purposes within the browser.

2. **Add the name of the class to the top of the box:**
   
   **Rectangle**
   
   Use the class's proper name (i.e., the same capitalization).

3. **Add the attributes to the middle section:**

   - `width:number = 0`
   - `height:number = 0`

   Here are the two attributes for the **Rectangle** class. Both are numbers with default values of 0.

4. **Add the constructor definition to the third part of the box:**

   ```typescript
   _construct(width:number = 0, height:number = 0):void
   ```

   This method is named `__construct`. It takes two arguments, both of type number, and both with default values of 0. The method does not return anything, so its return value is `void`.

   *continues on next page*
5. Add the `setSize()` method definition:

```
setSize(width:number = 0, height:number = 0):void
```

The `setSize()` method happens to be defined exactly like `__construct()`.

6. Add the `getArea()` method definition:

```
getArea():number
```

The `getArea()` method takes no arguments and returns a number.

7. Add the `getPerimeter()` method definition:

```
getPerimeter():number
```

The `getPerimeter()` method also takes no arguments and returns a number.

8. Add the `isSquare()` method definition:

```
isSquare():Boolean
```

This method takes no arguments but returns a Boolean value.

9. Save your design for later reference 👀.

**Tip** Be certain to update your class design should you later change your class definition.

**Tip** In the next chapter, in which more complex OOP theory is unveiled, you'll learn more UML techniques.
Better Documentation with phpDocumentor

Along with creating a UML class design, another new topic in this edition is creating better code documentation using phpDocumentor (www.phpdoc.org).

In my opinion, properly documenting one's code is so vitally important that I wish PHP would generate errors when it came across a lack of comments! Having taught PHP and interacted with readers for years, I am amazed at how often programmers omit comments, occasionally under the guise of waiting until later. Proper documentation is something that should be incorporated into code for your own good, for your client's, for your co-workers' (if applicable), and for the programmer in the future who may have to alter or augment your work—even if that programmer is you.

Although you can adequately document your code using simple comments, as I do in this book, there are two obvious benefits to adopting a formal phpDocumentor approach:

- It conveys many best practices and recommended styles.
- phpDocumentor will generate documentation, in HTML and other formats, for you.

The generated HTML can also be a valuable resource for anyone using your code, particularly your classes.

A The generated HTML documentation for the HelloWorld class.
phpDocumentor creates documentation by reading the PHP code and your comments. To facilitate that process, you would start writing your comments in a way that phpDocumentor understands. To begin, you'll use the docblock syntax:

```php
/**
 * Short description
 * Long description
 * Tags
 */
```

The short description should be a single line description. The long description can go over multiple lines and even use some HTML. Both are optional.

After the description, write one or more lines of tags. Each tag is prefaced by @, and phpDocumentor supports several kinds; which you use will depend on the thing you're documenting.

A docblock can be placed before any of the following:
- Class definition
- Function or method definition
- Variable declaration
- Constant definition
- File inclusion

A docblock should be written at the top of a script, in order to document the entire file (Script 4.8).

**Script 4.8** A more formally documented version of the **HelloWorld** class.

```php
<?php # Script 4.8 - HelloWorld.php #
/**
 * This page defines the HelloWorld class.
 */

*/

/**
 * The HelloWorld class says "Hello, world!" in different languages.
 */

class HelloWorld {

 /**
 * Function that says "Hello, world!" in different languages.
 * @param string $language Default is "English"
 * @returns void
 */
```

*script continues on next page*
To document a variable declaration, you use the `@var` tag, followed by the variable’s type (and optional description):

```bash
/**
 * @var string
 */
```

```
$name = 'Larry Ullman';
```

Notice that the docblock doesn’t need to reference the variable name, as phpDocumentor will be able to read that from the following line of code. The point of the docblock is to indicate the variable’s intended type.

To document methods and functions, use `@param` to detail the function’s parameters and `@return` to indicate the type of value the function returns (Script 4.8).

The details as to the possible types, and the full usage of all of phpDocumentor, can be found in the documentation ([www.phpdoc.org/docs/](http://www.phpdoc.org/docs/)).

Once you’ve written comments in the proper format, you can use the phpDocumentor tool to generate your documentation. To do that, you must first install phpDocumentor. The best way to install it is using PEAR ([http://pear.php.net](http://pear.php.net)), so you must have that installed, too. PEAR already comes installed with many all-in-one WAMP, MAMP, or LAMP stacks; check your associated documentation if you’re using one of these. If not, see the sidebar for some tips on installing PEAR.

```php
function sayHello($language = 'English') {
    // Put the greeting within P tags:
    echo '<p>

    // Print a message specific to a language:
    switch ($language) {
        case 'Dutch':
            echo 'Hallo, wereld!';
            break;
        case 'French':
            echo 'Bonjour, monde!';
            break;
        case 'German':
            echo 'Hallo, Welt!';
            break;
        case 'Italian':
            echo 'Ciao, mondo!';
            break;
        case 'Spanish':
            echo 'Hola, mundo!';
            break;
        default:
            echo 'Hello, world!';
            break;
    } // End of switch.

    // Close the HTML paragraph:
    echo '</p>);

    } // End of sayHello() method.

    } // End of HelloWorld class.
```
To use phpDocumentor:

1. Complete the phpDocumentor-type comments for a file (Script 4.8) or application.

   For simplicity’s sake, Script 4.8 shows a fully documented HelloWorld.php.

2. Access your computer via the command-line interface.

   My assumption is that you already know how to do this for your platform. If not, search the Web or use my support forums for answers.

3. Add the phpDocumentor PEAR channel:

   ```
   pear channel-discover pear.phpdoc.org
   ```

   This will allow you to download the latest version of the phpDocumentor directory from that site.

4. Install phpDocumentor:

   ```
   pear install phpdoc/phpDocumentor-alpha
   ```

   This instruction comes straight from the phpDocumentor Web site. It may change in time; check the site for the best, current instructions.
Installing PEAR Packages

One PEAR-related thing I do not discuss in this book is the installation process, for two good reasons. First, with the variations of available operating systems, it’s too tough to nail down comprehensive instructions for all potential readers. Second, experience tells me that many users are on hosted servers, where they cannot directly install anything. Still, installing PEAR is not impossibly hard, and once you master the installation of a single package, installing more is a snap. If you want to try your hand at installing PEAR packages, start by checking out the PEAR manual, which has instructions. If you’re still not clear as to what you should do, search the Web for articles on the subject, particular to your operating system, and/or post a question in the book’s supporting forum, where I’ll be happy to assist.

Some installation tips up front:
- You may need to invoke the `pear` installer as a superuser (or using `sudo`).
- Make sure that the location of your PEAR directory is in your PHP include path.
- Run the command `pear help install` to see what options are available.

If you are on a hosted server, the hosting company should be willing to install PEAR packages for you (which benefit every user on the server). If they won’t do that, you ought to consider a different hosting company (seriously). Barring that, you can install PHP and PEAR on your own computer in order to use phpDocumentor.

Note that on my system, in both Step 3 and Step 4, I had to preface these commands with `sudo`, to invoke the superuser, and include the full path to PEAR (both suggestions are made in the sidebar).

5. Move to the directory where your PHP scripts are:
   ```bash
   cd /path/to/folder
   ```

6. Document a single file using
   ```bash
   phpdoc -f HelloWorld.php -t docs
   ```
   That line tells phpDocumentor to parse the file `HelloWorld.php` and to write the output to the target (`-t`) directory `docs`, which would be a folder in that same directory. phpDocumentor will attempt to create that directory, if it does not exist.

7. Open `docs/index.html` in your browser.

Tip For the sake of saving precious book space, the code in this book will not be documented using the full phpDocumentor syntax.

Tip To view documentation mistakes, check out the generated errors.

Tip To have phpDocumentor document an entire project, you can have it parse the current directory using

```bash
phpdoc -d . -t docs
```

Tip If you want, you can edit the templates used by phpDocumentor to output HTML more to your liking.
Review and Pursue

If you have any problems with these sections, either in answering the questions or pursuing your own endeavors, turn to the book's supporting forum (www.LarryUllman.com/forums/).

Review

- How does OOP differ from procedural programming? (See page 120.)
- What is a class? What is an object? What is an attribute (or property)? What is a method? (See page 121.)
- What syntax do you use to create a class? To create an object? (See pages 121 and 124.)
- How do you create class methods? How do you call object methods? (See pages 121 and 124.)
- How do you create class attributes? How do you reference those attributes within the class? How do you reference those attributes using an object? (See pages 121, 124, and 127.)
- What is a constructor? How do you create one? When is a constructor called? (See page 133.)
- What is a destructor? How do you create one? When is a destructor called? (See page 136.)
- What is UML? How do you represent a class in UML? (See page 140.)
- What is phpDocumentor? What are the arguments for using it? (See page 143.)
- What is a docblock? (See page 144.)

Pursue

- Come up with another (relatively simple) class. Define and use it in PHP. Then model and document it using UML and phpDocumentor.
- Learn more about UML, if you are so inclined.
- Find UML software that you like (for your platform or online).
- Learn more about phpDocumentor, if you are so inclined.
- Add phpDocumentor-style comments to the Rectangle class and then generate its documentation.
Index

Symbols

<>, using with heredoc syntax, 31
& (ampersand), using with variables, 30
% (percent sign), using in strings, 41
:: (scope resolution operator), using, 172–175, 177
; (semicolon), use with stored functions, 109
" (quotation marks), using with classes, 152

A

abstract classes
  versus classes, 184
  creating, 186
  declaring attributes, 188
  defining constructors, 188
  Heron's Formula, 188
  versus interfaces, 191, 196
  Triangle class, 186–189
Abstract Factory pattern versus Factory, 224
abstract methods
  creating, 186
  declaring attributes, 188
  defining constructors, 188
  Heron's Formula, 188
access control
  establishing for methods, 165
  importance of, 166
  indicating in UML, 166
  in OOP, 151
private level, 165–166
protected level, 165
public level, 165
  restriction of, 165
accessor, explained, 171

add_page.html script, beginning, 324
add_page.php script
  beginning, 322
  submit button, 324
add_task.php script
  beginning, 10, 260
  for prepared statements, 266
  for SELECT query, 264
  for sprintf(), 39
Advanced PHP Debugger, downloading, 454
ampersand (&), using with variables, 30
Andrews, Tjobbe, 290
anonymous functions
  calling, 27
  downside, 27
  using, 27–29
antipatterns, explained, 232
Apache
  configuration, 67
  enabling URL rewriting, 71
  making improvements with, 285
array() function, replacing calls to, 2. See also multidimensional arrays; short array syntax
assertions, using with unit tests, 462
attributes
  in classes, 121
  protecting, 171
  rules for definition of, 127
  versus static variables, 177
autocompletion, support for, 387
__autoload() function, invoking, 136
autoload.php file, saving, 279
B
b type specifier, meaning of, 37
defining in OOP, 121–123
deriving from parents, 153–156
designing with UML, 140–142
defining in OOP, 121–123
backing up database, 356–357
designing with UML, 140–142
behavioral patterns
backtrace, printing, 50
explained, 215
using, 233
backing up database, 356–357
behavioral patterns
backtrace, printing, 50
behavioral patterns
explained, 215
using, 233
books1.xml document, beginning, 413
books1.xml file, opening, 416
bootstrap file
confirming module file, 60
creating, 57–60
header file, 60
main page, 57–60
purpose, 57
switch conditional, 59–60
validating, 59
browser cache, affecting, 75–79
C
c type specifier, meaning of, 37
caching. See also server caches
Ca type specifier, meaning of, 37
cache header types, 75
cache-control directives, 75
Caching. See also server caches
affecting, 76–79
pages, 75
CGI (Common Gateway Interface), versus CLI (command-line interface), 378
Cache-Control header type, 75, 79
checking, 75
CGI (Common Gateway Interface), versus CLI (command-line interface), 378
class attributes, accessing, 127–132
class design, benefits, 140
class versus object names, case-sensitivity, 126
class design, benefits, 140
classes. See also inheritance; OOP (object-oriented programming)
versus abstract classes, 184
defining for CMS with OOP example, 299–303
CLI installation testing
  on Mac OS X, 381–382
  on Unix, 381–382
  on Windows 7, 379–380
client URLs (cURL) utility. See cURL (client URLs) utility
__clone() method, defining, 197
CMS (content management system), 283
CMS with OOP example. See also OOP (object-oriented programming)
categories table, 286
comments table, 286
creating pages, 289
creating users, 289
creatorId, 288
database, 286–289
defining classes, 299–303
error view file, 297–298
footer for template, 291–293
header file for template, 290
header for template, 291–293
home page, 304–307
home page view, 306–307
HTML_QuickForm2, 312–319
MVC (Model-View-Controller) approach, 284–285
Page class, 299–301
pages, 284
pages table, 286, 288
pages versus posts, 286
site organization, 285
tags table, 286
template, 290–293
three-include approach for template, 290
User class, 301–303
user type structure, 289
users, 284
users table, 286–287, 289
utilities file, 294–296
viewing pages, 308–311
code documentation, importance of, 143
code library, organizing, 208
collection.dtd document, creating, 422
collection.xsd document, creating, 428
Color Blue HTML5 design, using, 52
command-line arguments
  alternative usage, 399
number.php script, 395
  using, 396–399
command-line script
  checking syntax without running, 394
  creating, 388–390
  running in Mac OS X, 394
  running in Unix, 394
  running in windows, 391–393
Company.php script, beginning, 208
Composite design, creating, 226–230
Composite pattern
  considering, 225
  described, 225
  example of, 232
  implementing, 225
  subclasses, 226
  using with Visitor pattern, 232
composite.php script, beginning, 231
composition
  “has a” relationship, 203
  indicating in UML, 203
  using, 209
compressing files, 354–362
config.inc.php script, beginning, 45
Config.php script, beginning for Singleton class, 217
configuration file, for modularized site, 45–51
  constants, assigning values to, 176
__construct() method, looking for, 135
constructors. See also destructors
calling directly, 135
creating in OOP, 133–135
declaring, 135
default, 135
inheriting, 157–160
for static members, 178–179
subclass, 158–160
syntax, 133
using, 133–135
content management system (CMS). See CMS with OOP example
content modules, creating, 61–63
creational patterns
explained, 215
using, 225
cron service
adding items to files, 363
asterisk (*) parameter, 363
crontab format, 363
establishing, 363–365
establishing for PHP file, 364–365
setting ranges with hyphen (-), 363
crontab file, using, 365
CRUD functionality, using iCrud interface for, 192–193
cURL (client URLs) utility
beginning transaction, 345
executing transaction, 345
invoking, 343
POST data, 345
POST method, 345
redirects, 345
timeout, 345
using, 343–346
cURL library, 344
curl_errno() function, 346
curl_getinfo() function, 346
curl.php script
creating, 343
running, 351

d
d type specifier, meaning of, 37
data
decrypting with MCrypt, 372–375
encrypting with MCrypt, 367–371
database file, creating for modularized site, 45
database-driven arrays
adding tasks, 10–16
connecting to database, 10
displaying tasks, 16
HTML form, 13
retrieving tasks, 14
securing task value, 14
selecting columns, 8
sorting tasks, 16
submission conditional, 14
using, 9
databases. See also zip codes
backing up, 356–357
distance calculations, 102–107
optimizing joins, 103
session functions, 84
session handlers, 85–91
session table, 82–83
SHOW WARNINGS command, 99
storage of session data, 82, 85–87
stores table, 100–101
zip codes, 96–99
db_backup.php document, creating, 355
db_sessions script, beginning, 85
DBG debugging tool, downloading, 454
debugging tools
Advanced PHP Debugger, 454
DBG, 454
Xdebug, 454
DECLARE statement, using with variables, 108
decrypting data with MCrypt, 372–375
delimiter, changing for stored functions, 109–110
demo document, creating, 137
design patterns
antipatterns, 232
behavioral, 215, 233
components, 214
Composite, 225–232
creational, 215, 225
Factory, 220–224
Gang of Four, 215
Iterator, 273–277
Singleton, 216–219
Strategy, 233–241
structural, 215, 225
destructors. See also constructors
creating, 136–139
inheriting, 157–160
for static members, 179
directories
protecting, 70
restricting access, 70
displaying results horizontally, 112–117
display.php script, beginning, 112
distance calculations, performing, 102–107
docblocks, using, 144–145
documentation
  importance of, 143
  viewing mistakes, 147
documenting
  functions, 145
  methods, 145
  variable declarations, 145
DTD, associating with XML file, 419–420
E
e type specifier, meaning of, 37
encapsulation
  explained, 166
  use in OOP, 120, 126, 140
encrypting data with Mcrypt, 367–371
error handling, purpose of, 460
error view file, creating, 297–298
error.html document, beginning, 297
Exception class, extending, 251–257
exception handling, purpose of, 460
exceptions, catching, 244–250, 259–260
Expat
  functions resource, 439
  parsing XML with, 433
expat.php document, creating, 434
Expires cache header type, 75, 79
F
f type specifier, meaning of, 37
Factory pattern
  versus Abstract Factory, 224
  consequence, 224
  creating, 220–224
  described, 220
  using, 220
  variation, 224
factory.php script
  for autoloading classes, 279
  beginning, 222
fetch() method, using, 262
file functions
  fgetc(), 404
  fgetcsv(), 404
  using on STDIN, 404
files, compressing, 354–362
final definition, using with functions, 163
footer.html file, saving, 56
footer.inc.php file, saving, 293
fopen()
  versus fsocketopen(), 338
  using, 328, 333
fscanf() function, using, 41, 400
fsocketopen()
  versus fopen(), 338
  using, 333–338
FTP port number, 333
function definitions
  anonymous functions, 27–29
  recursive, 17–24
  static variables, 24–26
function parameters
  making copies of variables, 30
  passing by reference, 30
  passing by value, 30
type hinting, 15
functions. See also stored functions
documenting, 145
final definition, 163
and references, 30
G
Gang of Four, 215
garbage collection, using with session handlers, 90
geolocation information, fetching, 340
get and set methods, using with classes, 132
get_quote.php document, creating, 329
getter, explained, 171
H
“has a” relationship, explained, 203
header() function, using in caching, 75–77
header.html file, saving, 55
header.inc.php script, using, 291
Index

HelloObject.php document, creating, 124–126
HelloWorld example
   analyzing, 126
   class documentation, 143–144
HelloWorld.php document, creating, 122
heredoc syntax
   comparing to nowdoc, 36
   encapsulating strings, 31–36
EOD delimiter, 32
EOT delimiter, 32
using, 31–36
Heron’s Formula, using with triangles, 188
hinting.php script
   beginning, 204
   for Iterator interface, 274
home page
   creating for CMS with OOP example, 304–307
try...catch block, 304
view, 306–307
horizontal results, displaying, 112–117
.htaccess overrides
   allowing, 67–69
AllowOverride directive, 68
Directory directive, 68
protecting directories, 70, 285
HTML tags versus XML tags, 410
HTML template
   creating, 52–56
   creating pages, 52–56
   footer file, 56
   header file, 53–54
HTMLQuickForm2
   add a page View file, 324–325
   adding pages, 322–325
   creating forms, 313, 322–323
   element types, 313
   filtering form data, 314
   HTML element types, 313
   logging out, 320–321
   login form, 312
   login View file, 318–319
login.php script, 315–318
   processing form data, 315–318
registerRule(), 315
   validating forms, 314, 322–323
   validation rules, 314
HTTP status codes, 334
httpd.conf file, opening, 68

\n
icrud interface, declaring, 192–193
IMAP port number, 333
index page
   confirming module file, 60
   creating, 57–60
   header file, 60
   main page, 57–60
   purpose, 57
   switch conditional, 59–60
   validating, 59
index.html script, beginning, 306
index.php script
   beginning, 57
   for home page, 304
inheritance. See also classes; objects; OOP (object-oriented programming)
   attributes, 150
   base class, 150
   child class, 150–151
   derived class, 150
   design, 160
   indicating in UML, 150
   “is a” relationships, 203
   members of classes, 150
   methods, 150
   parent class, 150–151
   process of, 152
   super class, 150
   terminology, 150
   using, 120, 209
inheriting
   classes, 152–156
   constructors, 157–160
   destructors, 157–160
instanceof keyword, using with classes, 152
interactive PHP CLI. See also CLI (command-line interface)
support for autocompletion, 387
using, 386–387
**interface** keyword, using, 191
**interface.php** script, beginning, 192
interfaces
versus abstract classes, 191, 196
associating classes with, 191
benefit of, 196
creating, 191
defining constructors, 194
indicating in UML, 196
meanings of, 196
versus traits, 200
using, 192–196
IP addresses, unreliability of, 342
IP geolocation
accuracy, 342
finding user’s location, 339–342
**gethostbyaddr()** function, 342
**gethostbyname()** function, 342
MaxMind option, 341
options, 341
performing, 339–342
**ip_geo.php** script, creating, 339
“is a” relationship, explained, 203
**iSort** interface, implementing, 235–236
**iSort** Strategy pattern, 239–240
**iSort.php** script, beginning, 235
Iterator design pattern
examples in SPL, 273
using, 273–277
**Iterator** interface
**current()** method, 274
**DirectoryIterator**, 277
**FilterIterator**, 277
implementing, 275–276
**key()** method, 274, 277
**LimitIterator**, 277
**next()** method, 274, 277
**rewind()** method, 274, 277
using, 274–277
**valid()** method, 274, 277

**J**
joins, optimizing, 103
JSON format, using with Web services, 348

**L**
lambdas
    calling, 27
downside, 27
using, 27–29
Last-Modified cache header type, 75, 78
LDAP port number, 333
load testing, explained, 476
local variables. See also variables
    creating for stored function, 110
    declaring, 108
login.html script, beginning, 318
login.php script
    creating, 315–318
    email address, 317
    form submission, 317
    password field, 317
validating form data, 317
logout.php script, beginning, 320

**M**
main.inc.php script, beginning, 61
max-age cache-control directive, meaning of, 75
MaxMind IP geolocation, features of, 341
MCrypt
decrypting data, 372–375
encrypting data, 367–371
using with PHP, 366
member access, controlling, 166–171. See also static members
methods
    accessors, 171
    constructors, 133–135
defining in OOP, 121
documenting, 145
establishing visibility of, 165
getters, 171
mutators, 171
overriding, 161–164, 173–174
mod_rewrite module, 285
  allowing .htaccess overrides, 67–69
  enabling URL rewriting, 71–74
  implementing, 72
  using, 67–74
Model-View-Controller (MVC), using with CMS and OOP, 284–285
modularity, use in OOP, 120
modularizing Web sites. See also Web sites
  comments, 48
  configuration file, 45–51
  content modules, 61–63
  creating database file, 45
  debugging level, 49
  email address for errors, 48
  error handling, 49–50
  explained, 44
  HTML template, 52–56
  index page, 51, 57–60
  printing error and backtrace, 50
  running script, 48
  search module, 64–66
  server-side constants, 49
  site structure, 50
multidimensional arrays. See also array() function
  adding tasks to databases, 10–16
  database-driven, 8–10
  defining, 6
  grade sorting function, 7
  name-sorting function, 6
  printing as defined, 7
  short array syntax, 2
  sorting, 4–8, 29
  for Strategy design, 240
  two-dimensional, 3–4
usort() function, 4
must-revalidate cache-control directive, meaning of, 75
mutator, explained, 171
MVC (Model-View-Controller), using with CMS and OOP, 284–285
mysql client, SHOW WARNINGS command, 99
MySQL database
  accessing, 83
  calculating distances, 103–107
N
namespace class, using, 210–211
namespace keyword, placement of, 211
__NAMESPACE__ constant, 211
namespace.php script, beginning, 210
namespaces
  defining, 207
  features of, 207
  limitations, 207
  referencing, 208, 211
  subnamespaces, 207
  using, 208–210
  using in multiple files, 211
networking
  accessing Web sites, 328–332
  classes in PEAR, 332
cURL, 343–346
  IP geolocation, 339–342
  sockets, 333–338
  Web services, 347–351
  Zend Framework classes, 332
no-cache directive, meaning of, 75
nowdoc syntax, comparing to heredoc, 36
number format, specifying printing of, 38
number2 script, creating, 396
number.php script, creating, 389
O
o type specifier, meaning of, 37
object versus class names, case-sensitivity, 126
object-oriented programming (OOP). See OOP (object-oriented programming)
objects. See also inheritance
  cloning, 197
  copying, 197
  creating from classes, 156
  creating in OOP, 124–126
  serializing, 294
  use in OOP, 120
OOP (object-oriented programming). See also classes; CMS with OOP example

$this attribute, 127–132
abstract classes, 184–190
access control, 120, 165–171
accessing class attributes, 127–132
attributes versus variables, 121
autoloading classes, 136
calling object methods, 125
classes, 120
composition, 203, 209
constructors, 133–135
controlling member access, 166–171
creating objects, 124–126
defining classes, 121–123
design approaches, 209
destructors, 136–139
encapsulation, 120, 126, 140, 166
inheritance, 120, 150, 209
installing PEAR packages, 147
interfaces, 191–196
methods, 184–190
modularity, 120
modularizing application files, 278
namespaces, 207–211
objects, 120
overriding, 120
overriding methods, 161–164
phpDocumentor, 143–147
polymorphism, 151
recommendation, 160
scope resolution operator (::), 172–175
static members, 176–181
taking actions with data, 120
taking actions with data, 120
theory, 120
traits, 197–202
type hinting, 203–206
visibility, 120, 123, 151
opcode caching, implementing, 473
overridden methods, referring to, 173–174
overriding methods, 161–164
use in OOP, 120

P
Page class
creating for CMS with OOP example, 299–301
getter methods, 300
page.html script, beginning, 311
Page.php script, beginning, 299
page.php script, beginning, 308
page-viewing page
catching exceptions, 310
Controller, 309
creating, 308–310
throwing exceptions, 310
validating page ID, 310
page-viewing View, creating, 311
parse_url() function
using with sockets, 333–334, 336
validating URLs, 338
parsing XML. See also XML (Extensible Markup Language)
changing case-folding, 439
event-based parser, 432
with Expat, 433
with PHP, 434–439
SimpleXML, 440–446
tree-based parser, 432
patterns
antipatterns, 232
behavioral, 215, 233
components, 214
Composite, 225–232
creational, 215, 225
Factory, 220–224
Gang of Four, 215
Iterator, 273–277
Singleton, 216–219
Strategy, 233–241
structural, 215, 225
pcntl (process control) extension, using with CLI, 403
PDO (PHP Data Objects)
calling quote() method, 262
catching exceptions, 259
changing error reporting, 261
PDO (continued)
  connecting to database, 258–259
  described, 258
  executing queries, 261–262
  prepared statements, 266–269
  preventing SQL injection attacks, 262
  running SELECT queries, 264
  SELECT queries, 262–263
  using, 260–261

PDO object, creating, 260

PEAR (PHP Extension and Application Repository)
  installing phpDocumentor in, 146
  networking classes, 332
  upgrading for unit testing, 461

PEAR packages, installing, 147

percent (%) sign, using in strings, 41

performance, improving, 473–475

Pet example inheritance design, 160

pets1.php script
  beginning, 153
  for overriding methods, 162

pets2.php script, for scope resolution operator (::), 172

PHP, parsing XML with, 434–439

PHP and server
  compressing files, 354–362
  establishing cron, 363–365
  MCrypt, 366–375

PHP CLI. See CLI (command-line interface)

PHP Data Objects (PDO). See PDO (PHP Data Objects)

PHP output, compressing, 362

phpDocumentor
  docblocks, 144–145
  features, 143–144
  installing, 145
  using, 146–147

PHPUnit. See also unit testing
  creating test cases, 463–465
  defining tests, 462–463
  downloading, 460
  installing, 461–462
  invoking assertion methods, 464
  running tests, 465–466

setting up tests, 467–470

setUp() method for testing, 467–468, 470

Simplestest alternative, 460

testing Rectangle class, 469

$this object, 464

upgrading PEAR, 461

phpunit command, executing, 465

polymorphism, use in OOP, 151

POP port number, 333

port numbers for sockets, 333

Pragma cache header type, 75

prepared statements
  performance benefits, 269
  try block, 266
  using through PDO, 266–269

printf() function
  formats, 37
  type specifiers, 37
  using, 37–38

printing
  backtrace, 50
  numbers and strings, 38

private cache-control directive, meaning of, 75

profile log, viewing in webgrind, 474–475

profiling scripts, 471–472

proxy server, explained, 75

proxy-revalidate cache-control directive, meaning of, 75

public cache-control directive, meaning of, 75

public variables, accessing, 169. See also variables

Q

queries, executing, 261–262

query caching, availability of, 473

query() method, using, 262

query results, displaying horizontally, 112–117

QuickForm2
  add a page View file, 324–325
  adding pages, 322–325
  creating forms, 313, 322–323
  element types, 313
  filtering form data, 314
  HTML element types, 313
logging out, 320–321
login form, 312
login View file, 318–319
login.php script, 315–318
processing form data, 315–318
registerRule(), 315
validating forms, 314, 322–323
validation rules, 314
quotation (") marks, using with classes, 152

R
read_mcrypt.php script, beginning, 372
Rectangle class
  constructor added to, 133–135
  using, 130–132, 201–202
Rectangle example, analyzing, 132
Rectangle.php script
  for constructors, 133–135
  creating, 128
  for tDebug trait, 200
recursive functions. See also static variables
  adding debugging line, 23
  adding tasks to array, 22
  calling, 21–22
  defining, 18
  foreach loop and function, 22
  looping through array, 21
  nested list of tasks, 19–20
  using, 17–23
references and functions, 30
remote server, using with PHP CLI, 384
results, displaying horizontally, 112–117
RSS feed
  Atom offshoot format, 451
  channel content, 447
  creating, 447–451
  generating, 448–449
rss.php script, beginning, 448

S
s type specifier, meaning of, 37
scanf() function, using, 41
scope resolution operator (::), using, 172–175, 177
scripts, profiling, 471–472
search module
creating, 64–66
printing caption, 66
printing results, 66
search.inc.php script, beginning, 64
SELECT queries
  executing, 262–263
  populating menu in form, 265
  running, 264
  setFetchMode() method, 262
  setting fetch mode, 264
try block, 264
semicolon (;), use with stored functions, 109
SEO, improving with mod_rewrite, 67–74
serializing objects, 294
server
  compressing files, 354–362
  establishing cron, 363–365
  MCrypt, 366–375
server caches, implementing, 473.
  See also caching
server commands, running, 374
service.php script, creating, 349
session data
  function for destruction of, 89
  storing as serialized array, 89
  storing in databases, 82, 85–87
session directory, changing for security, 82
session functions, defining, 84
session handlers
  creating, 85–91
  garbage collection function, 90
  using, 91–95
SessionHandlerInterface class, 270
sessions table, creating, 82–83
sessions.php script, beginning, 91
set and get methods, using with classes, 132
set_mcrypt.php script, beginning, 369
ShapeFactory class, using, 222–224
ShapeFactory.php script, beginning for
  Factory pattern, 220
Shape.php script, beginning, 186
short array syntax, using, 2. See also array() function
SHOW WARNINGS command, running, 99
Simpletest unit testing Web site, 460
SimpleXML
  asXML() method, 446
  using, 440–446
simplexml.php document, creating, 441
Singleton class, creating, 217–218
Singleton pattern
  Config class, 217–219
  described, 216
  implementing, 216
  UML representation, 216
sites. See also modularizing Web sites
  accessing, 328–332
  reading with PHP, 329–332
s-maxage cache-control directive,
  meaning of, 75
SMTP port number, 333
sockets
  connections, 334–335
  explained, 333
  fscanf() (fssockopen()), 333–338
FTP port, 333
GET request, 336
HEAD request, 336
HTTP status codes, 334
IMAP port, 333
LDAP port, 333
parse_url() function, 333–334, 336
POP port, 333
ports, 333
SMTP port, 333
SSH port, 333
SSL port, 333
Telnet port, 333
Web ports, 333
sort.php script
  for anonymous functions, 28
  beginning, 4
  for static variables, 24
SPL (Standard PHP Library)
  autoloading capability, 281
  autoloading classes, 278–280
  data structures, 278
exceptions, 273
file handling, 271
iterators, 273–277
SplFixedArray, 278
temporary files, 272
using, 270
SPL interfaces
  ArrayAccess, 279
  Countable, 279
SplFileObject, using, 272
SplTempFileObject, described, 272
sprintf() function
  formats, 37
  type specifiers, 37
  using, 39–41
  using with session data, 88
SQL injection attacks, preventing, 261–262
square.php script, creating, 158
SSH port number, 333
SSL port number, 333
Standard PHP Library (SPL). See SPL (Standard PHP Library)
static attributes
  versus class constants, 176
  using with static methods, 178–180
static members, creating, 178–181. See also member access
static variables. See also recursive functions; variables
  versus attributes, 177
  using, 24–26
static.php script, beginning, 178
stock quotes, retrieving, 329–332
stored functions. See also functions
  ; (semicolon) in code blocks, 109
  arguments section, 109
  changing delimiters, 109–110
  code section, 109
  creating, 109–111
  declaring, 108–112
  local variable, 110
stores table
  address selection, 101
  creating, 100–101
  populating, 101
Strategy design
  class definition, 239
  constructor, 237
  creating, 235–238
  display() method, 240
  iSort classes, 238–241
  iSort interface, 235
  multidimensional arrays, 240
  sort() method, 237–239
Strategy pattern
  described, 233
  example of, 234
  using, 233–234
strategy.php script, beginning, 238
strings. See also __toString() method
  encapsulating, 31–36
  percent signs, 41
  printing, 38
structural patterns
  explained, 215
  using, 225
subclass constructors, creating, 158–160
switch statement
  using with classes, 123
  using with index page, 59

T
  tags, HTML versus XML, 410
TDD (test-driven development), 461
tDebug trait, using, 200
Telnet port number, 333
temperature script, beginning, 400
$this attribute
  use in OOP, 127–132
  using with PHPUnit, 464
to-do list, nesting, 9
__toString() method, defining in classes, 184. See also strings
trait keyword, using, 197
trait.php script, beginning, 201
traits
  creating, 197
  incorporating into classes, 202
  versus interfaces, 200
  precedence, 202
  support for, 197
  using, 198–199
triangle
  calculating area of, 188
  setting sides of, 190
Triangle class
  creating, 186–189
  as extension of Shape, 187
  using, 189–190
Triangle.php script, beginning, 186
try block
  using with prepared statements, 266
  using with SELECT queries, 264
type hinting
  for arrays, 206
  function parameters, 15
  for functions, 206
  for interfaces, 206
  performing, 203
  triggering exceptions, 206
  using, 204–206
  using in functions, 206
type specifiers, 37

U
  u type specifier, meaning of, 37
UML (Unified Modeling Language)
  for classes, 140–142
  for composition, 203
  for inheritance, 150
  for interfaces, 196
  for visibility, 166
UML representation, of Singleton pattern, 216
unit testing. See also PHPUnit
  assertions, 462
  benefits, 460
  implementing, 460
  TDD (test-driven development), 461
  Xdebug debugging tool, 465
URL rewriting, enabling, 71–74
User class
  attributes of, 152
  creating for CMS with OOP example, 301–303
user input
  prompting for, 402
  taking in CLI, 400–404
user-defined functions
  anonymous functions, 27–29
  recursive functions, 17–23
  static variables, 24–26
User.php script, beginning, 301
usort() function, using, 4
utilities file
  catching PDO exceptions, 296
  database connection, 296
  serialize() function, 294
  serializing objects, 294
  starting session, 295
  writing, 294–296
utilities.inc.php script, beginning, 295
V
  variable declaration, documenting, 145
  variables, passing by reference, 30. See also public variables; static variables
view_tasks.php script
  beginning, 18
  for caching, 76
  header() function, 76–77
  for heredoc syntax, 32
  modifying, 33–34
visibility
  establishing for methods, 165
  importance of, 166
  indicating in UML, 166
  in OOP, 151
private level, 165–166
protected level, 165
public level, 165
restriction of, 165
visibility script, beginning, 166
Visitor pattern, using with Composite, 232
vprintf() function, using, 41
W
  warnings, showing, 99
Web port numbers, 333
Web services
  creating, 347–351
  JSON format, 348
  REST (Representational State Transfer), 347
  returning types of data, 348
  stateless, 347
  technologies related to, 410
  using, 342
Web sites. See also modularizing Web sites
  accessing, 328–332
  reading with PHP, 329–332
webgrind
  downloading, 471
  installing, 471
  loading in browser, 474
  using Xdebug with, 471–472
  viewing profile log in, 474–475
write_to_file.php script, opening, 272
X
  x type specifier, meaning of, 37
Xdebug debugging tool
  checking code coverage, 465
  customizing, 458
  downloading, 454
  ini_set() function, 458
  installation requirements, 454
  installing on Windows, 455–456
  on *nix systems, 454
  profiling in, 471
  using, 457–459
  using with webgrind, 471–472
XML (Extensible Markup Language). See also parsing XML
  adding books to file, 414
  &amp entity, 415
  &apos; entity, 415
  attributes, 415–418
  benefit, 424
  elements, 415–418
  entities, 415–418
Index

XSD document
complex types, 427
creating attributes, 427
defining elements, 426
incorporating, 426
mixed attribute on elements, 431
simple types, 427
using, 425
xmlns attribute, 425

Z
Zend Framework, network-related classes, 332
zip codes. See also databases
database, 96–99
importing data, 98
tables, 96
zlib
a+ mode, 355
b mode, 355
compressed binary files, 362
compressing files with, 354–362
f mode, 355
file open modes, 355
h mode, 355
a mode, 355
r mode, 355
r+ mode, 355
verifying support for, 354
w mode, 355
w+ mode, 355
x mode, 355
ZIP archives, 362