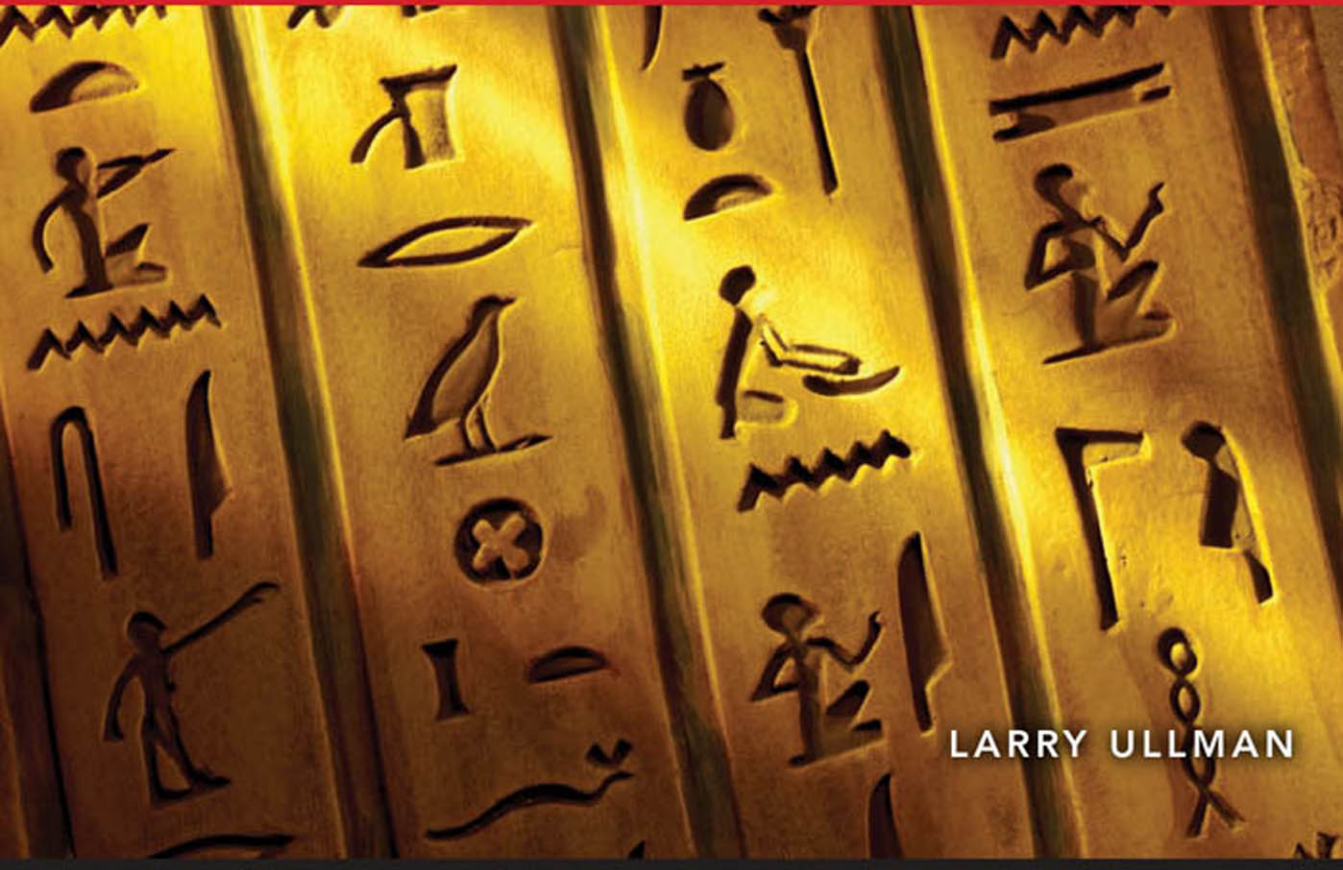
VISUAL QUICKPRO GUIDE

INCLUDES eBOOK

PHP Advanced and Object-Oriented Programming

Third Edition



LARRY ULLMAN

TWO WAYS TO LEARN—PRINT & eBOOK!

VISUAL QUICKPRO GUIDE

PHP Advanced and Object-Oriented Programming

LARRY ULLMAN

Peachpit Press

Visual QuickPro Guide

PHP Advanced and Object-Oriented Programming

Larry Ullman

Peachpit Press
1249 Eighth Street
Berkeley, CA 94710

Find us on the Web at: www.peachpit.com
To report errors, please send a note to: errata@peachpit.com
Peachpit Press is a division of Pearson Education.

Acquisitions Editor: Rebecca Gulick
Production Coordinator: Myrna Vladic
Copy Editor: Liz Welch
Technical Reviewer: Alan Solis
Compositor: Danielle Foster
Proofreader: Patricia Pane
Indexer: Valerie Haynes Perry
Cover Design: RHDG / Riezebos Holzbaur Design Group, Peachpit Press
Interior Design: Peachpit Press
Logo Design: MINE™ www.minesf.com

13-digit ISBN: 978-0-321-83218-4
10-digit ISBN: 0-321-83218-3

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

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

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/`**) 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.

4

Basic Object-Oriented Programming

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

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:

```
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:

```
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**:

```
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:

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

```
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 # Script 4.1 - HelloWorld.php
   ```

2. Begin defining the class:

   ```
   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.

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.)

```
1   <?php # Script 4.1 - HelloWorld.php
2   /*  This page defines the HelloWorld
    →class.
3    *  The class says "Hello, world!" in
     →different languages.
4    */
5   class HelloWorld {
6
7      // This method prints a greeting.
8      // It takes one argument: the
       →language to use.
9      // Default language is English.
10     function sayHello($language =
       →'English') {
11
12        // Put the greeting within P tags:
13        echo '<p>';
14
15        // Print a message specific to a
          →language:
16        switch ($language) {
17           case 'Dutch':
18              echo 'Hallo, wereld!';
19              break;
20           case 'French':
21              echo 'Bonjour, monde!';
22              break;
23           case 'German':
24              echo 'Hallo, Welt!';
25              break;
26           case 'Italian':
27              echo 'Ciao, mondo!';
28              break;
29           case 'Spanish':
30              echo '¡Hola, mundo!';
31              break;
32           case 'English':
33           default:
34              echo 'Hello, world!';
35              break;
36        } // End of switch.
37
38        // Close the HTML paragraph:
39        echo '</p>';
40
41        } // End of sayHello() method.
42
43  } // End of HelloWorld class.
```

3. Begin defining the first (and only) method:

```
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:

```
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**:

```
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;
  case 'English':
  default:
    echo 'Hello, world!';
    break;
} // End of switch.
```

The **switch** prints different messages based upon the chosen language. English is the default language, both in the **switch** and as the value of the **$language** argument (see Step 3). Obviously you can easily expand this **switch** to include more languages, like non-Western ones.

6. Complete the **sayHello()** method:

```
  echo '</p>';
} // End of sayHello() method.
```

You just need to close the HTML paragraph tag.

7. Complete the class and the PHP page:

```
}
```

8. Save the file as **HelloWorld.php**.

You've now created your first class. This isn't, to be clear, a *good* use of OOP, but it starts the process and you'll learn better implementations of the concept in due time.

Note that I'm not using a closing PHP tag, which is my policy for PHP scripts to be included by other files.

TIP **Class methods can also have a *visibility*, by preceding the function definition with the appropriate keyword. If not stated, all methods have an assumed definition of**

```
public function functionName() {...
```

TIP **The class `stdClass` is already in use internally by PHP and cannot be declared in your own code.**

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**:

```
$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:

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

(The **->** can be called the *object operator*.)

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.

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**):

```
<!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.

Script 4.2 In this page, PHP uses the defined class in order to say *Hello, world!* in several different languages.

```
1   <!doctype html>
2   <html lang="en">
3   <head>
4      <meta charset="utf-8">
5      <title>Hello, World!</title>
6      <link rel="stylesheet"
       → href="style.css">
7   </head>
8   <body>
9   <?php # Script 4.2 - hello_object.php
10  /* This page uses the HelloWorld class.
11   * This page just says "Hello, world!".
12   */
13
14  // Include the class definition:
15  require('HelloWorld.php');
16
17  // Create the object:
18  $obj = new HelloWorld();
19
20  // Call the sayHello() method:
21  $obj->sayHello();
22
23  // Say hello in different languages:
24  $obj->sayHello('Italian');
25  $obj->sayHello('Dutch');
26  $obj->sayHello('French');
27
28  // Delete the object:
29  unset($obj);
30  ?>
31  </body>
32  </html>
```

2. 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).

3. 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.

4. 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.

5. Say hello in a few more languages:

   `$obj->sayHello('Italian');`

   `$obj->sayHello('Dutch');`

   `$obj->sayHello('French');`

   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/phpvqp3/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:

```
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 B.

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.

B The resulting Web page (the examples will get better, I promise).

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 # Script 4.3 - Rectangle.php
```

2. Begin defining the class:

```
class Rectangle {
```

3. Declare the attributes:

```
public $width = 0;

public $height = 0;
```

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:

```
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 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.

```
1   <?php # Script 4.3 - Rectangle.php
2   /*  This page defines the Rectangle
    → class.
3    *  The class contains two attributes:
     → width and height.
4    *  The class contains four methods:
5    *  - setSize()
6    *  - getArea()
7    *  - getPerimeter()
8    *  - isSquare()
9    */
10
11  class Rectangle {
12
13     // Declare the attributes:
14     public $width = 0;
15     public $height = 0;
16
17     // Method to set the dimensions:
18     function setSize($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 *
          → $this->height);
26     }
27
28     // Method to calculate and return
       → the perimeter.
29     function getPerimeter() {
30        return ( ($this->width +
          → $this->height) * 2 );
31     }
32
33     // Method to determine if the
       → rectange
34     // is also a square.
35     function isSquare() {
36        if ($this->width ==
          → $this->height) {
37           return true; // Square
```

script continues on next page

Script 4.3 *continued*

```
38          } else {
39             return false; // Not a square
40          }
41       }
42
43    } // End of Rectangle class.
```

5. Create a method that calculates and returns the rectangle's area:

```
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:

```
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:

```
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:

```
} // 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**):

```
<!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:

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

3. Define the necessary variables and print an introduction:

```
$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:

```
$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:

```
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:

```
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:

```
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:

```
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

Script 4.4 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.

```
<!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
/*  This page uses the Rectangle class.
 *  This page shows a bunch of information about a rectangle.
 */

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

// Define the necessary variables:
$width = 42;
$height = 7;

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

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

// Assign the rectangle dimensions:
$r->setSize($width, $height);

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

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

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

// Delete the object:
unset($r);

?>
</body>
</html>
```

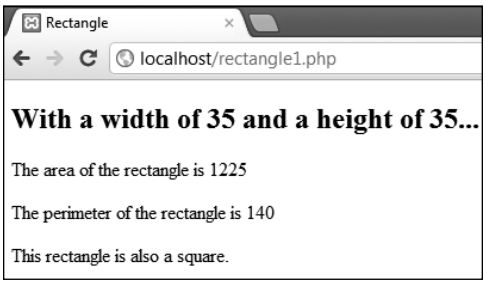
10. Test **rectangle1.php** by viewing it in your Web browser Ⓐ.
11. Change the variables' values in **rectangle1.php** and rerun it in your Web browser Ⓑ.

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:

```
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

```
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:

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

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

continues on next page

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   /* This page defines the Rectangle class.
3    * The class contains two attributes: width and height.
4    * The class contains five methods:
5    * - __construct()
6    * - setSize()
7    * - getArea()
8    * - getPermeter()
9    * - isSquare()
10   */
```

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**):

```
$width = 160;
$height = 75;
```

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

```
$r = new Rectangle($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 4.5 *continued*

```
11
12  class Rectangle {
13
14      // Declare the attributes:
15      public $width = 0;
16      public $height = 0;
17
18      // Constructor:
19      function __construct($w = 0,
        → $h = 0) {
20          $this->width = $w;
21          $this->height = $h;
22      }
23
24      // Method to set the dimensions:
25      function setSize($w = 0, $h = 0) {
26          $this->width = $w;
27          $this->height = $h;
28      }
29
30      // Method to calculate and return
        → the area:
31      function getArea() {
32          return ($this->width *
            → $this->height);
33      }
34
35      // Method to calculate and return
        → the perimeter:
36      function getPerimeter() {
37          return ( ($this->width +
            → $this->height) * 2 );
38      }
39
40      // Method to determine if the
        → rectange
41      // is also a square.
42      function isSquare() {
43          if ($this->width == $this->height)
{
44              return true; // Square
45          } else {
46              return false; // Not a square
47          }
48
49      }
50
51  } // End of Rectangle class.
```

Script 4.6 This new version of the script assigns the rectangle's dimensions when the object is created (thanks to the constructor).

```
1   <!doctype html>
2   <html lang="en">
3   <head>
4      <meta charset="utf-8">
5      <title>Rectangle</title>
6      <link rel="stylesheet"
       → href="style.css">
7   </head>
8   <body>
9   <?php # Script 4.6 - rectangle2.php
10  /* This page uses the revised Rectangle
    → class.
11   * This page shows a bunch of
     → information
12   * about a rectangle.
13   */
14
15  // Include the class definition:
16  require('Rectangle.php');
17
18  // Define the necessary variables:
19  $width = 160;
20  $height = 75;
21
22  // Print a little introduction:
23  echo "<h2>With a width of $width and a
    → height of $height...</h2>";
24
25  // Create a new object:
26  $r = new Rectangle($width, $height);
27
28  // Print the area.
29  echo '<p>The area of the rectangle
    → is ' . $r->getArea() . '</p>';
30
31  // Print the perimeter.
32  echo '<p>The perimeter of the rectangle
    → is ' . $r->getPerimeter() . '</p>';
33
34  // Is this a square?
35  echo '<p>This rectangle is ';
36  if ($r->isSquare()) {
37     echo 'also';
38  } else {
39     echo 'not';
40  }
41  echo ' a square.</p>';
42
43  // Delete the object:
44  unset($r);
45
46  ?>
47  </body>
48  </html>
```

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 Ⓐ.

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:**

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

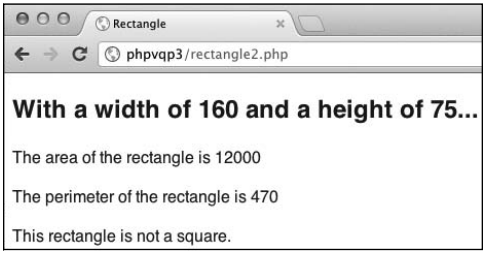
TIP **You can directly call a constructor (although you will rarely need to):**

```
$o = new SomeClass();
$o->__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).**



Ⓐ 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:

```
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

```
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**):

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Constructors and
  → Destructors</title>
  <link rel="stylesheet"
  → href="style.css">
</head>
<body>
<?php # Script 4.7 - demo.php
```

2. Begin defining the class:

```
class Demo {
```

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

3. Create the constructor:

```
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:

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

5. Complete the class:

```
}
```

   It's a very simple class!

continues on next page

Script 4.7 This script doesn't do anything except best convey when constructors and destructors are called.

```
1   <!doctype html>
2   <html lang="en">
3   <head>
4      <meta charset="utf-8">
5      <title>Constructors and Destructors</title>
6      <link rel="stylesheet" href="style.css">
7   </head>
8   <body>
9   <?php # Script 4.7 - demo.php
10  /*  This page defines a Demo class
11   *  and a demo() function.
12   *  Both are used to show when
13   *  constructors and destructors are called.
14   */
15
```

script continues on next page

6. Define a simple function that also creates an object:

```
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**:

```
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:

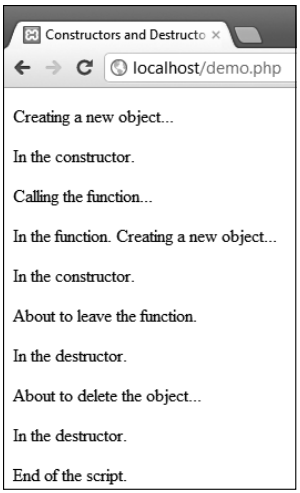
```
echo '<p>Calling the function...
→ </p>';

test();
```

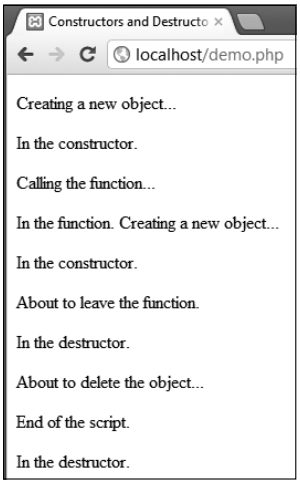
After printing the status statement, the function is called. Consequently, the function is entered, wherein *In the function. Creating a new object...* will first be printed. Then, in that function, a new object is created (called **$f**). Therefore, the constructor will be called again, and the *In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

Script 4.7 *continued*

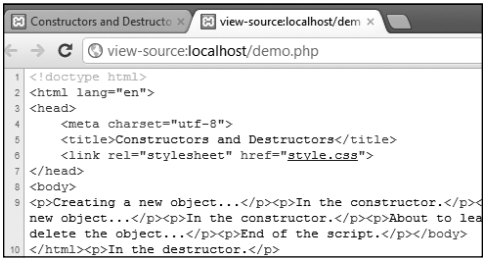
```
16  // Define the class:
17  class Demo {
18
19      // No attributes.
20
21      // Constructor:
22      function __construct() {
23          echo '<p>In the constructor.</p>';
24      }
25
26      // Destructor:
27      function __destruct() {
28          echo '<p>In the destructor.</p>';
29      }
30
31  } // End of Demo class.
32
33  // Define a test() function:
34  function test() {
35      echo '<p>In the function. Creating a
        → new object...</p>';
36      $f = new Demo();
37      echo '<p>About to leave the
        → function.</p>';
38  }
39
40  // Create the object:
41  echo '<p>Creating a new object...</p>';
42  $o = new Demo();
43
44  // Call the test() function:
45  echo '<p>Calling the function...</p>';
46  test();
47
48  // Delete the object:
49  echo '<p>About to delete the object...
    → </p>';
50  unset($o);
51
52  echo '<p>End of the script.</p>';
53  ?>
54  </body>
55  </html>
```



A The flow of the two objects' creation and destruction over the execution of the script is revealed by this script. In particular, you can see how the `test()` function's object, `$f`, lives and dies in the middle of this script.



B If you don't forcibly delete the object A, it will be deleted when the script stops running. This means that the `$o` object's destructor is called after the final printed message, even after the closing HTML tags C.



C The `$o` object's destructor is called as the very last script event, when the script stops running. Thus, the *In the destructor.* message gets sent to the browser after the closing HTML tag.

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 `$o` object:

```
echo '<p>About to delete the
→ object...</p>';

unset($o);
```

Once this object is deleted, its destructor is invoked.

10. Complete the page:

```
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
```

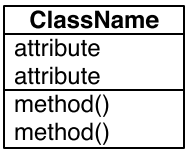
If the attribute had a default value, you could reflect that too:

```
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`**.

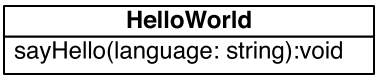


Ⓐ How UML represents a class graphically.

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.



B A UML representation of the simple `HelloWorld` class.

With this in mind, you can complete the class diagram for the `HelloWorld` class B. 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:

   ```
   __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 C.

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.

Rectangle
width:number = 0 height:number = 0
__construct(width:number = 0, height:number = 0):void setSize(width:number = 0, height:number = 0):void getArea():number getPerimeter():number isSquare():Boolean

C A UML representation of the simple **`Rectangle`** class.

Better Documentation with phpDocumentor

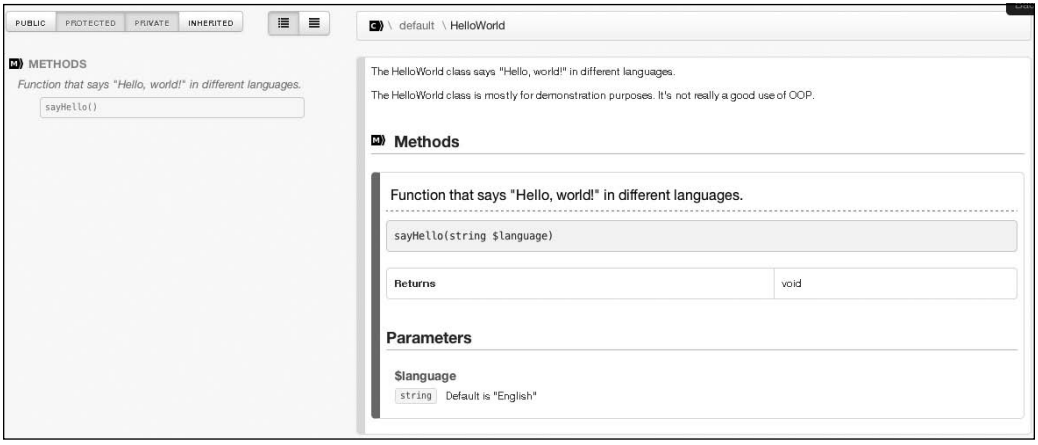
Along with creating a UML class design, another new topic in this edition is creating better code documentation using phpDocumentor (**`www.phpdoc.org`**).

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.



Ⓐ 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:

```
/**
 *
 * 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.

```
1   <?php # Script 4.8 - HelloWorld.php #2
2   /**
3    * This page defines the HelloWorld class.
4    *
5    * Written for Chapter 4, "Basic Object-Oriented Programming"
6    * of the book "PHP Advanced and Object-Oriented Programming"
7    * @author Larry Ullman <Larry@LarryUllman.com>
8    * @copyright 2012
9    */
10
11  /**
12   * The HelloWorld class says "Hello, world!" in different languages.
13   *
14   * The HelloWorld class is mostly for
15   * demonstration purposes.
16   * It's not really a good use of OOP.
17   */
18  class HelloWorld {
19
20      /**
21       * Function that says "Hello, world!" in different languages.
22       * @param string $language Default is "English"
23       * @returns void
24       */
```

script continues on next page

Script 4.8 *continued*

```
25      function sayHello($language =
        → 'English') {
26
27          // Put the greeting within P tags:
28          echo '<p>';
29
30          // Print a message specific to a
            → language:
31          switch ($language) {
32              case 'Dutch':
33                  echo 'Hallo, wereld!';
34                  break;
35              case 'French':
36                  echo 'Bonjour, monde!';
37                  break;
38              case 'German':
39                  echo 'Hallo, Welt!';
40                  break;
41              case 'Italian':
42                  echo 'Ciao, mondo!';
43                  break;
44              case 'Spanish':
45                  echo '¡Hola, mundo!';
46                  break;
47              case 'English':
48              default:
49                  echo 'Hello, world!';
50                  break;
51          } // End of switch.
52
53          // Close the HTML paragraph:
54          echo '</p>';
55
56          } // End of sayHello() method.
57
58  } // End of HelloWorld class.
```

To document a variable declaration, you use the **@var** tag, followed by the variable's type (and optional description):

```
/**
 * @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/**).

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**), 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.

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 ❶:

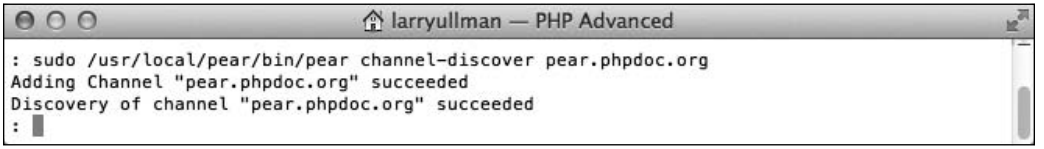
   ```
   pear channel-discover
   → pear.phpdoc.org
   ```

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

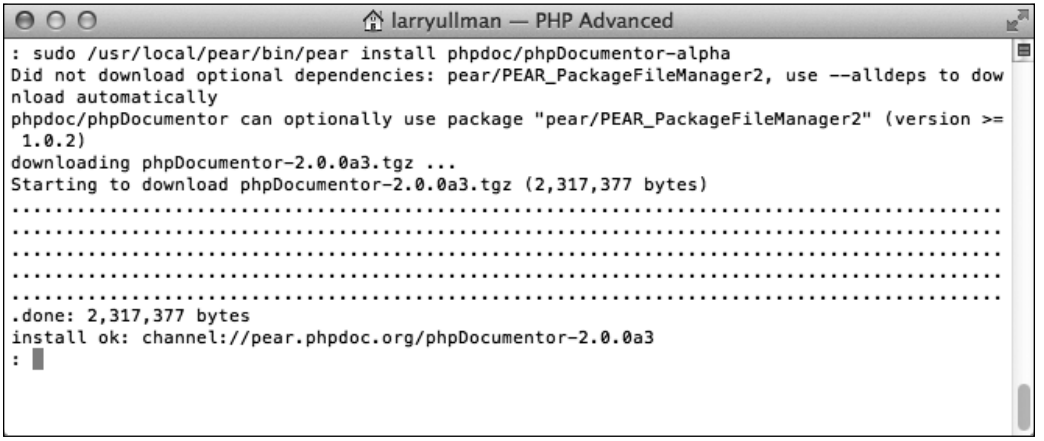
4. Install phpDocumentor ❷:

   ```
   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.



❶ Adding the phpDocumentor channel to my PEAR installation.



❷ Installing phpDocumentor in PEAR.

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:

   ```
   cd /path/to/folder
   ```

6. Document a single file using

   ```
   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 A.

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

```
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

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z