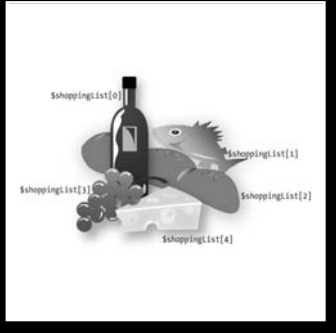
10 INTRODUCING THE BASICS OF PHP



This chapter is a cross between a crash course in PHP and a handy reference. It's aimed at readers who are completely new to PHP or who may have dabbled without really getting to grips with the language. The intention is not to teach you all there is to know but to arm you with sufficient knowledge to dig into Code view to customize Dreamweaver code with confidence. Dreamweaver's automatic code generation does a lot of the hard work for you, but you need to tweak the code to get the best out of it, and when it comes to sending an email from an online form, you have to do everything yourself.

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

- Writing and understanding PHP scripts
- Using variables to represent changing values
- Understanding the difference between single and double quotes
- Organizing related information with arrays
- Creating pages that make decisions for themselves
- Using loops and functions for repetitive work

If you're already comfortable with PHP, just glance at the section headings to see what's covered, because you might find it useful to refer to this chapter if you need to refresh your memory about a particular subject. Then move straight to the next chapter and start coding.

If you're new to PHP, don't try to learn everything at one sitting, or your brain is likely to explode from information overload. On the first reading, look at the headings and maybe the first paragraph or two under each one to get a general overview. Also read the section "Understanding PHP error messages."

Understanding what PHP is for

Back in the early 1990s, web pages consisted of nothing but text. Things didn't stand still for long, and it soon became possible to add images and scrolling text. But even if some things moved around the page in an irritating way, everything on the Web was **static** in the sense that the content was fixed at the time the developer created the page. Genuinely dynamic features began to be added around 1995 with the help of two distinct types of technology: client-side and server-side. The primary distinction between the two is concerned not with *how* dynamic features are generated but with *where*.

At its most basic level, the Internet involves a simple request and response between the user's computer (the **client**) and the remote website (the **server**), as illustrated in Figure 10-1. JavaScript is the most common example of a **client-side** technology. The scripts that control the Spry widgets you used in previous chapters are downloaded with the web page and loaded into the client's memory. When a user clicks a collapsible panel or tabbed interface, all the action takes place in the browser on the client computer.

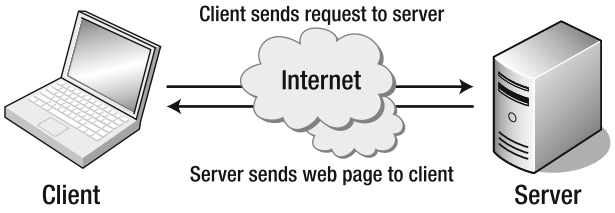


Figure 10-1. The basic relationship on the Internet is between client and server.

With **server-side** technology, on the other hand, all the action takes place on the web server before it's sent to the client. PHP is the most widely used server-side language for web development. In spite of its power, it's relatively easy to learn, and it has the advantage of being cross-platform. In other words, with only a handful of minor differences, it works the same on Windows, Mac OS X, and Linux.

Server-side technology encompasses a much broader range, but I'm concerned here with the way it integrates with the Web.

Increasing user interactivity with server-side technology

With a static web page, everything is fixed at the time of design. All text, links, images, and client-side scripts are hard-coded into the underlying markup. Dynamic web pages built with a server-side language like PHP work in a very different way. Instead of all content being embedded in the underlying code, much of it is automatically generated by the server-side language or drawn from a database. Figure 10-2 illustrates this extra stage in the process.

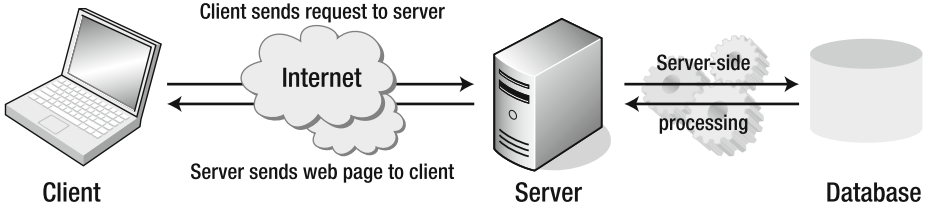


Figure 10-2. Server-side technology involves processing on the server before the web page is sent back to the client.

Generating content dynamically on the server makes it possible to offer the user a much richer variety of content. Perhaps the best known example is `http://www.amazon.com`. The Amazon catalog contains many thousands of items, something that would be impossible if it

were necessary to create and store a separate web page for every item. International news providers, such as the BBC (`http://www.bbc.com/news`) or CNN (`http://www.cnn.com`), are able to update their pages constantly in response to breaking news because most of the content is stored in a database. The web server uses server-side technology to extract the relevant information and build web pages on the fly. Although this involves extra processing, it's normally very quick, and the whole sequence appears seamless to the user.

By the end of this book, you will be able to create web pages that do the same: querying or searching a database, extracting the information, and displaying it as part of your website. You'll also be able to insert new material in the database and update or delete existing material. Admittedly, the projects in the remaining chapters won't be as grandiose as Amazon or a major news site, but they work on the same principles. It will involve getting your hands dirty from time to time with code, but Dreamweaver will do most of the hard work for you.

Writing PHP scripts

The web server processes your PHP code and sends only the results—usually as HTML—to the browser. Because all the action is on the server, you need to tell it that your pages contain PHP code. This involves two simple steps, namely:

- Give every page a PHP filename extension. Do not use anything other than `.php` unless you are told to specifically by your hosting company.
- Enclose all PHP code within PHP tags.

The opening tag is `<?php`, and the closing tag is `?>`. You may come across `<?` as a short version of the opening tag. However, `<?` doesn't work on all servers. Stick with `<?php`, which is guaranteed to work.

Embedding PHP in a web page

When somebody visits your site and requests a PHP page, the server sends it to the PHP engine, which reads the page from top to bottom looking for PHP tags. HTML passes through untouched, but whenever the PHP engine encounters a `<?php` tag, it starts processing your code and continues until it reaches the closing `?>` tag. If the PHP code produces any output, it's inserted at that point. Then, any remaining HTML passes through until another `<?php` tag is encountered.

You can have as many PHP code blocks as you like on a page, but they cannot be nested inside each other.

PHP doesn't always produce direct output for the browser. It may, for instance, check the contents of form input before sending an email message or inserting information into a database. So, some code blocks are placed above or below the main HTML code. You can

also store code in external files. Code that produces direct output, however, always goes where you want the output to be displayed.

A typical PHP page uses some or all of the following elements:

- **Variables** to act as placeholders for unknown or changing values
- **Arrays** to hold multiple values
- **Conditional statements** to make decisions
- **Loops** to perform repetitive tasks
- **Functions** to perform preset tasks

Ending commands with a semicolon

PHP is written as a series of commands or statements. Each **statement** normally tells the PHP engine to perform a particular action, and it must always be followed by a semicolon, like this:

```
<?php
do this;
now do something else;
finally, do that;
?>
```

PHP is not like JavaScript or ActionScript. It won't automatically assume there should be a semicolon at the end of a line if you leave it out. This has a nice side effect: you can spread long statements over several lines and lay out your code for ease of reading. PHP, like HTML, ignores whitespace in code. Instead, it relies on semicolons to indicate where one command ends and the next one begins.

To save space, I won't always surround code samples with PHP tags.

Using variables to represent changing values

A **variable** is simply a name you give to something that may change or that you don't know in advance. The *name* that you give to a variable remains constant, but the *value* stored in the variable can be changed at any time.

Although this concept sounds abstract, you use variables all the time in everyday life. When you meet somebody for the first time, one of the first things you ask is, "What's your name?" It doesn't matter whether the person you've just met is Tom, Dick, or Harry, *name* remains constant, but the value you store in it varies for different people. Similarly, with your bank account, money goes in and out all of the time (mostly out, it seems), but it doesn't matter whether you're scraping the bottom of the barrel or as rich as Croesus, the amount of money in your account is always referred to as the *balance*. In computer terms, *name* and *balance* are variables.

Naming variables

You can choose just about anything you like as the name for a variable, as long as you keep the following rules in mind:

- Variables always begin with $ (a dollar sign).
- The first character after the dollar sign cannot be a number.
- No spaces or punctuation are allowed, except for the underscore (_).
- Variable names are case-sensitive: `$name` and `$Name` are not the same.

A variable's name should give some indication of what it represents: `$name`, `$email`, and `$totalPrice` are good examples. Because you can't use spaces in variable names, it's a good idea to capitalize the first letter of the second or subsequent words when combining them (sometimes called **camel case**). Alternatively, you can use an underscore (for example, `$total_price`).

Don't try to save time by using really short variables. Using `$n`, `$e`, and `$tp` instead of descriptive ones makes code harder to understand. More important, it makes errors more difficult to spot.

> *Although you have considerable freedom in the choice of variable names, you can't use* `$this`*, because it has a special meaning in PHP object-oriented programming. It's also advisable to avoid using any of the keywords listed at* `http://docs.php.net/manual/en/reserved.php`.

Assigning values to variables

Variables get their values from a variety of sources, including the following:

- User input through online forms
- A database
- An external source, such as a news feed or XML file
- The result of a calculation
- Direct inclusion in the PHP code

Wherever the value comes from, it's always assigned in the same way with an equal sign (=), like this:

```
$variable = value;
```

Because it assigns a value, the equal sign is called the **assignment operator**. Although it's an equal sign, get into the habit of thinking of it as meaning "is set to" rather than "equals." This is because, in common with many other programming languages, PHP uses two equal signs (==) to mean "equals" when comparing items—something that catches out a lot of beginners (experienced PHP programmers are not immune to the occasional lapse, either).

Use the following rules when assigning a value to a variable:

- Text must be enclosed in single or double quotes (the distinction between the different types of quotes is explained later in the chapter).
- Numbers should not be in quotes—enclosing a number in quotes turns it into a string.

You can also use a variable to assign a value to another variable, for example:

```
$name = 'David Powers';
$author = $name;  // both $author and $name are now 'David Powers'
```

If the value of `$name` changes subsequently, it doesn't affect the value of `$author`. As this example shows, you don't use quotes around a variable when assigning its value to another. However, as long as you use double quotes, you can embed a variable in text like this:

```
$blurb = "$author has written several best-selling books on PHP.";
```

The value of `$blurb` is now "David Powers has written several best-selling books on PHP." There's a more detailed description on the use of variables with double quotes in "Choosing single or double quotation marks" later in the chapter.

*In common with other computer languages, PHP refers to a block of text as a **string**. This comes from the fact that text is a string of characters. From now on, I'll use the correct terminology.*

Displaying PHP output

The most common ways of displaying dynamic output in the browser are to use `echo` or `print`. The differences between the two are so subtle you can regard them as identical. I prefer `echo`, because it's one fewer letter to type. It's also the style used by Dreamweaver.

Put `echo` (or `print`) in front of a variable, number, or string like this to output it to the browser:

```
$name = 'David';
echo $name;     // displays David
echo 5;         // displays 5
echo 'David';   // displays David
```

You may see scripts that use parentheses with `echo` and `print`, like this:

```
echo('David');  // displays David
print('David'); // displays David
```

The parentheses make no difference. Unless you enjoy typing purely for the sake of it, leave them out.

The important thing to remember about `echo` *and* `print` *is that they work only with variables that contain a single value. You cannot use them to display more complex structures that are capable of storing multiple values.*

Commenting scripts for clarity and debugging

Even if you're an expert programmer, code is not always as immediately understandable as something written in your own human language. That's where comments can be a life-saver. You may understand what the code does five minutes after creating it, but when you come back to maintain it in six months' time—or if you have to maintain someone else's code—you'll be grateful for well-commented code.

In PHP, there are three ways to add comments. The first will be familiar to you if you write JavaScript. Anything on a line following a double slash is regarded as a comment and will not be processed:

```
// Display the name
echo $name;
```

You can also use the hash sign (#) in place of the double slash:

```
# Display the name
echo $name;
```

Either type of comment can go to the side of the code, as long as it doesn't go onto the next line:

```
echo $name;    // This is a comment
echo $name;    #  This is another comment
```

The third style allows you to stretch comments over several lines by sandwiching them between /* and */ (just like CSS comments):

```
/* You might want to use this sort of comment to explain
the whole purpose of a script. Alternatively, it's a
convenient way to disable part of a script temporarily.
*/
```

As the previous example explains, comments serve a dual purpose: they not only allow you to sprinkle your scripts with helpful reminders of what each section of code is for; they can also be used to disable a part of a script temporarily. This is extremely useful when you are trying to trace the cause of an error.

Choosing single or double quotation marks

As I mentioned earlier, strings must always be enclosed in single or double quotes. If all you're concerned about is what ends up on the screen, most of the time it doesn't matter

which quotes you use, but behind the scenes, PHP uses single and double quotes in very different ways:

- Anything between single quotation marks is treated as plain text.
- Anything between double quotation marks is processed.

Quotation marks need to be in matching pairs. If a string begins with a single quote, PHP looks for the next single quote and regards that as the end of the string. Since an apostrophe uses the same character as a single quote, this presents a problem. A similar problem arises when a string in double quotes contains double quotes. The best way to explain this is with a practical example.

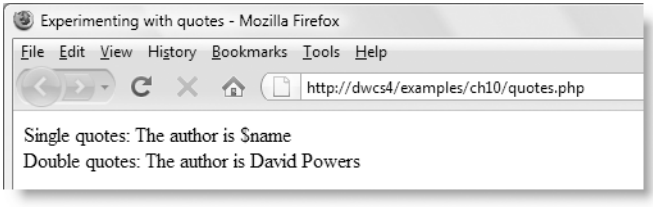
Experimenting with quotes

This simple exercise demonstrates the difference between single and double quotes and what happens when a conflict arises with an apostrophe or double quotes inside a string.

1. Create a new PHP page called `quotes.php` in `workfiles/ch10`. If you just want to look at the finished code, use `quotes.php` in `examples/ch10`.
2. Switch to Code view, and type the following code between the <body> tags:

```
<?php
$name = 'David Powers';
echo 'Single quotes: The author is $name<br />';
echo "Double quotes: The author is $name";
?>
```

3. Save the page, and load it into a browser. As you can see from the following screenshot, `$name` is treated as plain text in the first line but is processed and replaced with its value in the second line, which uses double quotes.

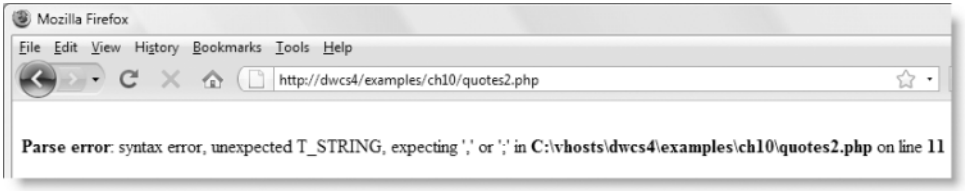


> *To display the output on separate lines, you have to include HTML tags, such as* `<br />`*, because* `echo` *outputs only the values passed to it—nothing more.*

4. Slightly change the text in lines 3 and 4 of the code, as follows:

```
echo 'Single quotes: The author's name is $name<br />';
echo "Double quotes: The author's name is $name";
```

As you type, the change in Dreamweaver syntax coloring should alert you to a problem, but save the page nevertheless, and view it in a browser (it's `quotes2.php` in `examples/ch10`). You should see something like this:



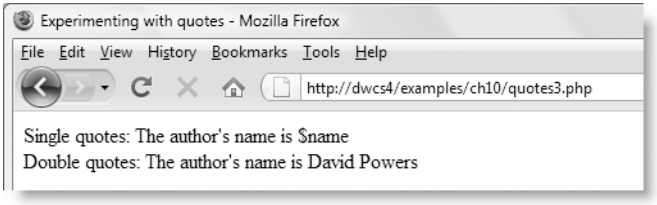
As far as PHP is concerned, an apostrophe and a single quote are the same thing, and quotes must always be in matching pairs. What's happened is that the apostrophe in `author's` has been regarded as the closing quote for the first line, what was intended as the closing quote of the first line becomes a second opening quote, and the apostrophe in the second line becomes the second closing quote. This is all quite different from what was intended—and if you're confused, is it any wonder that PHP is unable to work out what's meant to happen?

> *The meaning of parse error and other error messages is explained in "Understanding PHP error messages" later in this chapter.*

5. To solve the problem, insert a backslash in front of the apostrophe in the first sentence, like this (see `quotes3.php` in `examples/ch10`):

```
echo 'Single quotes: The author\'s name is $name<br />';
```

You should now see the syntax coloring revert to normal. If you view the result in a browser, it should display correctly like this:



Using escape sequences in strings

Using a backslash like this is called an **escape sequence**. It tells PHP to treat a character in a special way. Double quotes within a double-quoted string? You guessed it—escape them with a backslash:

```
echo "Swift's \"Gulliver's Travels\""; // displays the double quotes
```

The next line of code achieves exactly the same thing, but by using a different combination of quotes:

```
echo 'Swift\'s "Gulliver\'s Travels"';
```

When creating strings, the outside pair of quotes must match—any quotes of the same style in the string must be escaped with a backslash. However, putting a backslash in front of the opposite style of quote will result in the backslash being displayed. To see the effect, put a backslash in front of the apostrophe in the doubled-quoted string in the previous exercise.

So, what happens when you want to include a literal backslash? You escape it with a backslash (`\\`).

The backslash (`\\`) and the single quote (`\'`) are the only escape sequences that work in a single-quoted string. Because double quotes are a signal to PHP to process any variables contained within a string, there are many more escape sequences for double-quoted strings. Most of them are to avoid conflicts with characters that are used with variables, but three of them have special meanings: `\n` inserts a newline character, `\r` inserts a carriage return (needed mainly for Windows), and `\t` inserts a tab. Table 10-1 lists the main escape sequences supported by PHP.

Table 10-1. The main PHP escape sequences

Escape sequence	Character represented in double-quoted string
`\"`	Double quote
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash
`\$`	Dollar sign
`\{`	Opening curly brace
`\}`	Closing curly brace
`\[`	Opening square bracket
`\]`	Closing square bracket

The escape sequences listed in Table 10-1, with the exception of \\, work only in double-quoted strings. If you use them in a single-quoted string, they are treated as a literal backslash followed by the second character.

Joining strings together

PHP has a rather unusual way of joining strings. Although many other computer languages use the plus sign (+), PHP uses a period, dot, or full stop (.), like this:

```
$firstName = 'David';
$lastName = 'Powers';
echo $firstName.$lastName; // displays DavidPowers
```

As the comment in the final line of code indicates, when two strings are joined like this, PHP leaves no gap between them. Don't be fooled into thinking that adding a space after the period will do the trick. It won't. You can put as much space on either side of the period as you like; the result will always be the same, because PHP ignores whitespace in code. You must either include a space in one of the strings or insert the space as a string in its own right, like this:

```
echo $firstName.' '.$lastName; // displays David Powers
```

*The period—or **concatenation operator**, to give it its correct name—can be difficult to spot among a lot of other code. Make sure the font size in Code view is large enough to read without straining to see the difference between periods and commas. You can adjust the size in the* Fonts *category of the* Preferences *panel (*Edit *menu on Windows or* Dreamweaver *menu on a Mac).*

Adding to an existing string

Often you need to add more text at the end of an existing string. One way to do it is like this:

```
$author = 'David';
$author = $author.' Powers'; // $author is now 'David Powers'
```

Basically, this concatenates `Powers` (with a leading space) on the end of `$author` and then assigns everything back to the original variable.

Adding something to an existing variable is such a common operation that PHP offers a shorthand way of doing it—with the **combined concatenation operator**. Don't worry about the highfalutin name; it's just a period followed by an equal sign. It works like this:

```
$author = 'David';
$author .= ' Powers'; // $author is now 'David Powers'
```

There should be no space between the period and equal sign. You'll find this shorthand very useful when building the string to form the body of an email message in the next chapter.

How long can a string be? As far as PHP is concerned, there's no limit. In practice, you are likely to be constrained by other factors, such as server memory; but in theory, you could store the whole of War and Peace *in a string variable.*

Using quotes efficiently

Award yourself a bonus point if you spotted a better way of adding the space between `$firstName` and `$lastName` in the preceding example. Yes, that's right . . . Use double quotes, like this:

```
echo "$firstName $lastName"; // displays David Powers
```

Choosing the most efficient combination of quotation marks isn't easy when you first start working with PHP, but it can make your code a lot easier to use. The coding standard for the Zend Framework (`http://framework.zend.com/manual/en/coding-standard.html`) lays down the following rules:

- Use single quotes for literal strings (ones that contain no variables to be processed).
- When a literal string contains apostrophes, use double quotes around the whole string.
- Use double quotes when the string contains variables that need to be processed.

The Zend Framework is a set of advanced PHP scripts written by leading programmers, including members of the core PHP development team. By following its rules, you start out writing scripts the way an expert would. One of the main objectives is to make code efficient and readable, avoiding unnecessary escaping. I frequently see scripts written by inexperienced developers that contain lines like this:

```
echo "<img src=\"me.jpg\" width=\"300\" height=\"216\" alt=\"Me\" />";
```


Compare it with the following line, which wraps the whole literal string in single quotes:

```
echo '<img src="me.jpg" width="300" height="216" alt="Me" />';
```

It doesn't take a genius to work out which version is easier to read, not to mention type.

Special cases: true, false, and null

Although text should be enclosed in quotes, three special cases—`true`, `false`, and `null`—should never be enclosed in quotes unless you want to treat them as strings. The first two mean what you would expect; the last one, `null`, means "nothing" or "no value."

PHP makes decisions on the basis of whether something evaluates to `true` or `false`. Putting quotes around `false` has surprising consequences. The following code:

```
$OK = 'false';
```

does exactly the opposite of what you might expect: it makes `$OK` true! Why? Because the quotes around `false` turn it into a string, and PHP treats strings as `true` (see "The truth

according to PHP" later in this chapter). The other thing to note about `true`, `false`, and `null` is that they are *case-insensitive*. The following examples are all valid:

```
$OK = TRUE;
$OK = tRuE;
$OK = true;
```

Working with numbers

PHP can do a lot with numbers—from simple addition to complex math. Numbers can contain a decimal point or use scientific notation, but they must contain no other punctuation. Never use a comma as a thousands separator. The following examples show the right and wrong ways to assign a large number to a variable:

```
$million = 1000000;    // this is correct
$million = 1,000,000;  // this generates an error
$million = 1e6;        // this is correct
$million = 1e 6;       // this generates an error
```

When using scientific notation, the letter *e* can be uppercase or lowercase and optionally followed by a plus or minus sign. No spaces are permitted.

Negative numbers are preceded by a minus sign (use the hyphen on your keyboard or the minus key on a numeric keypad) with no space before the first digit, for example:

```
$loss = -50000;
```

Performing calculations

The standard arithmetic operators all work the way you would expect, although some of them look slightly different from those you learned at school. For instance, an asterisk (*) is used as the multiplication sign, and a forward slash (/) is used to indicate division.

Table 10-2 shows examples of how the standard arithmetic operators work. To demonstrate their effect, the following variables have been set:

```
$x = 20;
$y = 10;
$z = 3;
```

Table 10-2. Arithmetic operators in PHP

Operation	Operator	Example	Result
Addition	+	$x + $y	30
Subtraction	-	$x - $y	10

Operation	Operator	Example	Result
Multiplication	*	$x * $y	200
Division	/	$x / $y	2
Modulo division	%	$x % $z	2
Increment (add 1)	++	$x++	21
Decrement (subtract 1)	--	$y--	9

You may not be familiar with the modulo operator. This returns the remainder of a division, as follows:

```
26 % 5    // result is 1
26 % 27   // result is 26
10 % 2    // result is 0
```

A quirk with the modulo operator in PHP is that it converts both numbers to integers before performing the calculation. Consequently, if $z is 4.5 in Table 10-2, it gets rounded up to 5, making the result 0, not 2, as you might expect.

A practical use of the modulo operator is to work out whether a number is odd or even. $number % 2 will always produce 0 or 1.

The increment (++) and decrement (--) operators can come either before or after the variable. When they come before the variable, 1 is added to or subtracted from the value before any further calculation is carried out. When they come after the variable, the main calculation is carried out first, and then 1 is either added or subtracted. Since the dollar sign is an integral part of the variable name, the increment and decrement operators go before the dollar sign when used in front:

```
++$x
--$y
```

You can set your own values for $x, $y, and $z in calculation.php in examples/ch10 to test the arithmetic operators in action. The page also demonstrates the difference between putting the increment and decrement operators before and after the variable.

As noted earlier, numbers should not normally be enclosed in quotes, although PHP will usually convert to its numeric equivalent a string that contains only a number or that begins with a number.

Calculations in PHP follow the same rules as standard arithmetic. Table 10-3 summarizes the precedence of arithmetic operators.

Table 10-3. Precedence of arithmetic operators

Precedence	Group	Operators	Rule
Highest	Parentheses	()	Operations contained within parentheses are evaluated first. If these expressions are nested, the innermost is evaluated foremost.
Next	Multiplication and division	* / %	These operators are evaluated next. If an expression contains two or more operators, they are evaluated from left to right.
Lowest	Addition and subtraction	+ -	These are the final operators to be evaluated in an expression. If an expression contains two or more operators, they are evaluated from left to right.

If in doubt, use parentheses all the time to group the parts of a calculation that you want to make sure are performed as a single unit. For example:

```
4 * 5 - 2    // result is 18
4 * (5 - 2)  // result is 12
```

Combining calculations and assignment

You will often want to perform a calculation on a variable and assign the result back to the same variable. PHP offers the same convenient shorthand for arithmetic calculations as for strings. Table 10-4 shows the main combined assignment operators and their use.

Table 10-4. Combined arithmetic assignment operators used in PHP

Operator	Example	Equivalent to
+=	$a += $b	$a = $a + $b
-=	$a -= $b	$a = $a - $b
*=	$a *= $b	$a = $a * $b
/=	$a /= $b	$a = $a / $b
%=	$a %= $b	$a = $a % $b

Don't forget that the plus sign is used in PHP *only as an arithmetic operator*:

- **Addition**: Use += as the combined assignment operator.
- **Strings**: Use .= as the combined assignment operator.

Using arrays to store multiple values

Arrays are an important—and useful—part of PHP. You met one of PHP's built-in arrays, $_POST, in the previous chapter, and you'll work with it a lot more through the rest of this book. Arrays are also used extensively with a database, because you fetch the results of a search in a series of arrays.

An **array** is a special type of variable that stores multiple values rather like a shopping list. Although each item might be different, you can refer to them collectively by a single name. Figure 10-3 demonstrates this concept: the variable $shoppingList refers collectively to all five items—wine, fish, bread, grapes, and cheese.

Figure 10-3. Arrays are variables that store multiple items, just like a shopping list.

Individual items—or **array elements**—are identified by means of a number in square brackets immediately following the variable name. PHP assigns the number automatically, but it's important to note that the numbering always begins at 0. So, the first item in the array, wine, is referred to as $shoppingList[0], not $shoppingList[1]. And although there are five items, the last one (cheese) is $shoppingList[4]. The number is referred to as the array **key** or **index**, and this type of array is called an **indexed array**.

Instead of declaring each array element individually, you can declare the variable name once and assign all the elements by passing them as a comma-separated list to `array()`, like this:

```
$shoppingList = array('wine', 'fish', 'bread', 'grapes', 'cheese');
```

> *The comma must go* outside *the quotes, unlike American typographic practice. For ease of reading, it's recommended to insert a space following each comma, but omitting the space is perfectly valid.*

PHP numbers each array element automatically, so this creates the same array as in Figure 10-3. To add a new element to the end of the array, use a pair of empty square brackets like this:

```
$shoppingList[] = 'coffee';
```

PHP uses the next number available, so this becomes `$shoppingList[5]`.

Using names to identify array elements

Numbers are fine, but it's often more convenient to give array elements meaningful names. For instance, an array containing details of this book might look like this:

```
$book['title'] = 'Essential Guide to Dreamweaver CS4';
$book['author'] = 'David Powers';
$book['publisher'] = 'friends of ED';
```

This type of array is called an **associative array**. Note that the array key is enclosed in quotes (single or double; it doesn't matter). It mustn't contain any spaces or punctuation, except for the underscore.

The shorthand way of creating an associative array uses the => operator (an equal sign followed by a greater-than sign) to assign a value to each array key. The basic structure looks like this:

```
$arrayName = array('key1' => 'element1', 'key2' => 'element2');
```

So, this is the shorthand way to build the $book array:

```
$book = array('title'     => 'Essential Guide to Dreamweaver CS4',
              'author'    => 'David Powers',
              'publisher' => 'friends of ED');
```

It's not essential to align the => operators like this, but it makes code easier to read and maintain.

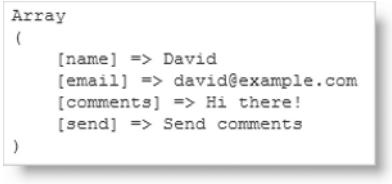
Technically speaking, all arrays in PHP are associative. This means you can use both numbers and strings as array keys in the same array. Don't do it, though, because it can produce unexpected results. It's safer to treat indexed and associative arrays as different types.

Inspecting the contents of an array with print_r()

As you saw in the previous chapter, you can inspect the contents of an array using `print_r()`. This is the code you inserted at the bottom of `feedback.php`:

```
<pre>
<?php if ($_POST) {print_r($_POST);} ?>
</pre>
```

It displays the contents of the array like this:



The `<pre>` tags are simply to make the output more readable. What really matters here is that `print_r()` displays the contents of an array. As explained earlier, `echo` and `print` work only with variables that contain a single value. However, `print_r()` is no good in a live web page; it's used only to inspect the contents of an array for testing purposes. To display the contents of an array in normal circumstances, you need to use a loop. This gives you access to each array element one at a time. Once you get to an element that contains a single value, you can use `echo` or `print` to display its contents. Loops are covered a little later.

Making decisions

Decisions, decisions, decisions . . . Life is full of decisions. So is PHP. They give it the ability to display different output according to circumstances. Decision making in PHP uses **conditional statements**. The most common of these uses `if` and closely follows the structure of normal language. In real life, you may be faced with the following decision (admittedly not very often if you live in Britain):

```
If the weather's hot, I'll go to the beach.
```

In PHP pseudo-code, the same decision looks like this:

```
if (the weather's hot) {
  I'll go to the beach;
}
```

The condition being tested goes inside parentheses, and the resulting action goes between curly braces. This is the basic decision-making pattern:

```
if (condition is true) {
  // code to be executed if condition is true
}
```

Confusion alert: I mentioned earlier that statements must always be followed by a semicolon. This applies only to the statements (or commands) inside the curly braces. Although called a conditional statement*, this decision-making pattern is one of PHP's control structures, and it shouldn't be followed by a semicolon. Think of the semicolon as a command that means "do it." The curly braces surround the command statements and keep them together as a group.*

The code inside the curly braces is executed *only* if the condition is `true`. If it's `false`, PHP ignores everything between the braces and moves on to the next section of code. How PHP determines whether a condition is `true` or `false` is described in the following section.

Sometimes, the `if` statement is all you need, but you often want a default action to be invoked. To do this, use `else`, like this:

```
if (condition is true) {
  // code to be executed if condition is true
} else {
  // default code to run if condition is false
}
```

What if you want more alternatives? One way is to add more `if` statements. PHP will test them, and as long as you finish with `else`, at least one block of code will run. However, it's important to realize that *all* `if` statements will be tested, and the code will be run in every single one where the condition equates to `true`. If you want only one code block to be executed, use `elseif` like this:

```
if (condition is true) {
  // code to be executed if first condition is true
} elseif (second condition is true) {
  // code to be executed if first condition fails
  // but second condition is true
} else {
  // default code to run if both conditions are false
}
```

You can use as many `elseif` clauses in a conditional statement as you like. It's important to note that *only the first one* that equates to `true` will be executed; all others will be ignored, even if they're also `true`. This means you need to build conditional statements in the order of priority that you want them to be evaluated. It's strictly a first-come, first-served hierarchy.

Although `elseif` *is normally written as one word, you can use* `else if` *as separate words.*

The truth according to PHP

Decision making in PHP conditional statements is based on the mutually exclusive **Boolean values**, `true` and `false` (the name comes from a 19th-century mathematician, George Boole, who devised a system of logical operations that subsequently became the basis of much modern-day computing). If the condition equates to `true`, the code within the conditional block is executed. If `false`, it's ignored. Whether a condition is `true` or `false` is determined in one of the following ways:

- A variable set explicitly to `true` or `false`
- A value PHP interprets implicitly as `true` or `false`
- The comparison of two values

Explicit true or false values

This is straightforward. If a variable is assigned the value `true` or `false` and then used in a conditional statement, the decision is based on that value. As explained earlier, `true` and `false` are case-insensitive and must *not* be enclosed in quotes.

Implicit true or false values

PHP regards the following as `false`:

- The case-insensitive keywords `false` and `null`
- Zero as an integer (`0`), a floating-point number (`0.0`), or a string (`'0'` or `"0"`)
- An empty string (single or double quotes with no space between them)
- An empty array
- A SimpleXML object created from empty tags

All other values equate to `true`.

This definition explains why `"false"` *(in quotes) is interpreted by PHP as* `true`*. The value* `-1` *is also treated as* `true` *in PHP.*

How comparisons equate to `true` or `false` is described in the next section.

Using comparisons to make decisions

Conditional statements often depend on the comparison of two values. Is this bigger than that? Are they both the same? If the comparison is `true`, the conditional statement is executed. If not, it's ignored.

To test for equality, PHP uses two equal signs (==) like this:

```
if ($status == 'administrator') {
  // send to admin page
} else {
  // refuse entry to admin area
}
```

Don't use a single equal sign in the first line like this:

```
if ($status = 'administrator') {
```

Doing so will open the admin area of your website to everyone. Why? This automatically sets the value of `$status` *to* `administrator`*; it doesn't compare the two values. To compare values, you must use two equal signs. It's an easy mistake to make, but one with potentially disastrous consequences.*

Size comparisons are performed using the mathematical symbols for less than (<) and greater than (>). Let's say you're checking the size of a file before allowing it to be uploaded to your server. You could set a maximum size of 50KB like this:

```
if ($bytes > 51200) {
  // display error message and abandon upload
} else {
  // continue upload
}
```

If you're wondering why I used 51200 instead of 50000, it's because when measuring computer storage capacity, a kilobyte is traditionally calculated as 1,024 (2^{10}) bytes. International standards organizations insist this should be called a kibibyte (KiB) instead of a kilobyte, but this doesn't seem to have caught on in general usage (`http://en.wikipedia.org/wiki/Kilobyte`).

Comparison operators

These compare two values (known as **operands** because they appear on either side of an operator). If both values pass the test, the result is `true` (or to use the technical expression, it **returns** `true`). Otherwise, it returns `false`. Table 10-5 lists the comparison operators used in PHP.

Table 10-5. PHP comparison operators used for decision-making

Symbol	Name	Use
==	Equality	Returns `true` if both operands have the same value; otherwise, returns `false`.
!=	Inequality	Returns `true` if both operands have different values; otherwise, returns `false`.

Symbol	Name	Use
<>	Inequality	This has the same meaning as !=. It's rarely used in PHP but has been included here for the sake of completeness.
===	Identical	Determines whether both operands are identical. To be considered identical, they must not only have the same value but also be of the same datatype (for example, both floating-point numbers).
!==	Not identical	Determines whether both operands are not identical (according to the same criteria as the previous operator).
>	Greater than	Determines whether the operand on the left is greater in value than the one on the right.
>=	Greater than or equal to	Determines whether the operand on the left is greater in value than or equal to the one on the right.
<	Less than	Determines whether the operand on the left is less in value than the one on the right.
<=	Less than or equal to	Determines whether the operand on the left is less in value than or equal to the one on the right.

Testing more than one condition

Frequently, comparing two values is not enough. PHP allows you to set a series of conditions using **logical operators** to specify whether all, or just some, need to be fulfilled.

All the logical operators in PHP are listed in Table 10-6. **Negation**—testing that the opposite of something is true—is also considered a logical operator, although it applies to individual conditions rather than a series.

Table 10-6. Logical operators used for decision-making in PHP

Symbol	Name	Use
&&	Logical AND	Evaluates to `true` if both operands are `true`. If the left-hand operand evaluates to `false`, the right-hand operand is never tested.
and	Logical AND	Exactly the same as &&, but it takes lower precedence.

Continued

Table 10-6. *Continued*

Symbol	Name	Use
`||`	Logical OR	Evaluates to `true` if either operand is `true`; otherwise, returns `false`. If the left-hand operand returns `true`, the right-hand operand is never tested.
`or`	Logical OR	Exactly the same as `||`, but it takes lower precedence.
`xor`	Exclusive OR	Evaluates to `true` if only one of the two operands returns `true`. If both are `true` or both are `false`, it evaluates to `false`.
!	Negation	Tests whether something is not `true`.

Technically speaking, there is no limit to the number of conditions that can be tested. Each condition is considered in turn from left to right, and as soon as a defining point is reached, no further testing is carried out. When using && or `and`, every condition must be fulfilled, so testing stops as soon as one turns out to be `false`. Similarly, when using `||` or `or`, only one condition needs to be fulfilled, so testing stops as soon as one turns out to be `true`.

```
$a = 10;
$b = 25;
if ($a > 5 && $b > 20) // returns true
if ($a > 5 || $b > 30) // returns true, $b never tested
```

The implication of this is that when you need all conditions to be met, you should design your tests with the condition most likely to return `false` as the first to be evaluated. When you need just one condition to be fulfilled, place the one most likely to return `true` first. If you want a particular set of conditions considered as a group, enclose them in parentheses.

```
if (($a > 5 && $a < 8) || ($b > 20 && $b < 40))
```

Operator precedence is a tricky subject. Stick with && and `||`, rather than `and` and `or`, and use parentheses to group expressions to which you want to give priority. The `xor` operator is rarely used.

Using the switch statement for decision chains

The `switch` statement offers an alternative to `if . . . else` for decision making. The basic structure looks like this:

```
switch(variable being tested) {
  case value1:
    statements to be executed
    break;
  case value2:
```

```
    statements to be executed
    break;
  default:
    statements to be executed
}
```

The case keyword indicates possible matching values for the variable passed to switch(). When a match is made, every subsequent line of code is executed until the break keyword is encountered, at which point the switch statement comes to an end.

You can group several instances of the case keyword together to apply the same block of code to them. For example:

```
switch($httpStatus) {
  case 200:
    $message = 'File OK';
    break;
  case 301:
  case 302:
  case 303:
  case 307:
  case 410:
    $message = 'File moved or does not exist';
    break;
  case 404:
    $message = 'File not found';
    break;
  default:
    $message = 'Other error';
}
```

Dreamweaver uses a switch statement in the GetSQLValueString() function (see Figure 15-1 in Chapter 15), which it inserts into pages that insert or update records in a database.

The main points to note about switch are as follows:

- The expression following the case keyword must be a number or a string.
- You can't use comparison operators with case. So, case > 100: isn't allowed.
- Each block of statements should normally end with break, unless you specifically want to continue executing code within the switch statement.
- If no match is made, any statements following the default keyword will be executed. If no default has been set, the switch statement will exit silently and continue with the next block of code.

Using the conditional (ternary) operator

The **conditional operator** (?:) is a shorthand method of representing a simple conditional statement. Because it uses three operands, it's also called the **ternary operator**. The basic syntax looks like this:

```
condition ? value if true : value if false;
```

What this means is that, if the condition to the left of the question mark is `true`, the value immediately to the right of the question mark is used. However, if the condition evaluates to `false`, the value to the right of the colon is used instead. Here is an example of it in use:

```
$age = 17;
$fareType = $age > 16 ? 'adult' : 'child';
```

The conditional operator can be quite confusing when you first encounter it, so let's break down this example section by section.

The first line sets the value of `$age` to `17`.

The second line sets the value of `$fareType` using the conditional operator. The condition is between the equal sign and the question mark—in other words, `$age > 16`.

If `$age` is greater than `16`, the condition evaluates to `true`, so `$fareType` is set to the value between the question mark and the colon—in other words, `'adult'`. Otherwise, `$fareType` is set to the value to the right of the colon—or `'child'`. The equivalent code using `if . . . else` looks like this:

```
if ($age > 16) {
  $fareType = 'adult';
} else {
  $fareType = 'child';
}
```

The `if . . . else` version is much easier to read, but the conditional operator is more compact, and it's used frequently by Dreamweaver. Most beginners hate this shorthand, but you need to understand how it works if you want to customize Dreamweaver code.

Using loops for repetitive tasks

Loops are huge time-savers, because they perform the same task over and over again yet involve very little code. They're frequently used with arrays and database results. You can step through each item one at a time looking for matches or performing a specific task. Loops frequently contain conditional statements, so although they're very simple in structure, they can be used to create code that processes data in often sophisticated ways.

Loops using while and do . . . while

The simplest type of loop is called a `while` loop. Its basic structure looks like this:

```
while (condition is true) {
  do something
}
```

The following code displays every number from 1 through 100 in a browser (you can see it in action in `while.php` in `examples/ch10`). It begins by setting a variable (`$i`) to 1 and then using the variable as a counter to control the loop, as well as display the current number onscreen.

```
$i = 1;  // set counter
while ($i <= 100) {
  echo "$i<br />";
  $i++; // increase counter by 1
}
```

A variation of the `while` loop uses the keyword `do` and follows this basic pattern:

```
do {
  code to be executed
} while (condition to be tested);
```

The only difference between a `do . . . while` loop and a `while` loop is that the code within the `do` block is executed at least once, even if the condition is never true. The following code (in `dowhile.php` in `examples/ch10`) displays the value of `$i` once, even though it's greater than the maximum expected.

```
$i = 1000;
do {
  echo "$i<br />";
  $i++; // increase counter by 1
} while ($i <= 100);
```

Dreamweaver uses a `do . . . while` loop in its Repeat Region server behavior to loop through the results of a database query (what Dreamweaver calls a **recordset**) and display them on your page.

The danger with creating `while` and `do . . . while` loops yourself is forgetting to set a condition that brings the loop to an end or setting an impossible condition. When this happens, you create an infinite loop that either freezes your computer or causes the browser to crash.

The versatile for loop

The `for` loop is less prone to generating an infinite loop, because you specify in the first line how you want the loop to work. The `for` loop uses the following basic pattern:

```
for (initialize counter; test; increase or decrease the counter) {
  code to be executed
}
```

The three expressions inside the parentheses control the action of the loop (note that they are separated by semicolons, not commas):

- The first expression initializes the counter variable at the start of the loop. You can use any variable you like, but the convention is to use `$i`. When more than one counter is needed, `$j` and `$k` are frequently used. This is the exception to the rule about using descriptive names for variables. The convention of using `$i` (or another single letter) as a counter is so deeply entrenched in programming and mathematic culture, it's unnecessary to use anything else.

- The second expression is a test that determines whether the loop should continue to run. This can be a fixed number, a variable, or an expression that calculates a value.
- The third expression shows the method of stepping through the loop. Most of the time, you will want to go through a loop one step at a time, so using the increment (++) or decrement (--) operator is convenient.

The following code does the same as the previous `while` loop, displaying every number from 1 to 100 (see `forloop.php` in `examples/ch10`):

```
for ($i = 1; $i <= 100; $i++) {
  echo "$i<br />";
}
```

There is nothing stopping you from using bigger steps. For instance, replacing `$i++` with `$i+=10` in this example would display 1, 11, 21, 31, and so on.

Looping through arrays with foreach

The final type of loop in PHP is used exclusively with arrays. It takes two forms, both of which use temporary variables to handle each array element. If you need to do something only with the value of each array element, the `foreach` loop takes the following form:

```
foreach (array_name as temporary_variable) {
  do something with temporary_variable
}
```

> *The* `foreach` *keyword is one word. Inserting a space between* `for` *and* `each` *doesn't work.*

The following example loops through the `$shoppingList` array and displays the name of each item (see `shopping_list.php` in `examples/ch10`):

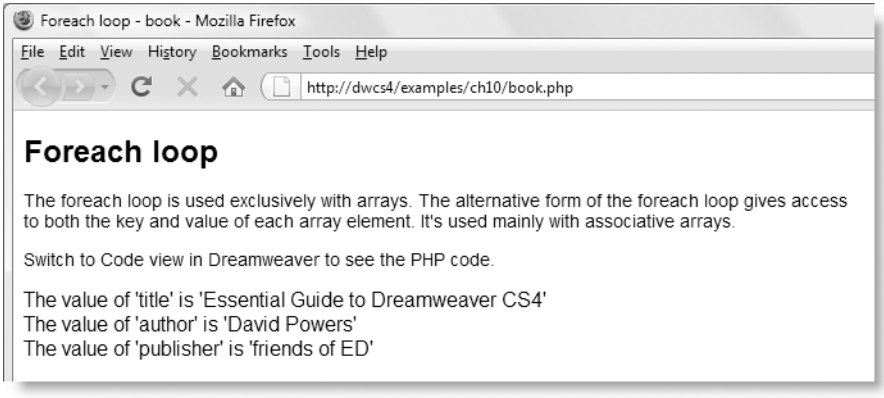
```
$shoppingList = array('wine', 'fish', 'bread', 'grapes', 'cheese');
foreach ($shoppingList as $item) {
  echo $item.'<br />';
}
```

The preceding example accesses only the value of each array element. An alternative form of the `foreach` loop gives access to both the key and the value of each element. It takes this slightly different form:

```
foreach (array_name as key_variable => value_variable) {
  do something with key_variable and value_variable
}
```

This next example uses the $book array from "Using names to identify array elements" earlier in the chapter and incorporates the key and value of each element into a simple string, as shown in the screenshot (see book.php in examples/ch10):

```
foreach ($book as $key => $value) {
  echo "The value of '$key' is '$value'<br />";
}
```



The use of $key and $value as the variables in a foreach loop has also become something of a convention. In this example, it makes sense because the loop is exposing the keys and values of array elements. However, it's a good idea to use descriptive variables where appropriate. For example, when looping through an array of book titles, it's much more meaningful to use something like this:

```
foreach ($titles as $title) {
  echo $title . '<br />';
}
```

Descriptive variables make code much easier to read and understand.

Breaking out of a loop

To bring a loop prematurely to an end when a certain condition is met, insert the break keyword inside a conditional statement. As soon as the script encounters break, it exits the loop. For example, the following loop comes to an end as soon as a banned word is found in $input:

```
foreach ($bannedWords as $word) {
  if (strpos($input, $word) !== false) {
    $reject = true;
    break;
  }
}
```

The `strpos()` function reports the position of a substring inside a longer string, counting from zero. If the presence of a single banned word is sufficient to reject `$input`, there's no point in looping through the whole array, so `break` terminates the loop as soon as the condition is met. (The reason for using `!== false` is to avoid a false negative; a matching word at the beginning of `$input` would return `0`, which PHP treats as `false`.)

To skip an iteration of the loop when a certain condition is met, use the `continue` keyword. Instead of exiting, it returns to the top of the loop and executes the next iteration. In the next example, the loop goes through an array of prices, counting how many items are less than $20.

```
$total = 0;
foreach ($prices as $price) {
  if ($price > 20) {
    continue;
  }
  $total++;
}
```

The `continue` keyword forces the script to abandon the rest of the current iteration if `$price` is higher than `20`, so `$total` isn't incremented. Of course, you could achieve the same result by using the following code:

```
$total = 0;
foreach ($prices as $price) {
  if ($price < 20) {
    $total++;
  }
}
```

But then it wouldn't demonstrate how `continue` works . . .

Using functions for preset tasks

Functions do things . . . lots of things, mind-bogglingly so in PHP. The last time I counted, PHP had nearly 3,000 built-in functions, and more have been added since. Don't worry: you'll only ever need to use a handful, but it's reassuring to know that PHP is a full-featured language capable of industrial-strength applications.

The functions you'll be using in this book do really useful things, such as send email, query a database, format dates, and much, much more. You can identify functions in PHP code, because they're always followed by a pair of parentheses. Sometimes the parentheses are empty. Often, though, the parentheses contain variables, numbers, or strings, like this:

```
$thisYear = date('Y');
```

This calculates the current year and stores it in the variable `$thisYear`. It works by feeding the string `'Y'` to the built-in PHP function `date()`. Placing a value between the parentheses

like this is known as **passing an argument** to a function. The function takes the value in the argument and processes it to produce (or **return**) the result. For instance, if you pass the string `'M'` as an argument to `date()` instead of `'Y'`, it will return the current month as a three-letter abbreviation (for example, Mar, Apr, May). The `date()` function is covered in detail in Chapter 17.

Some functions take more than one argument. When this happens, separate the arguments with commas inside the parentheses, like this:

```
$mailSent = mail($to, $subject, $message);
```

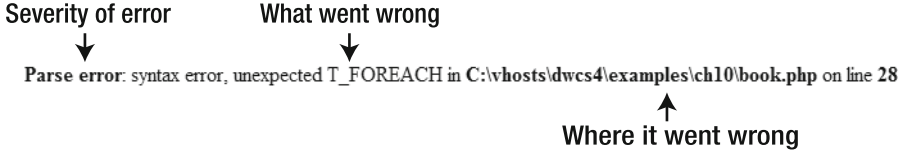
It doesn't take a genius to work out that this sends an email to the address stored in the first argument, with the subject line stored in the second argument and the message stored in the third one. You'll see how this function works in the next chapter.

> *You'll often come across the term* parameter *in place of* argument. *There is a technical difference between the two words, but for all practical purposes, they are interchangeable.*

As if the 3,000-odd built-in functions weren't enough, PHP lets you build your own custom functions. Even if you don't relish the idea of creating your own, throughout this book you'll use some that I have made. You use them in exactly the same way.

Understanding PHP error messages

There's one final thing you need to know about before savoring the delights of PHP: error messages. They're an unfortunate fact of life, but it helps a great deal if you understand what they're trying to tell you. The following illustration shows the structure of a typical error message:



The first thing to realize about PHP error messages is that they report the line where PHP discovered a problem. Most newcomers—quite naturally—assume that's where they have to look for their mistake. Wrong . . .

What PHP is telling you most of the time is that something unexpected has happened. In other words, the mistake frequently lies *before* that point. The preceding error message means that PHP discovered a `foreach` command where there shouldn't have been one. (Error messages always prefix PHP elements with `T_`, which stands for token. Just ignore it.)

Instead of worrying what might be wrong with the `foreach` command (probably nothing), start working backward, looking for anything that might be missing. Usually, it's a semicolon or closing quote. In this example, the error was caused by omitting the semicolon at the end of line 27 in `book.php`. In other words, the error was on the previous line, not the line in the error message.

Sometimes you'll see an error message that tells you it found a problem on or after the last line on the page. That normally means you left out a closing curly brace earlier in the script. Use the Balance Braces tool, as described in the next chapter, to find the cause of the problem.

There are five main categories of error, presented here in descending order of importance:

- **Fatal error**: Any HTML output preceding the error will be displayed, but once the error is encountered—as the name suggests—everything else is killed stone dead. A fatal error is normally caused by referring to a nonexistent file or function.
- **Parse error**: This means there's a mistake in your code, such as mismatched quotes, or a missing semicolon or closing brace. Like a fatal error, it stops the script in its tracks and doesn't even allow any HTML output to be displayed.
- **Warning**: This alerts you to a serious problem, such as a missing include file. (Include files are covered in Chapter 12.) However, the error is not serious enough to prevent the rest of the script from being executed.
- **Deprecated**: This is a new type of error introduced in PHP 5.3 that warns you about code that won't work in future versions. Don't say you haven't been warned.
- **Notice**: This advises you about relatively minor issues, such as the use of a nondeclared variable. Although you can turn off the display of notices, you should always try to eliminate the cause, rather than sweep the issue under the carpet. Any error is a threat to your output.

Hosting companies have different policies about the level of error checking. If error checking is set to a high level and the display of errors is turned off, any mistakes in your code will result in a blank screen. Even if your hosting company has a more relaxed policy, you still don't want mistakes to be displayed for all to see. Test your code thoroughly, and eliminate all errors before deploying it on a live website.

Another type of error, **strict**, was introduced in PHP 5.0, mainly for the benefit of advanced developers. Strict error messages are not displayed by default, but this will change in PHP 6. The official definition of a strict message is that it suggests changes to "ensure the best interoperability and forward compatibility of your code." Quite how this differs from deprecated is unclear, although the implication appears to be that deprecated means a feature will definitely be removed, whereas strict means a change is under consideration.

Chapter review

After that crash course, I hope you're feeling not like a crash victim but invigorated and raring to go. Although you have been bombarded with a mass of information, you'll discover that it's easy to make rapid progress with PHP. In the next chapter, you'll use most of the techniques from this chapter to send user input from an online form to your email inbox. To begin with, you'll probably feel that you're copying code without much comprehension, but I'll explain all the important things along the way, and you should soon find things falling into place.