The Benefits of PHP, MySQL, JavaScript, CSS, and HTML5

The world of Web 1.0, but it wasn't long before the rush was on to create Web 1.1, with the development of such browser enhancements as Java, JavaScript, JScript (Microsoft's slight variant of JavaScript), and ActiveX. On the server side, progress was being made on the Common Gateway Interface (CGI) using scripting languages such as Perl (an alternative to the PHP language) and *server-side scripting*—inserting the contents of one file (or the output of a system call) into another one dynamically.

Once the dust had settled, three main technologies stood heads and shoulders above the others. Although Perl was still a popular scripting language with a strong following,
PHP's simplicity and built-in links to the MySQL database program had earned it more than double the number of users.
JavaScript, which had become an essential part of the equation for dynamically manipulating Cascading Style Sheets (CSS) and HTML, now took on the even more muscular task of handling the client side of the Ajax process. Under Ajax, web pages perform data handling and send requests to web servers in the background—without the web user being aware that this is going on.
No doubt the symbiotic nature of PHP and MySQL helped propel them both forward, but what attracted developers to them in the first place? The simple answer has to be the ease with which you can use them to quickly create dynamic elements on websites.
MySQL is a fast and powerful, yet easy-to-use, database system that offers just about anything a website would need in order to find and serve up data to browsers.

When PHP allies with MySQL to store and retrieve this data, you have the fundamental parts required for the development of social networking sites and the beginnings of Web 2.0.
And when you bring JavaScript and CSS into the mix too, you have a recipe for building highly dynamic and interactive websites.

Using PHP

With PHP, it's a simple matter to embed dynamic activity in web pages. When you give pages the *.php* extension, they have instant access to the scripting language. From a developer's point of view, all you have to do is write code such as the following:

```
<?php
echo " Today is " . date("l") . ". ";
?>
Here's the latest news.
```

The opening <?php tells the web server to allow the PHP program to interpret all the following code up to the ?> tag. Outside of this construct, everything is sent to the client as direct HTML. So the text Here's the latest news. is simply output to the browser; within the PHP tags, the built-in date function displays the current day of the week according to the server's system time. The final output of the two parts looks like this:

Today is Wednesday. Here's the latest news.

PHP is a flexible language, and some people prefer to place the PHP construct directly next to PHP code, like this:

```
Today is <?php echo date("l"); ?>. Here's the latest news.
```

Using PHP, you have unlimited control over your web server. Whether you need to modify HTML on the fly, process a credit card, add user details to a database, or fetch information from a third-party website, you can do it all from within the same PHP files in which the HTML itself resides.

Using MySQL

Using PHP, you can make all these calls directly to MySQL without having to run the MySQL program yourself or use its command-line interface. This means you can save the results in arrays for processing and perform multiple lookups, each dependent on the results returned from earlier ones, to drill down to the item of data you need. For even more power, as you'll see later, there are additional functions built right into MySQL that you can call up for common operations and extra speed.

Using JavaScript

JavaScript, was created to enable scripting access to all the elements of an HTML document. In other words, it provides a means for dynamic user interaction such as checking email address validity in input forms, and displaying prompts such as "Did you really mean that?" Combined with CSS (see the following section), JavaScript is the power behind dynamic web pages that change in front of your eyes rather than when a new page is returned by the server.
However, JavaScript can also be tricky to use, due to some major differences in the ways different browser designers have chosen to implement it. This mainly came about when some manufacturers tried to put additional functionality into their browsers at the expense of compatibility with their rivals.
For now, let's take a look at how to use basic JavaScript, accepted by all browsers:

```
<script type="text/javascript">
document.write("Today is " + Date() );
</script>
```

This code snippet tells the web browser to interpret everything within the script tags as JavaScript, which the browser then does by writing the text Today is to the current document, along with the date, by using the JavaScript function Date. The result will look something like this:

Today is Sun Jan 01 2017 01:23:45

JavaScript was originally developed to offer dynamic control over the various elements within an HTML document, and that is still its main use. But more and more, JavaScript is being used for Ajax. This is a term for the process of accessing the web server in the background.

Using CSS

With the emergence of the CSS3 standard in recent years, CSS now offers a level of dynamic interactivity previously supported only by JavaScript. For example, not only can you style any HTML element to change its dimensions, colors, borders, spacing, and so on, but now you can also add animated transitions and transformations to your web pages, using only a few lines of CSS.
Using CSS can be as simple as inserting a few rules between <style> and </style> tags in the head of a web page, like this:

```
<style>
p {
text-align:justify;
font-family:Helvetica;
}
</style>
```

These rules will change the default text alignment of the <p> tag so that paragraphs contained in it will be fully justified and will use the Helvetica font.

And Then There's HTML5

As useful as all these additions to the web standards became, they were not enough for ever more ambitious developers. For example, there was still no simple way to manipulate graphics in a web browser without resorting to plug-ins such as Flash. And the same went for inserting audio and video into web pages. Plus, several annoying inconsistencies had crept into HTML during its evolution.

So, to clear all this up and take the Internet beyond Web 2.0 and into its next iteration, a new standard for HTML was created to address all these shortcomings. It was called *HTML5* and it began development as long ago as 2004, when the first draft was drawn up by the Mozilla Foundation and Opera Software (developers of two popular web browsers). But it wasn't until the start of 2013 that the final draft was submitted to the World Wide Web Consortium (W3C), the international governing body for web standards.

There's actually a great deal of new stuff in HTML (and quite a few things that have been changed or removed), but in summary, here's what you get:

Markup

Including new elements such as <nav> and <footer>, and deprecated elements like <font> and .

New APIs

Such as the <canvas> element for writing and drawing on a graphics canvas, <audio> and <video> elements, offline web applications, microdata, and local storage.

Applications

Including two new rendering technologies: MathML (Math Markup Language) for displaying mathematical formulae and SVG (Scalable Vector Graphics) for creating graphical elements outside of the new <canvas> element. However, MathML and SVG are somewhat specialist, and are so feature-packed they would need a book of their own, so I don't cover them here.

1.1 Configuration of PHP, Apache Web Server, MySQL and Open Source

The Apache Web Server

In addition to PHP, MySQL, JavaScript, CSS, and HTML5, there's a sixth hero in the dynamic Web: the web server; the Apache web server. We've discussed a little of what a web server does during the HTTP server/ client exchange, but it does much more behind the scenes. For example, Apache doesn't serve up just HTML files—it handles a wide range of files from images and Flash files to MP3 audio files, RSS (Really Simple Syndication) feeds, and so on. To do this, each element a web client encounters in an HTML page is also requested from the server, which then serves it up. But these objects don't have to be static files such as GIF images. They can all be generated by programs such as PHP scripts. That's right: PHP can even create images and other files for you, either on the fly or in advance to serve up later.

To do this, you normally have modules either precompiled into Apache or PHP or called up at runtime. One such module is the GD (Graphics Draw) library, which PHP uses to create and handle graphics. Apache also supports a huge range of modules of its own. In addition to the PHP module, the most important for your purposes as a web programmer are the modules that handle security. Other examples are the Rewrite module, which enables the web server to handle a varying range of URL types and rewrite them to its own internal requirements, and the Proxy module, which you can use to serve up often-requested pages from a cache to ease the load on the server.

About Open Source

Whether or not being open source is the reason these technologies are so popular has often been debated, but PHP, MySQL, and Apache *are* the three most commonly used tools in their categories. What can be said definitively, though, is that their being open source means that they have been developed in the community by teams of programmers writing the features they themselves want and need, with the original code

available for all to see and change. Bugs can be found and security breaches can be prevented before they happen.
There's another benefit: all these programs are free to use. There's no worrying about having to purchase additional licenses if you have to scale up your website and add more servers. And you don't need to check the budget before deciding whether to upgrade to the latest versions of these products.

Without using program code, let's summarize the contents of this chapter by looking at the process of combining some of these technologies into an everyday Ajax feature that many websites use: checking whether a desired username already exists on the site when a user is signing up for a new account. A good example of this can be seen with Gmail (see Figure 1-3).

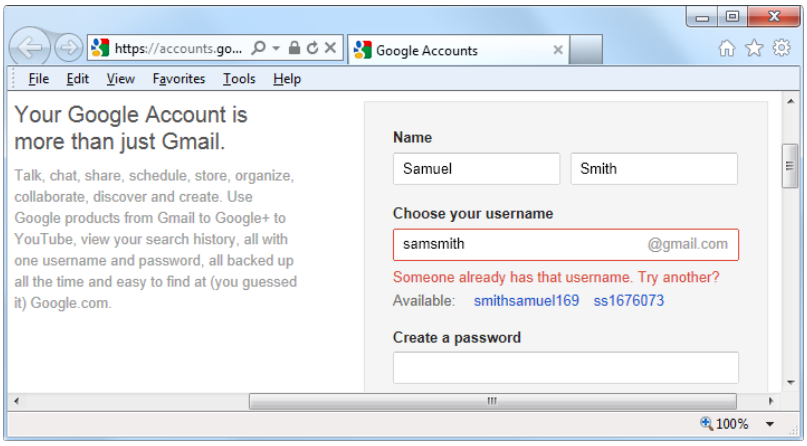


Figure 1-3. Gmail uses Ajax to check the availability of usernames
The steps involved in this Ajax process would be similar to the following:
1. The server outputs the HTML to create the web form, which asks for the necessary details, such as username, first name, last name, and email address.
2. At the same time, the server attaches some JavaScript to the HTML to monitor the username input box and check for two things: (a) whether some text has been typed into it, and (b) whether the input has been deselected because the user has clicked on another input box.
3. Once the text has been entered and the field deselected, in the background the JavaScript code passes the username that was entered back to a PHP script on the web server and awaits a response.
4. The web server looks up the username and replies back to the JavaScript regarding whether that name has already been taken.
5. The JavaScript then places an indication next to the username input box to show whether the name is one available to the user—perhaps a green checkmark or a red cross graphic, along with some text.
6. If the username is not available and the user still submits the form, the JavaScript interrupts the submission and reemphasizes (perhaps with a larger graphic and/or an alert box) that the user needs to choose another username.
7. Optionally, an improved version of this process could even look at the username requested by the user and suggest an alternative that is currently available.

1.2 Installing PHP for (Windows, Wamp server, XAMP server)

What Is a WAMP, MAMP, or LAMP?

WAMP, MAMP, and LAMP are abbreviations for "Windows, Apache, MySQL, and PHP," "Mac, Apache, MySQL, and PHP," and "Linux, Apache, MySQL, and PHP." These abbreviations describe a fully functioning setup used for developing dynamic Internet web pages.
WAMPs, MAMPs, and LAMPs come in the form of a package that binds the bundled programs together so that you don't have to install and set them up separately. This means you can simply download and install a single program, and follow a few easy prompts, to get your web development server up and running in the quickest time with a minimum hassle.

During installation, several default settings are created for you. The security configurations of such an installation will not be as tight as on a production web server, because it is optimized for local use. For these reasons, you should never install such a setup as a production server. But for developing and testing websites and applications, one of these installations should be entirely sufficient.

What is XAMPP?

XAMPP is the most popular PHP development environment. There are several available WAMP servers, each offering slightly different configurations, but out of the various open source and free options, the best is probably XAMPP.

XAMPP is a completely free, easy to install Apache distribution containing MariaDB, PHP, and Perl. The XAMPP open source package has been set up to be incredibly easy to install and to use.

Installing XAMPP on Windows:
Testing the Installation:

The first thing to do at this point is verify that everything is working correctly. To do this, you are going to try to display the default web page, which will have been saved in the server's document root folder (see Figure 2-13). Enter either of the following two URLs into the address bar of your browser:
localhost
127.0.0.1
The word *localhost* is used in URLs to specify the local computer, which will also respond to the IP address of 127.0.0.1, so you can use either method of calling up the document root of your web server.

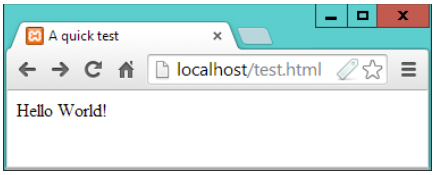
Accessing the document root
The *document root* is the directory that contains the main web documents for a domain. This is the one that is entered when a basic URL without a path is typed into a browser, such as *http://yahoo.com* or, for your local server, *http://localhost*. By default, XAMP uses the following location for this directory:
C:/xampp/htdocs

To ensure that you have everything correctly configured, you should now create the obligatory "Hello World" file. So create a small HTML file along the following lines using Windows Notepad or any other program or text editor, but not a rich word processor such as Microsoft Word (unless you save as plain text):

```
<html>
    <head>
        <title>A quick test</title>
    </head>
    <body>
        Hello World!
    </body>
</html>
```

Once you have typed this, save the file into the document root directory previously discussed, using the filename *test.htm*. If you are using Notepad, make sure that the "Save as type" box is changed from "Text Documents (*.txt)" to "All Files (*.*)". Or, if you prefer, you can save the file using the *.html* file extension; either is acceptable. You can now call this page up in your browser by entering one of the following URLs (according to the extension you used) in its address bar (see Figure 2-14):

http://localhost/test.html



Installing a LAMP on Linux

However, many Linux versions come preinstalled with a web server and MySQL, and the chances are that you may already be all set to go. To find out, try entering the following into a browser and see whether you get a default document root web page: http://localhost

If this works, you probably have the Apache server installed and may well have MySQL up and running too; check with your system administrator to be sure. If you don't yet have a web server installed, however, there's a version of XAMPP available that you can download at *apachefriends.org*. Installation is similar to the sequence shown in the Windows section, and if you need further assistance on using the suite, please refer to *apachefriends.org/faq_linux.html.*

Working Remotely

If you have access to a web server already configured with PHP and MySQL, you can always use that for your web development. But unless you have a high-speed connection, it is not always your best option. Developing locally allows you to test modifications with little or no upload delay. Accessing MySQL remotely may not be easy either. You may have to Telnet or SSH into your server to manually create databases and set permissions from the command line. Your web-hosting company will advise you on how best to do this and provide you with any password it has set for your MySQL access (as well as, of course, for getting into the server in the first place).

Using FTP

To transfer files to and from your web server, you will need an FTP program. If you go searching the Web for a good one, you'll find so many that it could take you quite a while to come across one with all the right features for you. Nowadays I always recommend FireFTP, because of these advantages:

• It is an add-on for the Firefox web browser, and will therefore work on any platform on which Firefox runs.
• Calling it up can be as simple as selecting a bookmark.
• It is one of the fastest and easiest-to-use FTP programs that I have encountered.

Using a Program Editor

Although a plain-text editor works for editing HTML, PHP, and JavaScript, there have been some tremendous improvements in dedicated program editors, which now incorporate very handy features such as colored syntax highlighting. Today's program editors are smart and can show you where you have syntax errors before you even run a program. Once you've used a modern editor, you'll wonder how you ever managed without one.

Using an IDE

As good as dedicated program editors can be for your programming productivity, their utility pales into insignificance when compared to *integrated development environments* (IDEs), which offer many additional features such as in-editor debugging and program testing, as well as function descriptions and much more. When developing with an IDE, you can set breakpoints and then run all (or portions) of your code, which will then stop at the breakpoints and provide you with information about the program's current state.

There are several IDEs available for different platforms, most of which are commercial, but there are some free ones too. Eclipse PDT , Komodo IDE , NetBeans , phpDesigner , PHPEclipse , PhpED , PHPEdit

Alternative WAMPs

Here's a selection of some of the best, in my opinion:

• EasyPHP: *easyphp.org*
• WAMPServer: *wampserver.com/en*
• Glossword WAMP: *glossword.biz/glosswordwamp*

General Language Features

Every user has specific reasons for using PHP to implement a mission-critical application, although one could argue that such motives tend to fall into four key categories: **practicality, power, possibility, and price.**

Practicality

Its ability to nest functions.
The following example produces a string of five alphanumeric characters suchas **a3jh8**:

$randomString = substr(md5(microtime()), 0, 5);

Language that allows the user to build powerful applications even with a minimum of knowledge. For instance, a useful PHP script can consist of as little as one line; unlike C, there is no need for the mandatory inclusion of libraries. For example, the following represents a complete PHP script, the purpose of which is to output the current date, in this case one formatted like **September 23, 2007**:

```
<?php echo date("F j, Y"); ?>
```

PHP is a *loosely typed* language, meaning there is no need to explicitly create, typecast, or destroy a variable, although you are not prevented from doing so. PHP handles such matters internally, creating variables on the fly as they are called in a script, and employing a best-guess formula for automatically typecasting variables. For instance, PHP considers the following set of statements to be perfectly valid:

```
<?php
$number = "5"; // $number is a string
$sum = 15 + $number; // Add an integer and string to produce integer
$sum = "twenty"; // Overwrite $sum with a string.
?>
```

PHP will also automatically destroy variables and return resources to the system when the script completes. In these and in many other respects, by attempting to handle many of the administrative aspects of programming internally, PHP allows the developer to concentrate almost exclusively on the final goal, namely a working application.

Power

PHP developers have almost 200 native libraries at their disposal, collectively containing well over 1,000 functions, in addition to thousands of third-party extensions. Although you're likely aware of PHP's ability to interface with databases, manipulate form information, and create pages dynamically, you might not know that PHP can also do the following:

- Create and manipulate Adobe Flash and Portable Document Format (PDF) files.
- Evaluate a password for guessability by comparing it to language dictionaries and easily broken patterns.
- Parse even the most complex of strings using the POSIX and Perl-based regular expression libraries.
- Authenticate users against login credentials stored in flat files, databases, and even Microsoft's Active Directory.
- Communicate with a wide variety of protocols, including LDAP, IMAP, POP3, NNTP, and DNS, among others.
- Tightly integrate with a wide array of credit-card processing solutions.

Possibility

PHP developers are rarely bound to any single implementation solution. On the contrary, a user is typically fraught with choices offered by the language. For example, consider PHP's array of database support options. Native support is offered for more than 25 database products, including Adabas D, dBase, Empress, FilePro, FrontBase, Hyperwave, IBM DB2, Informix, Ingres, InterBase, mSQL, Microsoft SQL Server, MySQL, Oracle, Ovrimos, PostgreSQL, Solid, Sybase, Unix dbm, and Velocis. In addition, abstraction layer functions are available for accessing Berkeley DB–style databases. Several generalized database abstraction solutions are also available, among the most popular being PDO (**www.php.net/pdo**) and MDB2

(**http://pear.php.net/package/MDB2**). Finally, if you're looking for an object relational mapping (ORM) solution, projects such as Propel (**www.propelorm.org**) should fit the bill quite nicely.
PHP's flexible string-parsing capabilities offer users of differing skill sets the opportunity to not only immediately begin performing complex string operations but also to quickly port programs of similar unctionality (such as Perl and Python) over to PHP. In addition to almost 100 string-manipulation functions, Perl-based regular expression formats are supported PHP offers comprehensive support for procedural programming and object-oriented paradigm PHP allows you to quickly capitalize on your current skill set with very little time investment.

Price

PHP is available free of charge! Since its inception, PHP has been without usage, modification, and redistribution restrictions. In recent years, software meeting such open licensing qualifications has been referred to as *open source software*.

1.3 PHP Structure and Syntax

Using Comments

There are two ways in which you can add comments to your PHP code. The first turns a single line into a comment by preceding it with a pair of forward slashes:

```
// This is a comment
```

This version of the comment feature is a great way to temporarily remove a line of code from a program that is giving you errors. For example, you could use such a comment to hide a debugging line of code until you need it, like this:

```
// echo "X equals $x";
```

You can also use this type of comment directly after a line of code to describe its action, like this:

```
$x += 10; // Increment $x by 10
```

When you need multiple-line comments, there's a second type of comment, which looks like

```
<?php
/* This is a section
of multiline comments
which will not be
interpreted */
?>
```

Basic Syntax

PHP is quite a simple language with roots in C and Perl, yet it looks more like Java. It is also very flexible, but there are a few rules that you need to learn about its syntax and structure.

Semicolons

You may have noticed in the previous examples that the PHP commands ended with a semicolon, like this:

```
$x += 10;
```

Probably the most common cause of errors you will encounter with PHP is forgetting this semicolon. This causes PHP to treat multiple statements like one statement, which it is unable to understand, prompting it to produce a Parse error message.

The $ symbol

The $ symbol has come to be used in many different ways by different programming languages. For example, if you have ever written in the BASIC language, you will have used the $ to terminate variable names to denote them as strings.
In PHP, however, you must place a $ in front of *all* variables. This is required to make the PHP parser faster, as it instantly knows whenever it comes across a variable. Whether your variables are numbers, strings, or arrays, they should all look something like those in Example 3-3.
Example 3-3. Three different types of variable assignment

```
<?php
$mycounter = 1; $mystring = "Hello"; $myarray = array("One", "Two", "Three"); ?>
```

1.7 Constants, Variables: Static and Global Variable.

Variables

Variable Declaration

A variable always begins with a dollar sign, $, which is then followed by the variable name. Variable names follow the same naming rules as identifiers. That is, a variable name can begin with either a letter or an underscore and can consist of letters, underscores, numbers, or other ASCII characters ranging from 127 through 255. The following are all valid variables:

- $color
- $operating_system
- $_some_variable
- $model

Note that variables are case sensitive. For instance, the following variables bear no relation to one another:

- $color
- $Color
- $COLOR

Interestingly, variables do not have to be explicitly declared in PHP as they do in a language such as C. Rather, variables can be declared and assigned values simultaneously. Nonetheless, just because you *can* do something doesn't mean you *should*. Good programming practice dictates that all variables should be declared prior to use, preferably with an accompanying comment. Once you've declared your variables, you can begin assigning values to them. Two methodologies are available for variable assignment: **by value and by reference.**

Value Assignment

Assignment by value simply involves copying the value of the assigned expression to the variable assignee. This is the most common type of assignment. A few examples follow:

```
$color = "red";
$number = 12;
$age = 12;
$sum = 12 + "15"; // $sum = 27
```

Keep in mind that each of these variables possesses a copy of the expression assigned to it. For example, $number and $age each possesses their own unique copy of the value 12. If you prefer that two variables point to the same copy of a value, you need to assign by reference.

Reference Assignment

PHP 4 introduced the ability to assign variables by reference, which essentially means that you can create a variable that refers to the same content as another variable does. Therefore, a change to any variable referencing a particular item of variable content will be reflected among all other variables referencing that same content. You can assign variables by reference by appending an ampersand (&) to the equal sign. Let's consider an example:

```
<?php
$value1 = "Hello";
$value2 =& $value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"
?>
```

http://localhost/myscripts/first.php

An alternative reference-assignment syntax is also supported, which involves appending the ampersand to the front of the variable being referenced. The following example adheres to this new syntax:

```
<?php
$value1 = "Hello";
$value2 = &$value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"  ?>
```

Example:
$username = "Fred Smith";
echo $username;

Or you can assign it to another variable like this: $current_user = $username;

Your first PHP program

```
<?php // test1.php
$username = "Fred Smith";
echo $username;
echo "<br>";
$current_user = $username;
echo $current_user;
?>
```

Now you can call it up by entering the following into your browser's address bar: http://localhost/test1.php

Numeric variables

Variables don't contain just strings—they can contain numbers too. If we return to the matchbox analogy, to store the number 17 in the variable $count, the equivalent would be placing, say, 17 beads in a matchbox on which you have written the word *count*: $count = 17;

You could also use a floating-point number (containing a decimal point); the syntax is the same: $count = 17.5;

Variable-naming rules

When creating PHP variables, you must follow these four rules:

• Variable names must start with a letter of the alphabet or the _ (underscore) character.

• Variable names can contain only the characters a-z, A-Z, 0-9, and _ (underscore).

• Variable names may not contain spaces. If a variable must comprise more than one word, it should be separated with the _ (underscore) character (e.g., $user_name).

• Variable names are case-sensitive. The variable $High_Score is not the same as the variable $high_score.

Variable Scope

However you declare your variables (by value or by reference), you can declare them anywhere in a PHP script. The location of the declaration greatly influences the realm in which a variable can be accessed, however. This accessibility domain is known as its *scope*. PHP variables can be one of four scope types:

- Local variables
- Function parameters
- Global variables
- Static variables

If you have a very long program, it's quite possible that you could start to run out of good variable names, but with PHP you can decide the *scope* of a variable. In other words, you can, for example, tell it that you want the variable $temp to be used only inside a particular function and to forget it was ever used when the function returns.

In fact, this is the default scope for PHP variables. Alternatively, you could inform PHP that a variable is global in scope and thus can be accessed by every other part of your program.

Local variables

Local variables are variables that are created within, and can be accessed only by, a function. They are generally temporary variables that are used to store partially processed results prior to the function's return.

One set of local variables is the list of arguments to a function. In the previous section, we defined a function that accepted a parameter named $timestamp. This is meaningful only in the body of the function; you can't get or set its value outside the function.

```
$x = 4;
function assignx () {
$x = 0;
printf("\$x inside function is %d <br />", $x);
}
assignx();
printf("\$x outside of function is %d <br />", $x);
```

Output:
$x inside function is 0
$x outside of function is 4

For another example of a local variable, take another look at the longdate function, which is modified slightly in Example 3-13.
Example 3-13. An expanded version of the longdate function

```
<?php
function longdate($timestamp)
{
$temp = date("l F jS Y", $timestamp);
return "The date is $temp";
}
?>
```

Here we have assigned the value returned by the date function to the temporary variable $temp, which is then inserted into the string returned by the function. As soon as the function returns, the value of $temp is cleared, as if it had never been used at all. Now, to see the effects of variable scope, let's look at some similar code in

Example 3-14. Here $temp has been created *before* we call the longdate function.
Example 3-14. This attempt to access $temp in function longdate will fail

```
<?php
$temp = "The date is ";
echo longdate(time());
function longdate($timestamp)
{
return $temp . date("l F jS Y", $timestamp);
}
?>
```

However, because $temp was neither created within the longdate function nor passed to it as a parameter, longdate cannot access it. Therefore, this code snippet outputs only the date, not the preceding text. In fact, it will first display the error message
Notice: Undefined variable: temp.

The reason for this is that, by default, variables created within a function are local to that function, and variables created outside of any functions can be accessed only by nonfunction code.
Some ways to repair Example 3-14 appear in Example 3-15 and Example 3-16.
Example 3-15. Rewriting to refer to $temp within its local scope fixes the problem

```
<?php
$temp = "The date is ";
echo $temp . longdate(time());
function longdate($timestamp)
{
return date("l F jS Y", $timestamp);
}
?>
```

Example 3-15 moves the reference to $temp out of the function. The reference appears in the same scope where the variable was defined.

Example 3-16. An alternative solution: passing $temp as an argument

```
<?php
$temp = "The date is ";
echo longdate($temp, time());
function longdate($text, $timestamp)
{
return $text . date("l F jS Y", $timestamp);
}
?>
```

Function Parameters

In PHP, as in many other programming languages, any function that accepts arguments must declare those arguments in the function header. Although those arguments accept values that come from outside of the function, they are no longer accessible once the function has exited.

Function parameters are declared after the function name and inside parentheses. They are declared much like a typical variable would be:

```
// multiply a value by 10 and return it to the caller
function x10 ($value) {
$value = $value * 10;
return $value;
}
```

Global variables

In contrast to local variables, a *global* variable can be accessed in any part of the program. To modify a global variable, however, it must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword global in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example:

```
$somevar = 15;
function addit() {
global $somevar;
$somevar++;
echo "Somevar is $somevar";
}
addit();
```

The displayed value of $somevar would be 16. However, if you were to omit this line, global $somevar; then the variable $somevar would be assigned the value 1 because $somevar would then be considered local within the addit() function. This local declaration would be implicitly set to 0 and then incremented by 1 to display the value 1.

An alternative method for declaring a variable to be global is to use PHP's $GLOBALS array. Reconsidering the preceding example, you can use this array to declare the variable $somevar to be global:

```
$somevar = 15;
function addit() {
$GLOBALS["somevar"]++;
}
addit();
echo "Somevar is ".$GLOBALS["somevar"];
```

This returns the following:

Somevar is 16

Regardless of the method you choose to convert a variable to global scope, be aware that the global scope has long been a cause of grief among programmers due to unexpected results that may arise from its careless use. Therefore, although global variables can be useful, be prudent when using them.

Static variables

In contrast to the variables declared as
function parameters, which are destroyed on the function's exit, a static variable does not lose its value when the function exits and will still hold that value if the function is called again. You can declare a variable as static simply by placing the keyword static in front of the variable name, like so:
STATIC $somevar;

Consider an example:

```
function keep_track() {
static $count = 0;
$count++;
echo $count;
echo "<br />";
}
keep_track();
keep_track();
keep_track();
```

What would you expect the outcome of this script to be? If the variable $count was not designated to e static (thus making $count a local variable), the outcome would be as follows:

```
1
1
1
```

However, because $count is static, it retains its previous value each time the function is executed. Therefore, the outcome is the following:

```
1
2
3
```

Example 3-17. A function using a static variable

```
<?php
function test()
{
static $count = 0;
echo $count;
$count++; }    ?>
```

Here the very first line of function test creates a static variable called $count and initializes it to a value of 0. The next line outputs the variable's value; the final one increments it.
The next time the function is called, because $count has already been declared, the first line of the function is skipped. Then the previously incremented value of $count is displayed before the variable is again incremented.
If you plan to use static variables, you should note that you cannot assign the result of an expression in their definitions. They can be initialized only with predetermined values (see Example 3-18).
Example 3-18. Allowed and disallowed static variable declarations

```
<?php
static $int = 0; // Allowed
static $int = 1+2; // Disallowed (will produce a Parse error)
```

```
static $int = sqrt(144); // Disallowed
?>
```

Constants

A *constant* is a value that cannot be modified throughout the execution of a program. Constants are particularly useful when working with values that definitely will not require modification, such as Pi (3.141592) or the number of feet in a mile (5,280). Once a constant has been defined, it cannot be changed (or redefined) at any other point of the program. Constants are defined using the define() function.

Defining a Constant

The define() function defines a constant by assigning a value to a name. Its prototype follows:
boolean define(string name, mixed value [, bool case_insensitive])

If the optional parameter *case_insensitive* is included and assigned TRUE, subsequent references to the constant will be case insensitive. Consider the following example in which the mathematical constant Pi is defined: define("PI", 3.141592);
The constant is subsequently used in the following listing:

```
printf("The value of Pi is %f", PI);
$pi2 = 2 * PI;
printf("Pi doubled equals %f", $pi2);
```

This code produces the following results:

```
The value of pi is 3.141592.
Pi doubled equals 6.283184.
```

Predefined Constants

PHP comes ready-made with dozens of predefined constants that you generally will be unlikely to use as a beginner to PHP. However, there are a few—known as the *magic constants*—that you will find useful. The names of the magic constants always have two underscores at the beginning and two at the end, so that you won't accidentally try to name one of your own constants with a name that is already taken. They are detailed in Table 3-5. The concepts referred to in the table will be introduced in future chapters.

Table 3-5. PHP's magic constants

__LINE__	The current line number of the file.
__FILE__	The full path and filename of the file. If used inside an include, the name of the included file is returned. In PHP 4.0.2, __FILE__ always contains an absolute path with symbolic links resolved, whereas in older versions it might contain a relative path under some circumstances.
__DIR__	The directory of the file. If used inside an include, the directory of the included file is returned. This is equivalent to *dirname*(__FILE__). This directory name does not have a trailing slash unless it is the root directory. (Added in PHP 5.3.0.)
__FUNCTION__	The function name. (Added in PHP 4.3.0.) As of PHP 5, returns the function name as it was declared (case-sensitive). In PHP 4, its value is always lowercase.
__CLASS__	The class name. (Added in PHP 4.3.0.) As of PHP 5, returns the class name as it was declared (casesensitive). In PHP 4, its value is always lowercased.
__METHOD__	The class method name. (Added in PHP 5.0.0.) The method name is returned as it was declared (casesensitive).
__NAMESPACE__	The name of the current namespace (case-sensitive). This constant is defined at compile time. (Added in PHP 5.3.0.)

1.8 Variable manipulation

Variable Typing

PHP is a very loosely typed language. This means that variables do not have to be declared before they are used, and that PHP always converts variables to the *type* required by their context when they are accessed.

For example, you can create a multiple-digit number and extract the *n*th digit from it simply by assuming it to be a string. In the following snippet of code, the numbers 12345 and 67890 are multiplied together, returning a result of 838102050, which is then placed in the variable $number, as shown in Example 3-10.

Example 3-10. Automatic conversion from a number to a string

```
<?php
$number = 12345 * 67890;
echo substr($number, 3, 1);
?>
```

At the point of the assignment, $number is a numeric variable. But on the second line, a call is placed to the PHP function substr, which asks for one character to be returned from $number, starting at the fourth position (remembering that PHP offsets start from zero). To do this, PHP turns $number into a nine-character string, so that substr can access it and return the character, which in this case is 1.

The same goes for turning a string into a number, and so on. In Example 3-11, the variable $pi is set to a string value, which is then automatically turned into a floatingpoint number in the third line by the equation for calculating a circle's area, which outputs the value 78.5398175.

Example 3-11. Automatically converting a string to a number

```
<?php
$pi = "3.1415927";
$radius = 5;
echo $pi * ($radius * $radius);
?>
```

1.9 Dynamic variables (Variable Variables)

PHP allows us to use dynamic variable. Dynamic variable is *variable variables*. We can name a variable with the value stored in another variable. That is, one variable contains the name of another variable.

On occasion, you may want to use a variable whose content can be treated dynamically as a variable in itself. Consider this typical variable assignment:

```
$recipe = "spaghetti";
```

Interestingly, you can treat the value spaghetti as a variable by placing a second dollar sign in front of the original variable name and again assigning another value:

```
$$recipe = "& meatballs";
```

This in effect assigns *& meatballs* to a variable named spaghetti. Therefore, the following two snippets of code produce the same result:

```
echo $recipe $spaghetti;
echo $recipe ${$recipe};
```

The result of both is the string *spaghetti & meatballs*.

1.10 Operators

Operators

Operators are the mathematical, string, comparison, and logical commands such as plus, minus, multiply, and divide. PHP looks a lot like plain arithmetic; for instance, the following statement outputs 8:

```
echo 6 + 2;
```

Before moving on to learn what PHP can do for you, take a moment to learn about the various operators it provides.

Operands
Operands are the inputs of an expression. You might already be familiar with the manipulation and use of operands not only through everyday mathematical calculations, but also through prior programming experience. Some examples of operands follow:
$a++; // $a is the operand
$sum = $val1 + val2; // $sum, $val1 and $val2 are operands

Expressions
An *expression* is a phrase representing a particular action in a program. All expressions consist of at least one operand and one or more operators. A few examples follow:

$a = 5; // assign integer value 5 to the variable $a
$a = "5"; // assign string value "5" to the variable $a
$sum = 50 + $some_int; // assign sum of 50 + $some_int to $sum
$wine = "Zinfandel"; // assign "Zinfandel" to the variable $wine
$inventory++; // increment the variable $inventory by 1

Operator Precedence
Operator precedence is a characteristic of operators that determines the order in which they evaluate the operands surrounding them. PHP follows the standard precedence rules used in elementary school math class. Consider a few examples: $total_cost = $cost + $cost * 0.06;
This is the same as writing
$total_cost = $cost + ($cost * 0.06);

because the multiplication operator has higher precedence than the addition operator.

Operator Associativity
The *associativity* characteristic of an operator specifies how operations of the same precedence (i.e., having the same precedence value, as displayed in Table 3-3) are evaluated as they are executed. Associativity can be performed in two directions, left-to-right or right-to-left. Left-to-right associativity means that the various operations making up the expression are evaluated from left to right. Consider the following example:
$value = 3 * 4 * 5 * 7 * 2;
The preceding example is the same as the following:
$value = ((((3 * 4) * 5) * 7) * 2);

Arithmetic operators
Arithmetic operators do what you would expect. They are used to perform mathematics. You can use them for the main four operations (plus, minus, times, and divide) as well as to find a modulus (the remainder after a division) and to increment or decrement a value (see Table 3-1).
Table 3-1. Arithmetic operators

+	Addition	$j + 1
-	Subtraction	$j - 6
*	Multiplication	$j * 11
/	Division	$j / 4
%	Modulus (division remainder)	$j % 9
++	Increment	++$j
--	Decrement	--$j

Table 3-8. *Increment and Decrement Operators*

Example	Label	Outcome
++$a, $a++	Increment	Increment $a by 1
--$a, $a--	Decrement	Decrement $a by 1

These operators can be placed on either side of a variable, and the side on which they are placed provides a slightly different effect. Consider the outcomes of the following examples:
$inv = 15; // Assign integer value 15 to $inv.
$oldInv = $inv--; // Assign $oldInv the value of $inv, then decrement $inv.
$origInv = ++$inv; // Increment $inv, then assign the new $inv value to $origInv.

Assignment operators
These operators are used to assign values to variables. They start with the very simple = and move on to +=, -=, and so on (see Table 3-2). The operator += adds the value on the right side to the variable on the left, instead of totally replacing the value on the left. Thus, if $count starts with the value 5, the statement
$count += 1;
sets $count to 6, just like the more familiar assignment statement:
$count = $count + 1;

PHP's *string operators* (see Table 3-7) provide a convenient way in which to concatenate strings together. There are two such operators, including the concatenation operator (.) and the concatenation assignment operator (.=) discussed in the previous section.

Example	Label	Outcome
$a = "abc"."def";	Concatenation	$a is assigned the string "abcdef"
$a .= "ghijkl";	Concatenation-assignment	$a equals its current value concatenated with "ghijkl"

Here is an example involving string operators:
// $a contains the string value "Spaghetti & Meatballs";
$a = "Spaghetti" . "& Meatballs";
$a .= " are delicious."
// $a contains the value "Spaghetti & Meatballs are delicious."

Example	Label	Outcome
$a = 5	Assignment	$a equals 5
$a += 5	Addition-assignment	$a equals $a plus 5
$a *= 5	Multiplication-assignment	$a equals $a multiplied by 5
$a /= 5	Division-assignment	$a equals $a divided by 5
$a .= 5	Concatenation-assignment	$a equals $a concatenated with 5

Comparison operators
Comparison operators (see Table 3-11), like logical operators, provide a method to direct program flow through an examination of the comparative values of two or more variables.

Table 3-11. *Comparison Operators*

Example	Label	Outcome
$a < $b	Less than	True if $a is less than $b
$a > $b	Greater than	True if $a is greater than $b
$a <= $b	Less than or equal to	True if $a is less than or equal to $b
$a >= $b	Greater than or equal to	True if $a is greater than or equal to $b
($a == 12) ? 5 : -1	Ternary	If $a equals 12, return value is 5; otherwise, return value is –1

Equality Operators

Example	Label	Outcome
$a == $b	Is equal to	True if $a and $b are equivalent
$a != $b	Is not equal to	True if $a is not equal to $b
$a === $b	Is identical to	True if $a and $b are equivalent and $a and $b have the same type

Logical operators

If you haven't used them before, logical operators may at first seem a little daunting. But just think of them the way you would use logic in English. For example, you might say to yourself, "If the time is later than 12 p.m. and earlier than 2 p.m., then have lunch." In PHP, the code for this might look something like the following (using military timing): if ($hour > 12 && $hour < 14) dolunch();

&& *And* $j == 3 **&&** $k == 2
and Low-precedence *and* $j == 3 **and** $k == 2
|| *Or* $j < 5 **||** $j > 10
or Low-precedence *or* $j < 5 **or** $j > 10
! *Not* ! ($j == $k)
xor *Exclusive or* $j **xor** $k

Example	Label	Outcome
$a && $b	AND	True if both $a and $b are true
$a AND $b	AND	True if both $a and $b are true
$a || $b	OR	True if either $a or $b is true
$a OR $b	OR	True if either $a or $b is true
!$a	NOT	True if $a is not true
NOT $a	NOT	True if $a is not true
$a XOR $b	Exclusive OR	True if only $a or only $b is true

Bitwise Operators

Example	Label	Outcome
$a & $b	AND	And together each bit contained in $a and $b
$a | $b	OR	Or together each bit contained in $a and $b
$a ^ $b	XOR	Exclusive—or together each bit contained in $a and $b
~ $b	NOT	Negate each bit in $b
$a << $b	Shift left	$a will receive the value of $b shifted left two bits
$a >> $b	Shift right	$a will receive the value of $b shifted right two bits

1.11 String in PHP

String concatenation

String concatenation uses the period (.) to append one string of characters to another. The simplest way to do this is as follows:

```
echo "You have " . $msgs . " messages.";
```

Assuming that the variable $msgs is set to the value 5, the output from this line of code will be the following:

You have 5 messages.

Just as you can add a value to a numeric variable with the += operator, you can append one string to another using .=, like this:

```
$bulletin .= $newsflash;
```

In this case, if $bulletin contains a news bulletin and $newsflash has a news flash, the command appends the news flash to the news bulletin so that $bulletin now comprises both strings of text.

String types

PHP supports two types of strings that are denoted by the type of quotation mark that you use. If you wish to assign a literal string, preserving the exact contents, you should use the single quotation mark (apostrophe), like this:

```
$info = 'Preface variables with a $ like this: $variable';
```

In this case, every character within the single-quoted string is assigned to $info. If you had used double quotes, PHP would have attempted to evaluate $variable as a variable.

On the other hand, when you want to include the value of a variable inside a string, you do so by using double-quoted strings:
echo "This week $count people have viewed your profile";

String Interpolation
To offer developers the maximum flexibility when working with string values, PHP offers a means for both literal and figurative interpretation. For example, consider the following string:
The $animal jumped over the wall.\n

You might assume that $animal is a variable and that \n is a newline character, and therefore both should be interpreted accordingly. However, what if you want to output the string exactly as it is written, or perhaps you want the newline to be rendered but want the variable to display in its literal form ($animal), or vice versa? All of these variations are possible in PHP, depending on how the strings are enclosed and whether certain key characters are escaped through a predefined sequence.

Double Quotes
Strings enclosed in double quotes are the most commonly used in PHP scripts because they offer the most flexibility. This is because both variables and escape sequences will be parsed accordingly.
Consider the following example:
<?php
$sport = "boxing";
echo "Jason's favorite sport is $sport.";
?>
This example returns the following:
Jason's favorite sport is boxing.

Escape Sequences
Escape sequences are also parsed. Consider this example:
<?php
$output = "This is one line.\nAnd this is another line.";
echo $output;
?>
This returns the following (as viewed from within the browser source):
This is one line.
And this is another line.

Sequence	Description
\n	Newline character
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\$	Dollar sign
\"	Double quote
\[0-7]{1,3}	Octal notation
\x[0-9A-Fa-f]{1,2}	Hexadecimal notation

Single Quotes
Enclosing a string within single quotes is useful when the string should be interpreted exactly as stated. This means that both variables and escape sequences will not be interpreted when the string is parsed. For example, consider the following single-quoted string:
print 'This string will $print exactly as it\'s \n declared.';

This produces the following:
This string will $print exactly as it's \n declared.

Note that the single quote located in *it's* was escaped. Omitting the backslash escape character will result in a syntax error. Consider another example:
print 'This is another string.\\';
This produces the following:

This is another string.\

In this example, the backslash appearing at the conclusion of the string has to be escaped; otherwise, the PHP parser would understand that the trailing single quote was to be escaped. However, if the backslash were to appear anywhere else within the string, there would be no need to escape it.

Curly Braces
While PHP is perfectly capable of interpolating variables representing scalar data types, you'll find that variables representing complex data types such as arrays or objects cannot be so easily parsed when embedded in an echo() or print() string. You can solve this issue by delimiting the variable in curly braces, like this:
echo "The capital of Ohio is {$capitals['ohio']}.";
Personally, I prefer this syntax, as it leaves no doubt as to which parts of the string are static and which are dynamic.

Heredoc
Heredoc syntax offers a convenient means for outputting large amounts of text. Rather than delimiting strings with double or single quotes, two identical identifiers are employed. An example follows:
```
<?php
$website = "http://www.romatermini.it";
echo <<<EXCERPT
<p>Rome's central train station, known as <a href = "$website">Roma Termini</a>,
was built in 1867. Because it had fallen into severe disrepair in the late 20th
century, the government knew that considerable resources were required to
rehabilitate the station prior to the 50-year <i>Giubileo</i>.</p>
EXCERPT;
?>
```

Several points are worth noting regarding this example:

- The opening and closing identifiers (in the case of this example, EXCERPT) must be identical. You can choose any identifier you please, but they must exactly match. The only constraint is that the identifier must consist of solely alphanumeric characters and underscores and must not begin with a digit or an underscore.
- The opening identifier must be preceded with three left-angle brackets (<<<).
- Heredoc syntax follows the same parsing rules as strings enclosed in double quotes. That is, both variables and escape sequences are parsed. The only difference is that double quotes do not need to be escaped.
- The closing identifier must begin at the very beginning of a line. It cannot be preceded with spaces or any other extraneous character. This is a commonly recurring point of confusion among users, so take special care to make sure your heredoc string conforms to this annoying requirement. Furthermore, the presence of any spaces following the opening or closing identifier will produce a syntax error.

Heredoc syntax is particularly useful when you need to manipulate a substantial amount of material but do not want to put up with the hassle of escaping quotes.

Outputting Data to the Browser
Of course, even the simplest of dynamic web sites will output data to the browser, and PHP offers several methods for doing so.

The print() Statement
The print() statement outputs data passed to it . Its prototype looks like this:

int print(argument)
All of the following are plausible print() statements:

```
<?php
print("<p>I love the summertime.</p>");
?>
<?php
$season = "summertime";
print "<p>I love the $season.</p>";
?>
<?php
print "<p>I love the
summertime.</p>";
?>
```

All these statements produce identical output:

I love the summertime.

The print() statement's return value is misleading because it will always return 1 regardless of outcome (the only outcome I've ever experienced using this statement is one in which the desired output is sent to the browser). This differs from the behavior of most other functions in the sense that their return value often serves as an indicator of whether the function executed as intended.

The echo() Statement

Alternatively, you could use the echo() statement for the same purposes as print(). While there are technical differences between echo() and print(), they'll be irrelevant to most readers and therefore aren't discussed here. echo()'s prototype looks like this:

void echo(string argument1 [, ...string argumentN])

To use echo(), just provide it with an argument just as was done with print():

```
echo "I love the summertime.";
```

As you can see from the prototype, echo() is capable of outputting multiple strings. The utility of this particular trait is questionable; using it seems to be a matter of preference more than anything else. Nonetheless, it's available should you feel the need. Here's an example:

```
<?php
$heavyweight = "Lennox Lewis";
$lightweight = "Floyd Mayweather";
echo $heavyweight, " and ", $lightweight, " are great fighters.";
?>
```

This code produces the following:

Lennox Lewis and Floyd Mayweather are great fighters.

Executing the following (in my mind, more concise) variation of the above syntax produces the same output:

```
echo "$heavyweight and $lightweight are great fighters.";
```

If you hail from a programming background using the C language, you might prefer using the printf() statement, introduced next, when outputting a blend of static text and dynamic information.

The printf() Statement

The printf() statement is ideal when you want to output a blend of static text and dynamic information stored within one or several variables. It's ideal for two reasons. First, it neatly separates the static and dynamic data into two distinct sections, allowing for easy maintenance. Second, printf() allows you to wield considerable control over how the dynamic information is rendered to the screen in terms of its type, precision, alignment, and position. Its prototype looks like this:

integer printf(string format [, mixed args])
For example, suppose you wanted to insert a single dynamic integer value into an otherwise static string: printf("Bar inventory: %d bottles of tonic water.", 100);
Executing this command produces the following: Bar inventory: 100 bottles of tonic water.

Table 3-1. *Commonly Used Type Specifiers*

Type	Description
%b	Argument considered an integer; presented as a binary number
%c	Argument considered an integer; presented as a character corresponding to that ASCII value
%d	Argument considered an integer; presented as a signed decimal number
%f	Argument considered a floating-point number; presented as a floating-point number
%o	Argument considered an integer; presented as an octal number
%s	Argument considered a string; presented as a string
%u	Argument considered an integer; presented as an unsigned decimal number
%x	Argument considered an integer; presented as a lowercase hexadecimal number
%X	Argument considered an integer; presented as an uppercase hexadecimal number

So what if you'd like to pass along two values? Just insert two specifiers into the string and make sure you pass two values along as arguments. For example, the following printf() statement passes in an integer and float value:
printf("%d bottles of tonic water cost $%f", 100, 43.20);
Executing this command produces the following:
100 bottles of tonic water cost $43.200000

Because this isn't the ideal monetary representation, when working with decimal values, you can adjust the precision using a precision specifier. An example follows:
printf("$%.2f", 43.2); // $43.20
Still other specifiers exist for tweaking the argument's alignment, padding, sign, and width. Consult the PHP manual for more information.

The sprintf() Statement

The sprintf() statement is functionally identical to printf() except that the output is assigned to a string rather than rendered to the browser. The prototype follows:
string sprintf(string format [, mixed arguments])
An example follows:
$cost = sprintf("$%.2f", 43.2); // $cost = $43.20

The Difference Between the echo and print Commands

So far, you have seen the echo command used in a number of different ways to output text from the server to your browser. In some cases, a string literal has been output. In others, strings have first been concatenated or variables have been evaluated. I've also shown output spread over multiple lines. But there is also an alternative to echo that you can use: print. The two commands are quite similar, but print is a function-like construct that takes a single parameter and has a return value (which is always 1), whereas echo is purely a PHP language construct. Since both commands are constructs, neither requires parentheses. By and large, the echo command will be a tad faster than print in general text output, because it doesn't set a return value. On the other hand, because it isn't implemented like a function, echo cannot be used as part of a more complex expression, whereas print can. Here's an example to output whether the value of a variable is TRUE or FALSE using print, something you could not perform in the same manner with echo, because it would display a Parse error message:
$b ? print "TRUE" : print "FALSE";
The question mark is simply a way of interrogating whether variable $b is TRUE or FALSE. Whichever command is on the left of the following colon is executed if $b is TRUE, whereas the command to the right is executed if $b is FALSE. Generally, though, the examples in this book use echo, and I recommend that you do so as well until you reach such a point in your PHP development that you discover the need for using print.

PHP's Supported Data Types

A *datatype* is the generic name assigned to any data sharing a common set of characteristics. Common data types include Boolean, integer, float, string, and array. PHP has long offered a rich set of data types, discussed next.

Scalar Data Types

Scalar data types are used to represent a single value. Several data types fall under this category, including Boolean, integer, float, and string.

Boolean

The Boolean datatype is named after George Boole (1815–1864), a mathematician who is considered to be one of the founding fathers of information theory. The *Boolean* data type represents truth, supporting only two values: TRUE and FALSE (case insensitive). Alternatively, you can use zero to represent FALSE, and any nonzero value to represent TRUE. A few examples follow:

```
$alive = false; // $alive is false.
$alive = 1; // $alive is true.
$alive = -1; // $alive is true.
$alive = 5; // $alive is true.
$alive = 0; // $alive is false.
```

Integer

An *integer* is representative of any whole number or, in other words, a number that does not contain fractional parts. PHP supports integer values represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal) numbering systems, although it's likely you'll only be concerned with the first of those systems. Several examples follow:

```
42 // decimal
-678900 // decimal
0755 // octal
0xC4E // hexadecimal
```

The maximum supported integer size is platform-dependent, although this is typically positive or negative 2^31 for PHP version 5 and earlier. PHP 6 introduced a 64-bit integer value, meaning PHP will support integer values up to positive or negative 2^63 in size.

Float

Floating-point numbers, also referred to as *floats*, *doubles*, or *real numbers*, allow you to specify numbers that contain fractional parts. Floats are used to represent monetary values, weights, distances, and a whole host of other representations in which a simple integer value won't suffice. PHP's floats can be specified in a variety of ways, several of which are demonstrated here:

```
4.5678
4.0
8.7e4
1.23E+11
```

String

Simply put, a string is a sequence of characters treated as a contiguous group. *Strings* are delimited by single or double quotes, although PHP also supports another delimitation methodology, which is introduced in the later "String Interpolation" section.

The following are all examples of valid strings:

```
"PHP is a great language"
"whoop-de-do"
```

'*9subway\n'
"123$%^789"

PHP treats strings in the same fashion as arrays (see the next section, “Compound Data Types,” for more information about arrays), allowing for specific characters to be accessed via array offset notation.
For example, consider the following string:
$color = "maroon";
You could retrieve a particular character of the string by treating the string as an array, like this:
$parser = $color[2]; // Assigns 'r' to $parser

Converting Between Data Types Using Type Casting
Converting values from one datatype to another is known as *type casting*. A variable can be evaluated once as a different type by casting it to another. This is accomplished by placing the intended type in front of the variable to be cast. A type can be cast by inserting one of the operators shown in Table 3-2 in front of the variable.

Table 3-2. *Type Casting Operators*

Cast Operators	**Conversion**
(array)	Array
(bool) or (boolean)	Boolean
(int) or (integer)	Integer
(object)	Object
(real) or (double) or (float)	Float
(string)	String

1.4 Creating PHP pages

Run: http://localhost/myscripts/first.php

1.5 Rules of PHP syntax

1.6 Integrating HTML with PHP

By default, PHP documents end with the extension *.php*. When a web server encounters this extension in a requested file, it automatically passes it to the PHP processor. Of course, web servers are highly configurable, and some web developers choose to force files ending with *.htm* or *.html* to also get parsed by the PHP processor, usually because they want to hide the fact that they are using PHP.

Your PHP program is responsible for passing back a clean file suitable for display in a web browser. At its very simplest, a PHP document will output only HTML. To prove this, you can take any normal HTML document such as an *index.html* file and save it as *index.php*, and it will display identically to the original.

To trigger the PHP commands, you need to learn a new tag. Here is the first part:

```
<?php
```

The first thing you may notice is that the tag has not been closed. This is because entire sections of PHP can be placed inside this tag, and they finish only when the closing part is encountered, which looks like this:

```
?>
```

A small PHP “Hello World” program might look like Example 3-1.

Example 3-1. Invoking PHP

```
<?php
echo "Hello world";
?>
```

The way you use this tag is quite flexible. Some programmers open the tag at the start of a document and close it right at the end, outputting any HTML directly from PHP commands. Others, however, choose to insert only the smallest possible fragments of PHP within these tags wherever dynamic scripting is required, leaving the rest of the document in standard HTML.

The latter type of programmer generally argues that their style of coding results in faster code, while the former say that the speed increase is so minimal that it doesn’t justify the additional complexity of dropping in and out of PHP many times in a single document.

As you learn more, you will surely discover your preferred style of PHP development, but for the sake of making the examples in this book easier to follow, I have adopted the approach of keeping the number of transfers between PHP and HTML to a minimum— generally only once or twice in a document.

By the way, there is a slight variation to the PHP syntax. If you browse the Internet for PHP examples, you may also encounter code where the opening and closing syntax looks like this:

```
<?
echo "Hello world";
?>
```

Although it’s not as obvious that the PHP parser is being called, this is a valid, alternative syntax that also usually works, but should be discouraged, as it is incompatible with XML and its use is now deprecated (meaning that it is no longer recommended and could be removed in future versions).

If you have only PHP code in a file, you may omit the closing ?>. This can be a good practice, as it will ensure that you have no excess whitespace leaking from your PHP files (especially important when you’re writing object-oriented code).