Chapter 1. Orientation and First Steps

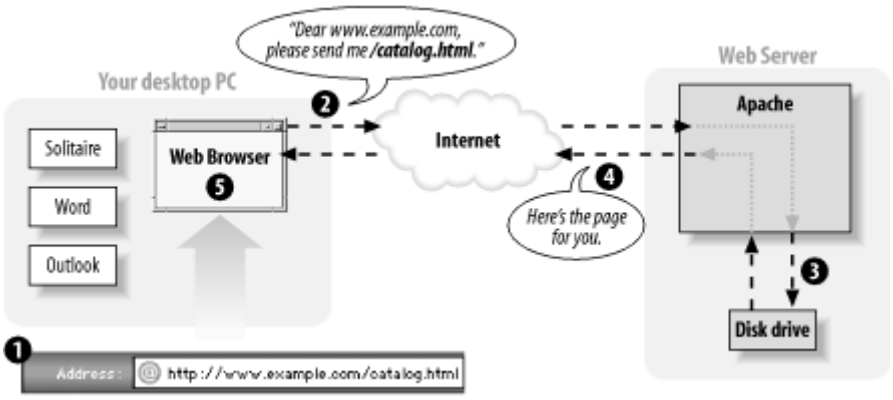
There are lots of great reasons to write computer programs in PHP. Maybe you want to learn PHP because you need to put together a small web site for yourself that has some interactive elements. Perhaps PHP is being used where you work and you have to get up to speed. This chapter provides context for how PHP fits into the puzzle of web site construction: what it can do and why it's so good at what it does. You'll also get your first look at the PHP language and see it in action.

1.1 PHP's Place in the Web World

PHP is a programming language that's used mostly for building web sites. Instead of a PHP program running on a desktop computer for the use of one person, it typically runs on a web server and is accessed by lots of people using web browsers on their own computers. This section explains how PHP fits into the interaction between a web browser and a web server.

When you sit down at your computer and pull up a web page using a browser such as Internet Explorer or Mozilla, you cause a little conversation to happen over the Internet between your computer and another computer. This conversation and how it makes a web page appear on your screen is illustrated in Figure 1-1.

Figure 1-1. Client and server communication without PHP



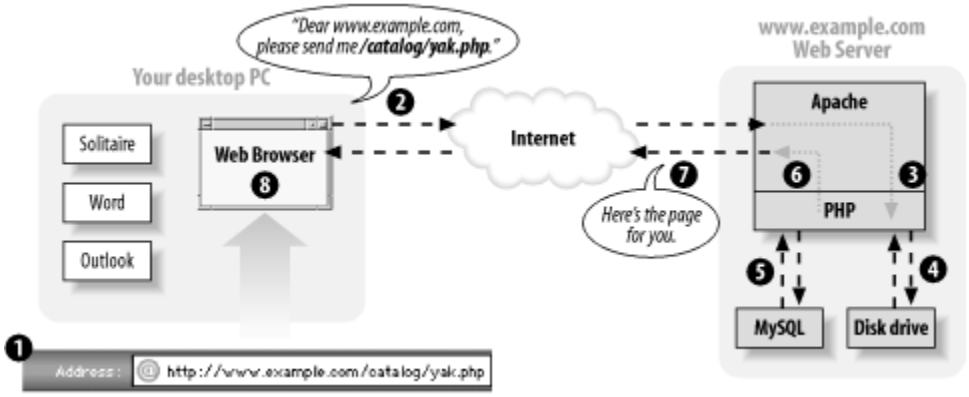
Here's what's happening in the numbered steps of the diagram:

1. You type *www.example.com/catalog.html* into the location bar of Internet Explorer.
2. Internet Explorer sends a message over the Internet to the computer named `www.example.com` asking for the */catalog.html* page.
3. Apache, a program running on the `www.example.com` computer, gets the message and reads the *catalog.html* file from the disk drive.
4. Apache sends the contents of the file back to your computer over the Internet as a response to Internet Explorer's request.
5. Internet Explorer displays the page on the screen, following the instructions of the HTML tags in the page.

Every time a browser asks for *http://www.example.com/catalog.html*, the web server sends back the contents of the same *catalog.html* file. The only time the response from the web server changes is if someone edits the file on the server.

When PHP is involved, however, the server does more work for its half of the conversation. Figure 1-2 shows what happens when a web browser asks for a page that is generated by PHP.

Figure 1-2. Client and server communication with PHP



Here's what's happening in the numbered steps of the PHP-enabled conversation:

1. You type *www.example.com/catalog/yak.php* into the location bar of Internet Explorer.
2. Internet Explorer sends a message over the Internet to the computer named `www.example.com` asking for the */catalog/yak.php* page.
3. Apache, a program running on the `www.example.com` computer, gets the message and asks the PHP interpreter, another program running on the `www.example.com` computer, "What does */catalog/yak.php* look like?"
4. The PHP interpreter reads the file */usr/local/www/catalog/yak.php* from the disk drive.
5. The PHP interpreter runs the commands in *yak.php*, possibly exchanging data with a database program such as MySQL.
6. The PHP interpreter takes the *yak.php* program output and sends it back to Apache as an answer to "What does */catalog/yak.php* look like?"
7. Apache sends the page contents it got from the PHP interpreter back to your computer over the Internet in response to Internet Explorer's request.
8. Internet Explorer displays the page on the screen, following the instructions of the HTML tags in the page.

"PHP" is a programming language. Something in the web server reads your PHP programs, which are instructions written in this programming language, and figures out what to do. The "PHP interpreter" follows your instructions. Programmers often say "PHP" when they mean either the programming language or the interpreter. In this book, I mean the language when I say "PHP." When I say "PHP interpreter," I mean the thing that follows the commands in the PHP programs you write and that generates web pages.

If PHP (the programming language) is like English (the human language), then the PHP interpreter is like an English-speaking person. The English language defines various words and combinations that, when read or heard by an English-speaking person, translate into various meanings that cause the person to do things such as feel embarrassed, go to the store

to buy some milk, or put on pants. The programs you write in PHP (the programming language) cause the PHP interpreter to do things such as talk to a database, generate a personalized web page, or display an image.

This book is concerned with the details of writing those programs — i.e., what happens in Step 5 of Figure 1-2 (although Appendix A contains details on configuring and installing the PHP interpreter on your own web server).

PHP is called a *server-side* language because, as Figure 1-2 illustrates, it runs on a web server. Languages and technologies such as JavaScript and Flash, in contrast, are called *client-side* because they run on a web client (like a desktop PC). The instructions in a PHP program cause the PHP interpreter on a web server to output a web page. The instructions in a JavaScript program cause Internet Explorer, while running on your desktop PC, to do something such as pop up a new window. Once the web server has sent the generated web page to the client (Step 7 in the Figure 1-2), PHP is out of the picture. If the page content contains some JavaScript, then that JavaScript runs on the client but is totally disconnected from the PHP program that generated the page.

A plain HTML web page is like the "sorry you found a cockroach in your soup" form letter you might get after dispatching an angry complaint to a bug-infested airline. When your letter arrives at airline headquarters, the overburdened secretary in the customer service department pulls the "cockroach reply letter" out of the filing cabinet, makes a copy, and puts the copy in the mail back to you. Every similar request gets the exact same response.

In contrast, a dynamic page that PHP generates is like a postal letter you write to a friend across the globe. You can put whatever you like down on the page — doodles, diagrams, haikus, and tender stories of how unbearably cute your new baby is when she spatters mashed carrots all over the kitchen. The content of your letter is tailored to the specific person to whom it's being sent. Once you put that letter in the mailbox, however, you can't change it any more. It wings its way across the globe and is read by your friend. You don't have any way to modify the letter as your friend is reading it.

Now imagine you're writing a letter to an arts-and-crafts-inspired friend. Along with the doodles and stories you include instructions such as "cut out the little picture of the frog at the top of the page and paste it over the tiny rabbit at the bottom of the page," and "read the last paragraph on the page before any other paragraph." As your friend reads the letter, she also performs actions the letter instructs her to take. These actions are like JavaScript in a web page. They're set down when the letter is written and don't change after that. But when the reader of the letter follows the instructions, the letter itself can change. Similarly, a web browser obeys any JavaScript commands in a page and pops up windows, changes form menu options, or refreshes the page to a new URL.

1.2 What's So Great About PHP?

You may be attracted to PHP because it's free, because it's easy to learn, or because your boss told you that you need to start working on a PHP project next week. Since you're going to use PHP, you need to know a little bit about what makes it special. The next time someone asks you "What's so great about PHP?", use this section as the basis for your answer.

1.2.1 PHP Is Free (as in Money)

You don't have to pay anyone to use PHP. Whether you run the PHP interpreter on a beat-up 10-year-old PC in your basement or in a room full of million-dollar "enterprise-class" servers, there are no licensing fees, support fees, maintenance fees, upgrade fees, or any other kind of charge.

Most Linux distributions come with PHP already installed. If yours doesn't, or you are using another operating system such as Windows, you can download PHP from http://www.php.net/. Appendix A has detailed instructions on how to install PHP.

1.2.2 PHP Is Free (as in Speech)

As an open source project, PHP makes its innards available for anyone to inspect. If it doesn't do what you want, or you're just curious about why a feature works the way it does, you can poke around in the guts of the PHP interpreter (written in the C programming language) to see what's what. Even if you don't have the technical expertise to do that, you can get someone who does to do the investigating for you. Most people can't fix their own cars, but it's nice to be able to take your car to a mechanic who can pop open the hood and fix it.

1.2.3 PHP Is Cross-Platform

You can use PHP with a web server computer that runs Windows, Mac OS X, Linux, Solaris, and many other versions of Unix. Plus, if you switch web server operating systems, you generally don't have to change any of your PHP programs. Just copy them from your Windows server to your Unix server, and they will still work.

While Apache is the most popular web server program used with PHP, you can also use Microsoft Internet Information Server and any other web server that supports the CGI standard. PHP also works with a large number of databases including MySQL, Oracle, Microsoft SQL Server, Sybase, and PostgreSQL. In addition, it supports the ODBC standard for database interaction.

If all the acronyms in the last paragraph freak you out, don't worry. It boils down to this: whatever system you're using, PHP probably runs on it just fine and works with whatever database you are already using.

1.2.4 PHP Is Widely Used

As of March 2004, PHP is installed on more than 15 million different web sites, from countless tiny personal home pages to giants like Yahoo!. There are many books, magazines, and web sites devoted to teaching PHP and exploring what you can do with it. There are companies that provide support and training for PHP. In short, if you are a PHP user, you are not alone.

1.2.5 PHP Hides Its Complexity

You can build powerful e-commerce engines in PHP that handle millions of customers. You can also build a small site that automatically maintains links to a changing list of articles or press releases. When you're using PHP for a simpler project, it doesn't get in your way with concerns that are only relevant in a massive system. When you need advanced features such as caching, custom libraries, or dynamic image generation, they are available. If you don't need them, you don't have to worry about them. You can just focus on the basics of handling user input and displaying output.

1.2.6 PHP Is Built for Web Programming

Unlike most other programming languages, PHP was created from the ground up for generating web pages. This means that common web programming tasks, such as accessing form submissions and talking to a database, are often easier in PHP. PHP comes with the capability to format HTML, manipulate dates and times, and manage web cookies — tasks that are often available only as add-on libraries in other programming languages.

1.3 PHP in Action

Ready for your first taste of PHP? This section contains a few program listings and explanations of what they do. If you don't understand everything going on in each listing, don't worry! That's what the rest of the book is for. Read these listings to get a sense of what PHP programs look like and an outline of how they work. Don't sweat the details yet.

When given a program to run, the PHP interpreter pays attention only to the parts of the program between PHP start and end tags. Whatever's outside those tags is printed with no modification. This makes it easy to embed small bits of PHP in pages that mostly contain HTML. The PHP interpreter runs the commands between `<?php` (the PHP start tag) and `?>` (the PHP end tag). PHP pages typically live in files whose names end in *.php*. Example 1-1 shows a page with one PHP command.

Example 1-1. Hello, World!

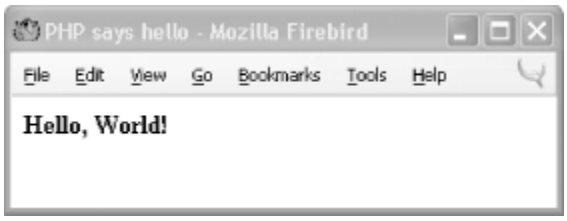
```
<html>
<head><title>PHP says hello</title></head>
<body>
<b>
<?php
print "Hello, World!";
?>
</b>
</body>
</html>
```

The output of Example 1-1 is:

```
<html>
<head><title>PHP says hello</title></head>
<body>
<b>
Hello, World!
</b>
</body>
</html>
```

In your web browser, this looks like Figure 1-3.

Figure 1-3. Saying hello with PHP



Printing a message that never changes is not a very exciting use of PHP, however. You could have included the "Hello, World!" message in a plain HTML page with the same result. More useful is printing dynamic data — i.e., information that changes. One of the most common sources of information for PHP programs is the user: the browser displays a form, the user enters information into that and hits the "submit" button, the browser sends that information to the server, and the server finally passes it on to the PHP interpreter where it is available to your program.

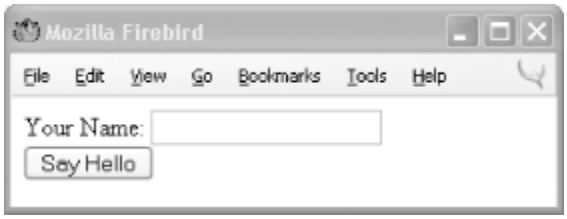
Example 1-2 is an HTML form with no PHP. The form consists simply of a text box named `user` and a Submit button. The form submits to *sayhello.php*, specified via the `<form>` tag's `action` attribute.

Example 1-2. HTML form for submitting data

```
<form method="POST" action="sayhello.php">
Your Name: <input type="text" name="user">
<br/>
<input type="submit" value="Say Hello">
</form>
```

Your web browser renders the HTML in Example 1-2 into the form shown in Figure 1-4.

Figure 1-4. Printing a form



Example 1-3 shows the *sayhello.php* program that prints a greeting to whomever is named in the form's text box.

Example 1-3. Dynamic data

```
<?php
print "Hello, ";
// Print what was submitted in the form parameter called 'user'
print $_POST['user'];
print "!";
?>
```

If you type `Ellen` in the text box and submit the form, then Example 1-3 prints `Hello, Ellen!`. Figure 1-5 shows how your web browser displays that.

Figure 1-5. Printing a form parameter

`$_POST` holds the values of submitted form parameters. In programming terminology, it is a *variable*, so called because you can change the values it holds. In particular, it is an *array* variable, because it can hold more than one value. This particular array is discussed in Chapter 6. Arrays are discussed in Chapter 4.

In this example, the line that begins with `//` is called a *comment line*. Comment lines are there for human readers of source code and are ignored by the PHP interpreter. Comments are useful for annotating your programs with information about how they work. Section 1.4.3, later in this chapter, discusses comments in more detail.

You can also use PHP to print out the HTML form that lets someone submit a value for `user`. This is shown in Example 1-4.

Example 1-4. Printing a form

```
<?php
print <<<_HTML_
<form method="post" action="$_SERVER[PHP_SELF]">
Your Name: <input type="text" name="user">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;
?>
```

Example 1-4 uses a string syntax called a *here document*. Everything between the `<<<_HTML_` and the `_HTML_` is passed to the `print` command to be displayed. Just like in Example 1-3, a variable inside the string is replaced with its value. This time, the variable is `$_SERVER[PHP_SELF]`. This is a special PHP variable that contains the URL (without the protocol or hostname) of the current page. If the URL for the page in Example 1-4 is *http://www.example.com/users/enter.php*, then `$_SERVER[PHP_SELF]` contains `/users/enter.php`.

With `$_SERVER[PHP_SELF]` as the form action, you can put the code for printing a form and for doing something with the submitted form data in the same page. Example 1-5 combines Examples Example 1-3 and Example 1-4 into one page that displays a form and prints a greeting when the form is submitted.

Example 1-5. Printing a greeting or a form

```
<?php
// Print a greeting if the form was submitted
```

```
if ($_POST['user']) {
    print "Hello, ";
    // Print what was submitted in the form parameter called 'user'
    print $_POST['user'];
    print "!";
} else {
    // Otherwise, print the form
    print <<<_HTML_
<form method="post" action="$_SERVER[PHP_SELF]">
Your Name: <input type="text" name="user">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;
}
?>
```

Example 1-5 uses the `if( )` construct to see whether the browser sent a value for the form parameter `user`. It uses that to decide which of two things to do: print a greeting or print a form. Chapter 3 talks about `if( )`. Using `$_SERVER[PHP_SELF]` and processing forms is discussed in Chapter 6.

PHP has a huge library of internal functions that you can use in your programs. These functions help you accomplish common tasks. One built-in function is `number_format( )`, which provides a formatted version of a number. Example 1-6 uses `number_format( )` to print out a number.

Example 1-6. Printing a formatted number

```
<?php print "The population of the US is about:";
print number_format(285266237);
?>
```

Example 1-6 prints:

```
The population of the US is about: 285,266,237
```

Chapter 5 is about functions. It shows you how to write your own and explains the syntax for calling and handling the results of functions. Many functions, including `number_format( )`, have a *return value*. This is the result of running the function. In Example 1-6, the data that second `print` statement is given to print is the return value from `number_format( )`. In this case, it's the the comma-formatted population number.

One of the most common types of programs written in PHP is one that displays a web page containing information retrieved from a database. When you let submitted form parameters control what is pulled from the database, you open the door to a universe of interactivity on your web site. Example 1-7 shows a PHP program that connects to a database server, retrieves a list of dishes and their prices based on the value of the form parameter `meal`, and prints those dishes and prices in an HTML table.

Example 1-7. Displaying information from a database

```
<?php
require 'DB.php';
// Connect to MySQL running on localhost with username "menu"
// and password "good2eaT", and database "dinner"
$db = DB::connect('mysql://menu:good2eaT@localhost/dinner');
// Define what the allowable meals are
$meals = array('breakfast','lunch','dinner');
// Check if submitted form parameter "meal" is one of
// "breakfast", "lunch", or "dinner"
if (in_array($meals, $_POST['meal'])) {
    // If so, get all of the dishes for the specified meal
    $q = $dbh->query("SELECT dish,price FROM meals WHERE meal LIKE '" .
                     $_POST['meal'] ."'");
    // If no dishes were found in the database, say so
    if ($q->numrows == 0) {
        print "No dishes available.";
    } else {
        // Otherwise, print out each dish and its price as a row
        // in an HTML table
        print '<table><tr><th>Dish</th><th>Price</th></tr>';
        while ($row = $q->fetchRow( )) {
            print "<tr><td>$row[0]</td><td>$row[1]</td></tr>";
        }
        print "</table>";
    }
} else {
    // This message prints if the submitted parameter "meal" isn't
    // "breakfast", "lunch", or "dinner"
    print "Unknown meal.";
}
?>
```

There's a lot going on in Example 1-7, but it's a testament to the simplicity and power of PHP that it takes only about 15 lines of code (without comments) to make this dynamic, database-backed web page. The following describes what happens in those 15 lines.

The `DB::connect( )` function at the top of the example sets up the connection to the MySQL database with appropriate authentication information such as a username and a password. These functions, like the other database functions used in this example (`query( )`, `numrows( )`, and `fetchRow( )`), are explained in more detail in Chapter 7.

Things in the program that begin with a `$`, such as `$db`, `$_POST`, `$q`, and `$row`, are variables. Variables hold values that may change as the program runs or that are created at one point in the program and are saved to use later. Chapter 2 talks about variables.

After connecting to the database, the next task is to see what meal the user requested. The `$meals` array is initialized to hold the allowable meals: `breakfast`, `lunch`, and `dinner`. The statement `in_array($meals, $POST['meal'])` checks whether the submitted form parameter `meal` (the value of `$_POST['meal']`) is in the `$meals` array. If not, execution skips down to the end of the example, after the last `else`, and prints `Unknown meal`.

If an acceptable meal was submitted, `query( )` sends a query to the database. For example, if the meal is `breakfast`, the query that is sent is as follows:

```
SELECT dish,price FROM meals WHERE meal LIKE 'breakfast'
```

Queries to MySQL and most other databases are written in a language called Structured Query Language (SQL). Appendix B provides the basics of SQL. The `query( )` function returns an identifier that we can use to get further information about the query.

The `numrows( )` function uses that identifier to see how many matching meals the query found in the database. If there are no applicable meals, the program prints `No dishes available`. Otherwise, it displays information about the matching meals.

The program prints the beginning of the HTML table. Then, it uses the `fetchRow( )` function to retrieve each dish that the query found. The `print` statement uses elements of the array returned by `fetchRow( )` to display one table row per dish.

1.4 Basic Rules of PHP Programs

This section lays out some ground rules about the structure of PHP programs. More foundational than the basics such as "how do I print something" or "how do I add two numbers", these proto-basics are the equivalent of someone telling you that you should read pages in this book from top to bottom and left to right, or that what's important on the page are the black squiggles, not the large white areas.

If you've had a little experience with PHP already or you're the kind of person that prefers playing with all the buttons on your new DVD player before going back and reading in the manual about how the buttons actually work, feel free to skip ahead to Chapter 2 now and flip back here later. If you forge ahead to write some PHP programs of your own, and they're behaving unexpectedly or the PHP interpreter complains of "parse errors" when it tries to run your program, revisit this section for a refresher.

1.4.1 Start and End Tags

Each of the examples you've already seen in this chapter uses `<?php` as the PHP start tag and `?>` as the PHP end tag. The PHP interpreter ignores anything outside of those tags. Text before the start tag or after the end tag is printed with no interference from the PHP interpreter.

A PHP program can have multiple start and end tag pairs, as shown in Example 1-8.

Example 1-8. Multiple start and end tags

```
Five plus five is:
<?php print 5 + 5; ?>
<p>
Four plus four is:
<?php
 print 4 + 4;
?>
<p>
<img src="vacation.jpg" alt="My Vacation">
```

The PHP source code inside each set of `<?php ?>` tags is processed by the PHP interpreter, and the rest of the page is printed as is. Example 1-8 prints:

```
Five plus five is:
10<p>
Four plus four is:
8<p>
<img src="vacation.jpg" alt="My Vacation">
```

Some older PHP programs use `<?` as a start tag instead of `<?php`. The `<?` is called the *short open tag*, since it's shorter than `<?php`. It's usually better to use the regular `<?php` open tag since it's guaranteed to work on any server running the PHP interpreter. The short tag can be turned on or off with a PHP configuration setting. Appendix A shows you how to modify your PHP configuration to control which open tags are valid in your programs.

The rest of the examples in this chapter all begin with the `<?php` start tag and end with `?>`. In subsequent chapters, not all the examples have start and end tags — but remember, your programs need them for the PHP interpreter to recognize your code.

1.4.2 Whitespace and Case-Sensitivity

Like all PHP programs, the examples in this section consist of a series of statements, each of which end with a semicolon. You can put multiple PHP statements on the same line of a program as long as they are separated with a semicolon. You can put as many blank lines between statements as you want. The PHP interpreter ignores them. The semicolon tells the interpreter that one statement is over and another is about to begin. No whitespace at all or lots and lots of whitespace between statements doesn't affect the program's execution. (*Whitespace* is programmer-speak for blank-looking characters such as space, tab, and newline.)

In practice, it's good style to put one statement on a line and to put blank lines between statements only when it improves the readability of your source code. The spacing in Examples Example 1-9 and Example 1-10 is bad. Instead, format your code as in Example 1-11.

Example 1-9. This PHP is too cramped

```
<?php print "Hello"; print " World!"; ?>
```

Example 1-10. This PHP is too sprawling

```
<?php

print "Hello";


print " World!";

?>
```

Example 1-11. This PHP is just right

```
<?php
print "Hello";
print " World!";
?>
```

In addition to ignoring whitespace between lines, the PHP interpreter also ignores whitespace between language keywords and values. You can have zero spaces, one space, or a hundred spaces between `print` and "`Hello, World!`" and again between "`Hello, World!`" and the semicolon at the end of the line.

Good coding style is to put one space between `print` and the value being printed and then to follow the value immediately with a semicolon. Example 1-12 shows three lines, one with too much spacing, one with too little, and one with just the right amount.

Example 1-12. Spacing

```
<?php
print        "Too many spaces"           ;
print"Too few spaces";
print "Just the right amount of spaces";
?>
```

Language keywords (such as `print`) and function names (such as `number_format`) are not case-sensitive. The PHP interpreter doesn't care whether you use uppercase letters, lowercase letters, or both when you put these keywords and function names in your programs. The statements in Example 1-13 are identical from the interpreter's perspective.

Example 1-13. Keywords and function names are case-insensitive

```
// These four lines all do the same thing
print number_format(285266237);
PRINT Number_Format(285266237);
Print number_format(285266237);
pRiNt NUMBER_FORMAT(285266237);
```

1.4.3 Comments

As you've seen in some of the examples in this chapter, comments are a way to explain to other people how your program works. Comments in source code are an essential part of any program. When you're coding, what you are writing may seem crystal clear to you at the time. A few months later, however, when you need to go back and modify the program, your brilliant logic may not be so obvious. That's where comments come in. By explaining in plain language how the programs work, comments make programs much more understandable.

Comments are even more important when the person who needs to modify the program isn't the original author. Do yourself and anyone else who might have occasion to read your source code a favor and fill your programs with a lot of comments.

Perhaps because they're so important, PHP provides many ways to put comments in your programs. One syntax you've seen already is to begin a line with `//`. This tells the PHP interpreter to treat everything on that line as a comment. After the

end of the line, the code is treated normally. This style of comment is also used in other programming languages such as C++, JavaScript, and Java. You can also put `//` on a line after a statement to have the remainder of the line treated as a comment. PHP also supports the Perl- and shell-style single-line comments. These are lines that begin with `#`. You can use `#` to start a comment in the same places that you can use `//`, but the modern style prefers `//` over `#`. Some single-line comments are shown in Example 1-14.

Example 1-14. Single-line comments with // or #

```
// This line is a comment
print "Smoked Fish Soup ";
print 'costs $3.25.';

# Add another dish to the menu
print 'Duck with Pea Shoots ';
print 'costs $9.50.';
// You can put // or # inside single-line comments
// Using // or # somewhere else on a line also starts a comment
print 'Shark Fin Soup'; // I hope it's good!
print 'costs $25.00!'; # This is getting expensive!

# Putting // or # inside a string doesn't start a comment
print 'http://www.example.com';
print 'http://www.example.com/menu.php#dinner';
```

For a multiline comment, start the comment with `/*` and end with `*/`. Everything between the `/*` and `*/` is treated as a comment by the PHP interpreter. Multiline comments are useful for temporarily turning off a small block of code. Example 1-15 shows some multiline comments.

Example 1-15. Multiline comments

```
/* We're going to add a few things to the menu:
   - Smoked Fish Soup
   - Duck with Pea Shoots
   - Shark Fin Soup
*/
print 'Smoked Fish Soup, Duck with Pea Shoots, Shark Fin Soup ';
print 'Cost: 3.25 + 9.50 + 25.00';

/* This is the old menu:
The following lines are inside this comment so they don't get executed.
print 'Hamburger, French Fries, Cola ';
print 'Cost: 0.99 + 1.25 + 1.50';
*/
```

There is no strict rule in PHP about which comment style is the best. Multiline comments are often the easiest to use, especially when you want to comment out a block of code or write a few lines describing a function. However, when you want to tack on a short explanation to the end of a line, a `//`-style comment fits nicely. Use whichever comment style you feel most comfortable with.

1.5 Chapter Summary

Chapter 1 covers:

- PHP's usage by a web server to create a response or document to send back to the browser.
- PHP as a server-side language, meaning it runs on the web server. This is in contrast to a client-side language such as JavaScript.
- What you sign up for when you decide to use PHP: it's free (in terms of money and speech), cross-platform, popular, and designed for web programming.
- How PHP programs that print information, process forms, and talk to a database appear.
- Some basics of the structure of PHP programs, such as the PHP start and end tags (`<?php` and `?>`), whitespace, case-sensitivity, and comments.