PHP 5 Ready

sitepoint

The PHP Anthology

Volume I: Foundations

By Harry Fuecks

Practical Solutions to Common Problems

The PHP Anthology, Volume I: Foundations (Chapters 1, 2, and 3)

Thank you for downloading the sample chapters of Harry Fuecks' book, *The PHP Anthology*, published by SitePoint.

This excerpt of *The PHP Anthology, Volume I: Foundations* includes the Summary of Contents, Information about the Author and SitePoint, Table of Contents, Preface, 3 chapters of the book, and all 4 of the Appendices.

You should have also received a separate file containing an excerpt of *The PHP Anthology, Volume II: Applications*.

We hope you find this information useful in evaluating these books.

Summary of Contents of this Excerpt

Summary of Additional Book Contents

Summary of Contents: Volume II

The PHP Anthology

Volume I: Foundations

by Harry Fuecks

The PHP Anthology, Volume I: Foundations

by Harry Fuecks

Editor: Georgina Laidlaw
Technical Editor: Kevin Yank
Cover Design: Julian Carroll
Printing History:
First Edition: December 2003

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood
VIC Australia 3066.

Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 0-9579218-5-3
Printed and bound in the United States of America

About The Author

Harry is a technical writer, programmer, and system engineer. He has worked in corporate IT since 1994, having completed a Bachelor's degree in Physics. He first came across PHP in 1999, while putting together a small Intranet. Today, he's the lead developer of a corporate Extranet, where PHP plays an important role in delivering a unified platform for numerous back office systems.

In his off hours he writes technical articles for SitePoint and runs phpPatterns (http://www.phppatterns.com/), a site exploring PHP application design.

Originally from the United Kingdom, he now lives in Switzerland. In May, Harry became the proud father of a beautiful baby girl who keeps him busy all day (and night!)

About SitePoint

SitePoint specializes in publishing fun, practical and easy-to-understand content for Web Professionals. Visit http://www.sitepoint.com/ to access our books, newsletters, articles and community forums.

For Natalie and Masha

Table of Contents

Preface

One of the great things about PHP is its vibrant and active community. Developers enjoy many online meeting points, including SitePoint Forums[1] where developers get together to help each other out with problems they face on a daily basis, from the basics of how PHP works, to solving design problems like "How do I validate a form?" As a way to get help, these communities are excellent—they're replete with all sorts of vital fragments you'll need to make your projects successful. But putting all that knowledge together into a solution that applies to your particular situation can be a problem. Often, community members assume other posters have some degree of knowledge; frequently, you might spend a considerable amount of time pulling together snippets from various posts, threads, and users (each of whom has a different programming style) to gain a complete picture.

The PHP Anthology is, first and foremost, a compilation of the best solutions provided to common PHP questions that turn up at the SitePoint Forums on a regular basis, combined with the experiences and insights I've gained from my work with PHP on a daily basis over the last four years.

What makes this book a little different from others on PHP is that it steps away from a tutorial style, and instead focuses on the achievement of practical goals with a minimum of effort. To that extent, you should be able to use many of the solutions provided here in a more or less "plug and play" manner, rather than having to read this book from cover to cover.

That said, threaded throughout these discussions is a "hidden agenda." As well as solutions, this book aims to introduce you to techniques that can save you effort, and help you reduce the time it takes to complete and later maintain your Web-based PHP applications.

Although it was originally conceived as a procedural programming language, in recent years PHP has proven increasingly successful as a language for the development of object oriented solutions. This was further compounded by the public opening in January 2003 of the PHP Application and Extension Repository[2] (PEAR), which provides a growing collection of reusable and well maintained solutions for architectural problems (such as Web form generation and validation) regularly encountered by PHP developers around the world.

[1] http://www.sitepointforums.com/
[2] http://pear.php.net/

The object oriented paradigm seems to scare many PHP developers, and is often regarded as "off limits" to all but the PHP gurus. What this book will show you is that you do *not* need a computer science degree to take advantage of the wealth of class libraries available in PHP today. Wherever possible in the development of the solutions provided in this book, I've made use of freely available libraries that I've personally found handy, and which have saved me many hours of development. Employing a class developed by someone else is often as easy as using any of the built-in functions PHP provides.

The emphasis this book places on taking advantage of reusable components to build your PHP Web applications reflects another step away from the focus of many current PHP-related books. Although you won't find extensive discussions of object oriented application design, reading *The PHP Anthology, Volume I: Foundations* and *Volume II: Applications* from cover to cover will, through a process of osmosis, help you take your PHP coding skills to the next level, setting you well on your way to constructing applications that can stand the test of time.

The PHP Anthology, Volume I: Foundations, will equip you with the essentials with which you need to be confident when working the PHP engine, including a fast-paced primer on object oriented programming with PHP (see Chapter 2). With that preparation out of the way, the book looks at solutions that could be applied to almost all PHP-based Web applications, the essentials of which you may already have knowledge of, but have yet to fully grasp.

Who should read this book?

If you have already gotten your feet wet with PHP, perhaps having read Kevin Yank's *Build Your Own Database Driven Website Using PHP & MySQL* (SitePoint, ISBN 0-9579218-1-0) and completed your first project or two with PHP, then this is the book for you.

Readers with a greater amount of PHP experience may like to skip ahead to *The PHP Anthology, Volume II: Applications* to learn how to put some of PHP's more advanced features to use, and refer back to *Volume I: Foundations* when they need an explanation of a more basic concept.

What's covered in this book?

Here's what you'll find in each of the chapters in this volume:

Chapter 1: PHP Basics

This chapter provides a summary of all the essentials you need in order to get around quickly in PHP, from how to use the manual, to understanding PHP error messages, and how includes work. There are also some tips for writing portable code, and we'll take a look at some of the main PHP configuration pitfalls.

Chapter 2: Object Oriented PHP

The second chapter includes a run-down of PHP's class syntax, as well as a primer that explains how all the key elements of the Object Oriented Paradigm apply to PHP. It's essential preparatory reading for later chapters in this anthology.

Chapter 3: PHP and MySQL

This chapter provides you with all the essentials of MySQL, PHP's favorite database. We start with the basics, covering important topics such as how to avoid SQL injection attacks. We then delve more deeply into many lesser known topics, such as MySQL `FULLTEXT` search facilities, how to repair corrupt tables and back up your database, and how to avoid writing SQL with `PEAR::DB_DataObject`. This chapter also serves as a "case study" in designing a class to handle connecting to, and the querying of, your MySQL database.

Chapter 4: Files

This fourth chapter is a survival guide to working with files in PHP. Here, we'll cover everything from gaining access to the local file system, to fetching files over a network using PHP's FTP client. We'll go on to learn how to create your own zipped archives with `PEAR::Archive_Tar`.

Chapter 5: Text Manipulation

This chapter covers the essentials of handling content on your site. We'll discuss string functions you can't live without, along with the process for validating and filtering user-submitted content. We'll look at how you can implement a BBCode system, and understand the practicalities involved in preventing cross site scripting exploits.

Chapter 6: Dates and Times

Here, you'll learn how to store dates in your database, and how to use PHP's date functions. We'll deal with the nuances of handling different time zones, and implement an online calendar. We'll see how easy it is to run batch jobs on your Website without access to the command line, and learn how to perform simple script performance measurements.

Chapter 7: Images
This chapter explores the creation of thumbnails, and how to "watermark" images on your site. We'll also discuss how you can prevent hot linking from other sites, and produce a few professional charts and graphs with `JpGraph`.

Chapter 8: Email
In this chapter, we deal specifically with email-related solutions, showing you how to take full advantage of email with PHP. We'll learn to send successfully HTML emails and attachments with help from `PHP Mailer`, and easily handle incoming mails delivered to your Web server, using PHP.

Chapter 9: Web Page Elements
The essentials of Web pages and navigation, such as tables with `PEAR::HTML_Table`, are covered here, along with the process for implementing paged result sets. We'll discuss the development of forms with `PEAR::HTML_QuickForm`, covering in some depth the handling of file uploads, and the construction of navigation menus. We'll also take a look at some tricks you can use with Apache to generate search engine friendly URLs.

Chapter 10: Error Handling
Understand PHP's error reporting mechanism, how to take advantage of PHP's customer error handling features, and how to handle errors gracefully in this action-packed chapter.

The Book's Website

Located at http://www.sitepoint.com/books/phpant1/, the Website that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains complete code for all the examples presented in this book.

Besides the PHP scripts themselves, the archive contains a number of shared libraries, which are bundled in the `SPLIB` directory. In order for the scripts that rely on these libraries to work as intended, you'll need to add this directory to PHP's `include_path` (see "How do I include one PHP script in another?" in Chapter 1 for full details on `include_path`). Doing this will also make it easier to use these libraries in your own projects.

For full instructions on how to install and use the code archive, consult the `readme.txt` file in the archive.

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's Website will always have the latest information about known typographical and code errors, and necessary updates for new releases of PHP and the various Web standards.

The SitePoint Forums

If you'd like to communicate with me or anyone else on the SitePoint publishing team about this book, you should join SitePoint's online community[4]. As I mentioned, the PHP forums[5], in particular, can offer an abundance of information above and beyond the solutions in this book.

In fact, you should join that community even if you *don't* want to talk to us, because there are a lot of fun and experienced Web designers and developers hanging out there. It's a good way to learn new stuff, get questions answered in a hurry, and just have fun.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune* and *The SitePoint Tech Times*. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of Web development. If nothing else, you'll get useful PHP articles and tips, but if you're interested in learning other technologies, you'll find them especially valuable. Go ahead and sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/—I'll wait!

[4] http://www.sitepointforums.com/
[5] http://www.sitepointforums.com/forumdisplay.php?forumid=34

Your Feedback

If you can't find your answer through the forums, or if you wish to contact us for any other reason, the best place to write is `<books@sitepoint.com>`. We have a well-manned email support system set up to track your inquiries, and if our support staff is unable to answer your question, they send it straight to me. Suggestions for improvements as well as notices of any mistakes you may find are especially welcome.

Acknowledgements

First and foremost, I'd like to thank the SitePoint team for doing such a great job in making this book possible, for being understanding as deadlines inevitably slipped past, and for their personal touch, which makes it a pleasure to work with them.

Particular thanks go to Kevin Yank, whose valuable technical insight and close cooperation throughout the process has tied up many loose ends and helped make *The PHP Anthology* both readable and accessible. Thanks also to Julian Szemere, whose frequent feedback helped shape the content of this anthology, and to Georgina Laidlaw, who managed to make some of my "late at night" moments more coherent.

A special thanks to the many who contribute to SitePoint Forums[7]. There's a long list of those who deserve praise for their selflessness in sharing their own practical experience with PHP. It's been fascinating to watch the PHP forums grow over the last three years, from discussing the basics of PHP's syntax, to, more recently, the finer points of enterprise application architecture. As a whole, I'm sure SitePoint's PHP community has made a very significant contribution to making PHP a popular and successful technology.

Finally, returning home, I'd like to thank Natalie, whose patience, love, and understanding throughout continue to amaze me. Halfway through writing this book, our first child, Masha, was born; writing a book at the same time was not always easy.

[7] http://www.sitepointforums.com/

1 PHP Basics

PHP is a programming language that's designed specifically for building Websites, and is both blessed and cursed with being remarkably easy to learn and use. Getting started is extremely simple. Before long, the typical beginner can put together a simple Website and experiment with the wealth of open source projects available through resources like HotScripts[1].

Unfortunately, the ease with which PHP-based sites can be developed also means you can quickly get yourself into trouble. As traffic to your site increases, along with the demand for more features and greater complexity, it's important to gain a more intimate understanding of PHP, and to research application designs and techniques that have proved successful on large Websites. Of course, you can't leap into programming and expect to know it all straight away. Even if you could, where would be the fun in that?

In this first chapter, I'll assume you've had a basic grounding in PHP, such as that provided in the first few chapters of Kevin Yank's *Build Your Own Database-Driven Website Using PHP & MySQL* (ISBN 0-9579218-1-0), and instead concentrate on the essentials of "getting around" in PHP.

In this chapter, you'll find out where to get help—a defence against those that bark "Read the manual!" at you—and how to deal with errors in your code. We'll

[1] http://www.hotscripts.com/

also discuss some general tips for keeping your code portable, and provide other essential roughage for your PHP diet. Not everything here fits under the heading of "basic"—there may also be a few surprises in store for the more experienced PHP developers, so keep your eyes peeled!

Be warned, though, that the discussion of PHP syntax is not the most invigorating of subjects—although it is essential to prepare for later chapters. If you start to struggle, remember the line from *The Karate Kid*: you must learn "wax on, wax off" before you can perform the flying kick.

Where do I get help?

PHP is the most widely-used Web scripting language, running on over ten million domains around the world[2]. For an open source technology that lacks any corporate funding whatsoever, its popularity may seem inexplicable. Yet PHP's success is no mystery; it has one of the most active and helpful online communities of any technology. Recent estimates place the number of PHP developers worldwide at around 500,000 and given the nature of the medium, it's fair to assume that a large proportion are active online. In other words, for developers of PHP-based Websites, help is only ever a few clicks away.

Reading the Manual

There's a well known four letter abbreviation, RTFM (I don't think it needs explaining here), which tends to be used to harass beginners in all areas of computing. While I can understand veterans might be unwilling to repeat endlessly the same, well documented instructions, I think the basic assumption should be that we all know how to read the manual in the first place.

The documentation for PHP is excellent, and is maintained by volunteers who make it their sole purpose to keep it up to date, understandable and relevant. The online version[3] is extremely easy to navigate and contains further know-how in the form of annotations from developers across the globe. The manual is one of the areas in which PHP is truly exceptional; software houses like Sun and Microsoft still have a long way to go to provide this quality of material to developers working on their platforms.

[2] http://www.php.net/usage.php
[3] http://www.php.net/manual/en/

The manual is also available in twenty-four different languages[4] but as you're reading this book I'll assume you're happy with the English version of the manual. It's broken down into five main sections plus appendices. It's worth knowing what kind of information can be found, and where—at least within the first four sections, which are the most relevant to the typical PHP developer.

Section I: Getting Started

http://www.php.net/getting-started

This section provides a short introduction to PHP with some basic examples. It then explains how to install PHP (describing all sorts of operating system-Web server combinations), and how to configure it in terms of modifying your `php.ini` file.

Not to be overlooked is the section on security, which covers the areas in which PHP developers often make mistakes that leave their applications open to abuse. Once again, the "price" of PHP's ease of use is that it won't always protect you from your worst mistakes, so it's worth getting started on security as early as possible in your PHP career. You'll find a summary of key security issues in Appendix C, as well as in discussions throughout this book, where appropriate.

Section II: Language Reference

http://www.php.net/langref

This section covers the fundamentals of PHP as a programming language. Some of these are essential to your being able to achieve anything with PHP, while others become useful as you look for ways to improve your technique. Reading the whole lot in one sitting may well be like reading a dictionary. Fortunately, it's possible to absorb much of the information contained in the language reference by reading the wealth of tutorials available online, and examining the code that's used in open source PHP applications. Certainly, as you read this book, I hope you'll pick up a thing or two about getting the most out of PHP. However, it is worth familiarizing yourself with the subjects contained in this section of the manual, and keeping them in the back of your mind for future reference.

[4] http://www.php.net/docs.php

Section III: Features

http://www.php.net/features

Covered here are the core elements of PHP that are generally focused on solving specific Web-related problems. Much of the Features section reads like an "executive summary" and, from a developers point of view, the information contained here may be better understood when you see it in action—for instance, in the examples we'll see throughout this book.

Section IV: Function Reference

http://www.php.net/manual/en/funcref.php

This section makes up the *real* body of the manual, covering all aspects of the functionality available within PHP. This is where you'll spend most of your time as you progress with PHP, so you'll be glad to hear the PHP group has made a concerted effort to make this section easy to get around. It's even fun, in an idle moment, just to trawl the manual and be amazed by all the things you can do with PHP. Yes, I *did* just describe reading a manual as "fun"!

The function reference is broken down into subsections that cover various categories of functions, each category corresponding to a **PHP extension**.

PHP Extensions

The notion of an extension can be a little confusing to start with, as many are distributed with the standard PHP installation. The String functions, which we'd be pretty hard-pressed to live without, are a case in point. In general, the PHP group distributes, as part of the default PHP installation, all the extensions they regard as being essential to developers.

Extensions regarded as "non-essential" functionality (i.e. they will be required by some, but not all developers) must be added separately. The important information appears under the heading "Installation" on the main page for each extension. Core extensions are described with the sentence "There is no installation needed to use these functions; they are part of the PHP core." Nonstandard extensions are examined in Appendix B.

Access to information within the Function Reference is available through the search field (top right) and searching within the "Function List". Note that searching within the function list examines *only* the Function Reference section

of the manual. To search the entire manual you need to search within "Online Documentation."

Another handy way to get around is to "short cut" to functions by passing the name of the topic you're interested in via the URL. For example, try entering the following in your browser's address field: http://www.php.net/strings. This will take you to http://www.php.net/manual/en/ref.strings.php, which is the main page for the Strings extension. Looking at this page, you'll see a list of all the functions made available by the extension; the same list is available in the menu on the left hand side.

Taking the `strpos` function as an example, enter the URL `http://www.php.net/strpos` (which takes you to http://www.php.net/manual/en/function.strpos.php). You will see the following information about the `strpos` function:

strpos

(PHP 3, PHP 4)

`strpos` -- Find position of first occurrence of a string

Description

```
int strpos (string haystack, string needle [, int offset])
```

Returns the numeric position of the first occurrence of `needle` in the `haystack` string. Unlike the `strrpos()`, this function can take a full string as the `needle` parameter and the entire string will be used.

If `needle` is not found, returns `FALSE`.

Line one contains the name of the function and line two lists the PHP versions in which the function is available. The third line tells you what the function actually does. In this case, it's a fairly terse explanation, but `strpos` really isn't a subject you can get excited about.

Under the Description heading is perhaps the most important line of all—the function's **signature**. This describes the **arguments** this function accepts and the value it **returns** in response. Reading from left to right, you have `int`, which tells you that the value returned by the function is an integer (in this case, the position of one piece of text within another). Next comes the name of the function itself, and then, in parentheses, the arguments this function takes, separated by commas.

Let's look at the argument `string haystack`. This says the first argument should be a string value, while `haystack` simply names the argument so that it can be referred to in the detailed description. Note that the third argument is placed inside square brackets, which means it's optional (i.e. you don't have to supply this argument).

Here's how you could use `strpos`:

File: **1.php**

```
<?php
$haystack = 'Hello World!';
$needle   = 'orld';

// Use the strpos() function
$position = strpos($haystack, $needle);

echo 'The substring "' . $needle . '" in "' .
     $haystack . '" begins at character ' . $position;
?>
```

Notice that I've used `strpos` similarly to the way it appears in the manual. I used the variable names `$haystack` and `$needle` to make clear the way each relates to the explanation in the manual, but you can use whatever variable names you like.

The function signature convention is used consistently throughout the manual, so once you're used to it, you'll be able to grasp quickly how to use functions you haven't tried before.

Get Help When Problems Arise

If you make a mistake using an in-built function in PHP 4.3.0, the default error reporting mechanism of PHP will display an error message with a link that takes you directly to the manual.

If you're ever in doubt, be sure to read through the comments submitted by other PHP developers, which appear at the bottom of every page in the manual. Usually, you will at least see an example of how the function is used, which may solve the particular dilemma you've run into. In many cases you'll also find alternative explanations and uses for a function, which help broaden your understanding.

Further Help

Outside the manual, there are literally thousands of online resources from which you can get further help. I would dare to say that 99% of all the common problems you'll encounter with PHP have already been answered somewhere, and are available online. That means the most obvious (but sometimes forgotten) place to begin is Google, where a quick search for "PHP strpos problem" will give you an idea of what I mean.

There are also some excellent sites where you can get answers directly from other PHP developers (for free, of course—it's part of the PHP ethic). Perhaps the three biggest in the English language are:

- SitePoint Forums: http://www.sitepointforums.com/
- Dev Shed Forums: http://forums.devshed.com/
- phpBuilder: http://www.phpbuilder.com/board/

Each of these uses vBulletin[16] to host an online discussion and, as such, have very friendly and easy-to-use interfaces. All have very active memberships and you should find most questions answered within twenty-four hours.

Note that when you ask for help on forums, the principle of "helping others to help yourself" is important. Don't post a message that says, "This script has a problem" and paste in your entire PHP script. Narrow the problem down–identify the area where you're having problems and post this code snippet along with other relevant information, such as error messages, the purpose of the code, your operating system, and so on. People offering to help generally don't want to spend more than a few minutes on your problem (they're doing it for free, after all), so saving them time will improve your chance of getting a helpful answer.

Less convenient, but perhaps the most effective last resorts are the PHP mailing lists[17], where beginners are encouraged to use the PHP General list. The lists are available for limited browsing[18], though it's possible to search some of them using the search field from the PHP Website[19] and selecting the list of your choice.

[16] http://www.vbulletin.com/
[17] http://www.php.net/mailing-lists.php
[18] http://news.php.net/
[19] http://www.php.net/

Zend, the company developing the core of the PHP engine, also hosts a fairly active forum[20] for general PHP questions.

If you want to be guaranteed an answer, it's worth investigating PHP Helpdesk[21], a service run by Tap Internet[22], who have partnered with Zend to offer PHP training.

How do I fix an error that PHP finds in my script?

There you are, half way through your latest and greatest script, and all of a sudden a test execution delivers this error:

```
Parse error: parse error, unexpected T_ECHO, expecting ',' or ';'
in c:\htdocs\sitepoint\phpbasics\2.php on line 5
```

The offending code here is as follows:

File: **2.php**

```
<?php
echo 'This is some code<br />';
echo 'Somewhere in here I\'ve got a ';
echo 'parse error!<br />'
echo 'But where is it?<br />';
?>
```

What you're dealing with here is known as a syntax error, and while you're new to PHP you may find yourself spending a lot of time hunting down such problems. As you get more experienced with PHP, tracking down syntax errors will become easier. You'll even come to know your own bad habits and probably be able to guess the error you made before you start the hunt (my own typical failings are forgetting the final quote when building SQL statements in a PHP string and leaving out commas when building arrays). Being familiar with PHP's error messages is a good idea, though.

In general terms, there are four basic types of errors you'll encounter in your PHP applications:

[20] http://www.zend.com/phorum/list.php?num=3
[21] http://www.phphelpdesk.com/
[22] http://www.tapinternet.com/

Syntax Errors

As in the example above, **syntax errors** occur when you break the rules of PHP's syntax. Syntax errors will usually result in a Parse Error message from PHP.

In the example above, the problem itself occurs on line 4:

```
echo 'parse error!<br />'
```

I forgot to add at the end of the line the semicolon (`;`) that's required to mark the termination of every statement. The PHP parser only noticed the problem on line five when it encountered another `echo` statement, as instructions may legally span more than one line. This is worth being aware of, as it sometimes makes errors hard to find—an error might actually have occurred prior to the line on which PHP noticed a problem.

Syntax errors can get particularly confusing in the case of large `if-else` or `while` statements where, for example, you've forgotten a closing parenthesis. Perhaps you have a long listing that's interspersed by blocks of HTML; finding that missing curly brace may be extremely difficult. However, as your coding technique improves and you start to take advantage of classes, breaking your code up into discrete blocks within which the code is short and easy to read, you'll find locating syntax errors much easier.

One further thing to be aware of is PHP's use of **tokens**. In the above error message, PHP complained about an "unexpected T_ECHO." A T_ECHO is a token representing an `echo` statement in your PHP script. The PHP parser breaks your code up into tokens so that it can analyze and process the script. Some of the tokens you'll see reported in parse errors are less obvious than others, so if you're unsure, it's worth looking at the manual on tokens[23].

If you're using PHP 4.3.0, you'll find it includes the so-called tokenizer extension[24], which allows you to see your script the way the PHP parser views it. For the sake of interest, here's how you could view the tokenizer's output:

File: **3.php**

```
<?php
/* Note: This script will only work with PHP 4.3.0 or later */

// Read a PHP script as a string
```

[23] http://www.php.net/tokens
[24] http://www.php.net/tokenizer

```
$script = file_get_contents('2.php');

// Fetch the tokens into an array
$tokens = token_get_all($script);

// Display
echo '<pre>';
print_r($tokens);
echo '</pre>';
?>
```

Semantic Errors

Semantic errors occur when you write code that obeys the rules of PHP's syntax, but which, when executed, breaks the "runtime rules" of PHP. For example, the `foreach` statement expects you to give it an array:

File: **4.php**

```
<?php
$variable = 'This is not an array';

foreach ($variable as $key => $value) {
  echo $key . ' : ' . $value;
}
?>
```

Because `$variable` was not an array, this script produces the following error message:

```
Warning: Invalid argument supplied for foreach() in
c:\htdocs\sitepoint\phpbasics\3.php on line 4
```

Semantic errors usually result in a Warning error message like this one.

Environment Errors

Environment errors occur when a system that's external to a PHP script causes a problem. For example, your MySQL server might have been down at the point at which your PHP script tried to connect to it. Perhaps you specified an incorrect path to a file you wanted to open, so PHP was unable to find the file.

These errors also occur when we take a PHP script that has been written on one system, and execute it on another system with a different environment. The

problem may simply be that the underlying directory structure or domain name of the Web server is different. It's common to deal with these types of issues by creating a central configuration script that stores all these environment variables.

PHP also has a number of settings in `php.ini` that can cause a script to fail on another system where the settings are different. I'll be looking at these in "How do I write portable PHP code?"; there's also summary information in Appendix A.

Logic Errors

Logic errors occur when an application runs perfectly as far as the PHP engine is concerned, but the code does something other than what you had intended. For example, imagine you have a mailing script that you want to use to send the same message to a few of the members of your online forum. To your horror, you discover upon executing the script that you've mailed the entire forum membership … twenty times!

These kinds of problems are the most difficult to find; users of Windows XP will be well acquainted with Windows updates—even big companies struggle with logic errors.

Critical to finding logic errors is your ability to test rigorously your code in a safe environment that's separate from your "live" Web server. Thankfully, PHP and related technologies like Apache and MySQL (if you're using them) are cross platform, which makes putting together an effective development environment easy even if the underlying operating systems are different.

You should also investigate **unit testing**, a facet of **Extreme Programming** (XP), to which you'll find an introduction in Volume II, Chapter 6. I've also suggested further reading at the end of this chapter.

In Chapter 10, I'll be taking a look at strategies for handling errors themselves, particularly environment errors. In particular, we'll discuss how you can record (or trap) errors for your analysis without displaying ugly messages to your applications users.

How do I include one PHP script in another?

Having discovered that writing thousand-line scripts may not be the best way to stay organized, you're probably looking for ways to break your code into separate files. Perhaps, while using someone else's Open Source application, you find yourself struggling to eliminate error messages like the one below:

```
Fatal error: Failed opening required 'script.php'
```

Mutual Inclusion

PHP provides four commands that allow you to add the contents of one PHP script to another, namely `include`, `require`, `include_once` and `require_once`. In each case, PHP fetches the file named in the command, then executes its contents. The difference between `include` and `require` is the way they behave should they be unable to find the script they were told to fetch.

`include` will generate a PHP warning message like this:

```
Warning: Failed opening 'script.php' for inclusion
```

This will allow the script that called the `include` command to continue execution.

By contrast, `require` results in a fatal error like the one shown above, which means the calling script will terminate, bringing everything to a halt. If the file that was required is critical to your application, having the script terminate is a very good thing.

The `include_once` and `require_once` commands behave similarly to their respective cousins, but if the script has already been included or required anywhere else (by any of the four commands), the statement will be ignored. At first glance, it may not be obvious how these commands can be used; surely you'll know how many times you've used an `include` command, right? Where the `_once` commands become extremely handy is in more complex applications in which you have PHP scripts that include other PHP scripts, which in turn include yet more PHP scripts. This is particularly important when you use libraries of classes (which we'll explore in Chapter 2), and those classes are being reused repeatedly by many scripts. One class may depend on another being available; using a `require_once` to include

the required class ensures it will always be available, yet causes no problem if the class happens to have been used elsewhere.

To see all this in action, let's make a script called `include_me.php`:

File: **include_me.php**

```
<?php
// include_me.php

echo 'I\'ve been included!<br />';
?>
```

Every time this script is included it will display the message "I've been included!" so we know it's worked.

Now, let's test the various ways we can include this file in another script:

File: **5.php**

```
<?php
// This works fine
echo '<br />Requiring Once: ';
require_once 'include_me.php';

// This works fine as well
echo '<br />Including: ';
include 'include_me.php';

// Nothing happens as file is already included
echo '<br />Including Once: ';
include_once 'include_me.php';

// This is fine
echo '<br />Requiring: ';
require 'include_me.php';

// Again nothing happens - the file is included
echo '<br />Requiring Once again: ';
require_once 'include_me.php';

// Produces a warning message as the file doesn't exist
echo '<br />Include the wrong file: ';
include 'include_wrong.php';

// Produces a fatal error and script execution halts
echo '<br />Requiring the wrong file: ';
```

```
require 'include_wrong.php';

// This will never be executed as we have a fatal error
echo '<br />Including again: ';
include 'include_me.php';
?>
```

Here's the output this generates (note that I've simplified the error messages at the end):

```
Requiring Once: I've been included!

Including: I've been included!

Including Once:

Requiring: I've been included!

Requiring Once again:

Include the wrong file:
Warning: Failed opening 'include_wrong.php' for inclusion

Requiring the wrong file:Fatal error: Failed opening required
'include_wrong.php'
```

Notice here that the first use of `include_once` does nothing (the file has already been included), as does the later use of `require_once`. Later on, when I try to include the wrong file (in this case, a file that doesn't exist), I get a warning message. However, execution continues to the next line where I try to require a file that doesn't exist. This time, PHP produces a fatal error and execution of the script halts, meaning the final attempt to include the file will never happen.

Be aware that the files you include needn't contain only PHP. The included file could simply contain HTML without any PHP.

Which Command to Use?

As a general practice, unless you have a special circumstance where some other behavior is needed, always use the `require_once` command to include one file in another. This is particularly important when you're placing PHP classes in separate files, and one class may depend on another. For the full story on classes, see Chapter 2.

PHP's four include commands should not be confused with the various file-related functions (discussed in Chapter 4); these are intended for fetching files without parsing them immediately as PHP scripts, thereby allowing you to work on their contents.

Note that throughout this book I'll be talking about "including" a file even when I'm using one of the `require` commands. This is a common convention for talking about PHP that stems from older programming languages used by the first PHP pioneers.

Path Finding

So far, I've only looked at including files in the same directory as the script that contains the include command. In practice, you'll usually want to organize files into subdirectories based on the job they do. This can be a source of much confusion, particularly when you're using third party code, as there are numerous alternative approaches to dealing with includes in other directories.

The first thing to be aware of is that all includes are calculated *relative* to the directory in which the main script (where execution began) resides. For example, imagine we have three files in the following locations:

```
/home/username/www/index.php
/home/username/www/includes/script.php
/home/username/www/another.php
```

First, let's consider `index.php`. The command `include 'includes/script.php';` will correctly include `script.php`, assuming `index.php` is the actual file requested.

But what if we use the following command in `script.php`:

```
include '../another.php'; // ???
```

If `script.php` is the page we're viewing, this command will correctly include `another.php`. However, if `index.php` is the page we're viewing, and it includes `script.php`, this command will fail, because the location of `another.php` is calculated relative to the location of `index.php`, *not* relative to `script.php`.

We have two choices. We can modify `script.php` so that it includes `another.php` as follows:

```
include 'another.php';
```

Alternatively, we can enter the full path to `another.php`, like this:

```
include '/home/username/www/another.php';
```

This leaves no doubt as to where `another.php` is located.

The PHP configuration file `php.ini` also contains the directive `include_path`. This allows you to specify directories from which files can be included, without the need to specify their locations when using one of the include commands.

This approach needs to be used with caution, as it may lead to strange results if an included file of the same name exists in more than one directory, yet it can be an effective means to solve include-related headaches. PHP's PEAR[25] class library, for example, relies on your adding the directory that contains PEAR's include files to the include path. Note also that it's not a good idea to specify too many locations in your include path, as this will slow PHP down when it tries to find the scripts you've included in your code.

If you're using Apache in a shared hosting environment, you may be able to override the value of `include_path` using a `.htaccess` file. Placed in the directory to which you want it to apply (it will also apply to all subdirectories), the file should contain something like this:

```
php_value include_path ".:/usr/local/lib/php:/home/user/phplib/"
```

The same can also be accomplished with the PHP function `ini_set`, for example:

```
ini_set('include_path', 'C:/phplib/');
```

This allows changes to be made at runtime from within a PHP script.

You'll find a reference to `php.ini` values in Appendix A.

How do I write portable PHP code?

Not all PHP installations are the same. Depending on version and configuration settings in `php.ini`, your script may or may not run correctly on another server where PHP is installed. However, there are some general good practices you can adopt to make life easier and minimize the need to rewrite code for other servers.

[25] http://pear.php.net/

Keep All Configuration Central

For most PHP applications, it will be necessary to provide information describing the environment in which the script will run, including database user names and passwords, directory locations, and so on. As a general rule, try to keep the majority of this information in a single place—maybe even a single file—so that when the information needs to be modified, you can do it all in the one place. That said, when building modular applications, you may want to store elements of the configuration that are local to a specific "module" with the module itself, rather than centrally.

How exactly you choose to store this information is a matter of personal choice. In some cases, it may be worth considering an XML file or storing some of the information in a database. It's also worth being aware of the `parse_ini_file` function, which I'll explore in Chapter 4.

A simple but effective mechanism is to place all the settings in a single file as PHP constants, which makes them available from any function or class in your application. For example:

File: **6.php**

```
<?php
// Configuration settings
define('DOMAIN', 'sitepoint.com');

// In another script
echo 'The domain is ' . DOMAIN;
?>
```

Constants need to be used with caution, though. To make your functions and classes reusable in other applications, they shouldn't depend on constants of a fixed name; rather, they should accept configuration information as arguments. In such cases, it's best to use PHP variables in your central configuration file, which you can then pass to functions and classes as required. If you look at Chapter 3, when connecting to MySQL we can identify a number of variables we need to have in a central location: the server host name, the user name, the password, and the name of the selected database.

Using the `require_once` command we looked at in the previous solution, we can create a file called, for instance, `config.php`, and place it outside the public Web directories. This helps ensure that no one accidentally browses to the file containing this critical information, which would place the site's security at risk.

Use the Full <?php ?> Tags

PHP supports a variety of tag styles to mark up sections of PHP code, including the short tags (`<? ?>`), and ASP-style tags (`<% %>`). These are controlled from `php.ini` with the settings `short_open_tag` and `asp_tags`. While you have these settings set to On, other people may not. The short tag style, for example, causes a problem when the PHP is mixed with XML documents, which use processing instructions like this:

```
<?xml version="1.0"?>
```

If we have a document which contains PHP and XML, and we have the `short_open_tag` turned on, PHP will mistake the XML processing instruction `<?xml` for a PHP opening tag.

It's possible that your code will need to run in environments where `short_open_tags` and `asp_tags` are both off. The best way to be sure that they are is to get into the habit of always using the `<?php ?>` tag style, otherwise there may be a lot of code rewriting to do in some dark future.

register_globals off

PHP is capable of turning incoming data into native PHP variables. This feature is controlled by the `register_globals` setting in `php.ini`. With `register_globals` switched on, if I point my browser at an address like http://www.mysite.com/index.php?logged_in=1, PHP will automatically create a variable `$logged_in` and assign it the value of `1`. The PHP group now recommends this setting be disabled because it presents a risk to security, as the previous example suggests.

So, in `php.ini` make sure the following code is in place:

```
register_globals = Off
```

This will force you to access incoming data via the special predefined **superglobal variables** (e.g. `$_GET['username']`), which means they won't conflict with variables you've created in your script.

Using a `.htaccess` file with Apache, the same result can be achieved with the following code:

```
php_flag register_globals off
```

Further information can be found in the PHP manual[26], and in Kevin Yank's article, *Write Secure Scripts with PHP 4.2!*[27] on SitePoint.

Magic Quotes

Magic quotes is a feature intended to help prevent security breaches in sites developed by PHP beginners.

It adds **escape characters** (see Chapter 5 for more information) to incoming URL query strings, form posts, and cookie data automatically, *before* your script is able to access any of these values. Should you insert the data directly into your database, there's no risk of someone being able to tamper with the database provided magic quotes functionality is switched on.

For beginners, this is certainly a useful way to prevent disasters. However, once you understand what **SQL injection attacks** are, and have developed the habit of dealing with them in your code, the magic quote functionality can become more of a problem than it's worth.

Magic quotes functionality is controlled by a PHP configuration setting, `magic_quotes_gpc`, which can be either on or off.

My own preference is to always have magic quotes switched off, and deal with escaping data for SQL statements myself. Unfortunately, this means the code I write won't port well to PHP installations where magic quotes is switched on (I'll end up with backslashes in my content). Thankfully, to deal with this problem, PHP provides the function `get_magic_quotes_gpc`, which can be used to find out whether magic quotes are switched on. To keep the code in this book portable, we'll use a simple file that strips out magic quotes, should the functionality be enabled:

File: **MagicQuotes/strip_quotes.php (in SPLIB)**

```
<?php
/**
 * Checks for magic_quotes_gpc = On and strips them from incoming
 * requests if necessary
 */
if (get_magic_quotes_gpc()) {
  $_GET    = array_map('stripslashes', $_GET);
  $_POST   = array_map('stripslashes', $_POST);
```

[26] http://www.php.net/registerglobals
[27] http://www.sitepoint.com/article/758

```
  $_COOKIE = array_map('stripslashes', $_COOKIE);
}
?>
```

If we include this at the start of any file in which we accept data from a query string, a form post, or a cookie, we'll remove any slashes added by magic quotes, should this functionality be switched on. This effectively gives us back what we started with.

The subject of SQL injection attacks is discussed in detail in "How do I solve database errors caused by quotes/apostrophes?" in Chapter 3. If you're not yet confident that you can protect yourself against SQL Injection attacks, use magic quotes. Once you're happy you have a full grasp of all the issues, switch the magic quotes functionality off and save yourself many headaches. Note that magic quotes can only be switched on or off using the `php.ini` file or one of Apache's `.htaccess` files. For more information, see Appendix A.

Call-Time Pass-By-Reference Off

A **reference** is like a "short cut" to the value of a variable. References are often required when we use PHP functions and classes, a subject we'll discuss further in Chapter 2. When you use a reference to a variable in calling a function or class method, it's defined as a **call-time pass-by-reference** Consider this example:

```
$result = myFunction(&$myVariable);
```

Here the `&` operator tells PHP to use a **reference** to the variable `$myVariable` as the argument, rather than creating a copy of its value. This is now generally regarded as bad practice, as it can make the job of understanding someone else's code extremely difficult.

Switch this off in `php.ini` using the following command:

```
allow_call_time_pass_reference = Off
```

Alternatively, switch it off in a `.htaccess` file as follows:

```
php_flag allow_call_time_pass_reference off
```

Write Reusable Code

It's easy to say, I know, but if you find yourself writing any more than one PHP script, you need to start thinking about ways to make your code reusable, before

you suffer premature hair loss. Technically, this isn't exactly an issue of portability as such, but if you end up working on other sites or applications, you'll appreciate having ready code that you can simply plug into your new project. Also, if you're writing code that other people will integrate with existing applications on their Websites, you need to package it in a form that doesn't place requirements on the code they're already using.

For example, if your application has some kind of user authentication system, will it integrate with the one they're already using—a system that already has a large database of users associated with it?

The best approach is to write **object oriented** code (the focus of Chapter 2) with a mind to creating reusable "components." Some people argue that writing object oriented code in PHP slows down the application's performance and should therefore be avoided at all costs. What they forget to mention is the drastic increase in *your* performance that object oriented programming delivers. After all, fast programmers cost more than fast microprocessors!

Some things to consider when measuring the potential of your code for reuse are:

- ❑ What happens when requirements change?
- ❑ How easy is it to add new features to your code?
- ❑ Are you still able to understand the code after a long period of time?
- ❑ Can your code be integrated easily with other applications?
- ❑ Will assumptions made in your code apply to your work on other sites?

You'll find throughout this book many hints and suggestions to encourage you to write reusable code, although an in-depth analysis of PHP applications design as a whole is beyond its scope. As you read this book, you should get a feeling for some of the critical factors as subjects for further investigation. You have one main responsibility to yourself as an experienced PHP developer: to keep expanding your general knowledge of the more esoteric aspects of software development, such as **design patterns** and **enterprise application architecture**, as a means to improve your development technique and, more importantly, save yourself time. The broader your knowledge, the lower the risk of failure when you land that big project.

Further Reading

- *Write Secure Scripts with PHP 4.2!*: http://www.sitepoint.com/article/758

  A tutorial that explains the importance of writing scripts with `register_globals` switched off.

- *Effortless (or Better!) Bug Detection with PHP Assertions*: http://www.sitepoint.com/article/1008

- *Using Strings*: http://www.zend.com/zend/tut/using-strings.php

  Zend provides a walk-through of the main functions available for working with strings.

- *String Theory*: http://www.devshed.com/Server_Side/PHP/StringTheory/

  DevShed offers an in depth look at strings, going as far as Posix extended regular expressions.

2 Object Oriented PHP

The object oriented paradigm is an approach to programming that's intended to encourage the development of maintainable and well structured applications. Many PHP coders regard **object oriented programming** (OOP) as some kind of mystic art, given that frequently, examples of PHP look only at **procedural**[1] approaches to problem solving. This is a shame, as there is much to be gained from adopting an object oriented approach to developing PHP applications, perhaps the most important being code reuse. A well written piece of object oriented code can easily be employed to solve the same problem in other projects; we can simply slot it in whenever we need it. There is a growing number of object oriented code repositories, such as PEAR[1] and PHP Classes[2], which can save you from hours of work spent solving well charted problems, and leave you free to focus on the specifics of your application.

In this chapter, you'll gain a practical grounding in writing object oriented PHP—and there'll be plenty of opportunities to get your hands dirty. There are many ways to teach OOP, and the topic provides endless room for discussion. In my opinion, the best approach is to dive in head first, seeing how procedural tasks can be accomplished with classes in PHP, and adding the theory as we go. This is the approach we'll take in this chapter. Throughout both volumes of *The*

[1]Procedural programming is the name given to non-object oriented programming. All the code we've seen in this book so far has been procedural in nature.

[1] http://pear.php.net/

[2] http://www.phpclasses.org/

PHP Anthology, I'll be using OOP, where appropriate, which should give you further examples to study. In particular, Volume II, Chapter 7 should provide some insight into why OOP is an effective way to structure your applications.

In practice, learning to use the object model provided by PHP requires us to achieve two goals, which usually have to be undertaken simultaneously:

- You'll need to learn the PHP class syntax and object oriented terminology.
- You must make the "mental leap" from procedural to object oriented code.

The first step is easy. It's the subject of the next solution, and further examples appear in later solutions that look at more advanced subjects.

The second step, the "mental leap", is both easy and challenging. Once you achieve it, you will no longer think about long lists of tasks that a single script should accomplish; instead, you'll see programming as the putting together of a set of tools to which your script will delegate work.

Jumping ahead a little—to give you a taste of things to come—here's a simple example that should be familiar to anyone who's worked with PHP for more than a week: connecting to MySQL and fetching some data. A common procedural approach looks like this:

```
<?php
// Procedural Example

// Connect to MySQL
$connection = mysql_connect('localhost', 'harryf', 'secret');

// Select desired database
mysql_select_db('sitepoint', $connection);

// Perform a query selecting five articles
$sql = 'SELECT * FROM articles LIMIT 0,5';
$result = mysql_query($sql, $connection);

// Display the results
while ($row = mysql_fetch_array($result)) {
  // Display results here
}
?>
```

In the above script, we've called directly PHP's MySQL functions, which act on the variables we pass to them. This generally results in our getting back a new variable with which we can perform further work.

An object oriented approach to solving the same problem might look like this:

```
<?php
// OOP Example

// Include MySQL class
require_once 'Database/MySQL.php';

// Instantiate MySQL class, connect to MySQL and select database
$db = new MySQL('localhost', 'harryf', 'secret', 'sitepoint');

// Perform a query selecting five articles
$sql = 'SELECT * FROM articles LIMIT 0,5';
$result = $db->query($sql); // Creates a MySQLResult object

// Display the results
while ($row = $result->fetch()) {
  // Display results here
}
?>
```

The detail of dealing with MySQL using PHP's MySQL functions has now been delegated to an **object** that's created from the `MySQL` **class** (which we'll use frequently throughout this book, and which is constructed in Chapter 3). Although this example may not make entirely clear the advantages of OOP, given that, in terms of the amount of code, it's very similar to the first example, what it *does* show is that some of the original script's complexity is now being taken care of by the `MySQL` class.

For example, we now no longer need to perform two steps to connect to the MySQL server, and then select a database; rather, we can handle both steps in one when we create the `MySQL` object. Also, should we later wish to have the script fetch the results from a different database, such as PostgreSQL, we could use the relevant class that provided the same **application programming interface** (API) as the `MySQL` class—and, to do so, we'd only need to change a single line of the above example. We'll do exactly that in Volume II, Chapter 7.

The object oriented approach really shows its worth in situations in which objects interact with each other. I'll leave further discussion of that to the solutions in this chapter, but it's an important concept. As you become fluent in object ori-

ented programming, you'll find that writing complex applications becomes as easy as putting together blocks of Lego.

I'll introduce the occasional **Unified Modelling Language** (UML) class diagram in this discussion. UML is a standard for describing object oriented programs with images. Don't worry if you haven't come across UML before; the relationship between the diagrams and the code will speak for itself.

What are the basics of object oriented PHP?

Assuming you have no knowledge of OOP, the best place to start is with the basic PHP syntax for classes. You can think of a **class** simply as a collection of functions and variables.

Read The Fine Manual

The PHP manual contains a wealth of information on OOP:

http://www.php.net/oop

Here, we'll develop a simple example that could help us generate HTML, which will demonstrate the basics of classes and objects in PHP. This isn't intended to be an example of great design; it's simply a primer in PHP syntax. Let's begin with a procedural script that builds a Web page. Then we'll gradually turn it into a PHP class:

File: **1.php**

```
<?php
// Generates the top of the page
function addHeader($page, $title)
{
  $page .= <<<EOD
<html>
<head>
<title>$title</title>
</head>
<body>
<h1 align="center">$title</h1>
EOD;
  return $page;
```

```
}

// Generates the bottom of the page
function addFooter($page, $year, $copyright)
{
  $page .= <<<EOD
<div align="center">&copy; $year $copyright</div>
</body>
</html>
EOD;
  return $page;
}

// Initialize the page variable
$page = '';

// Add the header to the page
$page = addHeader($page, 'A Prodecural Script');

// Add something to the body of the page
$page .= <<<EOD
<p align="center">This page was generated with a procedural
script</p>
EOD;

// Add the footer to the page
$page = addFooter($page, date('Y'), 'Procedural Designs Inc.');

// Display the page
echo $page;
?>
```

Of note in this example is our first look at **heredoc syntax**, which is an alternative method of writing PHP strings. Instead of surrounding the text with quotes, you begin it with `<<<EOD` and a new line, and end it with a new line and then `EOD`. The PHP Manual[4] can offer more detail on this if you're curious.

This procedural example uses two functions, `addHeader` and `addFooter`, along with a single global variable, `$page`. Perhaps this isn't a far cry from procedural scripts you've written yourself; maybe you've included in every page a file that contains functions such as `addHeader` and `addFooter`.

[4] http://www.php.net/types.string#language.types.string.syntax.heredoc

But how do we **refactor**[2] the above code to take on an object oriented form? First, we need a class into which we can place the two functions, `addHeader` and `addFooter`:

File: **2.php (excerpt)**

```
<?php
// Page class
class Page {
  // Generates the top of the page
  function addHeader($page, $title)
  {
    $page .= <<<EOD
<html>
<head>
<title>$title</title>
</head>
<body>
<h1 align="center">$title</h1>
EOD;
    return $page;
  }

  // Generates the bottom of the page
  function addFooter($page, $year, $copyright)
  {
    $page .= <<<EOD
<div align="center">&copy; $year $copyright</div>
</body>
</html>
EOD;
    return $page;
  }
}
```

Using the PHP keyword `class`, we can group the two functions, `addHeader` and `addFooter`, within the class. Functions placed inside a class are known as **member functions**, or, more commonly, **methods**. Unlike normal functions, methods must be called as part of the class:

[2]Refactoring is the process of restructuring code without actually changing what it does. This is usually done to ease future maintenance and expansion of the code that would be hindered by its current structure.

File: **2.php (excerpt)**

```
// Initialize the page variable
$page = '';

// Add the header to the page
$page = Page::addHeader($page, 'A Script Using Static Methods');

// Add something to the body of the page
$page .= <<<EOD
<p align="center">This page was generated with static class
methods</p>
EOD;

// Add the footer to the page
$page = Page::addFooter($page, date('Y'), 'Static Designs Inc.');

// Display the page
echo $page;
?>
```

Here, we've called the class methods `addHeader` and `addFooter` using the `::` operator. The script is practically the same as before; however, instead of calling our functions directly, we need to call them as shown here:

```
$page = Page::addHeader($page, 'A Script Using Static Methods');
```

Although this isn't a big improvement, it does let us collect our functions together by their job description. This allows us to call different functions by the same name, each nested separately inside a different class.

Static methods

So far, we've only used a class as a container for related functions. In object oriented parlance, functions that are designed to work this way are called **static methods**.

Actually, compared to most methods, static methods are about as boring as they sound. In the sections below, we'll see how you can *really* flex your object oriented muscles with some fully-fledged methods.

Classes and Objects

A **class** is a "blueprint" for an object. That is, unlike a function that you'd declare and use, a class merely describes a *type* of object. Before you can do useful work with it, you need to create an **object**—an **instance** of the class—using a process

called **instantiation**. Once you have an object, you can call the methods that are defined in the class.

Classes don't contain only functions—they can also contain variables. To make the `Page` class we developed above more useful, we might want to group some variables with the methods, then instantiate the class into an object. Here's the code for the revamped `Page` class; join me below for the explanation:

File: **3.php (excerpt)**

```
<?php
// Page class
class Page {

  // Declare a class member variable
  var $page;

  // The constructor function
  function Page()
  {
    $this->page = '';
  }

  // Generates the top of the page
  function addHeader($title)
  {
    $this->page .= <<<EOD
<html>
<head>
<title>$title</title>
</head>
<body>
<h1 align="center">$title</h1>
EOD;
  }

  // Adds some more text to the page
  function addContent($content)
  {
    $this->page .= $content;
  }

  // Generates the bottom of the page
  function addFooter($year, $copyright)
  {
    $this->page .= <<<EOD
```

```
<div align="center">&copy; $year $copyright</div>
</body>
</html>
EOD;
  }

  // Gets the contents of the page
  function get()
  {
    return $this->page;
  }
}
```

The `Page` class has become a lot more useful in this version of the example. First of all, we've added a **member variable** (also called a **field**), in which to store the HTML:

```
  // Declare a class member variable
  var $page;
```

The PHP keyword `var` is used to declare variables in classes. We can also assign values to variables as we declare them, but we cannot place function calls in the declaration. For example:

```
  // This is allowed
  var $page = '';
```

```
  // This is NOT allowed
  var $page = strtolower('HELLO WORLD');
```

After the variable declaration, we have a special method called the **constructor**. This method is automatically executed when the class is instantiated. The constructor function must always have the same name as the class.

Constructors have no return value

A constructor cannot return any value. It is used purely to set up the object in some way as the class is instantiated. If it helps, think of the constructor as a function that automatically returns the object once it has been set up, so there's no need for you to supply a return value yourself.

That said, you may still use the `return` command with no specified value to terminate the constructor immediately, if needed.

Inside the constructor we've used a special variable, `$this`:

```
  // The constructor function
  function Page()
  {
    $this->page = '';
  }
```

Within any method (including the constructor) `$this` points to the object in which the method is running. It allows the method to access the other methods and variables that belong to that particular object. The `->` (arrow) operator that follows `$this` is used to point at a property or method that's named within the object.

In the example above, the constructor assigns an empty string value to the `$page` member variable we declared at the start. The idea of the `$this` variable may seem awkward and confusing to start with, but it's a common strategy employed by other programming languages, such as Java[5], to allow class members to interact with each other. You'll get used to it very quickly once you start writing object oriented PHP code, as it will likely be required for almost every method your class contains.

Of the other class methods, `addHeader` and `addFooter` are almost the same as before; however, notice that they no longer return values. Instead, they update the object's `$page` member variable, which, as you'll see, helps simplify the code that will use this class. We've also used the `addContent` method here; with this, we can add further content to the page (e.g. HTML that we've formatted ourselves, outside the object). Finally, we have the `get` method, which is the only method that returns a value. Once we've finished building the page, we'll use this to create the HTML.

All these methods access the `$page` member variable, and this is no coincidence. The ability to tie PHP code (the methods) to the data (the variables) that it works on is the most fundamental feature of object oriented programming.

Here's the class in action:

File: **3.php (excerpt)**

```
// Instantiate the Page class
$webPage = new Page();

// Add the header to the page
$webPage->addHeader('A Page Built with an Object');
```

[5] http://java.sun.com/

```
// Add something to the body of the page
$webPage->addContent("<p> align=\"center\">This page was " .
  "generated using an object</p>\n");

// Add the footer to the page
$webPage->addFooter(date('Y'), 'Object Designs Inc.');

// Display the page
echo $webPage->get();
?>
```

To use the class, we've instantiated it with the `new` keyword. The object created from the class is placed in the `$webPage` variable. Through this variable, we have access to all the members of the object as we did with the `$this` variable above.

The first call to the `addHeader` method demonstrates the point:

```
$webPage->addHeader('A Page Built with an Object');
```

Only at the end, upon calling the `get` method, do we actually get anything back from the class. No longer do we need to worry about passing around a variable that contains the contents of the page—the class takes care of that.

Avoid output in classes

Instead of `get`, we could have endowed the `Page` class with a method called `write` to send the page code to the browser immediately. This would have made the code above slightly simpler, as the main script would not have had to `get` the code from the object and `echo` it itself. We avoided this for a reason.

It's usually a bad idea to output directly from inside a class (with statements and functions such as `echo` and `printf`); doing so will reduce the flexibility of your classes. Allowing the value to be retrieved from the class gives you the option of performing additional transformations on it before you send it to the browser, or use it for some other purpose entirely (like putting it in an email!).

Notice also that the number of lines of code we have to write to use the class is fewer than were required in the earlier examples. Although it's impossible to determine good application design by counting the number of lines of code, it is clear that the class has made the procedural code that uses it much simpler. From the point of view of people reading the code, it's already fairly clear what's going on, even without them having to look at the code for the `Page` class.

Understanding Scope

Write more than a few hundred lines of procedural PHP code and, no doubt, you'll run into a parser error or, worse still, a mysterious bug caused by your accidentally having used a function or variable name more than once. When you're including numerous files and your code grows increasingly complex, you may find yourself becoming more paranoid about this issue. How do you stop such naming conflicts from occurring? One approach that can help solve this problem is to take advantage of **scope** to hide variables and functions from code that doesn't need them.

A scope is a context within which the variables or functions you define are isolated from other scopes. PHP has three available scopes: the **global scope**, the **function scope**, and the **class scope**. Functions and variables defined in any of these scopes are hidden from any other scope. The function and class scopes are *local* scopes, meaning that function X's scope is hidden from function Y's scope, and vice versa.

The big advantage of classes is that they let you define variables and the functions that use them together in one place, while keeping the functions hidden from unrelated code. This highlights one of the key theoretical points about the object oriented paradigm. The procedural paradigm places most emphasis on functions, variables being treated as little more than a place to store data between function calls. The object oriented paradigm shifts the emphasis to variables; the functions "back" the variables and are used to access or modify them.

Let's explore this through an example:

```
<?php
// A global variable
$myVariable = 'Going global';

// A function declared in the global scope
function myFunction()
{
  // A variable in function scope
  $myVariable = 'Very functional';
}

// A class declared in the global scope
class MyClass {
  // A variable declared in the class scope
  var $myVariable = 'A class act';
```

```
  // A function declared in the class scope
  function myFunction()
  {
    // A variable in the function (method) scope
    $myVariable = 'Methodical';
  }
}
?>
```

In the above example, each of the `$myVariable` declarations is actually a separate variable. They can live together happily without interfering with each other, as each resides in a separate scope. Similarly, the two `myFunction` declarations are two separate functions, which exist in separate scopes. Thus PHP will keep all of their values separate for you.

Scope becomes important when you start to use object oriented programming in a significant way in your PHP applications. As many classes can have methods of the same name, you can design separate classes to deliver the same application programming interface (API). The scripts that use the classes can then use the same method calls, irrespective of which class was used to instantiate the object they're working with. This can be a very powerful technique in writing maintainable code. We'll look at this point more when we discuss polymorphism later in this chapter.

A Three Liner

Here's how we could make the class even easier to use:

File: **4.php (excerpt)**

```
<?php
// Page class
class Page {

  // Declare a class member variable
  var $page;
  var $title;
  var $year;
  var $copyright;

  // The constructor function
  function Page($title, $year, $copyright)
  {
    // Assign values to member variables
```

```
    $this->page = '';
    $this->title = $title;
    $this->year = $year;
    $this->copyright = $copyright;

    // Call the addHeader() method
    $this->addHeader();
  }

  // Generates the top of the page
  function addHeader()
  {
    $this->page .= <<<EOD
<html>
<head>
<title>$this->title</title>
</head>
<body>
<h1 align="center">$this->title</h1>
EOD;
  }

  // Adds some more text to the page
  function addContent($content)
  {
    $this->page .= $content;
  }

  // Generates the bottom of the page
  function addFooter()
  {
    $this->page .= <<<EOD
<div align="center">&copy; $this->year $this->copyright</div>
</body>
</html>
EOD;
  }

  // Gets the contents of the page
  function get()
  {
    // Keep a copy of $page with no footer
    $temp = $this->page;

    // Call the addFooter() method
    $this->addFooter();
```

```
    // Restore $page for the next call to get
    $page = $this->page;
    $this->page = $temp;

    return $page;
  }
}
```

This time, we've modified the constructor to accept all the variables needed for both the header and the footer of the page. Once the values are assigned to the object's member variables, the constructor calls the `addHeader` method, which builds the header of the page automatically:

```
  // The constructor function
  function Page($title, $year, $copyright)
  {
    // Assign values to member variables
    $this->page = '';
    $this->title = $title;
    $this->year = $year;
    $this->copyright = $copyright;

    // Call the addHeader() method
    $this->addHeader();
  }
```

As you can see, like member variables, methods can be called with the `$this` variable.

The `addHeader` method itself now fetches the data it needs from the member variables. For example:

```
<title>$this->title</title>
```

We've also updated the `get` method so that it calls the `addFooter` method before returning the contents of the `$page` member variable. This means that when we come to fetch the finished page, the footer is added automatically.

```
  // Gets the contents of the page
  function get()
  {
    // Keep a copy of $page with no footer
    $temp = $this->page;

    // Call the addFooter() method
```

```
    $this->addFooter();

    // Restore $page for the next call to get
    $page = $this->page;
    $this->page = $temp;

    return $page;
  }
```

It took a little work to make sure we could call `get` more than once, without adding extra footers to the page, but this complexity is neatly hidden within the class.

Using the class externally is now even easier:

File: **4.php (excerpt)**

```
// Instantiate the page class
$webPage = new Page('As Easy as it Gets', date('Y'),
  'Easy Systems Inc.');

// Add something to the body of the page
$webPage->addContent(
  "<p align=\"center\">It's so easy to use!</p>\n");

// Display the page
echo $webPage->get();
?>
```

Essentially, the page is now built using only three lines of code; I can also reuse this class to generate other pages. Represented as a UML diagram, the `Page` class is shown in Figure 2.1.

Figure 2.1. Page Class as UML

Page
-page : string -title : string -year : int -copyright : string
-addHeader() : void +addContent(in content : string) : void -addFooter() : void +get() : string

The member variables appear in the middle area, while methods appear in the bottom box. Also, the plus and minus signs are there to indicate to other developers which elements of the class are public (+) and which are private (-). Unlike languages such as Java, PHP does not enforce privacy on objects;[3] in the examples above, we could have accessed the `$page` member variable directly in our main script. Because we want the object to handle its own data without outside interference, we indicate in the UML diagram that only those members that have a + against them are available for public use. Those with a - are purely for internal use within the class.

That covers the basics of the class syntax in PHP, and should give you an idea of how classes compare with procedural code. With the syntax you've learnt, you should be able to write standalone classes containing the variables and functions you use frequently—a task that can really help tidy up your code and make it easier to maintain. This is a great start, but the *real* power of object oriented programming comes from using multiple objects and classes together. The rest of this chapter will look at some of the more advanced facets of the PHP object model, including references, inheritance, aggregation, and composition.

How do references work in PHP?

Most discussions of references in PHP begin with an opener like "references are confusing," which may add to the myth that surrounds them. In fact, references are a very simple concept to grasp, yet they're a concept that self-taught PHP developers only really need to consider once they begin writing object oriented applications. Until then, you're probably oblivious to the way PHP handles variables behind the scenes. Much of the confusion that exists around references has more to do with developers who are experienced with other languages like C++ or Java trying to work with PHP: Java, in particular, handles object references in almost the *opposite* way to the approach PHP takes in version 4.

References vs. Pointers

Developers who are familiar with compiled languages such as C++ or Java should note that references in PHP are not analogous to pointers in other languages.

A **pointer** contains an address in memory that *points to* a variable, and must be **dereferenced** in order to retrieve the variable's contents.

[3] Enforced privacy constraints on class members will be added in PHP 5.0.

In PHP, all variable names are linked with values in memory automatically. Using a reference allows us to link two variable names to the same value in memory, as if the variable names were the same. You can then substitute one for the other.

What Are References?

To understand references, we have to begin by understanding how PHP handles variables under normal circumstances (i.e. without references).

By default, when a variable is passed to anything else, PHP creates a copy of that variable. When I say "passed," I mean any of the following:

- Passing a variable to another variable:

```
<?php
  $color = 'blue';
  $settings['color'] = $color;
?>
```

`$settings['color']` now contains a *copy* of `$color`.

- Passing a variable as an argument to a function:

```
<?php
function isPrimaryColor($color)
{
  // $color is a copy
  switch ($color) {
    case 'red':
    case 'blue':
    case 'green':
      return true;
      break;
    default:
      return false;
  }
}

$color = 'blue';
if (isPrimaryColor($color)) {
  echo $color . ' is a primary color';
} else {
  echo $color . ' is not a primary color';
```

```
}
?>
```

When `$color` is passed to the function `isPrimaryColor`, PHP works with a copy of the original `$color` variable inside the function.

- The same applies when passing variables to class methods:

```
<?php
class ColorFilter {
  var $color;
  function ColorFilter($color)
  {
    // $color is a copy
    $this->color = $color;
    // $this->color is a copy of a copy
  }
  function isPrimaryColor()
  {
    switch ($this->color) {
      case 'red':
      case 'blue':
      case 'green':
        return true;
        break;
      default:
        return false;
    }
  }
}

$color = 'blue';
$filter = new ColorFilter($color);
if ($filter->isPrimaryColor() ) {
    echo ($color.' is a primary color');
} else {
    echo ($color.' is not a primary color');
}
?>
```

The original `$color` outside the class is passed to `ColorFilter`'s constructor. The `$color` variable inside the constructor is a copy of the version that was passed to it. It's then assigned to `$this->color`, which makes that version a copy of a copy.

All of these means of passing a variable create a copy of that variable's value; this is called **passing by value**.

Using a Reference

To pass using a reference, you need to use the reference operator `&` (ampersand). For example:

```
<?php
$color = 'blue';
$settings['color'] = &$color;
?>
```

`$settings['color']` now contains a *reference* to the original `$color` variable.

Compare the following examples, the first using PHP's default copying behavior:

```
<?php
$color = 'blue';
$settings['color'] = $color; // Makes a copy
$color = 'red'; // $color changes
echo $settings['color']; // Displays "blue"
?>
```

The second involves **passing by reference**:

```
<?php
$color = 'blue';
$settings['color'] = &$color; // Makes a reference
$color = 'red'; // $color changes
echo $settings['color']; // Displays "red"
?>
```

Passing by reference allows us to keep the new variable "linked" to the original source variable. Changes to either the new variable or the old variable will be reflected in the value of both.

So far, so good. You're probably wondering, "What's the big deal here? What difference does it make whether PHP copies or makes a reference to a variable, as long as we get what we expected?" For variables passed around a procedural program, you hardly ever need to worry about references. However, when it comes to objects interacting with one another, if you don't pass an object by reference, you may well get results you weren't expecting.

The Importance of References

Imagine you have a mechanism on your site that allows visitors to change the look and feel of the site—a user "control panel." It's likely that, to implement this sort of functionality, you'd have code that acts on a set of variables containing "look and feel" data, to modify them independently of the rest of the application's logic.

Representing this simply with classes, first, let's see the class that will store data-related to look and feel:

File: **5.php (excerpt)**

```
<?php
// Look and feel contains $color and $size
class LookAndFeel {
  var $color;
  var $size;
  function LookAndFeel()
  {
    $this->color = 'white';
    $this->size = 'medium';
  }
  function getColor()
  {
    return $this->color;
  }
  function getSize()
  {
    return $this->size;
  }
  function setColor($color)
  {
    $this->color = $color;
  }
  function setSize($size)
  {
    $this->size = $size;
  }
}
```

Next, we have a class that deals with rendering output:

File: **5.php (excerpt)**

```
// Output deals with building content for display
class Output {
  var $lookandfeel;
  var $output;

  // Constructor takes LookAndFeel as its argument
  function Output($lookandfeel)
  {
    $this->lookandfeel = $lookandfeel;
  }
  function buildOutput()
  {
    $this->output = 'Color is ' . $this->lookandfeel->getColor() .
      ' and size is ' . $this->lookandfeel->getSize();
  }
  function display()
  {
    $this->buildOutput();
    return $this->output;
  }
}
```

Notice the constructor for the `Output` class. It takes an instance of `LookAndFeel` as its argument so that, later, it can use this to help build the output for the page. We'll talk more about the ways classes interact with each other later in this chapter.

Here's how we use the classes:

File: **5.php (excerpt)**

```
// Create an instance of LookAndFeel
$lookandfeel = new LookAndFeel();

// Pass it to an instance of Output
$output = new Output($lookandfeel);

// Display the output
echo $output->display();
?>
```

This displays the following message:

```
Color is white and size is medium
```

Now, let's say that, in response to one of the options on your user control panel, you want to make some changes to the look and feel of the site. Let's put this into action:

File: **6.php (excerpt)**

```
$lookandfeel = new LookAndFeel(); // Create a LookAndFeel
$output = new Output($lookandfeel); // Pass it to an Output

// Modify some settings
$lookandfeel->setColor('red');
$lookandfeel->setSize('large');

// Display the output
echo $output->display();
```

Using the `setColor` and `setSize` methods, we change the color to "red" and the size to "large," right? Well, in fact, no. The output display still says:

```
Color is white and size is medium
```

Why is that? The problem is that we've only passed a *copy* of the `LookAndFeel` object to `$output`. So the changes we make to `$lookandfeel` have no effect on the copy that `$output` uses to generate the display.

To fix this we have to modify the `Output` class so that it uses a *reference* to the `LookAndFeel` object it is given. We do this by altering the constructor:

File: **7.php (excerpt)**

```
  function Output(&$lookandfeel)
  {
    $this->lookandfeel = &$lookandfeel;
  }
```

Notice that we have to use the reference operation twice here. This is because the variable is being passed twice—first to the constructor function, then again, to place it in a member variable.

Once we've made these changes, the display looks like this:

```
Color is red and size is large
```

In summary, passing by reference keeps the target variable "linked" to the source variable, so that if one changes, so does the other.

Good and Bad Practices

When working with classes and objects, it's a good idea to use references whenever an object is involved. Occasionally, you may have to do the same with an array, such as when you want to sort the array in a different section of code. But, for the most part, normal variables will not need this treatment, simply because, when your code reaches the level of complexity where you'd need to do so, you will (I hope!) be storing variables inside objects and passing the complete object by reference.

Let's look at some other situations in which you might need to use references…

```
// Make sure $myObject is a reference to
// the variable created by the new keyword
$myObject = &new MyClass();
```

This looks odd at first, but remember, a variable created by the `new` keyword is being passed here—even if you can't see it. The reference operator saves PHP from having to create a copy of the newly-created object to store in `$myObject`.

```
class Bar {
}

class Foo {
  // Return by reference
  function &getBar()
  {
    return new Bar();
  }
}

// Instantiate Foo
$foo = &new Foo();

// Get an instance of Bar from Foo
$bar = &$foo->getBar();
```

In the above example, you'll notice the `getBar` method in the `Foo` class. By preceding the function name with the reference operator, the value the function returns is passed by reference. Note that we also had to use a reference operator when assigning the return value of `getBar` to `$bar`. This technique is commonly used when a class method will return objects.

What's *bad* practice is the following:

```
function display($message) {
  echo $message;
}

$myMessage = 'Hello World!';

// Call time pass by reference - bad practice!
display(&$message);
```

That's known as a **call-time pass-by-reference**, which PHP controls with the following setting in `php.ini`:

```
allow_call_time_pass_reference = Off
```

By default, in recent PHP releases the above setting should be switched to `Off`; turning it on is "frowned upon" by PHP's makers. Switched off, PHP will generate warning errors every time a function call specifies an argument should be passed by reference. As such, it's good practice to leave this setting off.

The reason why call time pass by reference is a "bad thing" is that call time passing by reference can make code extremely difficult to follow. I've occasionally seen PHP XML parsers written using a call-time pass-by-reference—it's nearly impossible to gain any idea of what's going on.

The "decision" as to whether a variable is passed by reference or not is one that belongs to the *function* being called, not the code that calls it. The above code written correctly would look like this:

```
// Accept by reference - good practice
function display(&$message)
{
  echo $message;
}

$myMessage = 'Hello World!';

display($message);
```

Performance Issues

Depending on the scale of your application, there are some performance issues you might need to consider when using references.

In simple cases of copying one variable to another PHP's internal **reference counting** feature prevents unnecessary memory usage. For example,

```
$a = 'the quick brown fox';
$b = $a;
```

In the above example, the value of `$b` would not take up any extra memory, as PHP's internal reference counting will implicitly reference `$b` and `$a` to the same location in memory, until their values become different. This is an internal feature of PHP and affects performance without affecting behavior. We don't need to worry about it much.

In some cases, however, using a reference *is* faster, especially with large arrays and objects, where PHP's internal reference counting can't be used. and the contents must therefore be copied.

So, for best performance, you should do the following:

- With simple values such as integers and strings, avoid references whenever possible.
- With complex values such as arrays and objects, use references whenever possible.

References and PHP 5

With PHP 5, references will cease to be an issue because the default behavior of PHP, when passing objects, will be to pass by reference. If you ever need a copy of an object, you can use the special `__clone` method to create copies.

Essentially, the change brings PHP in line with the majority of object oriented programming languages like Java, and will certainly do a lot to reduce the confusion surrounding the subject. For now, though, and until PHP 5 has been widely adopted, knowing how references work is important.

How do I take advantage of inheritance?

Inheritance is one of the fundamental pieces of the object oriented paradigm and is an important part of its power. Inheritance is a relationship between different classes in which one class is defined as being a **child** or **subclass** of another. The child inherits the methods and member variables defined in the parent class, allowing it to "add value" to the parent.

The easiest way to see how inheritance works in PHP is by example. Let's say we have this simple class:

File: **8.php (excerpt)**

```
<?php
class Hello {
  function sayHello()
  {
    return 'Hello World!';
  }
}
```

Using the `extends` keyword, we can make a class that's a child of `Hello`:

File: **8.php (excerpt)**

```
class Goodbye extends Hello {
  function sayGoodbye()
  {
    return 'Goodbye World!';
  }
}
```

`Goodbye` is now a child of `Hello`. Expressed the other way around, `Hello` is the **parent** or **superclass** of `Goodbye`. Now, we can simply instantiate the child class and have access to the `sayHello` and the `sayGoodbye` methods using a single object:

File: **8.php (excerpt)**

```
$msg = &new Goodbye();

echo $msg->sayHello() . '<br />';
echo $msg->sayGoodbye() . '<br />';
?>
```

That example shows the basics of how inheritance works, but doesn't demonstrate its real power… This comes with the addition of overriding.

Overriding

What happens when we give a function in the child the same name as a function in the parent? An example:

File: **9.php (excerpt)**

```
<?php
class Hello {
  function getMessage()
  {
    return 'Hello World!';
  }
}

class Goodbye extends Hello {
  function getMessage()
  {
    return 'Goodbye World!';
  }
}
```

Both classes have the same method name, `getMethod`. This is perfectly acceptable to PHP—it makes no complaints about a method being declared twice.

Here's what happens when we use the classes:

File: **9.php (excerpt)**

```
$hello = &new Hello();
echo $hello->getMessage() . '<br />';

$goodbye = &new Goodbye();
echo $goodbye->getMessage() . '<br />';
?>
```

And the output is as follows:

```
Hello World!
Goodbye World!
```

Calling `getMessage` via the `$goodbye` object displays "Goodbye World!" The method in the child class is **overrides** the method in the parent class.

You can also have the child class make use of the parent class's method internally, while overriding it. For example:

File: **10.php**

```
<?php
class Hello {
  function getMessage()
  {
```

```
    return 'Hello World!';
  }
}

class Goodbye extends Hello {
  function getMessage()
  {
    $parentMsg = parent::getMessage();
    return $parentMsg . '<br />Goodbye World!';
  }
}

$goodbye = &new Goodbye();
echo $goodbye->getMessage() .'<br />';
?>
```

Using the `parent` keyword, we can call the parent class's method.

Note that we can also call the parent class by name to achieve exactly the same result:

```
class Goodbye extends Hello {
  function getMessage() {
    $parentMsg = Hello::getMessage();
    return $parentMsg . '<br />Goodbye World!';
  }
}
```

Notice that we've replaced the `parent` keyword with the name of the `Hello` class. The output is exactly the same. Using `parent`, however, saves you from having to remember the name of the parent class while working in the child, and is the recommended syntax.

A call such as `parent::getMessage()` or `Hello::getMessage()` from a *non-static* method is *not* the same as calling a static function. This is a special case where inheritance is concerned. The called function in the parent class retains access to the instance data, and is therefore not static. This may be demonstrated as follows:

File: **11.php**

```
<?php
class A {
    var $a = 1;
    function printA()
    {
```

```
        echo $this->a;
    }
}

class B extends A {
    var $a = 2;
    function printA()
    {
       parent::printA();
       echo "\nWasn't that great?";
    }
}

$b = new B();
$b->printA();
?>
```

The output generated from the above is as follows:

```
2
Wasn't that great?
```

PHP does not cascade constructors

Most object oriented languages, like Java, will run the constructor of the parent class automatically, before running an overriding constructor in the child class. This is called **cascading constructors**—it's a feature that PHP does not have.

If you create a constructor in a child class, be aware that you are completely overriding the parent class's constructor, and that you must call it explicitly from your new constructor if you still want the parent class to handle its share of the object initialization.

Overriding declared member variables is achieved in exactly the same way as methods, although you're unlikely to use this feature frequently.

Inheritance in Action

Now that you have a rough idea of how inheritance is used in PHP, it's time to look at an example that should give you a better idea of how inheritance can be applied.

The following example implements a simple navigation system for a Web page, generating the HTML that appears at the top of the page. By having one class

inherit from another, it becomes possible to add "crumb trail" navigation to the page when it's needed.

First up, the `StandardHeader` class deals with generating the HTML for the top of the page, as well as supplying the `setHeader` and `getHeader` methods to access the variable where the HTML is stored.

File: **12.php (excerpt)**

```
<?php
/**
 * A standard header for a Web page
 */
class StandardHeader {
  /**
   * The header HTML is stored here
   */
  var $header = '';

  /**
   * The constructor, taking the name of the page
   */
  function StandardHeader($title)
  {
    $html = <<<EOD
<html>
<head>
<title> $title </title>
</head>
<body>
<h1>$title</h1>
EOD;
    $this->setHeader($html);
  }

  /**
   * General method for adding to the header
   */
  function setHeader($string)
  {
    if (!empty($this->header)) {
      $this->header .= $string;
    } else {
      $this->header = $string;
    }
  }
```

```
  /**
   * Fetch the header
   */
  function getHeader()
  {
    return $this->header;
  }
}
```

Now, the subclass `CategoryHeader` brings extra functionality to its parent, adding the "bread crumb" links to the HTML that was generated. We don't need to re-create the `setHeader` and `getHeader` methods, as these are inherited from `StandardHeader` when `CategoryHeader` is instantiated.

File: **12.php (excerpt)**

```
/**
 * Subclass for dealing with Categories, building a breadcrumb
 * menu
 */
class CategoryHeader extends StandardHeader {
  /**
   * Constructor, taking the category name and the pages base URL
   */
  function CategoryHeader($category, $baseUrl)
  {
    // Call the parent constructor
    parent::StandardHeader($category);

    // Build the breadcrumbs
    $html = <<<EOD
<p><a href="$baseUrl">Home</a> >
<a href="$baseUrl?category=$category">$category</a></p>
EOD;
    // Call the parent setHeader() method
    $this->setHeader($html);
  }
}
```

Let's now put these two classes to use:

File: **12.php (excerpt)**

```
// Set the base URL
$baseUrl = '12.php';

// An array of valid categories
```

```
$categories = array('PHP', 'MySQL', 'CSS');

// Check to see if we're viewing a valid category
if (isset($_GET['category']) &&
    in_array($_GET['category'], $categories)) {

  // Instantiate the subclass
  $header = new CategoryHeader($_GET['category'], $baseUrl);
} else {

  // Otherwise it's the home page. Instantiate the Parent class
  $header = new StandardHeader('Home');
}

// Display the header
echo $header->getHeader();
?>
<h2>Categories</h2>
<p><a href="<?php echo $baseUrl; ?>?category=PHP">PHP</a></p>
<p><a href="<?php echo $baseUrl; ?>?category=MySQL">MySQL</a></p>
<p><a href="<?php echo $baseUrl; ?>?category=CSS">CSS</a></p>
</body>
</html>
```

As you can see, the controlling logic above looks for a `$_GET['category']` variable. If it exists, it creates an instance of `CategoryHeader`, displaying the navigation to allow users to find their way back to the home page. But if it doesn't exist, it creates an instance of the parent `StandardHeader` instead, which applies when users view the home page (and therefore does not require bread crumbs to find their way back).

In other words, inheritance allows us to add the extra functionality we need without having to reproduce the logic that already resides within the parent class; the existing methods and logic can be reused via the child subclass.

Inheritance provides a powerful mechanism to make classes that are modular, addressing a specific problem, while still making available shared methods and variables that can be used irrespective of the specific object we're dealing with.

Avoid Deep Inheritance Structures

As a general rule of thumb, when using inheritance to build class hierarchies, avoid going deeper than two generations.

Doing so is often a sign of a bad design, in which opportunities for classes to interact in different ways (see the next solution) were missed. In practice, having more than two generations of classes often leads to all sorts of debugging problems and makes the code difficult to maintain. For example, it can become hard to keep track of variable names you've used higher up in the hierarchy.

How do objects interact?

Aside from inheritance, there are other ways for objects to interact; for example, one object *uses* another object. In many ways, such interactions are more important than inheritance, and this is where the object oriented paradigm shows its real power.

There are two ways in which one object can use another: **aggregation** and **composition**.

Aggregation

Aggregation occurs when one object is given another object on "temporary loan." The second object will usually be passed to the first through one of the first's member functions. The first object is then able to call methods in the second, allowing it to use the functionality stored in the second object for its own purposes.

A common example of aggregation in action involves a database connection class. Imagine you pass a database connection class to some other class, which then uses the database connection class to perform a query. The class performing the query **aggregates** the database connection class.

Here's a simple example using the `MySQL` class, which we'll create in Chapter 3:

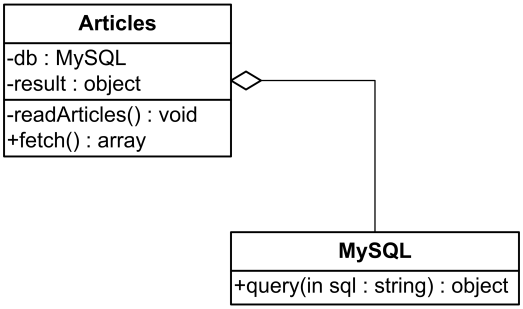
File: **13.php**

```
<?php
// Include the MySQL database connection class
require_once 'Database/MySQL.php';

// A class which aggregates the MySQL class
class Articles {
  var $db;
  var $result;
  // Accept an instance of the MySQL class
```

```
  function Articles(&$db)
  {
    // Assign the object to a local member variable
    $this->db = &$db;
    $this->readArticles();
  }
  function readArticles()
  {
    // Perform a query using the MySQL class
    $sql = "SELECT * FROM articles LIMIT 0,5";
    $this->result = &$this->db->query($sql);
  }
  function fetch()
  {
    return $this->result->fetch();
  }
}

// Create an instance of the MySQL class
$db = &new MySQL('localhost', 'harryf', 'secret', 'sitepoint');

// Create an instance of the Article class, passing it the MySQL
// object
$articles = &new Articles($db);

while ($row = $articles->fetch()) {
  echo '<pre>';
  print_r($row);
  echo '</pre>';
}
?>
```

In the above example, we instantiate the `MySQL` class outside the `Articles` class, then pass it to the `Articles` constructor as `Articles` is instantiated. `Articles` is then able to use the `MySQL` object to perform a specific query. In this case, `Articles` *aggregates* the `MySQL` object. Figure 2.2 illustrates this relationship with UML.

Figure 2.2. Aggregation



Composition

Composition occurs when one object "completely owns" another object. That is, the first object was responsible for creating (instantiating) the second object. There are many cases in which this can be useful, although, typically, composition is used when it's likely that the first object will be the only one that needs to use the second object.

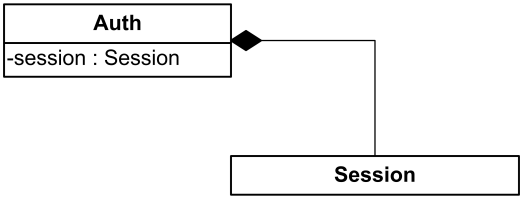
One example from Volume II, Chapter 1 is the `Auth` class, which **composes** an instance of the `Session` class, creating it in the constructor:

```
class Auth {
  …

  /**
   * Instance of Session class
   * @var Session
   */
  var $session;

  …

  function Auth (&$dbConn, $redirect, $md5 = true)
  {
    $this->dbConn = &$dbConn;
    $this->redirect = $redirect;
    $this->md5 = $md5;
    $this->session = &new Session();
    $this->checkAddress();
    $this->login();
  }
```

Because the `Auth` class needs to read and write to session variables, and only a limited number of other, unrelated classes in an application are likely also to need to use `Session`, it's logical that it gets to create its own `Session` object.

Figure 2.3 illustrates the composition in this example with UML.

Figure 2.3. Composition



Spotting the Difference

The general "thought test" to spot whether object A aggregates or composes object B is to ask, "What happens if object A dies? Will object B still be alive?" If object B outlives the death of object A, object A is said to aggregate object B. But if object B dies when object A dies, then object A is said to compose object B.

In terms of practical development, knowing when to apply aggregation or composition is important.

Aggregation has the advantage of lower overhead, because a single object will be shared by many other objects. Certainly, aggregating your database connection class is a good idea; composing it with every object that wants to make a query may require you to have multiple connections to your database, which will quickly halt your application when your site attracts high levels of traffic.

Composition has the advantage of making classes easier to work with from the outside. The code that uses the class doesn't have to worry about passing it the other objects it needs, which, in a complex application, can often become tricky and result in a design "work around." Composition also has the advantage that you know exactly which class has access to the composed object. With aggregation, another object sharing the aggregated object may do something to its state that "breaks" the object as far as the other classes that use it are concerned.

Polymorphism

Another powerful aspect of object oriented programming is polymorphism—the ability of different classes to share an **interface**.

An interface is one or more methods that let you use a class for a particular purpose. For example, you could have two database connection classes—one for MySQL and one for PostgreSQL. As long as they both offered a `query` method, you could use them interchangeably for running queries on different databases. The `query` method is a simple interface that the two classes share.

The classes sharing the same interface are often inherited from a parent class that makes the common methods available. Again, this is best understood by example.

First, we define an **abstract** base class, `Message`, which provides the common method `getMessage`. Beneath the `Message` class, we define **concrete** classes, each of which creates a specific message.

The terms "abstract" and "concrete" refer to class usage, in particular, whether a class is intended to be used directly or not. An abstract class is one in which some functionality or structure is to be shared by all subclasses, but is not intended to be used directly; typically, it has one or more empty methods that don't do anything useful. In other words, you're not supposed to create objects from an abstract class. A concrete class is a subclass of the abstract class from which you *can* create objects. Some languages, like Java, provide support for abstract classes within the language syntax—something PHP 4 doesn't offer. You can still use the *concept* of abstract classes when designing applications, though you might consider adding documentation to tell other developers working with the code that the class is abstract.

File: **14.php (excerpt)**

```
<?php
class Message {
  var $message;
  function setMessage($message)
  {
    $this->message = $message;
  }
  function getMessage()
  {
    return $this->message;
  }
}
```

```
class PoliteMessage extends Message {
  function PoliteMessage()
  {
    $this->setMessage('How are you today?');
  }
}

class TerseMessage extends Message {
  function TerseMessage()
  {
    $this->setMessage('Howzit?');
  }
}

class RudeMessage extends Message {
  function RudeMessage()
  {
    $this->setMessage('You look like *%&* today!');
  }
}
```

Now, we define the `MessageReader` class, which takes an array of `Message` objects through its constructor.

File: **14.php (excerpt)**

```
class MessageReader {
  var $messages;
  function MessageReader(&$messages) {
    $this->messages = &$messages;
    $this->readMessages();
  }
  function readMessages() {
    foreach ($this->messages as $message) {
      echo $message->getMessage() . '<br />';
    }
  }
}
```

The important thing to note here is that, as far as `MessageReader` is concerned, a "`Message` object" is any object that was instantiated from the `Message` class *or one of its subclasses*. Did you see how, inside the `readMessages` method, we call the `getMessage` method? This code will work on any object that has a `getMessage` method—including any subclass of `Message`.

Now, to prove the point, let's create some `Message` objects using our three subclasses at random:

File: **14.php (excerpt)**

```
$classNames =
  array('PoliteMessage', 'TerseMessage', 'RudeMessage');
$messages = array();
srand((float)microtime() * 1000000); // Prepares random shuffle
for ($i = 0; $i < 10; $i++) {
  shuffle($classNames);
  $messages[] = new $classNames[0]();
}
$messageReader = new MessageReader($messages);
?>
```

By creating the array `$classNames` and then repeatedly shuffling it, we can take the first element of the array and use it to create a new object:

```
  $messages[] = new $classNames[0]();
```

This is an example of a **variable function**. The expression `$classNames[0]` is evaluated to determine the name of the constructor (`PoliteMessage`, `TerseMessage`, or `RudeMessage`) to call.

Finally, the `$messages` array contains ten messages, randomly selected, and is passed to the constructor of `MessageReader` on instantiation.

Here's a sample result:

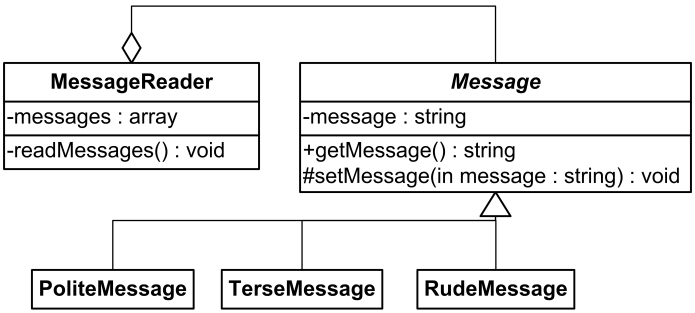
```
You look like *%&* today!
Howzit?
How are you today?
How are you today?
How are you today?
You look like *%&* today!
How are you today?
How are you today?
Howzit?
How are you today?
```

Each time we execute the script, the list is different.

Because all the concrete message classes share the same `getMethod` function (i.e. they implement the same interface), the `MessageReader` class is able to extract the data without knowing which particular type of message it's dealing with. The

ability for a group of related classes to work interchangeably is called **polymorphism**, and is illustrated in the UML diagram in Figure 2.4.

Figure 2.4. Polymorphism



This aspect of object oriented programming can be very powerful once you realize its worth. You might have a collection of objects representing HTML tags, for example, each being a subclass of a parent `HTMLTag` class, from which they all inherit a `render` method. Another class that handles the rendering of a page could take a collection of `HTMLTag` objects and create the page by calling each object's `render` method.

Further Reading

- *Object Oriented PHP: Paging Result Sets*: http://www.sitepoint.com/article/662
- *PHP References Explained*: http://www.zez.org/article/articleview/77/
- *PHP References*

  Part 1: http://www.onlamp.com/pub/a/php/2002/08/15/php_foundations.html

  Part 2: http://www.onlamp.com/pub/a/php/2002/09/12/php_foundations.html
- *PHP Reference Counting and Aliasing*: http://www.zend.com/zend/art/ref-count.php

3

PHP and MySQL

"On the Web today, content is king."
—Kevin Yank

In the "old days" of the Internet, most Web pages were nothing more than text files containing HTML. When people surfed to your site, your Web server simply made the file available to their browser, which parsed the contents and rendered something a human being could read. This approach was fine to start with, but as Websites grew and issues such as design and navigation became more important, developers realized that maintaining hundreds of HTML files was going to be a massive headache. To solve this problem, it became popular to separate variable content (articles, news items, etc.) from the static elements of the site—its design and layout.

Using a database as a repository to store variable content, a server side language such as PHP performs the task of fetching the data and placing it within a uniform "framework," the design and layout elements being reused. This means that modifying the overall look and feel of a site can be handled as a separate task from the addition or maintenance of content. Suddenly, running a Website is no longer a task that consumes a developer's every waking hour.

PHP supports all relational databases worth mentioning, including those commonly used in large companies, such as Oracle, IBM's DB2 and Microsoft's SQL Server. The two most noteworthy open source alternatives are PostgreSQL and

MySQL. Although PostgreSQL is arguably the better database, in that it supports more of the features that are common to relational databases, MySQL is better supported on Windows, and is a popular choice among Web hosts that provide support for PHP. These factors combine to make PHP and MySQL a very popular combination. This book is geared to the use of MySQL with PHP but it's important to remember that there are alternatives with full support for features such as stored procedures, triggers and constraints, many of which become important for applications with complex data structures.

This chapter covers all the common operations PHP developers have to perform when working with MySQL, from retrieving and modifying data, to searching and backing up a database. The examples focus on using a single table, so no discussion is made of table relationships here. For a full discussion of table relationships, see Kevin Yank's *Build Your Own Database Driven Website Using PHP & MySQL* (ISBN 0-9579218-1-0), or see an example of them in practice when we deal with user groups in Volume II, Chapter 1.

The examples used here work with a sample database called `sitepoint`, which contains the following single table:

File: **articles.sql**

```
CREATE TABLE articles (
  article_id INT(11)       NOT NULL AUTO_INCREMENT,
  title      VARCHAR(255)  NOT NULL DEFAULT '',
  intro      TEXT          NOT NULL,
  body       TEXT          NOT NULL,
  author     VARCHAR(255)  NOT NULL DEFAULT '',
  published  VARCHAR(11)   DEFAULT NULL,
  public     ENUM('0','1') NOT NULL DEFAULT '0',
  PRIMARY KEY (article_id),
  FULLTEXT KEY art_search (title, body, author)
)
```

A query to construct this table along with some sample data is available in the code archive, contained in the file `sql/articles.sql`. The table will be used for examples in later chapters of the book.

How do I access a MySQL database?

Connecting to MySQL with PHP couldn't be easier. It's essentially a two–step process; first connect to the MySQL database server itself, then inform MySQL of the database you want to connect to.

A Basic Connection

Here is a MySQL database connection in its simplest form:

File: **1.php**

```
<?php
$host   = 'localhost'; // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name

// Make connection to MySQL server
if (!$dbConn = mysql_connect($host, $dbUser, $dbPass)) {
  die('Could not connect to server');
}

// Select the database
if (!mysql_select_db($dbName, $dbConn)) {
  die('Could not select database');
}

echo 'Connection successful!';
// ... some code here using MySQL

// Close the connection when finished
mysql_close($dbConn);
?>
```

It's important to remember that MySQL is a separate server program, much like Apache. Both servers may run on the same physical computer (hence our use of `$host = 'localhost';` in the above example) but it's also possible to connect to MySQL on a remote computer, for example `$host = 'anothercomputer.com';`. To make matters a little more interesting, MySQL also has its own port number, which by default is 3306. PHP assumes that 3306 will be the port number but should you need to use a different one, all you need is `$host = 'anothercomputer.com:4321';`.

The other conceptual hurdle lies in understanding that a single MySQL server may provide access to many databases, which is why you need to select your database in PHP after connecting to the server.

Returning to the code above, there are a few things to note. First, I've placed in variables the values I need in order to connect to MySQL. This simply makes our lives easier; it's common to store this kind of information in separate files

that are included in every PHP script, making it possible to change many scripts at one time. We'll be looking at further tricks we can employ to make life easier in a moment.

The `mysql_connect` function does the work of connecting to a MySQL server. The value it returns is either a **link identifier** (a value supplied by PHP to identify the connection), or `FALSE`, meaning the connection failed.

```
if (!$dbConn = mysql_connect($host, $dbUser, $dbPass)) {
  die('Could not connect to server');
}
```

This `if` statement asks the question "Did I successfully connect to the MySQL server?" If not, it uses `die` to terminate the script.

Next, we've selected the database we want with `mysql_select_db`, using the same `if` statement technique:

```
if (!mysql_select_db($dbName, $dbConn)) {
```

Note that we provided the variable containing the link identifier as the second argument to `mysql_select_db`. We wouldn't usually need to do this (the argument is optional), but when a complex script juggles multiple database connections, this method can help ensure PHP knows which you're referring to.

Finally, we've used `mysql_close` to disconnect from the server again:

```
mysql_close($dbConn);
```

This occurs at the bottom of the script, once we've run some imaginary PHP code that used the connection. Closing the connection is generally optional—PHP automatically closes any connections after the script finishes[1].

[1]Connections made with `mysql_pconnect` are different. This function establishes a persistent connection to the database to be reused by multiple PHP scripts. Using a persistent connection makes your scripts slightly faster, as PHP no longer has to reconnect each time, but speed comes at a price: if your Website runs on a shared server, persistent connections may monopolize that server, resulting in other sites being unable to connect at times. In such environments, it's typical to either avoid `mysql_pconnect`, or configure MySQL so that connections are terminated the moment they stop doing anything, using a short connection timeout value.

Reusable Code

You've just seen the most simplistic way to connect to MySQL. It's often more useful, however, to "package" the above code in a function or a class so it can be reused.

As a function we could have:

File: **2.php**

```
<?php
function &connectToDb($host, $dbUser, $dbPass, $dbName)
{
  // Make connection to MySQL server
  if (!$dbConn = @mysql_connect($host, $dbUser, $dbPass)) {
    return false;
  }

  // Select the database
  if (!@mysql_select_db($dbName)) {
    return false;
  }

  return $dbConn;
}

$host   = 'localhost'; // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name

$dbConn = &connectToDb($host, $dbUser, $dbPass, $dbName);
?>
```

This reduces the process of connecting to MySQL and selecting a database to a single line (two if you count the `include` statement, which would point to a separate file containing the `connectToDb` function):

```
$dbConn = &connectToDb($host, $dbUser, $dbPass, $dbName);
```

Note that we've used the reference operator `&`. This operator and the role it plays were covered in detail in Chapter 2.

Be Lazy: Write Good Code

Scientists have now conclusively proven that knowledge of PHP is inversely proportional to free time but directly proportional to hair loss. The only way to prevent these effects is to learn how to write scalable, maintainable, and reusable code as early as possible. Taking advantage of classes and object orientation in PHP is a big step in the right direction. As a PHP developer, laziness is a virtue.

Going a step further, we can wrap this code in a class:

File: **Database/MySQL.php (in SPLIB) (excerpt)**

```
/**
 * MySQL Database Connection Class
 * @access public
 * @package SPLIB
 */
class MySQL {
  /**
   * MySQL server hostname
   * @access private
   * @var string
   */
  var $host;

  /**
   * MySQL username
   * @access private
   * @var string
   */
  var $dbUser;

  /**
   * MySQL user's password
   * @access private
   * @var string
   */
  var $dbPass;

  /**
   * Name of database to use
   * @access private
   * @var string
   */
  var $dbName;
```

```
/**
 * MySQL Resource link identifier stored here
 * @access private
 * @var string
 */
var $dbConn;

/**
 * Stores error messages for connection errors
 * @access private
 * @var string
 */
var $connectError;

/**
 * MySQL constructor
 * @param string host (MySQL server hostname)
 * @param string dbUser (MySQL User Name)
 * @param string dbPass (MySQL User Password)
 * @param string dbName (Database to select)
 * @access public
 */
function MySQL($host, $dbUser, $dbPass, $dbName)
{
  $this->host = $host;
  $this->dbUser = $dbUser;
  $this->dbPass = $dbPass;
  $this->dbName = $dbName;
  $this->connectToDb();
}

/**
 * Establishes connection to MySQL and selects a database
 * @return void
 * @access private
 */
function connectToDb()
{
  // Make connection to MySQL server
  if (!$this->dbConn = @mysql_connect($this->host,
      $this->dbUser, $this->dbPass)) {
    trigger_error('Could not connect to server');
    $this->connectError = true;
  // Select database
  } else if (!@mysql_select_db($this->dbName,$this->dbConn)) {
    trigger_error('Could not select database');
```

```
      $this->connectError = true;
    }
  }

  /**
   * Checks for MySQL errors
   * @return boolean
   * @access public
   */
  function isError()
  {
    if ($this->connectError) {
      return true;
    }
    $error = mysql_error($this->dbConn);
    if (empty($error)) {
      return false;
    } else {
      return true;
    }
  }
```

Now that may seem pretty overwhelming, but what's most important is not how the class itself is coded[2], but how you use it.

What's most important is that the task of connecting to MySQL is now reduced to the following:

File: **3.php**

```
<?php
// Include the MySQL class
require_once 'Database/MySQL.php';

$host   = 'localhost'; // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name

// Connect to MySQL
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);
?>
```

[2]In particular, the `trigger_error` function will be discussed in "How do I resolve errors in my SQL queries?" later in this chapter.

The point of using a class here is to get some practice using PHP's object model to deal with common tasks. If you're new to object oriented programming with PHP, the most important thing to remember at this stage is that you don't need to understand all the code you find in a class to be able to *use* it in your code.

We'll be making use of this class and others throughout the book to illustrate how object oriented programming aids the reuse of code and can save time when you're developing applications.

How do I fetch data from a table?

Being connected to a database is nice, sure. But what good is it if we can't get anything from it?

There are a number of ways to fetch data from MySQL, but the most widely used is probably `mysql_fetch_array` in conjunction with `mysql_query`.

We just need to add a little more to the `connectToDb` function we saw in "How do I access a MySQL database?" to fetch data from this table:

File: **4.php**

```
// Connect to MySQL
$dbConn = &connectToDb($host, $dbUser, $dbPass, $dbName);

// A query to select all articles
$sql = "SELECT * FROM articles ORDER BY title";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch rows from MySQL one at a time
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
  echo 'Title: '  . $row['title']  . '<br />';
  echo 'Author: ' . $row['author'] . '<br />';
  echo 'Body: '   . $row['body']   . '<br />';
}
```

Essentially, there are three steps to getting to your data:

1. First, place the necessary SQL query[3] in a string:

[3]If you are unfamiliar with Structured Query Language (SQL), I'll cover the basics throughout this chapter. For a more complete treatment, however, refer to *Build Your Own Database Driven Website Using PHP & MySQL, 2nd Edition* (ISBN 0–9579218–1–0).

```
$sql = "SELECT * FROM articles ORDER BY title";
```

It's handy to keep it in a separate variable, as when we get into writing more complex queries and something goes wrong, we can double-check our query with this one-liner:

```
echo $sql;
```

2. Next, tell MySQL to perform the query:

```
$queryResource = mysql_query($sql, $dbConn);
```

This can be confusing at first. When you tell MySQL to perform a query, it doesn't immediately give you back the results. Instead, it holds the results in memory until you tell it what to do next. PHP keeps track of the results with a **resource identifier**, which is what you get back from the `mysql_query` function. In the code above, we've stored the identifier in `$queryResource`.

3. Finally, use `mysql_fetch_array` to fetch one row at time from the set of results:

```
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC))
```

This places each row of the results in turn in the variable `$row`. Each of these rows will be represented by an array. By using the additional argument `MYSQL_ASSOC` with `mysql_fetch_array`, we've told the function to give us an array in which the keys correspond to column names in the table. If you omit the `MYSQL_ASSOC` argument, each column will appear twice in the array: once with a numerical index (i.e. `$row[0]`, `$row[1]`, etc.), and once with a string index (i.e. `$row['title']`, `$row['author']`, etc.). While this doesn't usually cause a problem, specifying the type of array value you want will speed things up slightly.

Using a `while` loop, as shown above, is a common way to process each row of the result set in turn. The loop effectively says, "Keep fetching rows from MySQL until I can't get any more", with the body of the loop processing the rows as they're fetched.

Forego Buffering on Large Queries

For large queries (that is, queries that produce large result sets), you can improve performance dramatically by telling PHP not to **buffer** the results of the query. When a query *is* buffered, the entire result set is retrieved from MySQL and stored in memory before your script is allowed to proceed. An

unbuffered query, on the other hand, lets MySQL hold onto the results until you request them, one row at a time (e.g. with `mysql_fetch_array`). Not only does this allow your script to continue running while MySQL performs the query, it also saves PHP from having to store all of the rows in memory at once.

PHP lets you perform unbuffered queries with `mysql_unbuffered_query`:

```
$queryResource = mysql_unbuffered_query($sql, $dbConn);
```

Of course, all good things come at a price—with unbuffered queries you can no longer use the `mysql_num_rows` function to count the number of rows. Obviously, as PHP doesn't keep a copy of the complete result set, it is unable to count the rows it contains! You also must fetch all rows in the result set from MySQL before you can make another query.

Although other functions exist for getting rows and cells from query results, like `mysql_fetch_object` and `mysql_result`, you can achieve more or less the same things with just `mysql_fetch_array`, and the consistency may help keep your code simple.

Fetching with Classes

Now that you're happy with the basics of fetching data from MySQL, it's time to build some more on the `MySQL` class from the last solution.

First, let's add a method to run queries from the class:

File: **Database/MySQL.php (in SPLIB) (excerpt)**

```
/**
 * Returns an instance of MySQLResult to fetch rows with
 * @param $sql string the database query to run
 * @return MySQLResult
 * @access public
 */
function &query($sql)
{
  if (!$queryResource = mysql_query($sql, $this->dbConn)) {
    trigger_error('Query failed: ' . mysql_error($this->dbConn)
                  . ' SQL: ' . $sql);
    return new MySQLResult($this, $queryResource);
  }
}
```

What this new method does is accept a variable containing an SQL statement, run it, then build a new object from another class, `MySQLResult` (described below). It then returns this object to the point where `query` was called.

Here's the code for that new class, `MySQLResult`:

File: **Database/MySQL.php (in SPLIB) (excerpt)**

```
/**
 * MySQLResult Data Fetching Class
 * @access public
 * @package SPLIB
 */
class MySQLResult {
  /**
   * Instance of MySQL providing database connection
   * @access private
   * @var MySQL
   */
  var $mysql;

  /**
   * Query resource
   * @access private
   * @var resource
   */
  var $query;

  /**
   * MySQLResult constructor
   * @param object mysql   (instance of MySQL class)
   * @param resource query (MySQL query resource)
   * @access public
   */
  function MySQLResult(&$mysql, $query)
  {
    $this->mysql = &$mysql;
    $this->query = $query;
  }

  /**
   * Fetches a row from the result
   * @return array
   * @access public
   */
  function fetch()
  {
```

```
    if ($row = mysql_fetch_array($this->query, MYSQL_ASSOC)) {
      return $row;
    } else if ( $this->size() > 0 ) {
      mysql_data_seek($this->query, 0);
      return false;
    } else {
      return false;
    }
  }

  /**
   * Checks for MySQL errors
   * @return boolean
   * @access public
   */
  function isError()
  {
    return $this->mysql->isError();
  }
}
```

Now, hold your breath just a little longer until you've seen what using these classes is like:

File: **5.php**

```
<?php
// Include the MySQL class
require_once 'Database/MySQL.php';

$host   = 'localhost'; // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name

// Connect to MySQL
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);

$sql = "SELECT * FROM articles ORDER BY title";

// Perform a query getting back a MySQLResult object
$result = $db->query($sql);

// Iterate through the results
while ($row = $result->fetch()) {
  echo 'Title: '  . $row['title']  . '<br />';
  echo 'Author: ' . $row['author'] . '<br />';
```

```
  echo 'Body: '    . $row['body']    . '<br />';
}
?>
```

If you're not used to object oriented programming, this may seem very confusing, but what's most important is to concentrate on how you can *use* the classes, rather than the detail hidden inside them. That's one of the joys of object oriented programming, once you get used to it. The code can get very complex behind the scenes, but all you need to concern yourself with is the simple "interface" (API) with which your code uses the class.

About APIs

It's common to hear the term API mentioned around classes. API stands for **Application Programming Interface**. What it refers to is the set of methods that act as "doors" to the functionality contained within a class. A well-designed API will allow the developer of the class to make radical changes behind the scenes without breaking any of the code that uses the class.

Compare using the MySQL classes with the earlier procedural code; it should be easy to see the similarities. Given that it's so similar, you may ask, "Why not stick to plain, procedural PHP?" Well, in this case, it hides many of the details associated with performing the query. Tasks like managing the connection, catching errors, and deciding what format to get the query results in are all handled behind the scenes by the class. Classes also make the implementation of global modifications (such as switching from MySQL to PostgreSQL) relatively painless (i.e. you could just switch to a `PostgreSQL` class that provided the same API).

How do I resolve errors in my SQL queries?

If something goes wrong when you try to deal with PHP and SQL together, it's often difficult to find the cause. The trick is to get PHP to tell you where the problem is, bearing in mind that you must be able to hide this information from visitors when the site goes live.

PHP provides the `mysql_error` function, which returns a detailed error message from the last MySQL operation performed.

It's best used in conjunction with the `trigger_error` function (which will be discussed in more detail in Chapter 10), which allows you to control the output of the error message. Let's modify the basic connection code we saw earlier:

File: **6.php (excerpt)**

```
// Make connection to MySQL server
if (!$dbConn = mysql_connect($host, $dbUser, $dbPass)) {
  trigger_error('Could not connect to server: ' . mysql_error());
  die();
}

// Select the database
if (!mysql_select_db($dbName)) {
  trigger_error('Could not select database: ' . mysql_error());
  die();
}
```

The same approach can be used with queries:

File: **6.php (excerpt)**

```
// A query to select all articles
$sql = "SELECT * FROM articles ORDER BY title";

// Run the query, identifying the connection
if (!$queryResource = mysql_query($sql, $dbConn)) {
  trigger_error('Query error ' . mysql_error() . ' SQL: ' . $sql);
}
```

It can be a good idea to return the complete query itself, as we've done in the above example, particularly when you've built it using PHP variables. This allows you to see exactly what query was performed and, if necessary, execute it directly against MySQL to identify exactly where it went wrong.

The `MySQL` class discussed above will automatically use `mysql_error` and `trigger_error` should it encounter a problem.

How do I add or modify data in my database?

Being able to fetch data from the database is a start, but how can you put it there in the first place?

Again, the answer is simple with PHP: use the `mysql_query` function combined with SQL commands `INSERT` and `UPDATE`. `INSERT` is used to create new rows in a table, while `UPDATE` is used to modify existing rows.

Inserting a Row

A simple `INSERT`, using the `articles` table defined at the start of this chapter, looks like this:

File: **7.php (excerpt)**

```
// A query to INSERT data
$sql = "INSERT INTO
          articles
        SET
          title  = '$title',
          body   = '$body',
          author = '$author'";

// Run the query, identifying the connection
if (!$queryResource = mysql_query($sql, $dbConn)) {
  trigger_error('Query error ' . mysql_error() . ' SQL: ' . $sql);
}
```

Updating a Row

Before you can use an `UPDATE` query, you need to be able to identify which row(s) of the table to update. In this example, I've used a `SELECT` query to obtain the unique `article_id` value for the article entitled "How to insert data":

File: **8.php (excerpt)**

```
// A query to select an article
$sql = "SELECT article_id FROM articles
        WHERE title='How to insert data'";

if (!$queryResource = mysql_query($sql, $dbConn)) {
  trigger_error('Query error ' . mysql_error() . ' SQL: ' . $sql);
}

// Fetch a single row from the result
$row = mysql_fetch_array($queryResource, MYSQL_ASSOC);

// A new title
$title = 'How to update data';
```

```
$sql = "UPDATE
          articles
        SET
          title='$title'
        WHERE
          article_id='" . $row['article_id'] . "'";

if (!$queryResource = mysql_query($sql, $dbConn)) {
  trigger_error('Query error ' . mysql_error() . ' SQL: ' . $sql);
}
```

In the above example, we used the `SELECT` query to find the ID for the row we wanted to update.

In practical Web applications, the `UPDATE` might occur on a page which relies on input from the Web browser, after the user has entered the value(s) using an HTML form, for example. It is possible that strings in this data might contain apostrophes, which would break the SQL, and impact upon security. In light of this, make sure you read "How do I solve database errors caused by quotes/apostrophes?", which covers SQL injection attacks.

Beware Global Updates

Be careful with `UPDATE` and remember to use a `WHERE` clause to indicate which rows to change.

For example, consider this query:

```
UPDATE articles SET title = 'How NOT to update data'
```

This will update *every* row of the table!

Another Class Action

Using the `MySQL` class last seen in "How do I fetch data from a table?", we can perform `INSERT` and `UPDATE` queries without any further modifications. Repeating the above examples using the class, we can first `INSERT` like this:

File: **`9.php` (excerpt)**

```
// Connect to MySQL
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);

$title  = 'How to insert data';
$body   = 'This is the body of the article';
```

```
$author = 'HarryF';

// A query to INSERT data
$sql = "INSERT INTO
          articles
        SET
          title  = '$title',
          body   = '$body',
          author = '$author'";

$db->query($sql);

if (!$db->isError()) {
  echo 'INSERT successful';
} else {
  echo 'INSERT failed';
}
```

We can `UPDATE` as follows:

File: **10.php (excerpt)**

```
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);

// A query to select an article
$sql = "SELECT article_id FROM articles
        WHERE title='How to insert data'";

$result = $db->query($sql);

$row = $result->fetch();

// A new title
$title = 'How to update data';

$sql = "UPDATE
          articles
        SET
          title='" . $title. "'
        WHERE
          article_id='" . $row['article_id'] . "'";

$db->query($sql);

if (!$db->isError()) {
  echo 'UPDATE successful';
} else {
```

```
  echo 'UPDATE failed';
}
```

How do I solve database errors caused by quotes/apostrophes?

Consider the following SQL statement:

```
INSERT INTO articles SET title='The PHP Anthology';
```

Perhaps the PHP script that made this query contained something like this:

```
<?php
$title = "The PHP Anthology";

$sql = "INSERT INTO articles SET title='$title';";

$result = mysql_query($sql, $dbConn);
?>
```

No problem so far, but look what happens if we change the title:

```
$title = "PHP's Greatest Hits";
```

Notice the apostrophe in the title? When we place this in the SQL statement, the query MySQL receives will be as follows:

```
INSERT INTO articles SET title='PHP's Greatest Hits';
```

See the problem? When MySQL reads that statement, it will only get as far as this:

```
INSERT INTO articles SET title='PHP'
```

The rest of the statement will cause a syntax error and the query will fail. It's easy enough to avoid this problem when you write the title yourself, but what happens when your script gets the value from user input?

The Great Escape

The solution is to **escape** the apostrophe character by adding a backslash before the apostrophe. The following query, for example, will work:

```
INSERT INTO articles SET title='PHP\'s Greatest Hits';
```

Backslashes and the ANSI SQL Standard

Note that using the backslash as an escape character is not standard ANSI SQL. If MySQL is the only database you'll ever use, the backslash may be acceptable, but the same SQL statement run on another database may well fail. According to ANSI SQL we should escape apostrophes with another single apostrophe:

```
INSERT INTO articles SET title='PHP''s Greatest Hits';
```

The question is, how do we make sure all our apostrophes are escaped when we build a query on the fly in PHP? Dealing with this situation has become rather confusing due to the number of alternative solutions:

- First we have the `php.ini` setting `magic_quotes_gpc`. **Magic quotes** is a feature of PHP which, when turned on, automatically escapes single and double quotes, as well as backslashes and null characters found in incoming GET, POST and cookie variables, by adding backslashes to the strings. This may sound great, but in practice it quickly makes for trouble, typically where forms are involved.

  Say you have a form which is used for editing articles. Your script takes the text the user enters and inserts it into MySQL. Now, if the user fails to complete some important field, you might want to re-display the details that have been entered in the form so far. With magic quotes on you'd have to strip out all the slashes it added to the values (with PHP's `stripslashes` function)!

  Then, what if you wanted to run the code on a server where `magic_quotes_gpc` is disabled? Your code would then have to check to see if magic quotes is switched on and bypass the use of `stripslashes`. Headaches are inevitable, and if you make a mistake and end up with spurious backslashes stored in your database[4], you may have a painful cleanup process ahead of you.

  Magic quotes is discussed in some detail in Chapter 1. If you *do* switch off `magic_quotes_gpc` as I advise, you should be aware of the potential risks to security. See the section called "SQL Injection Attacks" below and Appendix C.

[4]It continually amazes me how many professionally designed sites fail to handle character escaping properly! Keep an eye out for unexpected backslashes in your own Web travels. See Chapter 1 for my advice on how best to avoid this on your own sites.

- Next, we have the PHP function `addslashes`. Applied to any string, `addslashes` will use backslashes to escape single quotes, double quotes, backslashes and null characters. This makes it an effective means to escape strings for use in queries.

  If magic quotes is on, of course, you must *not* use `addslashes`, or characters would be escaped twice! To solve this conflict, you can check if magic quotes is enabled with the function `get_magic_quotes_gpc`, which returns `TRUE` if magic quotes is enabled and `FALSE` if it isn't. You can bundle up this test with a function as follows:

```
<?php
function safeAddSlashes($string)
{
  if (get_magic_quotes_gpc()) {
    return $string;
  } else {
    return addslashes($string);
  }
}
?>
```

- The third way, which is very similar to `addslashes`, uses the function `mysql_escape_string` or `mysql_real_escape_string` (the latter was added in PHP 4.3.0). These functions use the underlying MySQL C++ API (provided by the developers of MySQL, rather than the developers of PHP) to escape special characters.

  These functions escape null characters, line feeds, carriage returns, backslashes, single quotes, double quotes, and end-of-file characters. Since PHP 4.3.0, both functions have used the current character set for the connection between PHP and MySQL. There is, therefore, no difference between these two functions in the latest PHP versions, so feel free to stick with the shorter of the two, `mysql_escape_string`, if your server is up-to-date.

  As this method is, in effect, the built-in mechanism provided by MySQL for escaping strings, I recommend it over `addslashes` or `magic_quotes_gpc`. Of course, if you want your SQL to port well to other databases, you may want to consider "hiding" the function call within a class method, which allows you to swap out the class—including the escape mechanism—when moving to a different database.

Again, if you do not otherwise handle the magic quotes issue, you'll have to check whether `magic_quotes_gpc` is on:

```
<?php
function safeEscapeString($string)
{
  if (get_magic_quotes_gpc()) {
    return $string;
  } else {
    return mysql_real_escape_string($string);
  }
}
?>
```

The scripts in this book make use of the `MagicQuotes/strip_quotes.php` include file introduced in Chapter 1 and included in the code archive to effectively switch off magic quotes on servers where it is enabled, so the solutions that follow will use `mysql_real_escape_string` freely. I'd encourage you to do the same in your own scripts if you feel confident escaping quotes and other special characters yourself.

SQL Injection Attacks

An **SQL injection attack** occurs when an attacker exploits a legitimate user input mechanism on your site to send SQL code that your unsuspecting script will pass on to the database to execute. The golden rule: *escape all data from external sources* before letting it near your database. That rule doesn't just apply to `INSERT` and `UPDATE` queries, but also to `SELECT` queries.

No doubt many PHP developers have been saved from the worst SQL injection attacks by the limitations of MySQL, which will only allow a single SQL statement to be performed with each call to `mysql_query`. On other databases, the effect of an SQL injection can be disastrous, as an attacker can send a second query that, for example, deletes the entire contents of a table. With MySQL, however, problems can still occur, as the following code demonstrates:

```
$sql = "SELECT * FROM users
        WHERE username='" . $_POST['username'] . "'
        AND password='" . $_POST['password'] . "'";

echo 'Query: ' . $sql . '<br />';

$result = mysql_query($sql);
```

```
$rows = mysql_num_rows($result);

if ($rows > 0) {
  echo 'You are logged in!<br />';
} else {
  echo 'You are not allowed here!<br />';
}
?>
<form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">
<input type="text" name="username" /><br />
<input type="text" name="password" /><br />
<input type="submit" />
</form>
```

A savvy attacker could simply enter the following in the form's password field:

```
' OR username LIKE '%
```

Assuming magic quotes is disabled on your server, and you have no other measures in place to prevent it, this clever attack alters the meaning of the query:

```
SELECT * FROM users
WHERE username='' AND password='' OR username LIKE '%'
```

The modified query will select *all* records in the user table! When the script checks whether any users matched the supplied user name and password combination, it will see this big result set and grant access to the site!

This can be prevented if we escape the incoming variables:

```
$sql = "SELECT * FROM users
    WHERE username='" . safeEscapeString($_POST['username']) . "'
    AND password='" . safeEscapeString($_POST['password']) . "'";
```

In some cases, depending on the circumstances, this may not be necessary. But if you value your sleep, remember that golden rule: *escape all data from external sources*.

How do I create flexible SQL statements?

SQL is a powerful language for manipulating data. Using PHP, we can construct SQL statements out of variables, which can be useful for sorting a table by a single column or displaying a large result set across multiple pages.

Here is a simple example that lets us sort the results of a query by a table column:

File: **11.php (excerpt)**

```
// A query to select all articles
$sql = "SELECT * FROM articles";

// Initialize $_GET['order'] if it doesn't exist
if (!isset($_GET['order']))
  $_GET['order'] = FALSE;

// Use a conditional switch to determine the order
switch ($_GET['order']) {
  case 'author':
    // Add to the $sql string
    $sql .= " ORDER BY author";
    break;
  default:
    // Default sort by title
    $sql .= " ORDER BY title";
    break;
}

// Run the query, identifying the connection
if (!$queryResource = mysql_query($sql, $dbConn)) {
  trigger_error('Query error ' . mysql_error() . ' SQL: ' . $sql);
}
?>
<table>
<tr>
<th><a href="<?php echo $_SERVER['PHP_SELF']; ?>?order=title"
  >Title</a></th>
<th><a href="<?php echo $_SERVER['PHP_SELF']; ?>?order=author"
  >Author</a></th>
</tr>
<?php
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
  echo "<tr>\n";
  echo "<td>" . $row['title'] . "</td>";
  echo "<td>" . $row['author'] . "</td>";
  echo "</tr>\n";
}
?>
</table>
```

Within the `switch` statement, I've generated part of the SQL statement "on the fly," depending on a GET variable the script receives from the browser.

This general approach can be extended to `WHERE` clauses, `LIMIT` clauses, and anything else you care to consider with SQL. We'll look at this in more detail when we construct a paged result set in Chapter 9.

Persistence Layers: Database Interaction Without SQL

Persistence layers are becoming popular, and are well supported in PHP today. A persistence layer is a collection of classes that represents the tables in your database, providing you with an API through which all data exchanged between the database and the PHP application passes. This generally takes away the need for you to write SQL statements by hand, as the queries are generated and executed automatically by the PHP classes that represent the data.

Because SQL is a fairly well defined standard, it also becomes possible to have a persistence layer generated automatically. A program can examine your database schema and produce the classes that will automatically read and update it. This can be a very significant time saver; simply design your database, run the code generation tool, and the rest is "just" a matter of formatting a little (X)HTML.

A prime example of a persistence layer is PEAR::DB_DataObject[1], which builds on top of the PEAR::DB database abstraction library, and automatically generates a layer of classes with which to access your tables.

Persistence layers in general and PEAR::DB_DataObject in particular are discussed in "Do I really need to write SQL?".

How do I find out how many rows I've selected?

It's often useful to be able to count the number of rows returned by a query before you do anything with them, such as when you're splitting results across pages or producing statistical information. When selecting results, you can use either PHP or MySQL to count the number of rows for you.

Counting Rows with PHP

With PHP, the function `mysql_num_rows` returns the number of rows selected, but its application can be limited when you use unbuffered queries (see "How

[1] http://pear.php.net/package-info.php?package=DB_DataObject

do I fetch data from a table?"). The following code illustrates the use of `mysql_num_rows`:

File: **12.php (excerpt)**

```
// A query to select all articles
$sql = "SELECT * FROM articles ORDER BY title";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch the number of rows selected
$numRows = mysql_num_rows($queryResource);

echo $numRows . ' rows selected<br />';

// Fetch rows from MySQL one at a time
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
    echo 'Title: '  . $row['title']  . '<br />';
    echo 'Author: ' . $row['author'] . '<br />';
    echo 'Body: '   . $row['body']   . '<br />';
}
```

The `mysql_num_rows` function, demonstrated in the above example, takes a result set resource identifier and returns the number of rows in that result set.

Note that the related function, `mysql_num_fields`, can be used to find out how many columns were selected. This can be handy when you're using queries like `SELECT * FROM` table, but you don't know how many columns you've selected.

Counting Rows with MySQL

The alternative approach is to use MySQL's `COUNT` function within the query. This requires that you perform two queries—one to count the results and one to actually *get* the results—which will cost you a little in terms of performance.

Here's how you could use the MySQL `COUNT` function:

File: **13.php (excerpt)**

```
// A query to select all articles
$sql = "SELECT COUNT(*) AS numrows FROM articles";

// Query to count the rows returned
$queryResource = mysql_query($sql, $dbConn);
```

```
$row = mysql_fetch_array($queryResource, MYSQL_ASSOC);

echo $row['numrows'] . " rows selected<br />";

// A query to select all articles
$sql = "SELECT * FROM articles ORDER BY title";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch rows from MySQL one at a time
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
  echo 'Title: '  . $row['title']  . '<br />';
  echo 'Author: ' . $row['author'] . '<br />';
  echo 'Body: '   . $row['body']   . '<br />';
}
```

Notice we used an **alias** to place the result of the `COUNT` function?

```
SELECT COUNT(*) AS numrows FROM articles
```

We do this so that the number of rows can be identified later using `$row['numrows']`. The alternative would have been to omit the alias:

```
SELECT COUNT(*) FROM articles
```

This would require that we access the information as `$row['COUNT(*)']`, which can make the code confusing to read.

When we use the `COUNT` function, it becomes important to construct queries on the fly as we saw in "How do I create flexible SQL statements?". You need to make sure your `COUNT` query contains the same `WHERE` or `LIMIT` clauses you used in the "real" query. For example, if the query we're actually using to fetch data is:

```
SELECT * FROM articles WHERE author='HarryF'
```

In PHP, we'll probably want something like this:

File: **14.php (excerpt)**

```
// Define reusable "chunks" of SQL
$table = " FROM articles";
$where = " WHERE author='HarryF'";
$order = " ORDER BY title";

// Query to count the rows returned
```

```
$sql = "SELECT COUNT(*) as numrows" . $table . $where;

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

$row = mysql_fetch_array($queryResource, MYSQL_ASSOC);

echo $row['numrows'] . " rows selected<br />";

// A query to fetch the rows
$sql = "SELECT * " . $table . $where . $order;

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch rows from MySQL one at a time
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
    echo 'Title: '  . $row['title']  . '<br />';
    echo 'Author: ' . $row['author'] . '<br />';
    echo 'Body: '   . $row['body']   . '<br />';
}
```

Row Counting with Classes

Let's look again at the classes we've been developing throughout this section. We can add the ability to find out the number of rows selected by introducing the following method to the `MySQLResult` class:

File: **Database/MySQL.php (in SPLIB) (excerpt)**

```
  /**
   * Returns the number of rows selected
   * @return int
   * @access public
   */
  function size()
  {
    return mysql_num_rows($this->query);
  }
```

Here's how to use it:

File: **15.php (excerpt)**

```
// Connect to MySQL
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);
```

```
// Select all results for a particular author
$sql = "SELECT * FROM articles WHERE author='HarryF'";

$result = $db->query($sql);

echo "Found " . $result->size() . " rows";
```

Counting Affected Rows

It's also possible to find out how many rows were affected by an `UPDATE`, `INSERT` or `DELETE` query, using the PHP function `mysql_affected_rows`. Use of `mysql_affected_rows` is not common in typical PHP applications, but it could be a good way to inform users that, "You've just deleted 1854 records from the Customers table. Have a nice day!"

Unlike `mysql_num_rows`, which takes a result set resource identifier as its argument, `mysql_affected_rows` takes the database connection identifier. It returns the number of rows affected by the last query that modified the database, for the specified connection.

Here's how `mysql_affected_rows` can be used:

File: **16.php (excerpt)**

```
// Connect to MySQL
$dbConn = &connectToDb($host, $dbUser, $dbPass, $dbName);

// A query which updates the database
$sql = "UPDATE
          articles
        SET
          author='The Artist Formerly Known as...'
        WHERE
          author='HarryF'";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch the number of rows affected
$changedRows = mysql_affected_rows($dbConn);

echo $changedRows . ' rows changed<br />';
```

As situations in which `mysql_affected_rows` is needed are uncommon, I'll omit this from the `MySQLResult` class in the interests of keeping things simple.

After inserting a row, how do I find out its row number?

When you're dealing with `AUTO_INCREMENT` columns in database tables, it's often useful to be able to find out the ID of a row you've just inserted, so that other tables can be updated with this information. That, after all, is how relationships between tables are built. PHP provides the function `mysql_insert_id`, which, when given a link identifier, returns the ID generated by the last `INSERT` performed with that connection. Here's how `mysql_insert_id` can be used:

File: **17.php (excerpt)**

```
// A query to insert a row
$sql = "INSERT INTO
          articles
        SET
          title='How to use mysql_insert_id()',
          body='This is an example',
          author='HarryF'";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch the inserted ID
$insertID = mysql_insert_id($dbConn);

echo 'The new row has ID: ' . $insertID;
```

Class Insert ID

To use this functionality in our `MySQLResult` class, add the following method:

File: **Database/MySQL.php (in SPLIB) (excerpt)**

```
  /**
   * Returns the ID of the last row inserted
   * @return int
   * @access public
   */
  function insertID()
  {
    return mysql_insert_id($this->mysql->dbConn);
  }
```

As you might guess, using this method is quite straightforward:

File: **18.php (excerpt)**

```
// Include the MySQL class
require_once 'Database/MySQL.php';

$host   = 'localhost';   // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name

$db = &new MySQL($host, $dbUser, $dbPass, $dbName);

// A query to insert a row
$sql="INSERT INTO
        articles
      SET
        title='How to use mysql_insert_id()',
        body='This is an example',
        author='HarryF'";

$result = $db->query($sql);

echo 'The new row as ID: ' . $result->insertID();
```

How do I search my table?

Some people are just impatient; rather than trawling your site with the friendly navigation system you've provided, they demand information now! Hence PHP developers like you and I are required to implement search features to provide visitors a "short cut" to find the information they want. In the days of storing all content in the form of HTML files, this could be quite a problem, but now that you're using a database to store content, searching becomes much easier.

Select What You `LIKE`

The most basic form of search occurs against a single column, with the `LIKE` operator:

```
SELECT * FROM articles WHERE title LIKE 'How %'
```

The `%` is a wildcard character. The above statement will select all articles in which the title begins with the word "How." MySQL also has support for POSIX regular

expressions (the same as PHP's `ereg` functions). Using the `RLIKE` operator, we can compare a column using a regular expression:

```
SELECT * FROM articles WHERE title RLIKE '^How '
```

The above statement also selects every article in which the title begins with "How" followed by a space.

With some work, these operators provide everything needed to explore your data. Where the above approach becomes a burden is in performing a search against multiple columns. For example,

```
SELECT * FROM articles
WHERE title LIKE '%how%' OR body LIKE '%how%'
```

For larger tables, this can require you to write some very complicated and unpleasant queries.

FULLTEXT Searches

MySQL provides an alternative that does most of the work for you—the `FULLTEXT` index. Indexes in a database are much like the index of a book; they provide a means to locate information within the database quickly from an organized list. A `FULLTEXT` index allows you to search a table for particular words.

`FULLTEXT` indexes were introduced to MySQL with version 3.23. The implementation at this point was fairly limited but still useful for basic searching, which is what I'll demonstrate here. In MySQL version 4.0.1, this functionality was extended to provide a full Boolean search mechanism that gives you the ability to build something like Google™'s advanced search features. `FULLTEXT` indexes also allow each result to be returned with a "relevance" value so that, for example, the results of multiple word searches can be displayed in terms of how well each result matches that user's particular search.

To take advantage of `FULLTEXT` indexes, you first need to instruct MySQL to begin building an index of the columns you want to search:

```
ALTER TABLE articles ADD FULLTEXT art_search (title, body, author)
```

Once you've done that, you need to `INSERT` a new record (or modify an existing one) to get MySQL to build the index. You also need at least three records in the database for `FULLTEXT` searches to work, because non-Boolean searches will only return results if the search string occurred in less than 50% of the rows in

the table (if there are only two rows in the table, and your search matches one row, that makes 50%). One final thing to be aware of is that `FULLTEXT` searches will only match searches of more than three letters; the indexing mechanism ignores words of three characters or less, to avoid having to build a massive index. This is much like the index of a book; you'd be pretty surprised to discover in a book's index exactly which pages the word "the" appeared on!

Here's a basic `FULLTEXT` search:

```
SELECT * FROM articles
WHERE MATCH (title,body,author) AGAINST ('MySQL');
```

This search will return all rows where either the `title`, `body` or `author` contained the word "MySQL."

Another use for `FULLTEXT` indexes is in a search which returns the relevance for each result. For example:

File: **19.php (excerpt)**

```
// Select all rows but display relvance
$sql = "SELECT
          *, MATCH (title, body, author)
        AGAINST
          ('The PHP Anthology Released Long Word Matching')
        AS
          score
        FROM
          articles
        ORDER BY score DESC";

// Run the query, identifying the connection
$queryResource = mysql_query($sql, $dbConn);

// Fetch rows from MySQL one at a time
while ($row = mysql_fetch_array($queryResource, MYSQL_ASSOC)) {
  echo 'Title: '  . $row['title']  . '<br />';
  echo 'Author: ' . $row['author'] . '<br />';
  echo 'Body: '   . $row['body']   . '<br />';
  echo 'Score: '  . $row['score']  . '<br />';
}
```

The alias `score` now contains a value that identifies how relevant the row is to the search. The value is not a percentage, but simply a measure; 0 means no match was made at all. Matching a single word will produce a value around 1. The more words that match, the bigger the number gets, so a five word match

ranking will produce a relevance score around 13. MySQL's relevance algorithm is designed for large tables, so the more data you have, the more useful the relevance value becomes.

Overall, MySQL's `FULLTEXT` search capabilities provide a mechanism that's easy to implement and delivers useful results.

How do I back up my database?

The bigger a database becomes, the more nerve wracking it can be not to have a backup of the data it contains. What if your server crashes and everything is lost? Thankfully, MySQL comes with two alternatives: a command line utility called `mysqldump`, and a query syntax for backing up tables.

Here's how you can export the contents of a database from the command line with `mysqldump`:

```
mysqldump -uharryf -psecret sitepoint > sitepoint.sql
```

This command will log in to MySQL as user "harryf" (`-uharryf`) with the password "secret" (`-psecret`) and output the contents of the `sitepoint` database to a file called `sitepoint.sql`. The contents of `sitepoint.sql` will be a series of queries that can be run against MySQL, perhaps using the `mysql` utility to perform the reverse operation from the command line:

```
mysql -uharryf -psecret sitepoint < sitepoint.sql
```

Using the PHP function `system`, you can execute the above command from within a PHP script (this requires you to be logged in and able to execute PHP scripts from the command line). The following class puts all this together in a handy PHP form that you can use to keep regular backups of your site.

File: **`Database/MySQLDump.php` (in `SPLIB`)**

```
/**
 * MySQLDump Class<br />
 * Backs up a database, creating a file for each day of the week,
 * using the mysqldump utility.<br />
 * Can compress backup file with gzip of bzip2<br />
 * Intended for command line execution in conjunction with
 * cron<br />
 * Requires the user executing the script has permission to execute
 * mysqldump.
 * <code>
```

```
 * $mysqlDump = new MySQLDump('harryf', 'secret', 'sitepoint',
 *                            '/backups');
 * $mysqlDump->backup();
 * </code>
 * @access public
 * @package SPLIB
 */
class MySQLDump {
  /**
   * The backup command to execute
   * @private
   * @var string
   */
  var $cmd;

  /**
   * MySQLDump constructor
   * @param string dbUser (MySQL User Name)
   * @param string dbPass (MySQL User Password)
   * @param string dbName (Database to select)
   * @param string dest (Full dest. directory for backup file)
   * @param string zip (Zip type; gz - gzip [default], bz2 - bzip)
   * @access public
   */
  function MySQLDump($dbUser, $dbPass, $dbName, $dest,
                     $zip = 'gz')
  {
    $zip_util = array('gz'=>'gzip','bz2'=>'bzip2');
    if (array_key_exists($zip, $zip_util)) {
      $fname = $dbName . '.' . date("w") . '.sql.' . $zip;
      $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
                   ' ' . $dbName . '| ' . $zip_util[$zip] . ' >' .
                   $dest . '/' . $fname;
    } else {
      $fname = $dbName . '.' . date("w") . '.sql';
      $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
                   ' ' . $dbName . ' >' . $dest . '/' . $fname;
    }
  }

  /**
   * Runs the constructed command
   * @access public
   * @return void
   */
  function backup()
```

```
  {
    system($this->cmd, $error);
    if ($error) {
      trigger_error('Backup failed: ' . $error);
    }
  }
}
```

The `MySQLDump` class makes some assumptions about your operating system configuration. It assumes the `mysqldump` utility is available in the path of the user that executes this script. If the `gzip` or `bzip2` utilities are used, they also need to be present in the path of the user who executes this script. `bzip2` provides better compression than `gzip`, helping save disk space.

The following code demonstrates how this class can be used:

File: **20.php**

```
<?php
// Include the MySQLDump class
require_once 'Database/MySQLDump.php';

$dbUser = 'harryf';               // db User
$dbPass = 'secret';               // db User Password
$dbName = 'sitepoint';            // db name
$dest   = '/home/harryf/backups'; // Path to directory
$zip    = 'bz2';                  // ZIP utility to compress with

// Instantiate MySQLDump
$mysqlDump = new MySQLDump($dbUser, $dbPass, $dbName, $dest,
                           $zip);

// Perform the backup
$mysqlDump->backup();
?>
```

The `$dest` variable specifies the path to the directory in which the backup file should be placed. The filename that's created will be in this format:

`databaseName``.`*`dayOfWeek`*`.sql.`*`zipExtension`*

For example:

```
sitepoint.1.sql.bz2
```

The *`dayOfWeek`* element can be any number from 0 to 6 (0 being Sunday and 6 being Saturday). This provides a weekly "rolling" backup, the files for the following

week overwriting those from the previous week. This should provide adequate backups, giving you a week to discover any serious problems, and without requiring excessive disk space to store the files.

The use of a ZIP utility is optional. If the value of the `$zip` variable is not one of `gz` or `bz2`, then no compression will be made, although for large databases it's obviously a good idea to use a compression tool to minimize the amount of disk space required.

This class is intended for use with the `crontab` utility, which is a Unix feature that allows you to execute scripts on a regular (for example, daily) basis.

MySQL also provides the SQL statements `BACKUP TABLE` and `RESTORE TABLE`, which allow you to copy the contents of a table to another location on your file system. Unlike the `mysqldump` utility, tables backed up in this way preserve their original format (which is not human-readable) but this mechanism does not require access to a command line utility, so it could be executed via a Web page.

The general syntax for these statements is as follows:

```
BACKUP TABLE tbl_name[, tbl_name ...]
TO '/path/to/backup/directory'
```

```
RESTORE TABLE tbl_name[, tbl_name ...]
FROM '/path/to/backup/directory'
```

Note that on Windows systems it's best to specify paths using forward slashes (e.g. `C:/backups`).

By combining these with some of the "introspection" statements MySQL provides, we can backup our database using the `MySQL` class we built in this chapter. To start with, we need to get a list of tables in the database, which is quickly achieved using the `SHOW TABLES` query syntax:

File: **21.php (excerpt)**

```
<?php
// Include the MySQL class
require_once 'Database/MySQL.php';

$host   = 'localhost'; // Hostname of MySQL server
$dbUser = 'harryf';    // Username for MySQL
$dbPass = 'secret';    // Password for user
$dbName = 'sitepoint'; // Database name
```

```
$db = &new MySQL($host, $dbUser, $dbPass, $dbName);

// A query to show the tables in the database
$sql = "SHOW TABLES FROM sitepoint";

// Execute query
$result = $db->query($sql);
```

We also store the number of rows returned by this query to help us format the string we'll use to build the `BACKUP` query:

File: **21.php (excerpt)**

```
// Get the number of tables found
$numTables = $result->size();
```

Next, we loop through the results, building a comma-separated list of tables to back up:

File: **21.php (excerpt)**

```
// Build a string of table names
$tables = '';
$i = 1;
while ($table = $result->fetch()) {
  $tables .= $table['Tables_in_sitepoint'];
  if ($i < $numTables) {
    $tables .= ', ';
  }
  $i++;
}
```

Finally, we use the `BACKUP TABLE` query syntax to copy the tables to a directory of our choice (to which, of course, the script that executes this query needs permission to write):

File: **21.php (excerpt)**

```
// Build the backup query
$sql = "BACKUP TABLE $tables TO '/home/harryf/backup'";

// Perform the query
$db->query($sql);

if (!$db->isError()) {
  echo 'Backup succeeded';
} else {
  echo 'Backup failed';
```

```
}
?>
```

How do I repair a corrupt table?

Although it shouldn't happen, occasionally data stored in MySQL becomes corrupted. The are a number of (rare) circumstances where this can happen; Windows is particularly susceptible as it doesn't have the robust file locking mechanism of Unix-based systems. Servers with heavy loads, on which `INSERT` and `UPDATE` queries are common alongside `SELECT`s are also likely to suffer occasional corruption. Assuming you're using the MyISAM table type (you'll be using this unless you've specified otherwise), there's good news; in general , you should be able to recover all the data in a corrupt table.

Note that the information provided here represents a quick reference for those times when you need help fast. It's well worth reading the MySQL manual on Disaster Prevention and Recovery[2] so that you know exactly what you're doing.

MySQL provides two important utilities to deal with corrupt tables, as well as a handy SQL syntax for those who can get to the MySQL command line.

First, the `perror` utility can be run from the command line to give you a rough idea of what MySQL error codes mean. The utility should be available from the `bin` subdirectory of your MySQL installation. Typing `perror 145`, for example, will tell you:

```
145 = Table was marked as crashed and should be repaired
```

From the command line, you can then use the utility `myisamchk` to check the database files themselves:

```
myisamchk /path/to/mysql/data/table_name
```

To repair a corrupt table with `myisamchk`, the syntax is as follows:

```
myisamchk -r /path/to/mysql/data/table_name
```

Using SQL, you can also check and fix tables using a query like this:

```
CHECK TABLE articles
```

And this:

[2] http://www.mysql.com/doc/en/Disaster_Prevention.html

```
REPAIR TABLE articles
```

With luck, you'll need to use these commands only once or twice, but it's worth being prepared in advance so you can react effectively (without even a hint of panic creeping into your actions).

Do I really need to write SQL?

A good quality to posses as a programmer is laziness—the desire to do as much as possible with the minimum amount of effort. Although you may not want to cite it as one of your strong points in a job interview, being motivated to make life easier for yourself is a significant boon in developing a well designed application.

Now that you've read this chapter on PHP and MySQL, I think it's a good time to reveal that I hate SQL not because there's anything wrong with it, as such, but because it always causes me grief. If there's a syntax error in my PHP, for example, PHP will find it for me. But PHP *won't* find errors in SQL statements, and MySQL error messages can be less than revealing. If I'm hand coding SQL in an application, I'll spend a fair amount of time debugging it—time I could have spent taking it easy!

What if you could avoid having to write SQL statements altogether? If you think back to "How do I create flexible SQL statements?", where we constructed SQL strings "on the fly" based on incoming variables, you may have had an inkling that there would be some kind of generic solution to make generating SQL even easier. Well, there is! It's called PEAR::DB_DataObject[3].

`DB_DataObject` is a class that encapsulates the process of writing SQL statements in a simple API. It takes advantage of the native "grammar" of SQL and presents you with a mechanism that removes almost any need to write any SQL yourself. As an approach to dealing with databases, it's usually described as a **database persistence layer**, or, alternatively, as using the **Data Access Objects** (DAO) design pattern. You'll find further discussion of the general techniques used by `DB_DataObject` at the end of this chapter.

Here, I'll provide a short introduction to `DB_DataObject` to get you started, as it's a subject that could easily absorb a whole chapter if examined in depth. The DB_DataObject documentation[4] on the PEAR Website should provide you

[3] http://pear.php.net/DB_DataObject
[4] http://pear.php.net/manual/en/package.database.db-dataobject.php

with plenty of further help. The version we used here was 1.1; note that it requires that you have the `PEAR::DB` database abstraction library installed (see Appendix D for more information on installing PEAR libraries).

The first step in getting started with `DB_DataObject` is to point it at your database and tell it to generate the `DataObject` classes that will constitute your interface with the tables. `DB_DataObject` automatically examines your database, using MySQL's introspection functionality, and generates a class for each table in the database, as well as a configuration file containing the details of the columns defined by the table. To let `DB_DataObject` know where your database can be found, you need to provide it a configuration file like this one:

File: **db_dataobject.ini**

```
[DB_DataObject]
; PEAR::DB DSN
database        = mysql://harryf:secret@localhost/sitepoint
; Location where sitepoint.ini schema file should be created
schema_location = /htdocs/phpanth/SPLIB/ExampleApps/DataObject
; Location where DataObject classes should be created
class_location  = /htdocs/phpanth/SPLIB/ExampleApps/DataObject
; Prefix for including files from your code
require_prefix  = ExampleApps/DataObject
; Classes should be prefixed with this string e.g. DataObject_User
class_prefix    = DataObject_
; Debugging information: 0=off, 1=display sql, 2=display results,
; 3=everything
debug = 0
; Prevent SQL INSERT, UPDATE or DELETE from being performed
debug_ignore_updates = false
; Whether to die of error with a PEAR_ERROR_DIE or not
dont_die = false
```

The above `ini` file obeys the same formatting rules as `php.ini`. Most important is the first line, which is a PEAR::DB DSN string that defines the variables needed to connect to the database. This file is used both to generate the `DataObject` classes, and to use them in performing queries.

With that in place, we can use this script (which must be run from the command line) to generate the classes:

File: **22.php**

```
<?php
// Builds the DataObjects classes
$_SERVER['argv'][1] = 'db_dataobject.ini';
```

```
require_once 'DB/DataObject/createTables.php';
?>
```

This script automatically creates the class files we need in order to access the database. Here's an example developed for the `articles` table:

File: **ExampleApps/DataObject/Articles.php (in SPLIB)**

```
<?php
/**
 * Table Definition for articles
 */
require_once 'DB/DataObject.php';

class DataObject_Articles extends DB_DataObject
{

  ###START_AUTOCODE
  /* the code below is auto generated do not remove the above tag
*/

  var $__table = 'articles'; // table name
  var $article_id; // int(11)  not_null primary_key auto_increment
  var $title; // string(255)  not_null multiple_key
  var $intro; // blob(65535)  not_null blob
  var $body;  // blob(65535)  not_null blob
  var $author; // string(255)  not_null
  var $published; // string(11)
  var $public; // string(1)  not_null enum

  /* ZE2 compatibility trick*/
  function __clone() { return $this;}

  /* Static get */
  function staticGet($k,$v=NULL) {
    return DB_DataObject::staticGet('DataObject_Articles',$k,$v); }

  /* the code above is auto generated do not remove the tag below */
  ###END_AUTOCODE
}
?>
```

Let's now use this class to access the `articles` table:

File: **23.php**

```
<?php
// Include the DataObjects_Articles class
```

```
require_once 'ExampleApps/DataObject/Articles.php';

// Parse the database ini file
$dbconfig = parse_ini_file('db_dataobject.ini', true);

// Load Database Settings
// (note main PEAR class is loaded by Articles.php)
foreach ($dbconfig as $class => $values) {
  $options = &PEAR::getStaticProperty($class, 'options');
  $options = $values;
}

// Instantiate the DataObject_Articles class
$articles = new DataObject_Articles();

// Assign a value to use to search the 'Author' column
$articles->author = 'Kevin Yank';

// Perform the query
$articles->find();

echo 'Kevin has written the following articles:<br />';

// Loop through the articles
while ($articles->fetch()) {
  echo ' - ' . $articles->title . ', published: ' .
       date('jS M Y', $articles->published) . '<br />';
}
?>
```

First of all, where's the SQL? There isn't any—great! The `parse_ini_file` function is provided by PHP (see Chapter 4 for details) and deals with getting the variables from our `db_dataobject.ini` configuration file. The `foreach` loop makes the required variables available to `DB_DataObject` when we instantiate its auto-generated subclass `DataObject_Articles`. By assigning a value to the author property of the `$articles` object, we prepare a `WHERE` condition that `DataObject_Articles` should use when it queries the database. The query is actually performed by calling the `find` method (see the `DB_DataObject` documentation for full details), which in turn executes the following query:

```
SELECT * FROM articles WHERE articles.author = 'Kevin Yank'
```

To loop through the results, we use the `fetch` method. When it's called, `fetch` populates the properties of the `$articles` object with the current row result.

This allows us to access them again via the property names, as with `$articles->title`.

Further methods are provided to make the query more complex, for example, the `whereAdd` method:

File: **24.php (excerpt)**

```
// Instantiate the DataObject_Articles class
$articles = new DataObject_Articles();

// Assign a value to use to search the 'Author' column
$articles->author = 'Kevin Yank';

// Add a where clause
$articles->whereAdd('published > ' . mktime(0, 0, 0, 5, 1, 2002));

// Perform the query
$articles->find();
```

This allows us to add a further condition to the `WHERE` clause:

```
SELECT * FROM articles
WHERE published > 1020204000 AND articles.author = 'Kevin Yank'
```

There are other similar methods, so if these fail to provide what you need, you can use the `query` method to execute a hand-coded query. Note that if you find yourself needing to use the `query` method, it may be a good idea to create a subclass of the generated `DataObject` class, and wrap the query in a useful method name that describes it accurately.

`DB_DataObject` also deals effectively with table joins, which, although slightly more detailed than the example above, is certainly a time saver when compared with writing complex join queries by hand.

That concludes our short introduction to `DB_DataObject`, but this section should have given you a taste of what it can do for you. The big advantage is that it makes querying your database with SQL far less exhausting and error-prone. Also, by centralizing access to a particular table in a single class, it helps simplify dealing with changes to the table structure.

Further Reading

- *Beginning MySQL*: http://www.devshed.com/Server_Side/MySQL/Intro/

This article provides a solid summary of how to use SQL with MySQL.

- *Give me back my MySQL Command Line!*: http://www.sitepoint.com/article/627

  Kevin Yank shows how to put together a PHP script which can be used to simulate the MySQL command line via a Web page.

- *Optimizing your MySQL Application*: http://www.sitepoint.com/article/402

  This handy tutorial discusses the use of indexes in MySQL and how they can be used to improve performance.

- *Generating PHP Database Access Layers*: http://freshmeat.net/articles/view/843/

  This article provides an overview of persistence layers and related code generation with pointers to some useful tools.

- *Zend Tutorial on Fulltext Searches*: http://www.zend.com/zend/tut/tutorial-Ferrara.php

  This tutorial provides a detailed look at `FULLTEXT` searches.

- *Getting Started with MySQL Fulltext Searches*: http://www.devarticles.com/art/1/195

  This good tutorial delivers another detailed look at `FULLTEXT` searching.

- *Backing Up with MySQLDump*: http://www.sitepoint.com/article/678

  This tutorial that explores the ins and outs of the `mysqldump` utility.

- *Generating PHP Data Access Layers*: http://freshmeat.net/articles/view/843/

  This tutorial looks at database persistence layers in PHP from a fairly broad view as well as suggesting one or two open source projects which offer this functionality.

Appendix A: PHP Configuration

This is a quick reference to configuring PHP that covers the most important general settings you need to be aware of, either when running applications in a live environment, or because they impact security or the way you write code.

Configuration Mechanisms

The primary mechanism for configuring PHP is the `php.ini` file. As the master file, this provides you with control over all configuration settings. Entries generally take the format:

```
setting = value
```

Be sure to read the comments provided in the file before making changes, though. There are a few tricks, such as `include_path` using a colon (`:`) as a seperator on Unix, and a semicolon (`;`) on Windows.

Most Web hosts will not provide you access to your `php.ini` file unless you have root access to the system (which is typically not the case if you're using a cheap virtual hosting service). Your next alternative is to use `.htaccess` files to configure PHP (assuming the Web server is Apache).

An `.htaccess` file is a plain text file that you place in a public Web directory to determine the behavior of Apache when it comes to serving pages from that directory; for instance, you might identify which pages you'll allow public access to. Note that the effect of an `.htaccess` file is recursive—it applies to subdirectories as well.

To configure PHP with `.htaccess` files, your hosting provider must have the Apache setting `AllowOverride Options` or `AllowOverride All` applied to your Web directory in Apache's main `httpd.conf` configuration file. Assuming that is done, there are two Apache directives you can use to modify PHP's configuration:

`php_flag`
: used for settings that have boolean values (i.e. `on`/`off` or `1`/`0`) such as `register_globals`

`php_value`
: used to specify a string value for settings, such as you might have with the `include_path` setting

Here's an example `.htaccess` file:

```
# Switch off register globals
php_flag register_globals off

# Set the include path
php_value include_path ".;/home/username/pear"
```

The final mechanism controlling PHP's configuration is the group of functions `ini_set` and `ini_alter`, which let you modify configuration settings, as well as `ini_get`, which allows you to check configuration settings, and `ini_restore`, which resets PHP's configuration to the default value as defined by `php.ini` and any `.htaccess` files. Using `ini_set`, here's an example which allows us to avoid having to define our host, user name and password when connecting to MySQL:

```
ini_set('mysql.default_host', 'localhost');
ini_set('mysql.default_user', 'harryf');
ini_set('mysql.default_password', 'secret');

if (!mysql_connect()) {
  echo mysql_error();
} else {
  echo 'Success';
}
```

Be aware that PHP provides for some settings, such as `error_reporting`, alternative functions that perform effectively the same job as `ini_set`. Which you prefer is a matter of taste.

Note that certain settings, such as `register_globals`, can only be usefully modified by `php.ini` or `.htaccess`, because such settings influence PHP's behavior *before* it begins executing your scripts.

Furthermore, some configuration settings can be changed *only* in `php.ini`, such as `extension_dir`, which tells PHP the directory in which PHP extensions can be found. For a complete reference on controlling settings, refer to the PHP Manual[1].

[1] http://www.php.net/ini_set

Key Security and Portability Settings

Table A.1 shows the most important PHP settings that relate to the security and portability of your PHP scripts.

Table A.1. Key Security and Portability Settings

Setting	Notes
`register_globals` (default: `off`)	Automatically creates global variables from incoming HTTP request variables, such as GET and POST. For security and portability, it is highly recommended that you switch this off. See http://www.php.net/register_globals for more details.
`magic_quotes_gpc` (default: `off`)	Automatically escapes quotes in incoming HTTP request variables with a backslash, helping prevent SQL injection attacks. If you know what you're doing, it's usually better to switch this functionality off and handle this escaping yourself when inserting into a database, given the problems this feature can cause you with forms, as well as the performance overhead they introduce. See Chapter 1 for information on making your scripts compatible with this feature.
`call_time_pass_reference` (default: `off`)	Allows you to use variable references at call time (e.g. `htmlentities(&$string)`). To keep code clean and understandable, and to ensure portability, keep this functionality switched off.
`short_open_tag` (default: `on`)	Allows you to start a block of PHP code with just `<?` instead of the longer `<?php`. Also lets you write out PHP expressions with `<?=`, which is identical to `<?php echo`. While convenient, these shortcuts are not XML compliant, and can cause the PHP processor to become confused when it encounters XML processing instructions such as `<?xml version="1.0"?>`. Many people have `short_open_tag` switched off, so, for maximum portability, avoid the shortcuts and switch this feature off during development.
`asp_tags` (default: `off`)	Allows ASP style tags (`<% … %>`) as an alternative to the PHP open and close tags (`<?php … ?>`). Few people use these, so, for maximum portability, it's best to avoid them, and switch this feature off during development.

Setting	Notes
`error_reporting` (default: `E_ALL & ~E_NOTICE`)	When developing, and for maximum portability, it's best to set this to `E_ALL`, so that PHP will inform you of situations where, for example, a `$_GET` variable your code relies upon has not been initialized. This forces you to write code that is more secure and contains fewer logic errors, in order to avoid warnings. This also ensures that your code will run neatly on other servers configured this way.
`display_errors` (default: `on`)	Determines whether PHP sends error messages to the Web browser. When running your application in a live environment, it's generally better to switch this off, instead using PHP's logging mechanism to capture errors to a file, for example.
`open_basedir` (default: not set)	Allows you to restrict all PHP file operations to a given directory or below. This can be a good idea to prevent a script that is used to display the contents of files, for example, from being used to access sensitive files elsewhere on your server.
`allow_url_fopen` (default: `on`)	Allows you to specify remote file locations for use with functions like fopen (e.g. `fopen('http://www.sitepoint.com/','r');`). It's a handy tool but is also potentially a security risk for a badly written script. Switch it off if you know you don't need it.

Includes and Execution Settings

Table A.2 shows the most important PHP settings that relate to includes, and how well your PHP scripts run.

Table A.2. Includes and Execution Settings

Setting	Notes
`include_path` (default: `'.'`)	Allows you to specify relative and absolute paths that PHP should search when you use one of the `include` related commands. Make sure you have at least the current directory (`.`) specified, or most third party scripts will fail to work. On Unix systems, the list of directories is separated by colons (`:`), while on Windows the separator is a semi colon (`;`).
`auto_prepend_file` (default: not set)	PHP will execute the file(s) specified *before* executing any requested script. Useful for performing site-wide operations such as security, logging, defining error handlers, stripping backslashes added by the magic quotes feature, and so on. Useful for applications that you're sure you will only use yourself, but unsuitable for use in code you intend to distribute. Those unable to modify `php.ini` settings with `.htaccess` files will be unable to use such code. The list separator is the same as that used for the `include_path` setting.
`auto_append_file` (default: not set)	The twin of `auto_prepend_file`, executed *after* a requested script is executed.
`max_execution_time` (default: `30`)	Specifies the maximum execution time (in seconds) for which a PHP script run via a Web server may be allowed to execute. Generally, it's best to leave this as the default setting and use the `set_time_limit` function to extend the limit on a per-script basis. A value of `0` for either removes any limitations on script execution time.
`memory_limit` (default: `8M`)	The amount of memory PHP has available to it at runtime. Usually, the default is fine, but when handling very large XML documents, for example, or dealing with images, you may need to increase it. The bigger this value, and the more memory a script actually uses, the less memory is available for other applications running on your server.

Setting	Notes
`post_max_size` (default: `8M`)	The maximum amount of data that PHP will accept via an HTTP POST (e.g. a form that uploads an image). You may need to increase this if you have an application that will allow users to upload bigger files.

Error-Related Settings

Table A.3 shows the most important PHP settings that relate to the way PHP handles errors, in addition to `display_errors` and `error_reporting`, which are described in Table A.1.

Table A.3. Error-Related Settings

Setting	Notes
`log_errors` (default: `off`)	Allows you to log errors to a text file, in conjunction with `error_log` (below). Useful for a live site where you've switched off the display of errors to visitors.
`error_log` (default: not set)	A filename to which errors are logged when `log_errors` is switched on.
`ignore_repeated_errors` (default: `off`)	Using this, if the same error occurs from the same PHP script on the same line, the error will only be reported once per script execution. Helps prevent massive log files resulting from errors that occur in loops, when logging to a text file.
`ignore_repeated_source` (default: `30`)	Similar to `ignore_repeated_errors`, but, in this case, it suppresses repeated errors of the same type *throughout* a PHP script.
`report_memleaks` (default: `on`)	Make sure this is switched on, especially if you're using experimental versions or non-stable releases of PHP, otherwise you may end up crashing your server once leaked memory has eaten up all available space. `error_reporting` must be set to report warnings for this setting to apply.

Miscellaneous Settings

Table A.4 shows additional important settings that you should be aware of in your PHP configuration.

Table A.4. Miscellaneous Settings

Setting	Notes
`session.save_path` (default: `/tmp`)	If storing sessions in files on a Windows-based system, you will need to modify this setting to an available directory to which PHP can write session files.
`session.use_cookies` (default: `1`)	Use cookies to store the session ID on the client, rather than placing the session ID in the URL (which can present a greater risk to security).
`extension_dir` (default: `'./'`)	The path under which compiled PHP extensions can be found. On Windows-based systems, it might be something like this: `extension_dir = C:\php-4.3.2\extensions\`
`extension`	On Windows based systems only, this is used to identify all the extensions which should be loaded. The extensions specified should reside in the `extension_dir` path (above). For example: `extension = php_xslt.dll`

Appendix B: Hosting Provider Checklist

PHP, and, more generally, the LAMP combination of Linux, Apache, MySQL and PHP/Perl/Python, is widely available via literally thousands of Web hosts at very affordable prices. You can easily get quality Web hosting that will suit 90% of your needs for under $10 a month per site. That said, all PHP installations are not created equal, and depend largely on the configuration settings defined in `php.ini` as well as the extensions the host has installed for you. There are also a number of general issues relating to the amount of control you're given over your own environment, and these are important if you don't want big trouble later on.

This is a summary of the key issues you should investigate before paying for a hosting service. Contact potential providers and have them respond on each of these points. Follow up by asking for opinions from other people who know/have used the service in question. There are many online forums where you'll find people who are able to offer advice. Be aware, though, that the ratio of "knowledgable" to "ignorant" is stacked highly in favor of ignorance; gem up on technical detail so you're able to verify that the answers you were given were actually well-informed.

Some of the points I've provided here may seem a little extreme, but once you've been around the block a few times, you'll probably want to get value for your money, rather than spending your Saturday mornings fixing the problems your host made for you on Friday night.

General Issues

- ❑ **Require Linux and Apache (1.3)**

  From the point of view of performance and reliability, this is the best combination. Avoid any host using Apache 2.x (it's not yet completely stable with PHP). Ask for details of the Linux distribution. Although Red Hat and Suse are popular, you may find hosts using Debian (or, better yet, Rock Linux) know more about what they're doing.

- ❑ **Does the host provide you with SSH access to the server?**

SSH gives you a secure connection to the server to perform tasks from the Linux command line or transfer files with SCP (secure copy). Avoid any host who allows you to use telnet (a fundamentally insecure way to connect to a server over the Internet). For Windows users, Putty[1] makes an excellent command line tool over SSH, while WinSCP[2] provides a secure file transfer mechanism using an SSH connection. Oh, and don't transfer files with ftp—it's as insecure as telnet.

❑ **Is the host a reseller or do they maintain the server themselves?**

Resellers can provide significant value if you need help at a basic technical level (if, for example, you call yourself a newbie), but they generally have the same level of control over the server as you. Going "straight to the source" means you won't have to deal with delays when there are system problems, as you'll likely be dealing directly with those who maintain the server. The down side is that they tend to be less "newbie tolerant" so you may get answers—but not ones you can understand

❑ **To what degree does the host "overload" the server?**

Many Web hosting companies create far more accounts on a server than the maximum for which the system is specified. The best metric is the uptime command (to which you require access); this will tell you the server load averages over 1, 5 and 15 minutes. Ideally, the server should never have load averages above 1. Obviously, the problem isn't as simple as this, but once you see your server hit averages in excess of 5, you'll begin to experience significant delays in your PHP-based applications.

❑ **What is the hosting provider's policy on running scripts and programs from the command line?**

MySQLDump is a very handy tool for backing up your database, but it's no good if you can't run it. Some hosts automatically kill any command line application that executes for longer than a given time.

❑ **Does the host provide you access to cron, the Unix utility that allows you to schedule batch jobs?**

If so, make sure the host allows command line scripts to be executed. Some hosts have taken to implementing cron so that it executes scripts via a Web

[1] http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
[2] http://winscp.sourceforge.net/eng/

URL. This is no use if the script in question uses the MySQLDump application to back up your database—a PHP script executed via Apache will typically run as a user, which will not have the correct permissions required for the job.

PHP-Related Issues

❑ **Can you see the output of `phpinfo` on the server you will actually be assigned to?**

Some hosts may claim this is a security risk, but expert hosts know that security by obscurity is no substitute for *real* security. The information provided by `phpinfo` is *not* a security risk to hosting providers that know what they're doing, and have Linux, Apache, and firewalls correctly set up. What `phpinfo` tells you is the best way to confirm the facts.

❑ **Is PHP installed as an Apache module (not the CGI variant)?**

This provides much better performance.

❑ **Is the Apache setting AllowOverride set to Options or All?**

This will let you modify `php.ini` settings with `.htaccess` files.

❑ **Is PHP Safe Mode disabled?**

The `safe_mode` option in `php.ini` is, in theory, a way to make PHP secure, and prevent users from performing certain tasks or using certain functions that are security-sensitive. Safe Mode is nothing but a large headache if you're doing any serious work in PHP.

❑ **Check the upgrade policy of your host.**

Ask the host how much warning you will get before upgrades are performed. Check that they will provide you with a copy of the `php.ini` file they'll be using for the upgrade (before it happens). The number of hosts that, overnight, switch from `register_globals = on` to `register_globals = off` is considerable. Make sure you test your applications on your development system against the new version before the host performs the upgrade.

❑ **Ask for a list of installed PHP extensions.**

Confirm that these extensions match the requirements of your applications. Few hosts, for example, bother to provide the XSLT extension. Confirm also

that the host guarantees all extensions will remain available between PHP upgrades.

- ❑ **Will PHP be available for use from the command line?**

  If not, you might alternately require access to Perl or Python, or the ability to run shell scripts, if you're happy with those languages. Usually, running a serious Website will require that you have the ability to run routine batch jobs (with cron), for tasks like backups, mailing you the PHP error log, and so on.

- ❑ Last but not least, throw in one or two questions that will test your hosting providers' knowledge of PHP. Although it may not be their job to write PHP code, when you find yourself in the position of knowing a lot more about PHP than your host, the end result is depressing. It's important to have a host that understands your needs.

Appendix C: Security Checklist

Given that online PHP applications are exposed to essentially anyone and everyone, security should be one of, if not *the* top concern as you develop your applications. To some extent, the ease with which PHP applications can be developed is also one of its greatest weaknesses, in that, for beginners who aren't aware of the possible dangers, it's very easy to deploy an application for which the line of security resembles swiss cheese.

Make sure you're informed, and, if in any doubt, ask. The Open Web Application Security Project (OWASP)[1] is a corporate-sponsored community focused on raising awareness of Web security, and is an excellent source of information on potential dangers. They recently published a "Top 10" list of common security flaws in Web applications, the relevant points of which I've summarized here.

The Top Security Vulnerabilities

- **Unvalidated data**

  Never trust anything you get from a Web browser. The browser is completely outside of your control, and it's easy to fake values like the HTTP referrer. It's also easy to fake a hidden field in a form.

  More importantly, when dealing with forms, for example, validate the data carefully. Use a "deny all, permit a little" policy. For example, if a registration form has a field for the user name, allow only alphabetical characters and perhaps the numbers 0–9, rather than simply rejecting particular special characters. Use regular expressions to limit data to exactly what you require. Packages like PEAR::QuickForm, as you saw in Chapter 9, provide built-in mechanisms for validating forms and do a lot to help cover weaknesses you might otherwise neglect.

  Also, where things like include files are concerned, watch out for logic like this:

```
include($_GET['page']);
```

  Make sure you check the value of `$_GET['page']` against a list of files your code is designed to include:

[1] http://www.owasp.org/

```
$pages = array(
  'news.php', 'downloads.php', 'links.php'
);

if (in_array($_GET['page'], $pages)) {
  include $_GET['page'];
} else {
  include 'not_found.php';
}
```

Without such checks, it's very easy for an attacker to use code similar to this to execute other PHP scripts—even ones you didn't write.

❑ **Broken access control**

Fundamental logic of this form is easy to get wrong if you don't know what you're doing. For example, often, developers check a user name/password combination against a database using logic like this:

```
if ($numRows != 0) {
  // allow access ...
}
```

That means they let users in even if they found *more than one* matching entry in the database, which, if your site also has security holes like command injection flaws (see below), may provide attackers access to a lot more than you were expecting. It's easy to make mistakes in situations you think are secure when, in fact, the logic can be bypassed easily. In general, use respected third party libraries such as PEAR::Auth[2] and PEAR::LiveUser[3] wherever possible. Also, investigate Web testing frameworks such as SimpleTest[4], which provide the ability to test your site from the point of view of a Web browser.

❑ **Session and Cookie Vulnerabilities**

Watch out for session hijacking possibilities. On sites where you really need secure authentication (e.g. ecommerce sites), use SSL to serve the site to the browser, to ensure the conversation is encrypted and that no one is listening in. If you're passing session IDs via the URL, as you will for WML-based sites, make sure that you're not placing the session ID in URLs that point to remote sites. Also, when passing visitors to a remote site, forward them via an intermediate script that strips out any possible HTTP referrer information that

[2] http://pear.php.net/package/Auth
[3] http://pear.php.net/package/LiveUser
[4] http://www.lastcraft.com/simple_test.php

contains the session ID. In general, it's better to handle sessions with cookies. If you're working with your own cookie-based authentication, store an identifying session ID in the cookie only, not the user name and password.

❑ Cross Site Scripting (XSS)

By using the legitimate mechanisms your site provides, it's possible for attackers to post on your site, for example, JavaScript that results in other users giving away their session IDs, thereby allowing the attacker to hijack their session. Less serious, but equally embarrassing, is simply posting HTML that "scrambles" the layout of your page, perhaps closing a `table` tag prematurely. Use a "deny all, permit a little" approach, or, better yet, employ a separate markup language such as BBCode (see Chapter 5), while eliminating HTML with PHP functions like `strip_tags` and `htmlentities`. If you really want to allow HTML to be posted, consider building a filter based on PEAR::XML_HTMLSax[5] (see Volume II, Chapter 2).

❑ Command Injection

Command injection occurs when an attacker is able to influence the way PHP interacts with external systems, such as the file system or a database. An SQL injection is a prime example, which occurs when an attacker uses a form or URL to modify a database query. This was discussed in some detail in Chapter 3. The bottom line is: escape all data you receive from a user before you use it in a query.

❑ Error Handling

An experienced attacker will be able to gain a lot of important information about your system from your error messages. Although this comes under the heading of "security by obscurity" (which is no substitute for having a *really* secure application), for a live site, it's a good idea to instruct PHP to log error messages to a file, rather than display them to the browser. See Appendix A for details.

❑ Insecure Use of Cryptography

First of all, when it comes to cryptography, don't roll your own. Second, remember that if it's an algorithm that's meant to be decoded, then someone (other than you) is also capable of decoding it. Remember that, strictly speaking, MD5 is not an encryption algorithm (i.e. you cannot decrypt an

[5] http://pear.php.net/package/XML_HTMLSax

MD5 string to obtain the original data); it's a message digest algorithm. But if you don't need to decrypt a value then use MD5, which is available through PHP's `md5` function. This allows you to compare the encrypted versions of two pieces of data (e.g. a stored password and that entered by a user), which avoids the risks involved in working with encrypted values that could possibly be decrypted by an attacker.

- **Administration Flaws**

  Allowing an attacker to gain the same access you have to your site is clearly bad news. Avoid FTP and telnet in favor of SCP/SFTP and SSH, respectively. Linux distributions usually have the required client tools pre-installed. For Windows, check out putty[6] for SSH access and WinSCP[7] for SCP/SFTP. FTP and telnet expose your password to network sniffers. Make sure that any Web administration tools your host provides are used only over an SSL connection. If you're using third party software, such as phpBB, change the default administrator password immediately, and stay informed about potential security flaws.

- **Configuration and Patching**

  When installing PHP, the configuration file `php.ini-recommended` makes the best starting point to make sure you've got the package configured correctly.

  If you're using a hosting company, they should take care of most of the issues for you, such as patching software as vulnerabilities are announced. Still, it's worth staying up to date on your own, using sites like Security Focus[8] and others listed at DMOZ[9].

More information is available at PHP Advisory[10] although, sadly, the site is no longer being maintained.

[6] http://www.chiark.greenend.org.uk/~sgtatham/putty/
[7] http://winscp.sourceforge.net/eng/
[8] http://www.securityfocus.com/incidents/
[9] http://dmoz.org/Computers/Security/Mailing_Lists/
[10] http://www.phpadvisory.com/

Appendix D: Working with PEAR

PEAR[1], the **PHP Extension and Application Repository**, is the brainchild of Stig Bakken, and was inspired by Perl's CPAN[2].

As a project, it was originally conceived in 1999 and reached its first stable release in January 2003. It serves two purposes. First, it provides a library of PHP classes for solving common "architectural" problems, a number of which you've seen in this book. Second, under the title "PECL" (PHP Extension Code Library), PEAR provides a repository for extensions to PHP. PECL was originally intended to store "non standard" extensions that lay more on the fringes of PHP, but it has since evolved into the default repository for all extensions not included in the core PHP distribution. Here, I'll be concentrating on the PHP classes that PEAR provides.

Those who submit work and maintain the PEAR repository are all volunteers. Originally a small community of developers, the release of the first stable version of PEAR has seen their numbers grow significantly, and receive a greater focus from the PHP community as a whole. There's still a lot of work to be done to raise the standards to that of PHP itself, documentation being a key area in which there's still much room for improvement. If you're struggling, a good place to start is PHPKitchen's list of PEAR Tutorials[3]. That said, PEAR already offers significant value in terms of reducing the effort required in developing PHP applications.

But what does PEAR actually mean to you? Considering the capabilities of PEAR::SOAP, which was covered in Volume II, Chapter 2, attempting to write your own SOAP implementation first, *then* writing the "application" code that will use it is clearly a waste of time. Browsing the list of packages[4], you'll see that PEAR provides you many more classes, categorized by subject, to help prevent you having to reinvent wheels. It's important to understand the focus of PEAR classes is *architectural* issues, not application-level classes. In other words, PEAR is not Hotscripts; you won't find complete applications there; rather, you'll find code that can be reused in many different applications. Also important is that the PEAR developer community does its best to maintain and support the library, compared to, say, projects available via SourceForge[5], which are often individual

[1] http://pear.php.net/
[2] http://www.cpan.org/
[3] http://www.phpkitchen.com/staticpages/index.php?page=2003041204203962
[4] http://pear.php.net/packages.php
[5] http://www.sourceforge.net/

endeavours and come to a sudden end once the individuals in question stop contributing their time. Otherwise, there is some emphasis on maintaining a degree of standardization throughout the library. For example, all error handling should be performed using PEAR::Error, and the code should be documented using the PHPDoc standard, which means you should be able to extract the API documentation using PHPDocumentor[6] (see Volume II, Chapter 6) if you can't find it on the PEAR Website.

Be warned: the degree of integration between the packages within PEAR is currently fairly low when compared to, say, the Java class library. This means, in some cases, that you'll be confronted with decisions like whether to use PEAR::HTML_QuickForm's validation functionality, or PEAR::Validate, or both. It's a good idea to invest some time investigating which fits your development style up-front, rather than jumping straight in and using a PEAR class for a critical part of your application, only to discover later that it wasn't the best fit for the problem.

One important point to be clear on is that referring to "PEAR" can actually mean one of two things: the repository as a whole, or the PEAR front end (also known as the package manager), which provides tools for installing and upgrading the PEAR packages you use.

Note that it's *not* a requirement that you use the PEAR package manager to install PEAR packages. If you need to, you can download them directly from the PEAR Website and manually extract them to your PHP's include path. Make sure you check the dependencies listed on the site (these being other required packages) and be aware that most packages implicitly require PEAR "base" package[7] for tasks like error handling.

Installing PEAR

These days, the foundations of PEAR are provided with PHP distribution itself, but Web hosts typically fail to provide customers with their own default PEAR installation, so it's worth knowing how to go about doing this from scratch. The process can differ slightly between Unix and Windows based systems.

Step one is to make sure you can run PHP scripts via the command line. This is always possible if you type the full path to the PHP binary. For a Unix based system, you'd use the following:

[6] http://www.phpdoc.org/
[7] http://pear.php.net/package/PEAR

```
/usr/local/bin/php /home/username/scripts/my_script.php
```

For Windows, you'd use something like this:

```
c:\php\cli\php.exe c:\scripts\my_script.php
```

Note that in the Windows path above, we used the executable in the `cli` (command line interface) subdirectory of the PHP installation, this executable behaving slightly differently from that used by Apache to handle Web pages. PHP binary releases for Windows since 4.3.0 place the `cli` version of the PHP executable in this directory.

It's possible to make PHP much easier to use from the command line, though, by making some changes to your system's environment variables. For an in-depth discussion see *Replacing Perl Scripts with PHP Scripts*[8] on PHPBuilder[9].

Next, point your browser at http://pear.php.net/go-pear, where you'll see a PHP script. This script is used to install the PEAR package manager—the basis you'll need in order to install other PEAR packages. Download this to your computer (File, Save As) as `go-pear.php`. From here, you have a number of options.

Storing `go-pear.php` somewhere under your Web server's document root directory will allow you to run the script as a Web page. This behavior is still experimental, though, so there are no guarantees it'll work correctly. If you do use this approach, make sure that the script is not publicly available!

Better is to execute the `go-pear.php` script via the command line, for example:

```
/usr/local/bin/php /home/username/pear/go-pear.php
```

Or, on Windows:

```
c:\php\cli\php c:\pear\go-pear.php
```

This will start an interactive command line interface, which will ask you questions about how you would like PEAR installed. Note that the "installation prefix" is the directory in which PEAR (as well as any packages you install later) will be installed, and is referred to as `$prefix`, while `$php_dir` is the path to your PHP installation (in which `go-pear.php` will put PEAR-related documentation by default, unless you specify otherwise). Windows users should be aware that

[8] http://www.phpbuilder.com/columns/jayesh20021111.php3
[9] http://www.phpbuilder.com/

changing the installation prefix pops up a Windows "Browse" dialog box, through which you can specify the required directory.

With the installation options set to your requirements, the `go-pear.php` script will connect to the PEAR Website, and download all the packages required to set up the package manager (it also asks if you require additional packages, which are well worth having). Packages are installed in a subdirectory `pear` of the directory you specified as the installation prefix (so, in the above examples you'd end up with `c:\pear\pear` or `/home/username/pear/pear`).

Finally, if you let it, the `go-pear.php` installer will attempt to modify your `include_path` in `php.ini`. To do this manually, assuming you used the directories above, you'd specify the following:

```
include_path = ".:/home/username/pear/pear"
```

For Windows users, the path is as follows:

```
include_path = ".;c:\pear\pear"
```

Finally, to use the PEAR package manager from the command line, you need to set up some environment variables. For Windows users these can be automatically added to your Windows registry by right clicking on the file `PEAR_ENV.reg` and choosing Run. They may also be manually configured as environment variables via the Windows Control Panel. Users with Unix-based systems can configure them to be set up every time you log in, by editing the file `.profile` in your home directory (`/home/`*`username`*):

```
# Envinment variables
export PHP_PEAR_SYSCONF_DIR=/home/username/pear
export PHP_PEAR_INSTALL_DIR=/home/username/pear/pear
export PHP_PEAR_DOC_DIR=/home/username/pear/pear/docs
export PHP_PEAR_BIN_DIR=/home/username/pear
export PHP_PEAR_DATA_DIR=/home/username/pear/pear/data
export PHP_PEAR_TEST_DIR=/home/username/pear/pear/tests
export PHP_PEAR_PHP_BIN=/usr/local/bin/php
```

Finally, you need to add the PEAR command line script to your system path, which, on Windows, can be achieved through the System Control Panel application (on the *Advanced* tab, click Environment Variables), by appending `;c:\pear` to the `PATH` variable.

On Unix-based systems, add the following to your `.profile` script:

```
export PATH=$PATH;/home/username/pear
```

Once you've done all that, you're ready to move on and use the package manager in one of its many incarnations.

The PEAR Package Manager

Assuming you set PEAR up correctly, you can now use the command line interface to the PEAR package manager to install packages. For example, from the command line, type:

```
pear install HTML_Common
```

That will install the package HTML_Common from the PEAR Website. The package names for the command line are the same as those on the Website.

The PEAR Package Manager uses XML_RPC to communicate with the PEAR Website. If you're behind a proxy server or firewall, you will need to tell PEAR the domain name of the proxy server with:

```
pear config-set http_proxy proxy.your-isp.com
```

To unset the variable at some later stage, simply use:

```
pear config-set http_proxy ""
```

Now to add QuickForm to the installed PEAR packages, you simply need to type:

```
pear install HTML_QuickForm
```

Should another release of QuickForm be made after you've installed it, you can upgrade the version with:

```
pear upgrade HTML_QuickForm
```

If, for some reason, you later decide you don't need QuickForm any more, you can remove it using:

```
pear uninstall HTML_QuickForm
```

For a list of all PEAR commands, simply type **pear**.

Now, if you don't like command lines, there's also an (experimental) Web-based front end to PEAR (as well as a PHP-GTK front end, which is beyond the scope of this discussion). To use it, you first need to install it from the command line

(note that if you executed `go-pear.php` through your Web server, the Web-based front end is also installed for you). Type the following commands:

```
pear install Net_UserAgent_Detect
pear install Pager
pear install HTML_Template_IT
pear install PEAR_Frontend_Web
```

Note the first three packages are required by PEAR_Frontend_Web. With that done, you can launch the front end from your Web server using the following simple script:

```
<?php
// Optional if include path not set
# ini_set('include_path', 'c:\htdocs\PEAR');

require_once 'PEAR.php';

// For Windows users
# $pear_user_config = 'c:\windows\pear.ini';
// For Unix users
$pear_user_config = '/home/username/pear/pear/PEAR/pear.conf';

$useDHTML = TRUE; // Switch off for older browsers

require_once 'PEAR/WebInstaller.php';
?>
```

Installing Packages Manually

It's possible to install packages manually (although this involves more work), but it's important to watch the include paths carefully when doing so. First of all, create a directory that will be the base of all the PEAR classes you install. This directory *must* be in your include path. Next, install the main PEAR package[11]—download the latest *stable* version and extract it directly to the directory you've created, so that `PEAR.php` is in the root of this directory.

Installing further packages can be completed in more or less the same fashion, but you need to be careful which directory you extract to. For example, looking at PEAR::DB, the main `DB.php` file goes alongside the `PEAR.php` file in the root of the PEAR class directory, while further PEAR::DB-related files go in the subdirectory `DB`. The best way to check is to look at the `package.xml` file that comes

[11] http://pear.php.net/package/PEAR

with every PEAR package. This contains an element called `filelist`, which lists all the files contained in the package and the location at which they should be installed. For each `file`, check the `baseinstalldir` attribute which, if specified, tells you where, relative to the root PEAR class directory, the file should be placed. The `name` attribute specifies the path and filename *relative* to the `baseinstalldir` (or just the root PEAR class directory if there's no `baseinstalldir` attribute), where each file should be placed.

Index

This index covers both volumes of *The PHP Anthology*. Page references in another volume are prefixed with the volume number and appear in italics (e.g. *II-123* refers to page 123 of Volume II).

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

X

Z

What's Next?

If you've enjoyed these chapters from *The PHP Anthology*, why not order yourself a copy?

In the rest of *Volume I: Foundations*, you'll learn how to put PHP's basic features, such as file and text manipulation, image generation, and email, to full use in practical solutions built upon principles of object oriented software design. You'll also gain access to the code archive download, so you can try out all the examples without retyping!

Here are just a handful of things you'll learn to do with PHP in *Volume I: Foundations*:

- upload and download files over FTP
- create compressed ZIP/TAR files
- implement a bad word filter
- build an online calendar
- create thumbnail images
- display charts and graphs
- handle incoming mail
- make "search engine friendly" URLs
- implement a custom error handler
- And a whole lot more…

And don't forget *The PHP Anthology, Volume II: Applications*! It covers complex applications of PHP including XML processing, alternative content types like PDF files, site statistics, unit testing, design patterns, and plenty more besides!