PHP Object-Oriented Solutions

David Powers

friendsof
DESIGNER TO DESIGNER™
an Apress® company

PHP Object-Oriented Solutions

ISBN-13 (pbk): 978-1-4302-1011-5

ISBN-13 (electronic): 978-1-4302-1012-2

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

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

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

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

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor
Ben Renow-Clarke

Technical Reviewer
Seungyeob Choi

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

Project Manager
Beth Christmas

Copy Editors
Heather Lang and Damon Larson

Associate Production Director
Kari Brooks-Copony

Production Editor
Laura Esterman

Compositor
Molly Sharp

Proofreader
Patrick Vincent

Indexer
Toma Mulligan

Artist
April Milne

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

CONTENTS AT A GLANCE

CONTENTS

ABOUT THE AUTHOR

David Powers is the author of a series of highly successful books on PHP, including *PHP Solutions: Dynamic Web Design Made Easy* (friends of ED, ISBN: 978-1-59059-731-6) and *The Essential Guide to Dreamweaver CS3 with CSS, Ajax, and PHP* (friends of ED, ISBN: 978-1-59059-859-7). As a professional writer, he has been involved in electronic media for more than 30 years, first with BBC radio and television, both in front of the microphone (he was a BBC correspondent in Tokyo from 1987 to 1992) and in senior editorial positions. His clear writing style is valued not only in the English-speaking world—several of his books have been translated into Spanish and Polish.

Since leaving the BBC to work independently, David has devoted most of his time to web development, writing books, and teaching. He is active in several online forums, giving advice and troubleshooting PHP problems. David's expertise was recognized by his designation as an Adobe Community Expert in 2006.

When not pounding the keyboard writing books or dreaming of new ways of using PHP and other programming languages, David enjoys nothing better than visiting his favorite sushi restaurant. He has also translated several plays from Japanese.

ABOUT THE TECHNICAL REVIEWER

Seungyeob Choi is the lead developer and technology manager at Abraham Lincoln University in Los Angeles, where he has been developing various systems for online education. He built the university's learning platform and has been working on a development project for Student Lifecycle Management. Seungyeob has a PhD in computer science from the University of Birmingham, England.

ACKNOWLEDGMENTS

The book you're holding in your hand (or reading on the screen) owes its genesis to a tongue-in-cheek exchange with Steve Fleischer of Flying Tiger Web Design (`www.flyingtigerwebdesign.com`), who suggested I should write *Powers Object-Oriented PHP*. Actually, he phrased it rather differently. If you take the initial letters of the suggested title, you'll get the drift . . . But Steve had an important point: he felt that books on object-oriented programming (OOP) frequently assumed too much prior knowledge or weren't easily adaptable to PHP in a practical way. If you like what you find in this book, thank Steve for planting the idea in my brain. If you don't like it, blame me, because I'm the one responsible for writing it the way it is.

Thanks must also go to everyone at Apress/friends of ED for helping bring "my baby" into the world. Books are uncannily like real babies. This one took exactly nine months from conception to birth with the expert help of editor Ben Renow-Clarke, project manager Beth Christmas, and many other "midwives." I owe a particular debt of gratitude to Seungyeob Choi for his perceptive technical review. Seungyeob's eagle eye and deep knowledge of PHP and OOP saved me from several embarrassing mistakes. Any remaining errors are my responsibility alone.

I would also like to thank everyone who has supported me by buying this or any of my previous books. I realize not everyone can afford to buy books, but the royalties from new—not second-hand—books ensure that authors get some reward for all the hard effort that goes into writing. Even the most successful computer books can never aspire to the stratospheric heights of Harry Potter, so every little bit helps—and is much appreciated.

The biggest thanks of all must undoubtedly go to the developers of PHP, who have given the rest of the world a superb programming language that continues to go from strength to strength.

INTRODUCTION

My first experiments with object-oriented programming in PHP took place about six years ago. Unfortunately, the book that introduced me to the subject concentrated on the mechanics of writing classes and paid little heed to principles underlying OOP. As a result, I wrote classes that were closely intertwined with a specific project ("tightly coupled," to use the OOP terminology). Everything worked exactly the way I wanted, but the design had a fundamental flaw: the classes couldn't be used for any other project. Worse still, it was a large project—a bilingual, searchable database with more than 15,000 records—so any changes I wanted to make to it involved revising the whole code base.

The purpose of this book is to help you avoid the same mistake. Although most chapters revolve around mini-projects, the classes they use are project-neutral. Rather than being a "how to" cookbook, the aim is to help developers with a solid knowledge of PHP basics add OOP to their skill set.

So, what is OOP? To oversimplify, OOP groups together functions (known in OOP-speak as "methods") in classes. In effect, a class can be regarded as a function library. What makes OOP more powerful is the fact that classes can be extended to add new functionality. Since many of the new features added to PHP 5 are object-oriented, this means you can easily extend core PHP classes to add new functionality or simply make them work the way you want them to. In fact, Chapter 3 does precisely that: it extends the PHP `DateTime` class to make it easier to use. The project in Chapter 4 takes the PHP filter functions and hides them behind a much more user-friendly interface.

Chapter 5 shows how to create a class that retrieves a text file from a remote server by automatically detecting the most efficient available method. Chapters 6 and 7 cover two of the most important OOP features added to core PHP in version 5: SimpleXML and the Standard PHP Library (SPL). The XML theme continues in the final two chapters, which use the PHP `XMLWriter` class to generate XML on the fly from a database and show you how to create a news feed from your site.

The need for OOP has come about because PHP is being used increasingly for large-scale web applications. Object-oriented practices break down complex operations into simple units, each responsible for a defined task. This makes code much easier to test and maintain. However, ease of maintenance is just as important in small-scale projects, so OOP can play a

role in projects of any size. This is an introductory book, so the object-oriented solutions it contains are designed for use in small projects, but the principles they demonstrate apply equally to large-scale projects.

By the time you have finished this book, you should understand what OOP is and how to write PHP classes that conform to current best practices, making your code easier to maintain and deploy across multiple projects. The information contained in this book will also provide a solid foundation for anyone planning to use an object-oriented framework, such as the Zend Framework (`www.zend.com/en/community/framework`).

Although everything in this book is devoted to OOP, it's important to emphasize that OOP is only *part* of PHP. OOP helps you create portable, reusable code. Use it where appropriate, but there's no need to throw out all of your existing PHP skills or code.

Another important thing to emphasize is that all the code in this book requires a minimum of PHP 5, and preferably PHP 5.2 or 5.3. It has also been designed to work in PHP 6. *The code will not work in PHP 4*, nor will any support be provided for converting it to PHP 4. Even though at the time of publication, it's estimated that more than half of all PHP-driven websites still run on PHP 4, all support for PHP 4 officially ended on August 8, 2008. PHP 4 is dead. Long live PHP 5 (and PHP 6 when it's released). If you haven't yet made the switch from PHP 4, now is the time to do it.

Who should read this book

If you develop in PHP, but haven't yet got your feet wet with OOP, this is the book for you. No previous knowledge of OOP is necessary: Chapter 1 covers the basic theory and explains how OOP fits into PHP; Chapter 2 then goes into the mechanics of writing object-oriented code in PHP. The remaining seven chapters put all the theory into practice, showing you how to create and use your own classes and objects, as well as covering object-oriented features that have been built into core PHP since version 5.

You don't need to be a PHP expert to follow this book, but you do need to know the basics of writing your own PHP scripts. So, if you're comfortable with concepts such as variables, loops, and arrays, and have ever created a function, you should be fine. Throughout the book, I make extensive use of core PHP functions. In some cases, such as with the filter functions in Chapter 4, I go into considerable detail about how they work, because that knowledge is essential to understanding the chapter. Most of the time, though, I explain what the function is for and why I'm using it. If you want a more in-depth explanation, I expect you to look it up for yourself in the PHP online documentation at `http://docs.php.net/manual/en/`.

The book aims to be a gentle introduction to OOP in PHP, but it moves at a fairly fast pace. The code involved isn't particularly difficult, but it might take a little more time for some of the concepts to sink in. The best way to achieve this is to roll up your sleeves and start coding. Exercises at strategic points demonstrate what a particular section of code does and help reinforce understanding.

Using the download code

All the files necessary to work with this book can be downloaded from the friends of ED website by going to www.friendsofed.com/downloads.html and scrolling down to the link for *PHP Object-Oriented Solutions*. Download the ZIP file, and unzip its contents into a new folder inside your web server document root. I named the folder OopSolutions, but you can call it whatever you want. In addition to a series of folders named ch2_exercises through ch9_exercises, the folder should contain the following:

- Ch2: This contains example class definitions for use with ch2_exercises.
- class_docs: This contains full documentation in HTML format for all the classes developed in the book. Double-click index.html to view them in your browser.
- finished_classes: This contains a full set of completed class definitions.
- Pos: *This folder is empty*. It is where you should create your own versions of the class definitions as you work through each chapter. If you don't want to type out everything yourself, you need to copy each class definition from finished_classes to this folder for the files in the exercise folders for each chapter to work.

Understanding the file numbering system

Most download files have a filename ending in an underscore and a number before the .php filename extension (e.g., Book_01.php, Book_02.php). This is because the files represent a class definition or exercise at a particular stage of development.

If you are typing out the exercises and class definitions yourself, leave out the underscore and number (e.g., use Book.php instead of Book_01.php). Throughout the text, I indicate the number of the current version so you can compare the appropriate supplied version with your own, or simply use it directly if you don't want to type everything yourself.

To get the best out of this book, I strongly urge you to type out all the exercises and class definitions yourself. It's a lot of work, but hands-on practice really does reinforce the learning process.

What to do if things go wrong

Every effort has been made to ensure accuracy, but mistakes do slip through. If something doesn't work the way you expect, your first port of call should be www.friendsofed.com/book.html?isbn=9781430210115. A link to any known corrections since publication will be posted there. If you think you have found a mistake that's not listed, please submit an error report to www.friendsofed.com/errataSubmission.html. When friends of ED has finished with the thumbscrews and forced me to admit I'm wrong, we'll post the details for everyone's benefit on the friends of ED site.

If the answer isn't on the corrections page, scan the chapter subheadings in the table of contents, and try looking up a few related expressions in the index. Also try a quick search

through Google or one of the other large search engines. My apologies if all this sounds obvious, but an amazing number of people spend more time waiting for an answer in an online forum than it would take to go through these simple steps.

If you're still stuck, visit `www.friendsofed.com/forums/`. Use the following guidelines to help others help you:

- Always check the book's corrections page first. The answer may already be there.
- Search the forum to see if your question has already been answered.
- Give your message a meaningful subject line. It's likely to get a swifter response and may help others with a similar problem.
- Give the name of the book and a page reference to the point that's giving you difficulty.
- "It doesn't work" gives no clue as to the cause. "When I do so and so, x happens" is a lot more informative.
- If you get an error message, say what it contains.
- Be brief and to the point. Don't ask half a dozen questions at once.
- It's often helpful to know your operating system and which version of PHP you're using.
- Don't post the same question simultaneously in several forums. If you find the answer elsewhere, have the courtesy to close the forum thread and post a link to the answer.

Please be realistic in your expectations when asking for help in a free online forum. I'm delighted if you have bought one of my books and will try to help you if you run into problems; but I'm not always available and can't offer unlimited help. If you post hundreds of lines of code, and expect someone else to scour it for mistakes, don't be surprised if you get a rather curt answer or none at all. And if you do get the help that you need, keep the community spirit alive by answering questions that you know the answer to.

Layout conventions

To keep this book as clear and easy to follow as possible, the following text conventions are used throughout.

Important words or concepts are normally highlighted on the first appearance in **bold type**.

Code is presented in `fixed-width font`.

New or changed code is normally presented in **`bold fixed-width font`**.

Pseudocode and variable input are written in *`italic fixed-width font`*.

Menu commands are written in the form Menu ➤ Submenu ➤ Submenu.

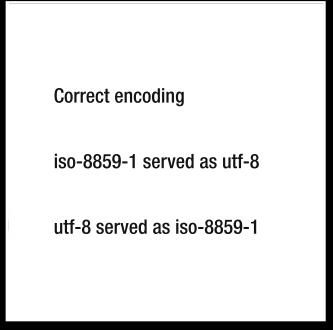
Where I want to draw your attention to something, I've highlighted it like this:

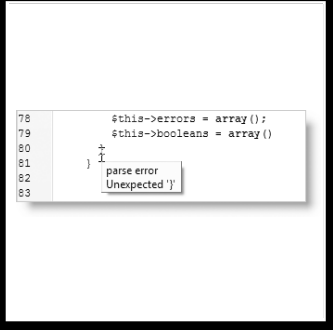
Ahem, don't say I didn't warn you.

Sometimes code won't fit on a single line in a book. Where this happens, I use an arrow like this: ➥.

```
This is a very, very long section of code that should be written all ➥
on the same line without a break.
```

1 WHY OBJECT-ORIENTED PHP?





Let's get things straight right from the start: PHP (PHP Hypertext Preprocessor) is not an object-oriented language, but it does have extensive object-oriented features. These underwent comprehensive revision and enhancement when PHP 5 was released in July 2004, and the PHP 5 object-oriented programming (OOP) model remains essentially unchanged in PHP 6. The purpose of this book is to help you leverage those features to make your code easier to reuse in a variety of situations. I assume you're familiar with basic PHP concepts, such as variables, arrays, and functions. If you're not, this isn't the book for you—at least not yet. I suggest you start with a more basic one, such as my *PHP Solutions: Dynamic Web Design Made Easy* (friends of ED, ISBN13: 978-1-59059-731-6).

> *The techniques and code used in this book require PHP 5 or PHP 6. They will* not *work with PHP 4.*

In this introductory chapter, you'll learn about the following topics:

- How OOP evolved and the thinking behind it
- What an object is and how it differs from a class
- What terms such as encapsulation, inheritance, and polymorphism really mean
- How the object-oriented model has developed in PHP
- Which tools make it easier to work with classes in PHP

I don't intend to bombard you with dense theory. The emphasis will be on gaining practical results with a minimum of effort. If you're lazy or in a hurry, you can just use the PHP classes in the download files (available from `www.friendsofed.com/downloads.html`) and incorporate them into your own scripts. However, you'll get far more out of this book if you type out the code yourself, and follow the description of how each section works and fits into the overall picture.

The techniques taught in this book are intended to improve the way you work with PHP, not replace everything you've learned so far. However, should you decide to delve deeper into OOP, they lay a solid foundation for further study. You'll find the knowledge in this book indispensible if you intend to use a PHP framework, such as the Zend Framework (`www.zend.com/en/community/framework`). Although frameworks take a lot of the hard work out of writing code, without a working knowledge of OOP, you'll be completely lost.

So what is OOP, and how does it fit into PHP?

Understanding basic OOP concepts

Object-oriented programming (OOP) is one of those great buzzwords that tend to mystify or intimidate the uninitiated. Part of the mystique stems from the fact that OOP was originally the preserve of graduates in computer science—a mystique deepened by concepts with obscure sounding names, such as encapsulation, polymorphism, and loose coupling. But OOP is finding its way increasingly into web development. ActionScript 3, the

language behind Adobe Flash and Flex, is a fully fledged OOP language, and the many JavaScript frameworks, like jQuery (`http://jquery.com/`) and script.aculo.us (`http://script.aculo.us`), that have recently become so popular—although not 100 percent OOP—make extensive use of objects.

In spite of all the high sounding words, the underlying principles of OOP are very simple. To begin with, let's take a look at why OOP developed.

How OOP evolved

Object-oriented programming traces its roots back to the 1960s, when computer programmers realized that increasingly complex programs were becoming harder to maintain. Programs sent a series of instructions to the computer to be processed sequentially, in much the same way as PHP is usually written. This approach—known as **procedural programming**—works fine for short, simple scripts, but once you get beyond more than a few hundred lines of code, it becomes increasingly difficult to spot mistakes. If you make a change to part of the program's logic, you need to ensure that the same change is reflected throughout.

The answer was to break up long, procedural code into discrete units of programming logic. In many ways, this is similar to creating custom functions. However, OOP takes things a step further by removing all functions from the main script, and grouping them in specialized units called classes. The code inside the class does all the dirty work—the actual manipulation of data—leaving the main script like a set of high-level instructions. To take a common example that will be familiar to PHP developers, before using input from an online form, you need to make sure it doesn't contain anything that could be used to sabotage your database or relay spam. The procedural approach looks at the specific project, and writes tailor-made code, usually a series of conditional statements designed to check each input field in turn. For instance, this sort of code is commonly used to make sure a username is the right length:

```
if (strlen($username) < 6 || strlen($username) > 12) {
  $error['username'] = 'Username must be between 6 and 12 characters';
}
```

OOP looks at programming in a more generic way. Instead of asking "How do I validate *this* form?" the object-oriented approach is to ask "How do I validate *any* form?" It does so by identifying common tasks and creating generic functions (or methods, as they're called in OOP) to handle them. Checking the length of text is one such task, so it makes sense to have a method that checks the length of any input field and automatically generates the error message. The method definition is tucked away inside the class file, leaving something like this in the main script:

```
$val->checkTextLength('username', 6, 12);
```

At this stage, don't worry about what the code looks like or how it works (this object-oriented approach to input validation is explained fully in Chapter 4). Don't worry about the terms, class, and method, either; they will be described shortly.

The approach taken by OOP has two distinct advantages, namely:

- **Code reusability**: Breaking down complex tasks into generic modules makes it much easier to reuse code. Class files are normally separate from the main script, so they can be quickly deployed in different projects.
- **Easier maintenance and reliability**: Concentrating on generic tasks means each method defined in a class normally handles a single task. This makes it easier to identify and eliminate errors. The modular nature of code stored outside the main script means that, if a problem does arise, you fix it in just one place. Once a class has been thoroughly tried and tested, you can treat it like a black box, and rely on it to produce consistent results.

This makes developing complex projects in teams a lot easier. Individual developers don't need to concern themselves with what happens inside a particular unit; all that matters is that it produces the expected result.

So, how's it done? First, let's take a look at the basic building blocks of OOP: classes and objects.

> *See* `http://en.wikipedia.org/wiki/Object-oriented_programming` *for a more detailed background to OOP.*

Using classes and objects

Many computer books begin explaining OOP by using a car as an example of an **object**, describing the number of wheels or color of the bodywork as typical properties, and accelerate or brake as methods. Although this is a conceptually appealing way of illustrating some basic OOP terminology, it has nothing to do with building a web site, which involves practical things such as processing forms, validating input, and so on.

So, forget all about cars—and even objects—for the moment. Let's think in terms of code. The fundamental building block of all object-oriented code is called a **class**. This is simply a collection of related variables and functions, all wrapped up in a pair of curly braces and labeled with the name of the class. In OOP-speak, a variable associated with a class is referred to as a **property**, and a function is called a **method**—nothing scary or mysterious at all. If you have built up a library of your own custom PHP functions for reuse in different projects, creating a class will seem very familiar. We'll look at the actual syntax in the next chapter, but the main difference is that a class groups everything together.

> *Not all variables in a class are properties. A property is a special type of variable that can be used anywhere inside a class and sometimes outside as well. The distinction between ordinary variables and properties will become clearer later.*

So, if you want to validate user input, you could create a class called `Validator` (by convention, class names always begin with an initial capital) to group together a series of methods (in other words, functions) to perform such tests as these:

- Is this a number?
- Is it within a specified range?
- Is this a valid email address?
- Does this text have potentially malicious content?

In fact, you'll build just such a class in Chapter 4. Since other developers are likely to create classes for similar purposes, it's recommended that you prefix class names with a three- or four-letter prefix followed by an underscore. All classes in this book will be prefixed with `Pos_` (for **P**HP **O**bject-Oriented **S**olutions), so the class in Chapter 4 will be called `Pos_Validator`.

When you want to use any of the class's properties or methods, you need to create an **instance** of the class by using the `new` keyword like this:

```
$val = new Pos_Validator();
```

This creates an **object** called `$val`. As you can see, it looks just like an ordinary PHP variable, so what is an object? In this particular case, it's principally a device that gives you access to all the methods defined in the `Pos_Validator` class. Without `$val`, you have no way of using them. In addition to methods, a class can have properties. In the case of `Pos_Validator`, one of them stores an array of required fields that the user has failed to fill in; another property stores an array of error messages. Because of the way the class has been designed, these arrays are populated automatically, holding the information until you're ready to use it. In programming terms, you might think of an object as a supercharged multidimensional array that controls functions as well as variables. However, in conceptual terms, the `$val` object is the tool that validates the user input. It uses its methods to run specific validation tests, and stores all the results. In other applications, objects can be envisaged in a similar way to real life objects. An e-commerce application might use a `Product` class to represent items of stock, or an online forum might have a `Member` class to represent individual contributors.

Let's take a quick look at an example of the `Pos_Validator` class in action. As I said earlier, you need to create an instance of the class to be able to use it. This is known as **instantiating an object**. As you probably noticed, when instantiating the `$val` object earlier, I placed a pair of empty parentheses after the name of the class. In the same way as you pass arguments to functions, you can also pass arguments to an object at the time of instantiation. The `Pos_Validator` class accepts two arguments, both of them optional. The first optional argument is an array of required form fields. So, the script to instantiate a `Pos_Validator` object to validate a simple form might look like this:

```
// create an array of required form fields
$required = array('age', 'name', 'comments');

// instantiate a validator object, and pass it the $required array
$val = new Pos_Validator($required);
```

The way you access an object's properties and methods is with the -> operator (a dash followed by a greater-than sign, with no space in between). Even if you don't know anything about OOP, it shouldn't take long to work out what the following code does (try to guess, and then read the next paragraph to see if you were right):

```
// use class methods to validate individual fields
$val->isInt('age');
$val->removeTags('name', 0, 0, 1);
$val->checkTextLength('comments', 5, 500);
$val->removeTags('comments', 0, 0, 1);
$val->isEmail('email');

// validate the input and get any error messages
$filtered = $val->validateInput();
$missing = $val->getMissing();
$errors = $val->getErrors();
```

> *To save space, opening and closing PHP tags have been omitted throughout this book, except where required for clarity.*

The `$val` object begins by checking if `age` is an integer. It then removes HTML tags from the `name` field, checks that the `comments` field contains between 5 and 500 characters, and strips all tags from it before checking that the `email` field contains a properly formed email address. The final three lines validate the input, and get the names of missing fields and details of errors. It might look mysterious at the moment, but it's a lot easier to read than dozens of lines of conditional statements.

Another advantage is that objects are independent of each other, even if they're instances of the same class. You can create two separate instances of the `Pos_Validator` class to validate user input from both the `$_POST` and `$_GET` arrays. Because the objects are separate, you can identify where an error message has come from and take appropriate action.

Each object acts like a black box, keeping the data passed to each one completely separate from the other. The black box analogy also applies to one of the main concepts behind OOP: encapsulation.

Protecting data integrity with encapsulation

The idea of **encapsulation** is to ensure that each part of an application is self-contained and doesn't interfere with any others, except in a clearly defined manner. OOP breaks down complex tasks into their component parts, so it's necessary to ensure that changing the value of a property doesn't trigger an unintended chain effect through other parts of the application. When defining a property in a class, you must specify whether it's public, private, or protected. Public properties are accessible to all parts of a PHP script, both inside and outside the class definition, and their values can be changed in the same way as any variable. Protected and private properties, on the other hand, are hidden from external scripts, so they cannot be changed arbitrarily.

Methods can also be public, protected, or private. Since methods allow objects to do things, such as validate input, you frequently need them to be public. However, protected and private methods are useful for hiding the inner workings of a class from the end user.

You'll see how this works in the next two chapters when you start working with actual code, but one of the properties of the `Pos_Validator` class is `$_inputType`, which determines whether the input being validated comes from the `$_POST` or `$_GET` array. To prevent the value of `$_inputType` from being changed, the class definition declares it protected like this:

```
protected $_inputType;
```

The value of `$_inputType` is set internally by the class at the time of instantiating the object. If you attempt to change it directly, PHP generates a fatal error, bringing everything to a grinding halt. Inconvenient though this might sound, this preserves the integrity of your code by preventing an attacker from tricking a validation routine to handle variables from the wrong type of source. As long as a class is well designed, encapsulation prevents the values of important properties from being changed except by following the rules laid down by the class.

It's a common convention to begin the names of protected and private properties with an underscore as a reminder that the property's value should be changed only in strictly controlled circumstances. You'll learn more about public, protected, and private properties and methods in the next chapter.

Encapsulation also makes the final code much simpler and easier to understand, and this is where the example of a car as an object begins to make sense. Unless you're a motor mechanic or enthusiast, you don't need to know the details of the internal combustion engine to get in a car and drive. It doesn't matter whether it's an old-fashioned gas guzzler or one that runs on biofuel; all you need to do is turn on the ignition and put your foot down on the accelerator. What this means in terms of OOP is that you can create a class with a method called `accelerate()`, and the user doesn't need to worry about the internal code. As long as the `accelerate()` method performs the expected task, the user is happy.

This leaves the developer free to make improvements to the method's internal code without forcing users to make similar changes throughout their own scripts. If you're working on your own, this might not seem all that important, as you're both the developer and end user. However, if you're working in a team, or decide to use a third-party class or framework, knowing what goes on inside the black box of the object is irrelevant. All you want to know is that it works and provides consistent results.

Encapsulation is a great advantage for the end user, but it places an important responsibility on the developer to ensure that changes to the internal code don't produce unexpected changes in output. If a method is expected to return a string, it shouldn't suddenly return an array. The black box must work consistently. Otherwise, all dependent code will be affected, defeating the whole purpose of OOP.

Closely related to this is another key feature of OOP: polymorphism.

Polymorphism is the name of the game

In spite of its obscure-sounding name, **polymorphism** is a relatively simple concept. It applies to both methods and properties and means using the same name in different contexts. If that doesn't make it any clearer, an example from the real world should help. The word "polymorphism" comes from two Greek words meaning "many shapes." A human head is a very different shape from a horse's head, but its function is basically the same (eating, breathing, seeing, and so on), so we use the same word without confusion. OOP applies this to programming by allowing you to give the same name to methods and properties that play similar roles in different classes.

Each class and object is independent, so method and property names are intrinsically associated with the class and any objects created from it. There's no danger of conflicts, so when a method or property is used similarly in different classes, it makes sense to use the same name each time. Continuing the example from the previous section, `accelerate()` makes a car go faster, but the way this is achieved depends on its type. In a regular car, you put your foot down on the accelerator pedal; but in a car specially adapted for a wheelchair user, the accelerator is usually on the steering wheel; and in a child's pedal car, you need to move your legs backward and forward quickly. There's no confusion, because each type of car is different, and they all achieve the same effect in different ways. It doesn't matter how many new classes are created to cover different types of cars, you can use `accelerate()` for all of them, leaving the implementation of how they go faster encapsulated inside the class. This is far more convenient than having to use `footDown()`, `squeezeHandle()`, or `pedalFaster()` depending on the type of car. Polymorphism and encapsulation go hand in hand, with polymorphism providing a common interface and encapsulation taking care of the inner details.

In a web site context, you might create different classes to interact with MySQL and SQLite databases. Although the code needed to connect to each database and run queries is different, the concepts of connecting and running queries are common to both, so it makes sense to give both classes `connect()` and `query()` methods, and a `$_result` property. A MySQL object will automatically use the code encapsulated in its black box, and a SQLite object will do likewise. But thanks to polymorphism, both classes use methods and properties with common names.

Contrast this to the need in procedural programming to use different functions, such as `mysql_connect()` and `sqlite_open()`. If you want to change the database your web site uses, you need to change every single line of database code. With the object-oriented approach, the only changes you need to make are the connection details and instantiating a MySQL object instead of a SQLite object, or vice versa. As long as your SQL is database-neutral, the rest of the code should work seamlessly.

This brings us to the final basic concept in OOP: inheritance.

Extending classes through inheritance

Since a class is simply a collection of related functions and variables, one way of adding new functionality is to amend the code. In the early stages of development, this is usually the correct approach, but a fundamental aim of OOP is reusability and reliability. Once a

class has been developed and tested, it should be a stable component that users can rely on. Once the wheel has been invented, there's no need to reinvent it—but you can improve it or adapt it for specialized uses. However, there's no need to code everything from scratch; you can base a new class on an existing one. OOP classes are extensible through **inheritance**.

Just as you have inherited certain characteristics from your parents, and developed new ones of your own, a **child class** or **subclass** in OOP can inherit all the features of its **parent** (or **superclass**), adapt some of them, and add new ones of its own. Whereas humans have two parents, in PHP, a child class can have only one parent. (Some object-oriented languages, such as Python and C++, permit inheritance from more than one parent class, but PHP supports only single inheritance.)

The subclass automatically inherits all the properties and methods of its superclass, which can be a great timesaver if the superclass contains a lot of complex code. Not only can you add new methods and properties of your own, but you can also **override** existing methods and properties (this is polymorphism at play), adapting them to the needs of the new class. You see this in action in Chapter 3, when you extend the built-in PHP `DateTime` class. The extended class inherits all the basic characteristics of the `DateTime` class and creates an object to store a date, time, and time zone. Some of the original class's methods, such as for setting and getting the time zone, are fine as they are, so they are inherited directly. However, the original `DateTime` class doesn't check a date for validity, so you'll override some methods to improve their reliability, as well as adding new methods to format and perform calculations with dates.

Generally speaking, you can extend any class: one you have built yourself, a third-party one, or any of those built into PHP. However, you cannot extend a class or method that has been declared final. I explain the significance of final classes and methods in the next chapter, and in Chapter 3, you'll learn how to inspect a class to find out which properties and methods can be inherited and/or overridden.

Deciding on a class hierarchy

The ability to create subclasses through inheritance is undoubtedly one of the main benefits of OOP, but it also poses a dilemma for the developer: how to decide what each class should do. The object-oriented solutions in this book take a relatively simple approach, either extending an existing PHP class or creating a class that stands on its own. However, if you plan to go more deeply into OOP, you will need to give considerable thought to the structure of your inheritance hierarchy.

Say, for example, that you have an e-commerce site that sells books. If you create a Book class, you run into problems as soon as you decide to sell DVDs as well. Although they share a lot in common, such as price, weight, and available stock, DVDs don't need a property that stores the number of pages, and books don't have a playing time. Add T-shirts to your *product* range, and the inheritance problems become even worse. Hopefully, you picked up the clue in the previous sentence: start with a generic concept, such as product, and use inheritance to add the specific details.

Inheritance is extremely powerful, but there is a danger of overusing it. So, to round out this brief overview of OOP principles, I want to take a quick look at two principles of best practice: loose coupling and design patterns.

Using best practice

Once you appreciate the advantages of OOP, there's a temptation to go overboard and use it for everything, particularly creating lots of child classes. Well designed classes are said to be **loosely coupled**. This means that changes to one part of the code don't have a domino effect forcing changes elsewhere. Loose coupling is achieved by giving classes and objects clearly defined tasks, so that one class is not dependent on the way another works. For example, you might have two classes: one to query a database, and the other to display the results. If the second expects a `mysql_result` resource, it's tightly coupled to the class performing the query. You can't switch to using a different database without changing both classes. If the first class returned an array instead, the second class would continue working regardless of where the data came from.

The general advice about loose coupling is to avoid coding for a particular project. However, this is easier said than done. At some stage, you need to get down to the specifics of the project in hand, and it's often necessary to create classes that you won't be able to reuse elsewhere. Don't worry about this too much. When creating a new class, just ask yourself whether the same technique could be useful in other projects. If it could be, then you know it should be loosely coupled—made more generic.

Many of the problems you try to solve, while new to you, are likely to be the same issues that countless other developers have come across before. If you can find a tried and tested way of doing something, it's often best to adopt that solution, and spend your time tackling issues specific to your own project. Over the years, the accumulated wisdom of OOP developers has been crystallized into what are known as design patterns. A **design pattern** isn't a block of code that you can pick off the shelf and plug into your project; it describes an approach to a problem and a suggested solution. The `Pos_Validator` class in Chapter 4 is an example of the Facade design pattern, the purpose of which is to define "a higher-level interface that makes the subsystem easier to use." PHP 5.2 introduced a set of filter functions designed to validate user input. Unfortunately, it relies on a large number of predefined constants, such as `FILTER_FLAG_ALLOW_THOUSAND`, that are difficult to remember and tedious to type out. The `Pos_Validator` class encapsulates this complexity and hides it behind a set of user-friendly methods.

In the course of this book, I make use of some design patterns and describe them briefly at the appropriate point. However, this isn't a book about PHP design patterns. The emphasis is on learning how to write PHP classes and put them to practical use in the context of website development. If you want to study design patterns in detail, I suggest *PHP Objects, Patterns, and Practice, Second Edition* by Matt Zandstra (Apress, ISBN13: 978-1-59059-909-9). Another good book is *Head First Design Patterns* by Eric Freeman and Elizabeth Freeman (O'Reilly, ISBN13: 978-0-596-00712-6). Even though all the examples in the second book are written in Java, they are easy to understand, and the unconventional approach brings the subject to life.

> *The names and descriptions of most design patterns come from* Design Patterns: Elements of Reusable Object-Oriented Software *by Gamma, Helm, Johnson, and Vlissides (Addison-Wesley, ISBN13: 978-0201633610), affectionately known as the "Gang of Four (GoF) book."*

How OOP has evolved in PHP

As I said before, PHP is not an object-oriented language. In fact, support for OOP wasn't added until PHP 3. Unfortunately, the way OOP was originally incorporated into PHP lacked many essential features. The biggest problem was the way variables were handled internally, resulting in unexpected behavior. These shortcomings weren't addressed in PHP 4 because the main emphasis was on preserving backwards compatibility.

The addition of support for OOP was unexpectedly popular, but it was impossible to rectify the shortcomings without breaking existing scripts. So, when PHP 5 was released in July 2004, the way classes and objects work in PHP was changed radically. PHP 4 objects are incompatible with those designed for PHP 5. The good news is that, apart from a few advanced features beyond the scope of this book, the way PHP 6 handles objects is identical to PHP 5.

> *All the code in this book is designed to work in both PHP 5 and PHP 6. To ensure full compatibility, you should be using a minimum of PHP 5.2.*

OOP since PHP 5

PHP's handling of objects was completely rewritten in PHP 5 to improve performance and conform to standards common to other object-oriented languages. Aside from a long list of new features, the biggest change from PHP 3 and 4 is the way objects and their properties are handled. Take the following line of code:

```
$objectB = $objectA;
```

In PHP 3 and 4, this makes a *copy* of `$objectA` and stores it as `$objectB`. Both objects then act independently of each other; changes made to `$objectA` don't affect `$objectB`, and vice versa. This is known as **copying by value** and is the way PHP handles variables. In short, PHP 3 and 4 treated objects like any other variable.

Since PHP 5, objects are treated differently from other variables. Instead of making a copy of `$objectA`, the previous line of code stores a *reference* to `$objectA` in `$objectB`. Both variables refer to *the same object*; changes made to one affect the other. This is known as **copying by reference**. If you find this difficult to grasp, it's like adopting a nickname in an online forum. In public, you might call yourself Haven'tAClue, but you remain the same person. To make a copy of an object since PHP 5, you need to use the `clone` keyword like this:

```
$objectB = clone $objectA;
```

> *The* `clone` *keyword is used only with objects. All other variables act the same way as in PHP 4. To learn more about references in PHP, see* `http://docs.php.net/manual/en/language.references.php`.

Other important differences include the addition of the following features:

- Modifiers to control access to properties and methods (essential for encapsulation)
- A unified constructor name, `__construct()`
- Support for explicitly cleaning up resources through a destructor function
- Support for interfaces and abstract classes
- Final classes
- Static classes, properties, and methods
- Automatic class loading

All these features are covered in the remaining chapters. If you have worked with objects and classes in PHP 4, you'll find some things familiar, but I advise you to forget most of what you already know. The new OOP model is very different.

Preparing for PHP 6

PHP 6 has been a long time in the making. It was originally expected to come out in early 2007. The timetable then slipped to the end of 2007, but even as 2008 dawned, the months rolled by with still no sign of PHP 6. One factor behind the delay was the need to continue supporting PHP 4, which still represented nearly three-fourths of all PHP installations at the end of 2007. Since PHP 5 was released in 2004, this meant the development team was maintaining two major releases at the same time as trying to develop the next one. The pressure was too great, so a decision was made to terminate support for PHP 4. All support came to an end on August 8, 2008 after the release of the final security update (PHP 4.4.9). By the time you read this, PHP 4 should have been consigned to the dustbin of history. It served the web community well, but it's time to move on. If you're still using PHP 4, you're living on borrowed time.

With PHP 4 out of the way, the development team could finally concentrate on the future of PHP, rather than patching up the past, but the task is enormous. The main goal of PHP 6 is to make it Unicode-compliant. Computers store letters and characters by assigning a number to each one. The problem is that different encoding systems have evolved to cope with the different writing systems used around the world. To make things worse, different computer operating systems don't always use the same numbers to indicate a specific character. Unicode changes all that by providing a unique number for every character, no matter what the platform, no matter what the program, no matter what the language (`www.unicode.org/standard/WhatIsUnicode.html`).

If you work exclusively in English, and never use accented characters, the switch to Unicode is nothing to worry about, as the 26 letters of the alphabet and basic punctuation use the same encoding in both Latin 1 (iso-8859-1) and the most common Unicode standard (utf-8). However, as Figure 1-1 shows, accented characters cause major problems if you mix encodings. Even English-speaking Britain isn't safe, as the encoding for the pound sterling symbol (£) is different.

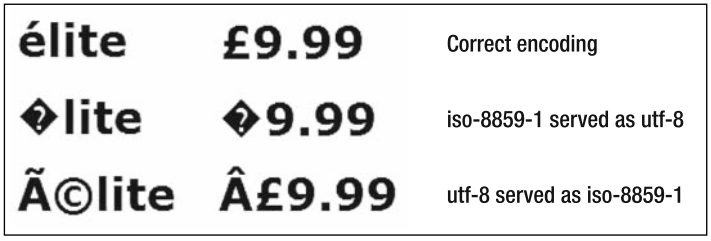


Figure 1-1. Mixing character encoding results in garbled output onscreen.

Since PHP manipulates character data, making PHP 6 Unicode-compliant means updating thousands of functions. It has also generated a vigorous debate about whether to make Unicode the default. As of mid-2008, a final decision had still not been made. The problems posed by the transition to Unicode resulted in a decision to bring forward many of the features originally planned for PHP 6. The most important of these, support for namespaces in OOP, was introduced in PHP 5.3.

One of the core developers, Andi Gutmans, is on record as saying "the migration path may be extremely hard moving from PHP 5 to PHP 6" (`http://marc.info/?l=php-internals&m=120096128032660&w=2`), and there is a widespread expectation that PHP 5 will remain the common standard for a long time to come. Even if you decide to postpone the move to PHP 6, it's important to make sure you don't use code that will break when you finally make the transition. In addition to becoming Unicode-compliant, PHP 6 is dropping support for many deprecated features that could be lurking in existing scripts. The following guidelines should help you future-proof your PHP applications:

- Unify the way you gather and store data, making sure that the same encoding, preferably utf-8, is used throughout.
- Be aware that versions of MySQL prior to 4.1 do not support utf-8. Any data imported from older versions needs to be converted.
- Eliminate `$HTTP_*_VARS` from existing scripts, and replace them with the shorter equivalents, such as `$_POST` and `$_GET`.
- PHP 6 does not support `register_globals`. Make sure you don't have any scripts that rely on `register_globals`.
- Magic quotes have been removed from PHP 6.
- Replace all `ereg_` functions with their `preg_` equivalents, and use Perl-compatible regular expressions (PCRE). Support for `ereg_` and Portable Operating System Interface (POSIX) regular expressions is turned off by default.

Those are the main issues you need to address to prepare for PHP 6, as code that relies on deprecated features just won't work. However, since they already represent best practice, there's nothing arduous about implementing these guidelines. Other best practice that you should adopt includes the following:

- Always use the full opening PHP tag, `<?php`. It's the only one guaranteed to work on all servers.
- Although function and class names are not case-sensitive, always treat them as such, because there are moves to make them case-sensitive in the same way as variables.

*The code in this book was developed before the final release of PHP 6. Any changes that affect its operation in PHP 6 will be listed on the friends of ED web site (*www.friendsofed.com*) and my web site at* http://foundationphp.com/pos/.

Choosing the right tools to work with PHP classes

PHP code is written and stored on the web server in plain text. The web server compiles PHP scripts into byte code at runtime, so there's no need for any special tools or a compiler. All you need is a text editor to write the scripts and a PHP-enabled server to run them on. Since you need to be familiar with PHP basics before embarking on this book, I assume that you already have access to a web server. Although you can test the code on a remote web server, you'll find a local testing environment is much more efficient. Detailed instructions for setting up a local testing environment are in my earlier books, *PHP Solutions: Dynamic Web Design Made Easy*, *Foundation PHP for Dreamweaver 8*, and *The Essential Guide to Dreamweaver CS3 with CSS, Ajax, and PHP* (all friends of ED), so I won't go over the same ground here.

If you don't want to configure a testing environment yourself, I suggest you try XAMPP for Linux or Windows (www.apachefriends.org/en/xampp.html) or MAMP for Mac OS X (www.mamp.info/en/mamp.html). Both are easy to set up and have a good reputation. Alternatively, you might want to invest in a specialized PHP integrated development environment (IDE), such as Zend Studio for Eclipse (www.zend.com/en/products/studio/) or PhpED (www.nusphere.com/products/phped.htm). Both have built-in PHP servers for testing.

Let's take a quick look at choosing a suitable program to write PHP classes.

Using a specialized script editor

Although you can write PHP classes in Notepad or TextEdit, you can make your life a lot easier by choosing a specialized editor with built-in support for PHP. A good script editor should offer at least the following features:

- **Line numbering**: Being able to find a specific line quickly makes troubleshooting a lot easier, because PHP error messages always identify the line where a problem was encountered.
- **A "balance braces" feature**: Parentheses, square brackets, and curly braces must always be in matching pairs, but the opening and closing ones can be dozens or even hundreds of lines apart. Some editors automatically insert the closing one as soon as you type the opening one of the pair; others simply provide a way of identifying the matching pairs. Either way, this is a huge time-saver.
- **Syntax coloring**: Most specialized script editors highlight different parts of code in distinctive colors. If your code is in an unexpected color, it's a sure sign that you have made a typing mistake.