BONUS CHAPTER 46

Using PHP

IN THIS CHAPTER

- Introduction to PHP
- Basic Functions
- Handling HTML Forms
- Databases
- References

This chapter introduces you to the world of PHP programming, from the point of view of using it as a web scripting language and as a command-line tool. PHP originally stood for *personal home page* because it was a collection of Perl scripts designed to ease the creation of guest books, message boards, and other interactive scripts commonly found on home pages. However, since those early days, it has received two major updates (PHP 3 and PHP 4), plus a substantial revision in PHP 5, which is the version bundled with Ubuntu.

Part of the success of PHP has been its powerful integration with databases—its earliest uses nearly always took advantage of a database back end. In PHP 5, two big new data storage mechanisms were introduced: SQLite, which is a powerful and local database system; and SimpleXML, which is an *application programming interface (API)* designed to make *Extensible Markup Language (XML)* parsing and querying easy. As you will see over time, the PHP developers did a great job: Both SQLite and SimpleXML are easy to learn and use.

NOTE

Many packages for PHP are available from the Ubuntu repositories. The basic package is just called `php5`, but you might also want to add extensions such as `php5-ldap`, `php5-mysql`, or `php5-pgsql`.

Introduction to PHP

In terms of the way it looks, PHP is a cross between Java and Perl, having taken the best aspects of both and merged them successfully into one language. The Java parts include a powerful object-orientation system, the capability to throw program exceptions, and the general style of writing that both languages borrowed from C. Borrowed from Perl is the "it should just work" mentality where ease of use is favored over strictness. As a result, you will find a lot of "there is more than one way to do it" in PHP. This also means that it is possible to accomplish tasks in ways that are less than ideal or without consideration for good security. Many criticize PHP for this, but for simple tasks or if written carefully, it can be a pretty good language and is easy to understand and use, especially for quick website creation.

Entering and Exiting PHP Mode

Unlike PHPs predecessors, you embed your PHP code inside your HTML as opposed to the other way around. Before PHP, many websites had standard HTML pages for most of their content, linking to Perl CGI pages to do back-end processing when needed. With PHP, all your pages are capable of processing and containing HTML. This is a huge factor in PHPs popularity.

Each PHP file is processed by PHP that looks for code to execute. PHP considers all the text it finds to be HTML until it finds one of four things:

- `<?php`
- `<?`
- `<%`
- `<script language="php">`

The first option is the preferred method of entering PHP mode because it is guaranteed to work.

When in PHP mode, you can exit it by using `?>` (for `<?php` and `<?`); `%>` (for `<%`); or `</script>` (for `<script language="php">`). This code example demonstrates entering and exiting PHP mode:

```
In HTML mode
<?php
  echo "In PHP mode";
?>
In HTML mode
In <?php echo "PHP"; ?> mode
```

Variables

All variables in PHP start with a dollar sign (`$`). Unlike many other languages, PHP does not have different types of variable for integers, floating-point numbers, arrays, or

Booleans. They all start with a `$`, and all are interchangeable. As a result, PHP is a weakly typed language, which means you do not declare a variable as containing a specific type of data; you just use it however you want to.

Save the code in Listing 46.1 into a new file called `ubuntu1.php`.

LISTING 46.1 Testing Types in PHP

```
<?php
  $i = 10;
  $j = "10";
  $k = "Hello, world";
  echo $i + $j;
  echo $i + $k;
?>
```

To run that script, bring up a console and browse to where you saved it. Then type this command:

```
matthew@seymour:~$ php ubuntu1.php
```

If PHP is installed correctly, you should see the output `2010`, which is really two things. The `20` is the result of 10 + 10 (`$i` plus `$j`), and the `10` is the result of adding 10 to the text string `Hello, world`. Neither of those operations are really straightforward. Whereas `$i` is set to the number `10`, `$j` is actually set to be the text value `"10"`, which is not the same thing. Adding 10 to 10 gives 20, as you would imagine, but adding 10 to `"10"` (the string) forces PHP to convert `$j` to an integer on-the-fly before adding it.

Running `$i + $k` adds another string to a number, but this time the string is `Hello, world` and not just a number inside a string. PHP still tries to convert it, though, and converting any non-numeric string into a number converts it to 0. So, the second `echo` statement ends up saying `$i + 0`.

As you should have guessed by now, calling `echo` outputs values to the screen. Right now, that prints directly to your console, but internally PHP has a complex output mechanism that enables you to print to a console, send text through Apache to a web browser, send data over a network, and more.

Now that you have seen how PHP handles variables of different types, it is important that you understand the selection of types available to you, as shown in Table 46.1.

TABLE 46.1 PHP Variable Types

Type	Stores
`integer`	Whole numbers; for example, 1, 9, or 324809873
`float`	Fractional numbers; for example, 1.1, 9.09, or 3.141592654
`string`	Characters; for example, “a”, “sfdgh”, or “Ubuntu Unleashed”

Type	Stores
`boolean`	True or false
`array`	Several variables of any type
`resource`	Any external data

The first four can be thought of as simple variables, and the last three as complex variables. Arrays are simply collections of variables. You might have an array of numbers (the ages of all the children in a class); an array of strings (the names of all Wimbledon tennis champions); or even an array of arrays, known as a *multidimensional array*. Arrays are covered in more depth in the next section because they are unique in the way in which they are defined.

Objects are used to define and manipulate a set of variables that belong to a unique entity. Each object has its own personal set of variables, as well as functions that operate on those variables. Objects are commonly used to model real-world things. You might define an object that represents a TV, with variables such as `$CurrentChannel` (probably an integer), `$SupportsHiDef` (a Boolean), and so on.

Of all the complex variables, the easiest to grasp are resources. PHP has many extensions available to it that allow you to connect to databases, manipulate graphics, or even make calls to Java programs. Because they are all external systems, they need to have types of data unique to them that PHP cannot represent using any of the six other data types. So, PHP stores their custom data types in resources—data types that are meaningless to PHP but can be used by the external libraries that created them.

Arrays

Arrays are one of our favorite parts of PHP because the syntax is smart and easy to read and yet manages to be as powerful as you could want. You need to know four pieces of jargon to understand arrays:

- An array is made up of many *elements*.
- Each element has a *key* that defines its place in the array. An array can have only one element with a given key.
- Each element also has a *value*, which is the data associated with the key.
- Each array has a *cursor*, which points to the current key.

The first three are used regularly; the last one less often. The array cursor is covered later in this chapter in the section “Basic Functions,” but we look at the other three now. With PHP, your keys can be almost anything: integers, strings, objects, or other arrays. You can even mix and match the keys so that one key is an array; another is a string, and so on. The one exception to all this is floating-point numbers; you cannot use floating-point numbers as keys in your arrays.

There are two ways of adding values to an array: with the `[]` operator, which is unique to arrays; and with the `array()` pseudo-function. You should use `[]` when you want to add items to an existing array and use `array()` to create a new array.

To sum all this up in code, Listing 46.2 shows a script that creates an array without specifying keys, adds various items to it both without keys and with keys of varying types, does a bit of printing, and then clears the array.

LISTING 46.2 Manipulating Arrays

```
<?php
  $myarr = array(1, 2, 3, 4);

  $myarr[4] = "Hello";
  $myarr[] = "World!";
  $myarr["elephant"] = "Wombat";
  $myarr["foo"] = array(5, 6, 7, 8);

  echo $myarr[2];
  echo $myarr["elephant"];
  echo $myarr["foo"][1];

  $myarr = array();
?>
```


The initial array is created with four elements, to which we assign the values `1`, `2`, `3`, and `4`. Because no keys are specified, PHP automatically assigns keys for us starting at 0 and counting upward—giving keys `0`, `1`, `2`, and `3`. Then we add a new element with the `[]` operator, specifying `4` as the key and `"Hello"` as the value. Next, `[]` is used again to add an element with the value `"World!"` and no key and then again to add an element with the key `"elephant"` and the value `"wombat"`. The line after that demonstrates using a string key with an array value—an array inside an array (a multidimensional array).

The next three lines demonstrate reading back from an array, first using a numeric key, then using a string key, and then using a string key and a numeric key. Remember, the `"foo"` element is an array in itself, so that third reading line retrieves the array and then prints the second element (arrays start at 0, remember). The last line blanks the array by simply using `array()` with no parameters, which creates an array with elements and assigns it to `$myarr`.

The following is an alternative way of using `array()` that allows you to specify keys along with their values:

```
$myarr = array("key1" => "value1", "key2" => "value2",
7 => "foo", 15 => "bar");
```

Which method you choose really depends on whether you want specific keys or want PHP to pick them for you.

Constants

Constants are frequently used in functions that require specific values to be passed in. For example, a popular function is `extract()`, which takes all the values in an array and places them into variables in their own right. You can choose to change the name of the variables as they are extracted using the second parameter—send it a 0 and it overwrites variables with the same names as those being extracted, send it a 1 and it skips variables with the same names, send it a 5 and it prefixes variables only if they exist already, and so on. Of course, no one wants to have to remember a lot of numbers for each function, so you can instead use `EXTR_OVERWRITE` for 0, `EXTR_SKIP` for 1, `EXTR_PREFIX_IF_EXISTS` for 5, and so on, which is much easier.

You can create constants of your own by using the `define()` function. Unlike variables, constants do not start with a dollar sign. Code to define a constant looks like this:

```
<?php
  define("NUM_SQUIRRELS", 10);
  define("PLAYER_NAME", "Jim");
  define("NUM_SQUIRRELS_2", NUM_SQUIRRELS);
  echo NUM_SQUIRRELS_2;
?>
```

That script demonstrates how you can set constants to numbers, strings, or even the value of other constants, although that doesn't really get used much!

References

Using the equal sign (=) copies the value from one variable to another so they both have their own copy of the value. Another option here is to use references, which is where a variable does not have a value of its own; instead, it points to another variable. This enables you to share values and have variables mutually update themselves.

To copy by reference, use the `&` symbol, as follows:

```
<?php
  $a = 10;
  $b = &$a;
  echo $a . "\n";
  echo $b . "\n";

  $a = 20;
  echo $a . "\n";
  echo $b . "\n";

  $b = 30;
  echo $a . "\n";
  echo $b . "\n";
?>
```

If you run that script, you will see that updating `$a` also updates `$b`, but also that updating `$b` updates `$a`.

Comments

Adding short comments to your code is recommended and usually a requirement in larger software houses. In PHP, you have three options for commenting style: `//`, `/* */`, and `#`. The first option (two slashes) instructs PHP to ignore everything until the end of the line. The second (a slash and an asterisk) instructs PHP to ignore everything until it reaches `*/`. The last (a hash symbol) works like `//` and is included because it is common among shell scripting languages.

This code example demonstrates the difference between `//` and `/* */`:

```
<?php
  echo "This is printed!";
  // echo "This is not printed";
  echo "This is printed!";
  /* echo "This is not printed";
  echo "This is not printed either"; */
?>
```

It is generally preferred to use `//` because it is a known quantity. However, it is easy to introduce coding errors with `/* */` by losing track of where a comment starts and ends.

NOTE

Contrary to popular belief, having comments in your PHP script has almost no effect on the speed at which the script executes. What little speed difference exists is wholly removed if you use a code cache.

Escape Sequences

Some characters cannot be typed, and yet you will almost certainly want to use some of them from time to time. For example, you might want to use an ASCII character for a new line, but you cannot type it. Instead, you need to use an escape sequence: `\n`. Similarly, you can print a carriage return character with `\r`. It is important to know both of these because, on the Windows platform, you need to use `\r\n` to get a new line. If you do not plan to run your scripts anywhere else, you need not worry about this.

Going back to the first script you wrote, recall that it printed `2010` because you added 10 + 10 and then 10 + 0. You can rewrite that using escape sequences, like this:

```
<?php
  $i = 10;
  $j = "10";
  $k = "Hello, world";
  echo $i + $j;
```

```
  echo "\n";
  echo $i + $k;
  echo "\n";
?>
```

This time, PHP prints a new line after each of the numbers, making it obvious that the output is `20` and `10` rather than `2010`. Note that the escape sequences must be used in double quotation marks because they will not work in single quotation marks.

Three common escape sequences are `\\`, which means "ignore the backslash"; `\"`, which means "ignore the double quote"; and `\'`, which means "ignore the single quote." This is important when strings include quotation marks inside them. If we had a string such as `"Are you really Conan O'Brien?"`, which has a single quotation mark in it, this code would not work:

```
<?php
  echo 'Are you really Conan O'Brien?';
?>
```

PHP would see the opening quotation mark, read all the way up to the *O* in O'Brien, and then see the quotation mark following the *O* as being the end of the string. The Brien? part would appear to be a fragment of text and would cause an error. You have two options here: You can either surround the string in double quotation marks or escape the single quotation mark with `\'`.

If you choose the escaping route, it will look like this:

```
echo 'Are you really Conan O\'Brien?';
```

Although they are a clean solution for small text strings, be careful with overusing escape sequences. HTML is particularly full of quotation marks, and escaping them can get messy:

```
$mystring = "<img src=\"foo.png\" alt=\"My picture\"
width=\"100\" height=\"200\" />";
```

In that situation, you are better off using single quotation marks to surround the text simply because it is a great deal easier on the eye.

Variable Substitution

PHP allows you to define strings using three methods: single quotation marks, double quotation marks, or heredoc notation. Heredoc is not discussed in this chapter because it is fairly rare compared to the other two methods, but single quotation marks and double quotation marks work identically, with one minor exception: variable substitution.

Consider the following code:

```
<?php
  $age = 25
  echo "You are ";
  echo $age;
?>
```

That is a particularly clumsy way to print a variable as part of a string. Fortunately, if you put a variable inside a string, PHP performs *variable substitution*, replacing the variable with its value. That means we can rewrite the code like this:

```
<?php
  $age = 25
  echo "You are $age";
?>
```

The output is the same. The difference between single quotation marks and double quotation marks is that single-quoted strings do not have their variables substituted. Here's an example:

```
<?php
  $age = 25
  echo "You are $age";
  echo 'You are $age';
?>
```

The first `echo` prints `You are 25`, but the second one prints `You are $age`.

Operators

Now that you have data values to work with, you need some operators to use, too. You have already used `+` to add variables together, but many others in PHP handle arithmetic, comparison, assignment, and other operators. *Operator* is just a fancy word for something that performs an operation, such as addition or subtraction. However, *operand* might be new to you. Consider this operation:

```
$a = $b + c;
```

In this operation, `=` and `+` are operators, and `$a`, `$b`, and `$c` are operands. Along with `+`, you also already know – (subtract), `*` (multiply), and `/` (divide), but Table 46.2 provides some more.

TABLE 46.2 PHP Operators

Operator	What It Does
=	Assigns the right operand to the left operand.
==	Returns true if the left operand is equal to the right operand.
!=	Returns true if the left operand is not equal to the right operand.
===	Returns true if the left operand is identical to the right operand. This is not the same as ==.
!==	Returns true if the left operand is not identical to the right operand. This is not the same as !=.
<	Returns true if the left operand is smaller than the right operand.
>	Returns true if the left operand is greater than the right operand.
<=	Returns true if the left operand is equal to or smaller than the right operand.
&&	Returns true if both the left operand and the right operand are true.
||	Returns true if either the left operand or the right operand is true.
++	Increments the operand by one.
—	Decrements the operand by one.
+=	Increments the left operand by the right operand.
-=	Decrements the left operand by the right operand.
.	Concatenates the left operand and the right operand (joins them together).
%	Divides the left operand by the right operand and returns the remainder.
|	Performs a bitwise OR operation. It returns a number with bits that are set in either the left operand or the right operand.
&	Performs a bitwise AND operation. It returns a number with bits that are set both in the left operand and the right operand.

At least 10 other operators are not listed; to be fair, however, you're unlikely to use them. Even some of the ones in this list are used infrequently (bitwise AND, for example). Having said that, the bitwise OR operator is used regularly because it allows you to combine values.

Here is a code example demonstrating some of the operators:

```
<?php
  $i = 100;
  $i++; // $i is now 101
  $i—; // $i is now 100 again
  $i += 10; // $i is 110
  $i = $i / 2; // $i is 55
  $j = $i; // both $j and $i are 55
  $i = $j % 11; // $i is 0
?>
```

The last line uses modulus, which takes some people a little bit of effort to understand. The result of `$i % 11` is `0` because `$i` is set to `55`, and modulus works by dividing the left operand (`55`) by the right operand (`11`) and returning the remainder. 55 divides by 11 exactly 5 times, and so has the remainder 0.

The concatenation operator, a period (`.`), sounds scarier than it is: It just joins strings together. For example:

```
<?php
echo "Hello, " . "world!";
echo "Hello, world!" . "\n";
?>
```

There are two "special" operators in PHP that are not covered here and yet are used frequently. Before we look at them, though, it is important that you see how the comparison operators (such as `<`, `<=`, and `!=`) are used inside conditional statements.

Conditional Statements

In a *conditional statement*, you instruct PHP to take different actions depending on the outcome of a test. For example, you might want PHP to check whether a variable is greater than 10 and, if so, print a message. This is all done with the `if` statement, which looks like this:

```
if (your condition) {
  // action to take if condition is true
} else {
  // optional action to take otherwise
}
```

The `your condition` part can be filled with any number of conditions you want PHP to evaluate, and this is where the comparison operators come into their own. For example:

```
if ($i > 10) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

PHP looks at the condition and compares `$i` to `10`. If it is greater than 10, it replaces the whole operation with `1`; otherwise, it replaces it with `0`. So, if `$i` is `20`, the result looks like this:

```
if (1) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

In conditional statements, any number other than 0 is considered to be equivalent to the Boolean value `true`; so `1` always evaluates to `true`. There is a similar case for strings: If your string has any characters in it, it evaluates to `true`, with empty strings evaluating to `false`. This is important because you can then use that `1` in another condition through `&&` or `||` operators. For example, if you want to check whether `$i` is greater than `10` but less than `40`, you could write this:

```
if ($i > 10 && $i < 40) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

If you presume that `$i` is set to `50`, the first condition `($i, 10)` is replaced with `1` and the second condition (`$i < 40`) is replaced with `0`. Those two numbers are then used by the `&&` operator, which requires both the left and right operands to be `true`. Whereas `1` is equivalent to `true`, `0` is not, so the `&&` operand is replaced with `0` and the condition fails.

`=`, `==`, `===`, and similar operators are easily confused and often the source of programming errors. The first, a single equal sign, assigns the value of the right operand to the left operand. However, all too often you see code like this:

```
if ($i = 10) {
  echo "The variable is equal to 10!";
} else {
  echo "The variable is not equal to 10";
}
```

That is incorrect. Rather than checking whether `$i` is equal to `10`, it assigns `10` to `$i` and returns `true`. What is needed is `==`, which compares two values for equality. In PHP, this is extended so that there is also `===` (three equal signs), which checks whether two values are identical, more than just equal.

The difference is slight but important: If you have a variable with the string value `"10"` and compare it against the number value of `10`, they are equal. Thus, PHP converts the type and checks the numbers. However, they are not identical. To be considered identical, the two variables must be equal (that is, have the same value) and be of the same data type (that is, both are strings, both are integers, and so on).

NOTE

It is common practice to put function calls in conditional statements rather than direct comparisons. For example:

```
if (do_something()) {
```

If the `do_something()` function returns `true` (or something equivalent to `true`, such as a nonzero number), the conditional statement evaluates to `true`.

Special Operators

The ternary operator and the execution operator work differently from those we have seen so far. The ternary operator is rarely used in PHP, thankfully, because it is really just a condensed conditional statement. Presumably it arose through someone needing to make a code occupy as little space as possible because it certainly does not make PHP code any easier to read.

The ternary operator works like this:

```
$age_description = ($age < 18) ? "child" : "adult";
```

Without explanation, that code is essentially meaningless; however, it expands into the following five lines of code:

```
if ($age < 18) {
  $age_description = "child";
} else {
  $age_description = "adult";
}
```

The ternary operator is so named because it has three operands: a condition to check (`$age < 18` in the previous code), a result if the condition is `true` (`"child"`), and a result if the condition is `false` (`"adult"`). Although we hope you never have to use the ternary operator, it is at least important to know how it works in case you stumble across it.

The other special operator is the execution operator, which is the backtick symbol, `` ` ``. The position of the backtick key varies depending on your keyboard, but it is likely to be just to the left of the 1 key (above Tab). The execution operator executes the program inside the backticks, returning any text the program outputs. For example:

```
<?php
  $i = `ls -l`;
  echo $i;
?>
```

That executes the `ls` program, passing in `-l` (a lowercase *L*) to get the long format, and stores all its output in `$i`. You can make the command as long or as complex as you like, including piping to other programs. You can also use PHP variables inside the command.

Switching

Having multiple `if` statements in one place is ugly, slow, and prone to errors. Consider the code in Listing 46.3.

LISTING 46.3 How Multiple Conditional Statements Lead to Ugly Code

```
<?php
  $cat_age = 3;

  if ($cat_age == 1) {
    echo "Cat age is 1";
  } else {
    if ($cat_age == 2) {
      echo "Cat age is 2";
    } else {
      if ($cat_age == 3) {
        echo "Cat age is 3";
      } else {
        if ($cat_age == 4) {
          echo "Cat age is 4";
        } else {
          echo "Cat age is unknown";
        }
      }
    }
  }
?>
```

Even though it certainly works, it is a poor solution to the problem. Much better is a `switch/case` block, which transforms the previous code into what's shown in Listing 46.4.

LISTING 46.4 Using a `switch/case` Block

```
<?php
  $cat_age = 3;

  switch ($cat_age) {
    case 1:
      echo "Cat age is 1";
      break;
    case 2:
      echo "Cat age is 2";
      break;
    case 3:
      echo "Cat age is 3";
      break;
    case 4:
      echo "Cat age is 4";
      break;
    default:
```

```
      echo "Cat age is unknown";
  }
?>
```

Although it is only slightly shorter, it is a great deal more readable and much easier to maintain. A `switch`/`case` group is made up of a `switch()` statement in which you provide the variable you want to check, followed by numerous `case` statements. Notice the `break` statement at the end of each `case`. Without that, PHP would execute each `case` statement beneath the one it matches. Calling `break` causes PHP to exit the `switch`/`case`. Notice also that there is a default `case` at the end that catches everything that has no matching case.

It is important that you do not use `case default`: but merely `default`:. Also, it is the last `case` label, so it has no need for a `break` statement because PHP exits the `switch`/`case` block there anyway.

Loops

PHP has four ways you can execute a block of code multiple times: `while`, `for`, `foreach`, and `do...while`. Of the four, only `do...while` sees little use; the others are popular, and you will certainly encounter them in other people's scripts.

The most basic loop is the `while` loop, which executes a block of code for as long as a given condition is `true`. So, we can write an infinite loop—a block of code that continues forever—with this PHP:

```
<?php
  $i = 10;
  while ($i >= 10) {
    $i += 1;
    echo $i;
  }
?>
```

The loop block checks whether `$i` is greater or equal to `10` and, if that condition is `true`, adds `1` to `$i` and prints it. Then it goes back to the loop condition again. Because `$i` starts at `10` and we only ever add numbers to it, that loop continues forever. With two small changes, we can make the loop count down from 10 to 0:

```
<?php
  $i = 10;
  while ($i >= 0) {
    $i -= 1;
    echo $i;
  }
?>
```

So, this time we check whether `$i` is greater than or equal to 0 and subtract 1 from it with each loop iteration. `while` loops are typically used when you are unsure of how many times the code needs to loop because while keeps looping until an external factor stops it.

With a `for` loop, you specify precise limits on its operation by giving it a declaration, a condition, and an action. That is, you specify one or more variables that should be set when the loop first runs (the *declaration*), you set the circumstances that will cause the loop to terminate (the *condition*), and you tell PHP what it should change with each loop iteration (the *action*). That last part is what really sets a `for` loop apart from a `while` loop: You usually tell PHP to change the condition variable with each iteration.

We can rewrite the script that counts down from 10 to 0 using a `for` loop:

```
<?php
  for($i = 10; $i >= 0; $i -= 1) {
    echo $i;
  }
?>
```

This time you do not need to specify the initial value for `$i` outside the loop, and neither do you need to change `$i` inside the loop; it is all part of the `for` statement. The actual amount of code is really the same, but for this purpose the for loop is arguably tidier and therefore easier to read. With the `while` loop, the `$i` variable was declared outside the loop and so was not explicitly attached to the loop.

The third loop type is `foreach`, which is specifically for arrays and objects, although it is rarely used for anything other than arrays. A `foreach` loop iterates through each element in an array (or each variable in an object), optionally providing both the key name and the value.

In its simplest form, a `foreach` loop looks like this:

```
<?php
  foreach($myarr as $value) {
    echo $value;
  }
?>
```

This loops through the `$myarr` array you created earlier, placing each value in the `$value` variable. You can modify that so you get the keys as well as the values from the array, like this:

```
<?php
  foreach($myarr as $key => $value) {
    echo "$key is set to $value\n";
  }
?>
```

As you can guess, this time the array keys go in `$key` and the array values go in `$value`. One important characteristic of the `foreach` loop is that it goes from the start of the array to the end and then stops—and by *start* we mean the first item to be added rather than the lowest index number. This script shows this behavior:

```
<?php
  $array = array(6 => "Hello", 4 => "World",
                 2 => "Wom", 0 => "Bat");
  foreach($array as $key => $value) {
    echo "$key is set to $value\n";
  }
?>
```

If you try this script, you will see that `foreach` prints the array in the original order of `6`, `4`, `2`, `0` rather than the numeric order of `0`, `2`, `4`, `6`.

The `do...while loop` works like the `while` loop, with the exception that the condition appears at the end of the code block. This small syntactical difference means a lot, though, because a `do...while` loop is always executed at least once. Consider this script:

```
<?php
  $i = 10;
  do {
    $i -= 1;
    echo $i;
  } while ($i < 10);
?>
```


Without running the script, what do you think it will do? One possibility is that it will do nothing; `$i` is set to `10`, and the condition states that the code must loop only while `$i` is less than `10`. However, a `do...while` loop always executes once, so what happens is that `$i` is set to `10` and PHP enters the loop, decrements `$i`, prints it, and then checks the condition for the first time. At this point, `$i` is indeed less than `10`, so the code loops, `$i` is decremented again, the condition is rechecked, `$i` is decremented again, and so on. This is in fact an infinite loop and so should be avoided!

If you ever want to exit a loop before it has finished, you can use the same `break` statement that you used earlier to exit a `switch/case` block. This becomes more interesting if you find yourself with *nested* loops (loops inside of loops). This is a common situation to be in. For example, you might want to loop through all the rows in a chessboard and, for each row, loop through each column. Calling `break` exits only one loop or `switch/case`, but you can use `break 2` to exit two loops or `switch/cases`, or `break 3` to exit three, and so on.

Including Other Files

Unless you are restricting yourself to the simplest programming ventures, you will want to share code among your scripts at some point. The most basic need for this is to have

a standard header and footer for your website, with only the body content changing. However, you might also find yourself with a small set of custom functions you use frequently, and it would be an incredibly bad move to simply copy and paste the functions into each of the scripts that use them.

The most common way to include other files is with the `include` keyword. Save this script as `include1.php`:

```
<?php
  for($i = 10; $i >= 0; $i -= 1) {
    include "echo_i.php";
  }
?>
```

Then save this script as `echo_i.php`:

```
<?php
  echo $i;
?>
```

If you run `include1.php`, PHP loops from 10 to 0 and includes `echo_i.php` each time. For its part, `echo_i.php` just prints the value of `$i`, which is a crazy way of performing an otherwise simple operation, but it does demonstrate how included files share data. Note that the `include` keyword in `include1.php` is inside a PHP block, but we reopen PHP inside `echo_i.php`. This is important because PHP exits PHP mode for each new file, so you always have a consistent entry point.

Basic Functions

PHP has a vast number of built-in functions that enable you to manipulate strings, connect to databases, and more. There is not room here to cover even 10 percent of the functions; for more detailed coverage of functions, check the “References” section at the end of this chapter.

Strings

Several important functions are used for working with strings, and there are many more less-frequently used ones for which there is not enough space here. We are going to look at the most important here, ordered by difficulty—easiest first!

The easiest function is `strlen()`, which takes a string as its parameter and returns the number of characters in there, like this:

```
<?php
  $ourstring = "  The Quick Brown Box Jumped Over The Lazy Dog  ";
  echo strlen($ourstring);
?>
```

We will be using that same string in subsequent examples to save space. If you execute that script, it outputs `48` because 48 characters are in the string. Note the two spaces on either side of the text, which pad the 44-character phrase up to 48 characters. You can fix that padding with the `trim()` function, which takes a string to trim and returns it with all the whitespace removed from either side. This is a commonly used function because all too often you encounter strings that have an extra new line at the end or a space at the beginning. This cleans it up perfectly.

Using `trim()`, you can turn the 48-character string into a 44-character string (the same thing, without the extra spaces), like this:

```
echo trim($ourstring);
```

Keep in mind that `trim()` returns the trimmed string, so that outputs `"The Quick Brown Box Jumped Over The Lazy Dog"`. You can modify that so `trim()` passes its return value to `strlen()` so that the code trims it and then outputs its trimmed length:

```
echo strlen(trim($ourstring));
```

PHP always executes the innermost functions first, so the previous code takes `$ourstring`, passes it through `trim()`, uses the return value of `trim()` as the parameter for `strlen()`, and prints it.

Of course, everyone knows that boxes do not jump over dogs; the usual phrase is "the quick brown fox." Fortunately, there is a function to fix that problem: `str_replace()`. Note that it has an underscore in it. PHP is inconsistent on this matter, so you really need to memorize the function name.

The `str_replace()` function takes three parameters: the text to search for, the text to replace it with, and the string you want to work with. When working with search functions, people often talk about *needles* and *haystacks*. In this situation, the first parameter is the needle (the thing to find), and the third parameter is the haystack (what you are searching through).

So, you can fix the error and correct *box* to *fox* with this code:

```
echo str_replace("Box", "Fox", $ourstring);
```

There are two little addendums to make here. First, note that we have specified `"Box"` as opposed to `"box"` because that is how it appears in the text. The `str_replace()` function is a *case-sensitive* function, which means it does not consider `"Box"` to be the same as `"box"`. If you want to do a non-case-sensitive search and replace, you can use the `stri_replace()` function, which works in the same way.

The second addendum is that because you are actually changing only one character (*B* to *F*), you need not use a function at all. PHP enables you to read (and change) individual characters of a string by specifying the character position inside braces ({ and }). As with arrays, strings are zero based, which means in the `$ourstring` variable `$ourstring{0}` is `T`, `$ourstring{1}` is `h`, `$ourstring{2}` is `e`, and so on. You could use this instead of `str_replace()`, like this:

```
<?php
  $ourstring = "  The Quick Brown Box Jumped Over The Lazy Dog  ";
  $ourstring{18} = "F";
  echo $ourstring;
?>
```

You can extract part of a string using the `substr()` function, which takes a string as its first parameter, a start position as its second parameter, and an optional length as its third parameter. Optional parameters are common in PHP. If you do not provide them, PHP assumes a default value. In this case, if you specify only the first two parameters, PHP copies from the start position to the end of the string. If you specify the third parameter, PHP copies that many characters from the start. You can write a simple script to print `"Lazy Dog"` by setting the start position to `38`, which, remembering that PHP starts counting string positions from 0, copies from the 39th character to the end of the string:

```
echo substr($ourstring, 38);
```

If you just want to print the word `"Lazy`," you need to use the optional third parameter to specify the length as `4`, like this:

```
echo substr($ourstring, 38, 4);
```

You can also use the `substr()` function with negative second and third parameters. If you specify just parameter one and two and provide a negative number for parameter two, `substr()` counts backward from the end of the string. So, rather than specifying `38` for the second parameter, you can use `-10`, so it takes the last 10 characters from the string. Using a negative second parameter and positive third parameter counts backward from the end string and then uses a forward length. You can print `"Lazy"` by counting 10 characters back from the end and then taking the next 4 characters forward:

```
echo substr($ourstring, -10, 4);
```

Finally, you can use a negative third parameter, too, which also counts back from the end of the string. For example, using `"-4"` as the third parameter means to take everything except the last four characters. Confused yet? This code example should make it clear:

```
echo substr($ourstring, -19, -11);
```

That counts 19 characters backward from the end of the string (which places it at the `O` in `Over`) and then copies everything from there until 11 characters before the end of the string. That prints `Over The`. You could write the same thing using `–19` and `8`, or even `29` and `8`; there is more than one way to do it.

Moving on, the `strpos()` function returns the position of a particular substring inside a string; however, it is most commonly used to answer the question, "Does this string contain a specific substring?" You need to pass it two parameters: a haystack and a needle. (Yes, that's a different order from `str_replace()`.)

In its most basic use, `strpos()` can find the first instance of `Box` in your phrase, like this:

```
echo strpos($ourstring, "Box");
```

This outputs `18` because that is where the `B` in `Box` starts. If `strpos()` cannot find the substring in the parent string, it returns `false` rather than the position. Much more helpful, though, is the ability to check whether a string contains a substring; a first attempt to check whether your string contains the word `The` might look like this:

```
<?php
  $ourstring = "The Quick Brown Box Jumped Over The Lazy Dog";
  if (strpos($ourstring, "The")) {
    echo "Found 'The'!\n";
  } else {
    echo "'The' not found!\n";
  }
?>
```

Note that we have temporarily taken out the leading and trailing whitespace from `$ourstring` and are using the return value of `strpos()` for our conditional statement. This reads, "If the string is found then print a message; if not, print another message." Or does it?

Run the script, and you will see it print the `"not found"` message. The reason for this is that `strpos()` returns `false` if the substring is not found and otherwise returns the position where it starts. If you recall, any nonzero number equates to `true` in PHP, which means that `0` equates to `false`. With that in mind, what is the string index of the first `The` in your phrase? Because PHP's strings are zero based and you no longer have the spaces on either side of the string, the `The` is at position `0`, which your conditional statement evaluates to `false` (hence, the problem).

The solution here is to check for identicality. We know that `0` and `false` are equal, but they are not identical because `0` is an integer, whereas `false` is a Boolean. So, you need to rewrite the conditional statement to see whether the return value from `strpos()` is identical to `false`. If it is, the substring was not found:

```
<?php
  $ourstring = "The Quick Brown Box Jumped Over The Lazy Dog";
  if (strpos($ourstring, "The") !== false) {
    echo "Found 'The'!\n";
  } else {
    echo "'The' not found!\n";
  }
?>
```

Arrays

Working with arrays is no easy task, but PHP makes it easier by providing a selection of functions that can sort, shuffle, intersect, and filter them. As with other functions, there is only space here to choose a selection; this is by no means a definitive reference to PHP's array functions.

The easiest function to use is `array_unique()`, which takes an array as its only parameter and returns the same array with all duplicate values removed. Also in the realm of "so easy you do not need a code example" is the `shuffle()` function, which takes an array as its parameter and randomizes the order of its elements. Note that `shuffle()` does not return the randomized array; it uses your parameter as a reference and scrambles it directly. The last too-easy-to-demonstrate function is `in_array()`, which takes a value as its first parameter and an array as its second and returns `true` if the value is in the array.

With those out of the way, we can focus on the more interesting functions, two of which are `array_keys()` and `array_values()`. They both take an array as their only parameter and return a new array made up of the keys in the array or the values of the array, respectively. The `array_values()` function is an easy way to create a new array of the same data, just without the keys. This is often used if you have numbered your array keys, deleted several elements, and want to reorder it.

The `array_keys()` function creates a new array where the values are the keys from the old array, like this:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  $mykeys = array_keys($myarr);
  foreach($mykeys as $key => $value) {
    echo "$key = $value\n";
  }
?>
```

That prints `"0 = foo"`, `"1 = bar"`, and `"2 = baz"`.

Several functions are used specifically for array sorting, but only two get much use: `asort()` and `ksort()`, the first of which sorts the array by its values and the second of which sorts the array by its keys. Given the array `$myarr` from the previous example, sorting by the values would produce an array with elements in the order `bar/blue`, `baz/green`, and `foo/red`. Sorting by key would give the elements in the order `bar/blue`, `baz/green`, and `foo/red`. As with the `shuffle()` function, both `asort()` and `ksort()` do their work *in place*, meaning they return no value, directly altering the parameter you pass in. For interest's sake, you can also use `arsort()` and `krsort()` for reverse value sorting and reverse key sorting, respectively.

This code example reverse sorts the array by value and then prints it as before:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  arsort($myarr);
```

```
  foreach($myarr as $key => $value) {
    echo "$key = $value\n";
  }
?>
```

Previously when discussing constants, we mentioned the `extract()` function that converts an array into individual variables; now it is time to start using it for real. You need to provide three variables: the array you want to extract, how you want the variables prefixed, and the prefix you want used. Technically, the last two parameters are optional, but practically you should always use them to properly namespace your variables and keep them organized.

The second parameter must be one of the following:

- **`EXTR_OVERWRITE`**—If the variable exists already, overwrites it.
- **`EXTR_SKIP`**—If the variable exists already, skips it and moves on to the next variable.
- **`EXTR_PREFIX_SAME`**—If the variable exists already, uses the prefix specified in the third parameter.
- **`EXTR_PREFIX_ALL`**—Prefixes all variables with the prefix in the third parameter, regardless of whether it exists already.
- **`EXTR_PREFIX_INVALID`**—Uses a prefix only if the variable name would be invalid (for example, starting with a number).
- **`EXTR_IF_EXISTS`**—Extracts only variables that already exist. We have never seen this used.

You can also, optionally, use the bitwise `OR` operator, |, to add in `EXTR_REFS` to have `extract()` use references for the extracted variables. In general use, `EXTR_PREFIX_ALL` is preferred because it guarantees name spacing. `EXTR_REFS` is required only if you need to be able to change the variables and have those changes reflected in the array.

This next script uses `extract()` to convert `$myarr` into individual variables, `$arr_foo`, `$arr_bar`, and `$arr_baz`:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  extract($myarr, EXTR_PREFIX_ALL, 'arr');
?>
```

Note that the array keys are `"foo"`, `"bar"`, and `"baz"` and that the prefix is `"arr"`, but that the final variables will be `$arr_foo`, `$arr_bar`, and `$arr_baz`. PHP inserts an underscore between the prefix and array key.

Files

As you have learned from elsewhere in the book, the UNIX philosophy is that everything is a file. In PHP, this is also the case: A selection of basic file functions is suitable for

opening and manipulating files, but those same functions can also be used for opening and manipulating network sockets. We cover both here.

Two basic read and write functions for files make performing these basic operations easy. They are `file_get_contents()`, which takes a filename as its only parameter and returns the file's contents as a string, and `file_put_contents()`, which takes a filename as its first parameter and the data to write as its second parameter.

Using these two, you can write a script that reads all the text from one file, `filea.txt`, and writes it to another, `fileb.txt`:

```
<?php
  $text = file_get_contents("filea.txt");
  file_put_contents("fileb.txt", $text);
?>
```

Because PHP enables you to treat network sockets like files, you can also use `file_get_contents()` to read text from a website, like this:

```
<?php
  $text = file_get_contents("http://www.slashdot.org");
  file_put_contents("fileb.txt", $text);
?>
```

The problem with using `file_get_contents()` is that it loads the whole file into memory at once; that's not practical if you have large files or even smaller files being accessed by many users. An alternative is to load the file piece by piece, which can be accomplished through the following five functions: `fopen()`, `fclose()`, `fread()`, `fwrite()`, and `feof()`. The `f` in those function names stands for *file*, so they open, close, read from, and write to files and sockets. The last function, `feof()`, returns `true` if the end of the file has been reached.

The `fopen()` function takes a bit of learning to use properly, but on the surface it looks straightforward. Its first parameter is the filename you want to open, which is easy enough. However, the second parameter is where you specify how you want to work with the file, and you should specify one of the following:

- **`r`**—Read-only; it overwrites the file.
- **`r+`**—Reading and writing; it overwrites the file.
- **`w`**—Write-only; it erases the existing contents and overwrites the file.
- **`w+`**—Reading and writing; it erases the existing content and overwrites the file.
- **`a`**—Write-only; it appends to the file.
- **`a+`**—Reading and writing; it appends to the file.
- **`x`**—Write-only, but only if the file does not exist.
- **`a+`**—Reading and writing, but only if the file does not exist.

Optionally, you can also add `b` (for example, `a+b` or `rb`) to switch to binary mode. This is recommended if you want your scripts and the files they write to work smoothly on other platforms.

When you call `fopen()`, you should store the return value. It is a resource known as a *file handle*, which the other file functions all need to do their jobs. The `fread()` function, for example, takes the file handle as its first parameter and the number of bytes to read as its second, returning the content in its return value. The `fclose()` function takes the file handle as its only parameter and frees up the file.

So, you can write a simple loop to open a file, read it piece by piece, print the pieces, and then close the handle:

```
<?php
  $file = fopen("filea.txt", "rb");
  while (!feof($file)) {
    $content = fread($file, 1024);
    echo $content;
  }
  fclose($file);
?>
```

That only leaves the `fwrite()` function, which takes the file handle as its first parameter and the string to write as its second. You can also provide an integer as the third parameter, specifying the number of bytes you want to write of the string, but if you exclude this, `fwrite()` writes the entire string.

If you recall, you can use `a` as the second parameter to `fopen()` to append data to a file. So, you can combine that with `fwrite()` to have a script that adds a line of text to a file each time it is executed:

```
<?php
  $file = fopen("filea.txt", "ab");
  fwrite($file, "Testing\n");
  fclose($file);
?>
```

To make that script a little more exciting, you can stir in a new function, `filesize()`, that takes a filename (not a file handle, but an actual filename string) as its only parameter and returns the file's size in bytes. Using that new function brings the script to this:

```
<?php
  $file = fopen("filea.txt", "ab");
  fwrite($file, "The filesize was" . filesize("filea.txt") . "\n");
  fclose($file);
?>
```

Although PHP automatically cleans up file handles for you, it is still best to use `fclose()` yourself so that you are always in control.

Miscellaneous

Several functions do not fall under the other categories and so are covered here. The first one is `isset()`, which takes one or more variables as its parameters and returns `true` if they have been set. It is important to note that a variable with a value set to something that would be evaluated to `false`—such as `0` or an empty string—still returns `true` from `isset()` because it does not check the value of the variable. It merely checks that it is set; hence, the name.

The `unset()` function also takes one or more variables as its parameters, simply deleting the variable and freeing up the memory. With these two, you can write a script that checks for the existence of a variable and, if it exists, deletes it (see Listing 46.5).

LISTING 46.5 Setting and Unsetting Variables

```
<?php
  $name = "Ildiko";
  if (isset($name)) {
    echo "Name was set to $name\n";
    unset($name);
  } else {
    echo "Name was not set";
  }

  if (isset($name)) {
    echo "Name was set to $name\n";
    unset($name);
  } else {
    echo "Name was not set";
  }
?>
```

That script runs the same `isset()` check twice, but it `unset()`s the variable after the first check. As such, it prints `"Name was set to Ildiko"` and then `"Name was not set"`.

Perhaps the most frequently used function in PHP is `exit`, although purists will tell you that it is in fact a language construct rather than a function. `exit` terminates the processing of the script as soon as it is executed, meaning subsequent lines of code are not executed. That is really all there is to it; it barely deserves an example, but here is one just to make sure:

```
<?php
  exit;
  echo "Exit is a language construct!\n";
?>
```

That script prints nothing because the `exit` comes before the `echo`.

One function we can guarantee you will use a lot is `var_dump()`, which dumps out information about a variable, including its value, to the screen. This is invaluable for arrays because it prints every value and, if one or more of the elements is an array, it prints all the elements from those, and so on. To use this function, just pass it a variable as its only parameter:

```
<?php
  $drones = array("Graham", "Julian", "Nick", "Paul");
  var_dump($drones);
?>
```

The output from that script looks like this:

```
array(4) {
  [0]=>
  string(6) "Graham"
  [1]=>
  string(6) "Julian"
  [2]=>
  string(4) "Nick"
  [3]=>
  string(4) "Paul"
}
```


The `var_dump()` function sees a lot of use as a basic debugging technique because it is the easiest way to print variable data to the screen to verify it.

Finally, we briefly discuss regular expressions; with the emphasis on *briefly* because regular expression syntax is covered elsewhere in this book and the only unique thing relevant to PHP are the functions you use to run the expressions. You have the choice of either *Perl-Compatible Regular Expressions (PCRE)* or POSIX Extended Regular Expressions, but there really is little to choose between them in terms of functionality offered. For this chapter, we use the PCRE expressions because, to the best of our knowledge, they see more use by other PHP programmers.

The main PCRE functions are `preg_match()`, `preg_match_all()`, `preg_replace()`, and `preg_split()`. We start with `preg_match()` because it provides the most basic functionality by returning `true` if one string matches a regular expression. The first parameter to `preg_match()` is the regular expression you want to search for, and the second is the string to match. So, if you want to check whether a string has the word `Best`, `Test`, `rest`, `zest`, or any other word containing `est` preceded by any letter of either case, you could use this PHP code:

```
$result = preg_match("/[A-Za-z]est/", "This is a test");
```

Because the test string matches the expression, `$result` is set to `1` (`true`). If you change the string to a nonmatching result, you get `0` as the return value.

The next function is `preg_match_all()`, which gives you an array of all the matches it found. However, to be most useful, it takes the array to fill with matches as a by-reference parameter and saves its return value for the number of matches that were found.

We suggest you use `preg_match_all()` and `var_dump()` to get a feel for how the function works. This example is a good place to start:

```
<?php
  $string = "This is the best test in the west";
  $result = preg_match_all("/[A-Za-z]est/", $string, $matches);
  var_dump($matches);
?>
```

That outputs the following:

```
array(1) {
  [0]=>
  array(3) {
    [0]=>
    string(4) "best"
    [1]=>
    string(4) "test"
    [2]=>
    string(4) "west"
  }
}
```

If you notice, the `$matches` array is actually multidimensional in that it contains one element, which itself is an array containing all the matches to your regular expression. The reason for this is because your expression has no *subexpressions*, meaning no independent matches using parentheses. If you had subexpressions, each would have its own element in the `$matches` array containing its own array of matches.

Moving on, `preg_replace()` is used to change all substrings that match a regular expression into something else. The basic manner of using this is quite easy: You search for something with a regular expression and provide a replacement for it. However, a more useful variant is *back referencing*, using the match as part of the replacement. For our example, we will imagine you have written a tutorial on PHP but want to process the text so each reference to a function is followed by a link to the PHP manual.

PHP manual page URLs take the form www.php.net/<*somefunc*> (for example, www.php.net/preg_replace). The string we need to match is a function name, which is a string of alphabetic characters, potentially also mixed with numbers and underscores and terminated with two parentheses, `()`. As a replacement, we will use the match we found, surrounded in HTML emphasis tags (`<em></em>`), and then with a link to the relevant PHP manual page. Here is how that looks in code:

```
<?php
  $regex = "/([A-Za-z0-9_]*)\(\)/";
  $replace = "<em>$1</em> (<a href=\"http://www.php.net/$1\">manual</A>)";
  $haystack = "File_get_contents()is easier than using fopen().";
  $result = preg_replace($regex, $replace, $haystack);
  echo $result;
?>
```

The `$1` is our back reference; it will be substituted with the results from the first subexpression. The way we have written the regular expression is very exact. The `[A-Za-z0-9_]*` part, which matches the function name, is marked as a subexpression. After that is `\(\)`, which means the exact symbols (and), not the regular expression meanings of them, which means that `$1` in the replacement will contain `fopen` rather than `fopen()`, which is how it should be. Of course, anything that is not back referenced in the replacement is removed, so we have to put the `()` after the first `$1` (not in the hyperlink) to repair the function name.

After all that work, the output is perfect:

```
<em>File_get_contents()</em> (<a href="http://www.php.net/
file_get_contents">manual</A>) is easier than using <em>fopen()
</em> (<a href="http://www.php.net/fopen">manual</A>).
```


Handling HTML Forms

Given that PHPs primary role is handling web pages, you might wonder why this section has been left so late in the chapter. It is because handling HTML forms is so central to PHP that it is essentially automatic.

Consider this form:

```
<form method="POST" action="thispage.php">
User ID: <input type="text" name="UserID" /><br />
Password: <input type="password" name="Password" /><br />
<input type="submit" />
</form>
```

When a visitor clicks Submit, `thispage.php` is called again, and this time PHP has the variables available to it inside the `$_REQUEST` array. Given that script, if the user enters `12345` and `frosties` as her user ID and password, PHP provides you with `$_REQUEST['UserID']` set to `12345` and `$_REQUEST['Password']` set to `frosties`. Note that it is important that you use HTTP `POST` unless you specifically want `GET`. `POST` enables you to send a great deal more data and stops people from tampering with your URL to try to find holes in your script.

Is that it? Well, almost. That tells you how to retrieve user data, but you should be sure to sanitize it so users do not try to sneak HTML or JavaScript into your database as something you think is innocuous. PHP gives you the `strip_tags()` function for this purpose. It takes a string and returns the same string with all HTML tags removed.

Databases

The ease with which PHP can be used to create dynamic, database-driven websites is the key reason to use it for many people. The stock build of PHP comes with support for MySQL, PostgreSQL, SQLite, Oracle, Microsoft SQL Server, ODBC, plus several other popular databases, so you are sure to find something to work with your data.

If you want to, you can learn all the individual functions for connecting to and manipulating each database PHP supports, but a much smarter, or at least easier, idea is to use `PEAR::DB`, which is an abstraction layer over the databases that PHP supports. You write your code once, and—with the smallest of changes—it works on every database server.

PEAR is the script repository for PHP, and it contains numerous tools and prewritten solutions for common problems. `PEAR::DB` is perhaps the most popular part of the PEAR project, but it is worth checking out the PEAR site to see whether anything else catches your eye.

To get basic use out of `PEAR::DB`, you need to learn how to connect to a database, run a SQL query, and work with the results. This is not a SQL tutorial, so we have assumed you are already familiar with the language. For the sake of this tutorial, we have also assumed you are working with a database called `dentists` and a table called `patients` that contains the following fields:

- **ID**—The primary key, auto-incrementing integer for storing a number unique to each patient
- **Name**—A varchar(255) field for storing a patient name
- **Age**—Integer
- **Sex**—1 for male, 2 for female
- **Occupation**—A varchar(255) field for storing a patient occupation

Also for the sake of this tutorial, we use a database server on IP address `10.0.0.1`, running MySQL, with username `ubuntu` and password `alm65z`. You need to replace these details with your own; use `localhost` for connecting to the local server.

The first step to using `PEAR::DB` is to include the standard `PEAR::DB` file, `DB.php`. Your PHP will be configured to look inside the PEAR directory for `include()` files, so you do not need to provide any directory information.

PEAR::DB is object oriented, and you specify your connection details at the same time as you create the initial DB object. This is done using a URL-like system that specifies the database server type, username, password, server, and database name all in one. After you have specified the database server here, everything else is abstracted, meaning you only need to change the connection line to port your code to another database server.

This first script connects to our server and prints a status message (see Listing 46.6).

LISTING 46.6 Connecting to a Database Through PEAR::DB

```
<?php
  include("DB.php");
  $dsn = "mysql://ubuntu:alm65z@10.0.0.1/dentists";
  $conn = DB::connect($dsn);
  if (DB::isError($conn)) {
    echo $conn->getMessage() . "\n";
  } else {
    echo "Connected successfully!\n";

  }
?>
```


You should be able to see how the connection string breaks down. It is server name first, then a username and password separated by a colon, then an @ symbol followed by the IP address to which to connect, and then a slash and the database name. Notice how the call to connect is DB::connect(), which calls PEAR::DB directly and returns a database connection object for storage in $conn. The variable name $dsn was used for the connection details because it is a common acronym standing for data source name.

If DB::connect() successfully connects to a server, it returns a database object we can use to run SQL queries. If not, we get an error returned that we can query using functions such as getMessage(). In the previous script, we print the error message if we fail to connect, but we also print a message if we succeed. Next, we change that so we run an SQL query if we have a connection.

Running SQL queries is done through the query() function of our database connection, passing in the SQL we want to execute. This then returns a query result that can be used to get the data. This query result can be thought of as a multidimensional array because it has many rows of data, each with many columns of attributes. This is extracted using the fetchInto() function, which loops through the query result converting one row of data into an array that it sends back as its return value. You need to pass in two parameters to fetchInto() specifying where the data should be stored and how you want it stored. Unless you have unusual needs, specifying DB_FETCHMODE_ASSOC for the second parameter is a smart move.

Listing 46.7 shows the new script.

LISTING 46.7 Running a Query Through `PEAR::DB`

```
<?php
  include("DB.php");
  $dsn = "mysql://ubuntu:alm65z@10.0.0.1/dentists";
  $conn = DB::connect($dsn);
  if (DB::isError($conn)) {
    echo $conn->getMessage() . "\n";
  } else {
    echo "Connected successfully!\n";
    $result = $conn->query("SELECT ID, Name FROM patients;");
    while ($result->fetchInto($row, DB_FETCHMODE_ASSOC)) {
      extract($row, EXTR_PREFIX_ALL, 'pat');
      echo "$pat_ID is $pat_Name\n";
    }
  }
?>
```

The first half is identical to the previous script, with all the new action happening if we get a successful connection.

Going along with the saying "never leave to PHP what you can clean up yourself," the current script has problems. We do not clean up the query result, and we do not close the database connection. If this code were being used in a longer script that ran for several minutes, this would be a huge waste of resources. Fortunately, we can free up the memory associated with these two by calling `$result->free()` and `$conn->disconnect()`. If we add those two function calls to the end of the script, it is complete.

References

- **https://secure.php.net/**—The best place to look for information is the PHP online manual. It is comprehensive, well written, and updated regularly.
- **www.phpbuilder.com**—A large PHP scripts and tutorials site where you can learn new techniques and also chat with other PHP developers.
- **www.zend.com**—The home page of a company founded by two of the key developers of PHP. Zend develops and sells proprietary software, including a powerful IDE and a code cache, to aid PHP developers.
- **http://pear.php.net/**—The home of the PEAR project contains a large collection of software you can download and try, and it has thorough documentation for it all.
- **www.phparch.com/**—There are quite a few good PHP magazines around, but *PHP Architect* probably leads the way. It posts some of its articles online for free, and its forums are good, too.

- Quality books on PHP abound, and you are certainly spoiled for choice. For beginning developers, the best available is *PHP and MySQL Web Development* (Sams Publishing), ISBN: 0-672-32916-6. For a concise, to-the-point-book covering all aspects of PHP, check out *PHP in a Nutshell* (O'Reilly), ISBN: 0-596-10067-1. Finally, for advanced developers, you can consult *Advanced PHP Programming* (Sams Publishing), ISBN: 0-672-32561-6.