CHAPTER 6

Working with Strings

Considered pretty much a staple of any programming language, the ability to work with, maneuver, and ultimately control strings is an important part of a daily programming routine. PHP, unlike other languages, has no trouble using data typing to handle strings. Thanks to the way PHP is set up—that is to say, that when a value is assigned to a variable, the variable automatically typecasts itself—working with strings has never been easier. PHP 5 has not done much to improve upon, or modify, the way strings are handled in PHP itself, but it has provided you with new and improved class functionality so you can more readily create tools to help support PHP's somewhat clunky string function naming conventions.

This chapter's focus will be threefold. First, we will offer a bit of a refresher course on how PHP's versatile string functions can be quite an asset to the aspiring developer. Second, we will display some real-world examples not just on how using strings is both important and practical while deploying applications but also on how to use them to your best advantage. Last, we will use what we have learned and apply it towards a fully functional, working example of a possible string dilemma (see Table 6-1).

Table 6-1. *PHP 5 String Functions*

Function	Description
`substr_count()`	Counts the number of substring occurrences
`strstr()`	Finds the first occurrence of a string
`strchr()`	Can be used as an alias of `strstr()`
`strrchr()`	Finds the last occurrence of a character in a string
`stristr()`	Performs the same functionality as `strstr()` but is case-insensitive
`substr_replace()`	Replaces text within a portion of a string
`strpos()`	Finds the position of the first occurrence of a string
`substr()`	Returns a piece of a string
`strlen()`	Returns the length of a string
`strtok()`	Splits a string into smaller tokens
`explode()`	Returns an array of substrings of a target string, delimited by a specific character
`implode()`	Takes an array of items and puts the items together, delimited by a specific character

Continued

Table 6-1. *Continued*

Function	Description
`join()`	Acts as an alias to `implode()`
`str_split()`	Converts a string to an array
`strtoupper()`	Converts an entire string to uppercase characters
`strtolower()`	Converts an entire string to lowercase characters
`ucfirst()`	Changes a given string's first character into uppercase
`ucwords()`	Changes a given string's first character of each word into uppercase
`trim()`	Strips whitespace from the beginning and end of a string
`chop()`	Acts as an alias for `rtrim()`
`rtrim()`	Strips whitespace from the end of a string only
`ltrim()`	Strips whitespace from the beginning of a string only
`strcmp()`	Performs a string comparison

Manipulating Substrings

One of the common occurrences developers will come across is the problem of deducing what is needed, where certain aspects are, or even what order is necessary from a string. *Substrings* make up part of a full string. Since manipulating different portions of a string is a common task while building applications, PHP has granted you the use of some rather powerful functions. None of the string functions need to be included, and the basic string functions are prepackaged with the PHP 5 release, thus removing the problem of including them as extensions or packaging them with libraries. Table 6-2 lists the functions that prove useful (and sometimes quite invaluable) when working with substrings.

Table 6-2. *PHP 5 Substring Functions*

Function	Description
`substr_count()`	Counts the number of substring occurrences
`strstr()`	Finds the first occurrence of a string
`strchr()`	Can be used as an alias of `strstr()`
`strrchr()`	Finds the last occurrence of a character in a string
`stristr()`	Performs the same functionality as `strstr()` but is case-insensitive
`substr_replace()`	Replaces text within a portion of a string
`strops()`	Finds the position of the first occurrence of a string
`substr()`	Returns a piece of a string

6-1. Testing for Substrings

The first thing you might do when working with substrings is test a string for occurrences of a specific substring. You can put this sort of functionality to use in almost any kind of application. The most obvious usage of this sort of algorithm would be when building a search engine. Depending on what exactly it is you are building the search engine to do, testing for substrings manually with PHP may not always be the most efficient plan. For instance, if you were searching something in a database, more than likely it would be beneficial to do a query comparison using the built-in SQL engine to find whether a substring exists, but certainly sometimes you will not have the luxury of letting another system's workhorse do the brunt of the task for you.

Obviously, if you were going to do a search within a given block of text using PHP, it might be a smart move to check whether any instances of the search query appear within the text; there is no point in proceeding with your search algorithm if you find no instance of the search term. Thankfully, PHP has a nicely built function, `substr_count()`, that is perfect for the task. The standard definition for the `substr_count()` function is as follows:

```
int substr_count ( string haystack, string needle )
```

That is to say, you provide the block of text you want to search as the first parameter for the function (`string haystack`), and then you provide the substring you want to obtain as the second parameter (`string needle`). The function will then provide you with a return value of the number of occurrences of the `needle` parameter from the `haystack` search block.

The following block of code is basically a mundane search engine.

The Code

```
<?php
  function searchtext ($haystack, $needle){
    //First, let's deduce whether there is any point in going on with our little
    //string hunting charade.
    if (substr_count ($haystack, $needle) == 0){
      echo "No instances were found of this search query";
    } else {
      //Now, we will go through the haystack, find out the
      //different positions that the string occurs at, and then output them.

      //We will start searching at the beginning.
      $startpos = 0;
      //And we will set a flag to stop searching once there are no more matches.
      $lookagain = true;

      //Now, we search while there are still matches.
      while ($lookagain){
        if ($pos = strpos ($haystack, $needle, $startpos)){
          echo "The search term \"$needle\" was found at position: ➥
$pos<br /><br />";
          //We increment the position we are searching in order to continue.
          $startpos = $pos + 1;
        } else {
```

```
          //If there are no more matches, then we want to break out of the loop.
          $lookagain = false;
        }
      }

      echo "Your search for \"$needle\" within \"$haystack\" ➥
returned a total of \"" . substr_count ($haystack, $needle) . "\" matches.";
    }
  }

  searchtext ("Hello World!","o");
?>
```

```
The search term "o" was found at position: 4
The search term "o" was found at position: 7
Your search for "o" within "Hello World!" returned a total of "2" matches.
```

How It Works

The previous function is a bare-bones search engine that is primed to take a block of text and then output not only whether there are any matches within the block of text but exactly where those matches occur. For ease of use and cleanliness, the total number of matches found is displayed at the bottom. Take a moment to review the function, and we will discuss how exactly it does what it does by using the built-in PHP string functions.

By using `substr_count()` to make sure there are no instances of the substring, you let the user know that there were no matches if, indeed, you find no matches.

The next matter that this function attends to is the meat and potatoes of this function; it actually loops through the search string and outputs the position of all instances of the substring.

```
//Now, we search while there are still matches.
while ($lookagain){
  if ($pos = strpos ($haystack, $needle, $startpos)){
    echo "The search term \"$needle\" was found at position:  $pos<br /><br />";
    //We increment the position we are searching in order to continue.
    $startpos = $pos + 1;
   } else {
    //If there are no more matches, then we want to break out of the loop.
   $lookagain = false;
  }
}
```

By using the `strpos()` function, which outputs the position within the string of the given substring, you can go through the entire block of search text, outputting where exactly the search term falls.

Now, this is a scaled-down idea for a search engine, but by using these basic concepts you can see the power you get by taking advantage of built-in string functions.

6-2. Counting the Occurrences of a Substring

As discussed in the previous recipe, counting the number of occurrences of a substring within a search string is a simple process. PHP has included the function `substr_count()` that does the work for you. Naturally, there are more manual ways of doing this (consider adding a counter to the previous script as it loops), but since this is a stable function that has been around for some time, there is no need to reinvent the wheel. Thus, when it comes time in the previous function to output a total search tally, you simply call the `substr_count()` function, as in the following example.

The Code

```
<?php
  //Let's find the number of o's in our string.
  $counter = substr_count ("Hello World!","o");
  echo "There are " . $counter . " instance (s) of \"o\" in Hello World!.";
?>
```

```
There are 2 instances of "o" in Hello World!.
```

How It Works

As you can see, by providing the `substr_count()` function with a `needle` substring and a `haystack` string to search through, you can determine quickly and easily how many instances of a given substring exist within the supplied string.

6-3. Accessing Substrings

A fairly common day-to-day activity you might be required to perform while programming would be to access, and then do something with, a certain substring. PHP has a rather versatile and powerful function that will allow you to do just that. Aptly named `substr()`, this function will allow you to access any part of a string using the concise method detailed here:

```
string substr ( string string, int start [, int length] )
```

Basically, the function `substr()` takes as arguments the string you want to divide, the position you want to begin dividing from, and (optionally) the end point at which you want to stop dividing. The function then returns a nicely packaged (in the form of a string) substring ready for your use and (potentially) abuse.

As you can imagine, the `substr()` function is a handy tool. You can use it in many real-world applications, and it is a great help with everything from validation to properly formatted output. Imagine, for instance, that you have been tasked with building a content management system (CMS) that takes in information from a client and then outputs it onto the home page of a website. Sound easy? Now imagine that the design for the website has been built so the

height of the block where the text is supposed to be output is big enough to handle only 300 characters of text. If the amount of text outputted exceeds that, the site could potentially "break," causing the entire design to look flawed; this is not good considering this is an important client.

In addition, consider that the client is not sure at all how many characters to use and even if you had informed them to use only 300, there is still a real chance they would try their luck anyway. How then can you guarantee that the design will not break? Well, this sounds like a lovely test for our good friend Mr. `substr()` and his buddy Ms. `strlen()`.

The Code

```
<?php

  $theclientstext = "Hello, how are you today? I am fine!";

  if (strlen ($theclientstext) >= 30){
    echo substr ($theclientstext,0,29);
  } else {
    echo $theclientstext;
  }

?>
```

```
Hello, how are you today? I a
```

How It Works

The first thing this block of code does is check to make sure the text provided by the client is within the length you need it to be:

```
if (strlen ($theclientstext) >= 30){
```

If it happens to fall outside the range of acceptable length, you then use the lovely `substr()` function to echo only the portion of the text that is deemed acceptable. If the client has entered a proper block of text, then the system merely outputs the text that was entered, and no one is the wiser.

By using the function `substr()`, you have averted a potential disaster. People browsing the site will see nothing but a slightly concatenated set of verbiage, so the site's integrity remains sound. This sort of rock-solid validation and programming can save business relationships, as clients are seldom fond of having their site appear "broken" to potential customers or intrigued individuals.

6-4. Using Substring Alternatives

You can consider the `substr()` function as something of a jack of all trades. It can get you whatever you are looking for in a string with the greatest of ease. Sometimes, however, it may not be necessary to go to the trouble of using such a versatile function. Sometimes it is just

easier to use a more specialized function to accomplish a task; fortunately, PHP has a fairly decent selection of such methods.

For instance, if you are interested in using only the first instance of a substring, you can use the function `strstr()` (or `strchr()`, which is merely an alias of the former), which takes a block of text and a search value as arguments (the proverbial haystack and needle). If you are not concerned with the case of the subjects, the function `stristr()` will take care of any problems you may have. Alternatively, you may be interested in obtaining the last instance of a substring within a block of text. You can accomplish this particular maneuver with the `strrchr()` function, also available from PHP. The prototypes for `strstr()` and `stristr()` are as follows:

```
string strstr ( string haystack, string needle )
string stristr ( string haystack, string needle )
```

The Code

```
<?php
  $url = "www.apress.com";
  $domain = strstr ($url, ".");
  echo $domain;
?>
```

```
.apress.com
```

How It Works

In this example in which you are attempting to find the domain name of the current string, the `strstr()` function finds the first instance of the dot (.) character and then outputs everything starting with the first instance of the dot. In this case, the output would be ".apress.com".

6-5. Replacing Substrings

How often do you find yourself using the search-and-replace function within your word processor or text editor? The search-and-replace functionality found within such applications is a testament to how much easier it is to do things using a computer rather than manually. (How helpful would it be to have such a function while, say, skimming the local newspaper for classified ads?) Thankfully, PHP has heard the cries of the people and has provided a function called `substr_replace()` that can quickly turn the tedious task of scanning and editing a large block of text into a lofty walk through the park where you let PHP do your task for you while you grab yourself another coffee (preferably a white-chocolate mocha…). The `substr_replace()` function is defined as follows:

```
string substr_replace ( string str, string replacmnt, int start [, int len] )
```

The function `substr_replace()` is a powerful and versatile piece of code. While you can access the core functionality of it easily and painlessly, the depth and customization you can accomplish through the function is rather daunting. Let's start with the basics. If you want to simply make a replacement to the substring, and you want to start from the beginning and

replace the entire instance (say, by changing the ever-so-clichéd "Hello World!" into the more "l33t" phrase "H3110 W0r1d!" and hence proving your "l33t" status), you could simply invoke the `substr_replace()` function as shown in the following example.

The Code

```
<?php
  //By supplying no start or length arguments,
  //the string will be added to the beginning.
  $mystring = substr_replace("Hello World", "H3110 W0r1d!", 0, 0);
  echo $mystring . "<br />"; //Echoes H3110 W0r1d!Hello World

  //Where if we did this:
  $mystring = substr_replace("Hello World", "0 w0", 4, 4);
  echo $mystring; //Echoes Hell0 w0rld.

?>
```

```
H3110 W0r1d!Hello World
Hell0 w0rld
```

How It Works

This is not all that useful, is it? Happily, the `substr_replace()` function can do much more than that. By changing the third argument (the start position) and the last argument (which is optional and represents a length of characters that you want to replace), you can perform some pretty powerful and dynamic operations. Let's say you simply want to add the catchy "H3110 W0r1d!" phrase to the front of a string. You could perform this operation by simply using the `substr_replace()` function as follows:

```
<?php
 substr_replace("Hello World", "H3110 W0r1d!", 0, 0);
?>
```

You can also do some pretty fancy operations by changing the start and length arguments of the function from positive to negative values. By changing the start value to a negative number, you can start the function counting from the end of the string rather than from the beginning. By changing the length value to a negative number, the function will use this number to represent the number of characters from the end of the given string argument at which to stop replacing the text.

Processing Strings

Now that we have gone into how to manipulate and use the more intricate substrings contained within a string value, it is only natural to get right into using strings for more powerful applications. In any given piece of software, it is likely that some sort of string processing will be involved. Be it a block of text that is being collected from an interested Internet user (for example, an e-mail address for a newsletter) or a complete block of text for use in a CMS, text is here to stay, so it is important to be able to put it to good use.

Of particular note in this day and age is security. No matter what form of content is being submitted, and no matter the form it takes (query strings, post variables, or database submittal), it is important to be able to validate both when collecting the necessary information and when outputting it. By knowing what is available to you in the form of string processing, you can quickly turn a security catastrophe into a well-managed, exception-handled occurrence. In the next recipes, we will show what you can do with the current string functions available through PHP and what you can do to help preserve the integrity of a data collection.

6-6. Joining and Disassembling Strings

The most basic functionality of strings is joining them. In PHP joining strings is easy. The simplest way to join a string is to use the dot (.) operator. For example:

```
<?php
  $string1 = "Hello";
  $string2 = " World!";
  $string3 = $string1 . $string2;
?>
```

The end result of this code is a string that reads "Hello World!" Naturally, this is the easiest way to do things; in the real world, applications will likely call for a more specific approach. Thankfully, PHP has a myriad of solutions available to take care of the issue.

A common, and rather inconvenient, dilemma that rears its ugly head is dealing with dates. With the help of Jon Stephen's date class (see Chapter 5), you will not have to deal with this issue; rather, you may have to deal with date variables coming from the database. Generally, at least in MySQL, dates can either be stored as type `date` or be stored as type `datetime`. Commonly this means they will be stored with a hyphen (`-`) delimiting the month from the day from the year. So, this can be annoying when you need just the day or just the month from a given string. PHP has the functions `explode()`, `implode()`, and `join()` that help you deal with such situations. The prototypes for the functions `implode()` and `explode()` are as follows:

```
string implode ( string glue, array pieces )
array explode ( string separator, string string [, int limit] )
```

Consider the following block of code:

```
<?php
  //Break the string into an array.
  $expdate = explode ("-","1979-06-23");
  echo $expdate[0] . "<br />"; //echoes 1979.
  //Then pull it back together into a string.
  $fulldate = implode ("-", $expdate);
  echo $fulldate; //Echoes 1979-06-23.
?>
```

```
1979
1979-06-23
```

This block of code will create an array called `$expdate` that will contain three values: 1979, 06, and 23. Basically, `explode()` splits a string at every occurrence of the character specified and packs the individual contents into an array variable for ease of use. Now, if you want to simply display the year an individual was born (a famous author perhaps?), you can easily manage to do so, like this:

```
<?php
  echo $expdate[0];
?>
```

```
1979
```

Similarly, if you then want to repackage the contents of an array into a delimited string, you can use the function `implode()` by doing something like this:

```
<?php
  $fulldate = implode ("-", $expdate);
  echo $fulldate;
?>
```

```
1979-06-23
```

The result of this line of code will repackage the array of date fragments back into a fully functioning string delimited by whatever character you choose as an argument, in this case the original hyphen. The `join()` function acts as an alias to `implode()` and can be used in the same way; however, for the sake of coherence, the `explode()`/`implode()` duet is probably the better way to do things if for nothing more than clarity's sake.

By using `explode()` and `implode()` to their fullest, you can get away with some classy and custom maneuvers. For example, if you want to group like fields into just one hidden field, perhaps to pass along in a form, you can implode them into one string value and then pass the string value in the hidden field for easy explosion when the data hits your processing statement.

The `strtok()` function performs a similar task to `explode()`. Basically, by entering strings into the `strtok()` function, you allow it to "tokenize" the string into parts based on a dividing character of your choosing. The tokens are then placed into an array much like the `explode()` function. Consider the following prototype for `strtok()`:

```
string strtok ( string str, string token )
```

The Code

```
<?php
  $anemail = "lee@babinplanet.ca";
  $thetoken = strtok ($anemail, "@");
  while ($thetoken){
    echo $thetoken . "<br />";
    $thetoken = strtok ("@");
  }
?>
```

```
Lee
babinplanet.ca
```

How It Works

As you can see, the `strtok()` function skillfully breaks the string down into highly useable tokens that can then be applied to their desired task.

In this example, say you want to tokenize the string based upon the at (`@`) symbol. By using `strtok()` to break the string down at the symbol, you can cycle through the string outputting the individual tokens one at a time. The `strtok()` function differs from the `explode()` function in that you can continue to cycle through the string, taking off or outputting different elements (as per the dividing character), where the `explode()` function simply loads the individual substrings into an array from the start.

Further, sometimes you will probably prefer to split a string up without using a dividing character. Let's face it, strings don't always (and in fact rarely do) follow a set pattern. More often than not, the string will be a client- or customer-submitted block of text that reads coherently across, left to right and up to down (just like the book you currently hold in your hands). Fortunately, PHP has its answer to this as well; you can use a function called `str_split()`. The definition of `str_split()` is as follows:

```
array str_split ( string string [, int split_length] )
```

Basically, `str_split()` returns an array filled with a character (or blocks of characters) that is concurrent to the string that was placed as an argument. The optional `length` argument allows you to break down a string into chunks of characters. For example, take note of the following block of code:

```
<?php
  $anemail = "lee@babinplanet.ca";
  $newarray = str_split($anemail);
?>
```

This instance would cause an array that looks like this:

```
Array {
  [0] => l
  [1] => e
  [2] => e
  [3] => @
  [4] => b
  [5] => a
  [6] => b
  [7] => i
  [8] => n
  [9] => p
  [10] => l
  [11] => a
  [12] => n
  [13] => e
  [14] => t
  [15] => .
  [16] => c
  [17] => a
}
```

You can also group the output into blocks of characters by providing the optional `length` argument to the function call. For instance:

```
$newarray = str_split ("lee@babinplanet.ca",3);
```

In this case, the output array would look like this:

```
Array {
  [0] => lee
  [1] => @ba
  [2] => bin
  [3] => pla
  [4] => net
  [5] => .ca
}
```

6-7. Reversing Strings

While we are on the subject of working with strings, we should note that you can also reverse strings. PHP provides a bare-bones, yet highly functional, way to take a string and completely reverse it into a mirror image of itself. The prototype of the function `strrev()`, which performs the necessary deed, is as follows:

```
string strrev ( string string )
```

Therefore, you can take a basic string, such as the fan favorite "Hello World," and completely reverse it by feeding it into the `strrev()` function as an argument.

The Code

```
<?php
  $astring = "Hello World";
  echo strrev ($astring);
?>
```

```
dlroW olleH
```

How It Works

The output for such code would change the value of "Hello World" into the rather more convoluted "dlroW olleH" string. Quite apart from those who prefer to read using a mirror, the `strrev()` function can come in handy in a myriad of ways ranging from using encryption to developing Internet-based games.

6-8. Controlling Case

From time to time, it can be important to control the case of text strings, particularly from user-submitted data. For instance, if you have created a form that allows a customer to create an account with your site and allows them to enter their preferred username and password, it is probably a good idea to force a case-sensitive submittal. Confusion can occur if a client creates a password that contains one wrongly created capital letter, especially when using a password field (with all characters turned into asterisks). If the client meant to enter "mypass" but instead entered "myPass" accidentally, an exact string match would not occur.

PHP has several ways to control the case of a string and hence remove the potential for such a disaster. The ones most relevant to the previous problem are the functions `strtoupper()` and `strtolower()`. The prototypes for these two functions are as follows:

```
string strtoupper ( string string )
string strtolower ( string str )
```

These functions do what you would expect them to do. The function `strtoupper()` turns an entire block of text into uppercase, and `strtolower()` changes an entire string into lowercase. By using either of these functions, you can quickly turn troubles with case sensitivity into things of the past.

The Code

```
<?php
  //The value passed to use by a customer who is signing up.
  $submittedpass = "myPass";
  //Before we insert into the database, we simply lowercase the submittal.
  $newpass = strtolower ($submittedpass);

  echo $newpass; //Echoes mypass
?>
```

```
mypass
```

How It Works

This code will work fine if there was a user mistake when entering a field or if you want all the values in your database to be a certain case, but what about checking logins? Well, the code can certainly apply there as well; the following block of code will check for a valid username and password match:

```
<?php
  if (strcmp (strtolower ($password), strtolower ($correctpassword) == 0){
    //Then we have a valid match.
  }
?>
```

This function also uses the `strcmp()` function, which is described in more detail later in this chapter (see recipe 6-12).

By turning both the correct password and the user-submitted password into lowercase, you alleviate the problem of case sensitivity. By comparing the two of them using the `strcmp()` function (which returns a zero if identical and returns a number greater than zero if the first string is greater than the second, and vice versa), you can find out whether you have an exact match and thusly log them in properly.

Besides turning an entire block of text into a specific case, PHP can also do some interesting things regarding word-based strings. The functions `ucfirst()` and `ucwords()` have the following prototypes:

```
string ucfirst ( string str )
string ucwords ( string str )
```

Both functions operate on the same principle but have slightly differing scopes. The `ucfirst()` function, for instance, changes the first letter in a string into uppercase. The `ucwords()` does something slightly handier; it converts the first letter in each word to uppercase. How does it determine what a word is? Why, it checks blank spaces, of course. For example:

```
<?php
  $astring = "hello world";
  echo ucfirst ($astring);
?>
```

```
Hello world
```

This would result in the function outputting the "Hello world" phrase. However, if you changed the function slightly, like so:

```
<?php
  $astring = "hello world";
  echo ucwords ($astring);
?>
```

you would get the (far more satisfying) result of a "Hello World" phrase:

```
Hello World
```

As you can see, controlling the case of strings can be both gratifying and powerful; you can use this feature to control security in your applications and increase readability for your website visitors.

6-9. Trimming Blank Spaces

A potentially disastrous (and often overlooked) situation revolves around blank spaces. A frequent occurrence is for website visitors (or CMS users) to enter content that contains a myriad of blank spaces into forms. Of particular frequency is the copy-and-paste flaw. Some people may compose text in a word processor or perhaps copy text from another web browser. The problem occurs when they then try to paste the submission into a form field. Although the field may look properly filled out, a blank space can get caught either at the beginning or at the end of the submittal, potentially spelling disaster for your data integrity goal. PHP has a few ways to deal with this.

The more common way of removing blank space is by using PHP's `trim()`, `ltrim()`, and `rtrim()` functions, which go a little something like this:

```
string trim ( string str [, string charlist] )
string ltrim ( string str [, string charlist] )
string rtrim ( string str [, string charlist] )
```

The `trim()` function removes all whitespace from the front and back of a given string; `ltrim()` and `rtrim()` remove it exclusively from the front or back of a string, respectively. By providing a list of characters to remove to the optional `charlist` argument, you can even specify what you want to see stripped. Without any argument supplied, the function basically strips away certain characters that should not be there; you can use this without too much concern if you are confident about what has to be removed and what does not.

The Code

```
<?php
  $blankspaceallaround = " somepassword ";
  //This would result in all blank spaces being removed.
  echo trim ($blankspaceallaround) . "<br />";
  //This would result in only the blank space at the beginning being trimmed.
  echo ltrim ($blankspaceallaround) . "<br />";
  //And, as you can imagine, only the blank space at the end would be trimmed here.
  echo rtrim ($blankspaceallaround) . "<br />";
?>
```

How It Works

For security purposes and all-around ease of use, it makes sense to use `trim()` on pretty much any field you encounter. Blank spaces cannot be seen and more often than not will cause trouble for the individual who entered them. Particularly disastrous are login fields that can be next to impossible to decipher should some unruly blank spaces make their appearance. It is highly recommended that you take care of any information that is integral to the system (validation, please!), and using the trim functions provides the means to an end in that regard.

As a side note, data storing is not the only place this sort of validation can come in handy. Pretty much any form consisting of user submittal can benefit from a little extra cleanliness. Search queries with blank spaces accidentally entered at the beginning or end of a search term can provide a frustrating experience for visitors to your website, for instance.

6-10. Wrapping Text

Sometimes it is not always a matter of ensuring a proper submittal of data that makes string manipulation so important; it is frequently important to ensure that strings are displaying properly to the end user. There is no point in having a beautiful set of information that displays in a choppy, chunky manner. Once again, PHP comes to your rescue by providing a couple of clever text formatting functions.

We will first talk about the function `nl2br()`, whose prototype is as follows:

```
string nl2br ( string string )
```

Basically, `nl2br()` changes any new line characters found in the data string into `<br />` Hypertext Markup Language (HTML) code. This can be extremely handy when building CMS type systems with end users who are unfamiliar with HTML code. Which would you consider easier out of the following two choices? First, is it easier teaching clients who have absolutely no technical expertise whatsoever (and no time for any) how to use the cryptic `<br />` every time they want a new line, or, second, is it easier just telling them to hit the Enter key whenever they want a new line? If you chose the second option, go grab yourself a cookie (we recommend the white-chocolate, macadamia-nut variety).

Basically, the `nl2br()` function can be a lifesaver because it allows your client (or whoever is entering information) to enter text into a text area in a way that looks normal to them. Then, rather than displaying one big chunk of run-on text on the website, you can allow the already formatted text to "automagically" display using this function.

The Code

```
<?php
  $astring = "Hello\nWorld\n\nHow are you?";
  echo nl2br ($astring);
?>
```

This block of code would result in something that looks like this:

```
Hello
World

How are you?
```

How It Works

The `nl2br()` function is nice if the person submitting the data is aware of carriage returns and whatnot, but what if they just feel like copying and pasting a huge block of text into your painstakingly prepared web layout? Well, there is a simple way of dealing with this sort of occurrence as well, using the highly useful `wordwrap()` function that has the following prototype:

```
string wordwrap ( string str [, int width [, string break [, bool cut]]] )
```

By using this function, you can set a block of text to wrap to a width of your choosing and then even choose the character you want to break it with. Consider the following block of code as an example:

```
<?php
  $textblock = "See spot run, run spot run.  See spot roll, roll spot roll!";
  echo wordwrap ($textblock, 20, "<br />");
?>
```

This would create a paragraph whereby the text block would go only to a width of 20 and then break into a new line. Not only does this help lay out the page in a more readable format, it can also be a lifesaver in certain circumstances. The output would look something like this:

```
See spot run, run
spot run. See spot
roll, roll spot roll!
```

Unfortunately, while HTML layout elements such as tables or divs can contain text and wrap the text automatically, they do have one interesting flaw. Basically, HTML will wrap text only if there is a blank space contained (that is, a new word). Sadly, this does not encompass the end result of someone entering a word that is really long and does not contain a blank space. For example:

```
<?php
  $alongstring = "Hellllllllllllllllllllllllllllllooooooooooo World";
?>
```

Now, if the very long "Hello" happened to be contained by a certain design HTML wrapper and it exceeded the length of the wrapper, the design could potentially break. But if you put the `wordwrap()` function to good use, you should be safe even in such an eventuality.

6-11. Checking String Length

A common occurrence that is quite easily handled in PHP is attempting to find out how long a string is. This can come in handy in multitudes of places, including validation of form elements, output of user-submitted data, and even database insertion preparation. PHP's `strlen()` function will instantly retrieve for you the length of any given string. The prototype for `strlen()` is as follows:

```
int strlen ( string string )
```

Since validation and security are such vital issues, it is important to know a few common string types that should always be checked for proper length. First up is data that will soon be inserted into a database and that has been submitted from a form by a user. Without going too in depth into MySQL (Chapter 15 goes into more detail in that regard), we will just begin by saying that certain data fields in a database can handle only a certain size field. If a string field, for instance, goes into a database field that cannot take the length of the string, an error will definitely be generated; and that is no fun for anyone. What is the simple way around this problem? You can simply validate the string's length using `strlen()`, as shown in the following example.

The Code

```
<?php
  //Define a maximum length for the data field.
  define ("MAXLENGTH", 10);
  if (strlen ("Hello World!") > MAXLENGTH){
    echo "The field you have entered can be only " ➥
 . MAXLENGTH . " characters in   length.";
  } else {
    //Insert the field into the database.
  }
?>
```

```
The field you have entered can be only 10 characters in length.
```

How It Works

As you can see, by checking to make sure the length of the string is less than the maximum length that your database field will allow, you prevent a potential tragedy. You can use this in many occasions such as making sure a password submitted by a user is at least a certain number of characters in length and when outputting user-submitted text that could potentially break a design to which a CMS has been applied.

6-12. Comparing Strings

No matter what language you are programming in, comparing values becomes a common dilemma. Unlike in most programming languages, however, PHP makes comparing easy, at least on the surface. The easiest way to compare two strings is with the == operator. The == operator, in PHP, basically determines an exact equal match when using a conditional statement. The following block of code shows how to use it.

The Code

```
<?php
  $stringone = "something";
  $stringtwo = "something";
  if ($stringone == $stringtwo){
    echo "These two strings are the same!";
  }
?>
```

```
These two strings are the same!
```

How It Works

However, sometimes a simple string comparison as shown previously just will not cut it. Sometimes a more precise comparison is in order; once again PHP has given you an answer in the form of `strcmp()`.

```
int strcmp ( string str1, string str2 )
```

The function `strcmp()` does slightly more than your average == operator as well. Not only does it check for an exact binary match between strings, but it can also return a result that lets you know if a string is greater than or less than the other. More specifically, if the value returned is less than zero, then string 1 is less than string 2; and, as you might expect, if the returned value is greater than zero, then string 1 is greater than string 2.

A real-world way in which you may want to use a full-on binary comparison function such as `strcmp()` is when dealing with usernames and passwords. Quite realistically, it is not good enough for a string to be "almost" the same as the other one. What we mean by that is if blank spaces get in the way or some such circumstance, occasionally the == operator will return a match even when the two strings are not completely identical. By using the `strcmp()` function, you can be assured that if the two values are not a complete and absolute match, the function will not return you a zero.

PHP also has a few other cousin functions to the mighty `strcmp()` that are a little more advanced and provide slightly different functionality. The more similar function available is the `strncmp()` function, which does almost the same thing as `strcmp()` but adds the benefit of being able to choose the length of the characters you want to compare. The `strncmp()` function has a prototype that looks like this:

```
int strncmp ( string str1, string str2, int len )
```

Similarly, should you not be interested in case sensitivity when comparing strings, you can use the functions `strcasecmp()` and `strncasecmp()`, which look like this:

```
int strcasecmp ( string str1, string str2 )
int strncasecmp ( string str1, string str2, int len )
```

Basically, these two functions do exactly what their case-sensitive counterparts do, only they completely ignore case sensitivity. The slightly confusing part of the `strncmp()` and `strncasecmp()` functions is the `len` argument. What this means is that it will compare `len` amount of characters from the first string with the second string. For example:

```
<?php
  if (strncmp ("something","some",4) == 0){
    echo "A correct match!";
  }
?>
```

```
A correct match!
```

6-13. Comparing Sound

A common use for comparing strings has always been a search engine. By entering appropriate terms as arguments, you can then compare them against similar fields using string comparison. In quite a few modern-day applications, direct string comparisons may not be enough to satisfy the ever-growing need for a powerful search application.

To help make search engines a touch friendlier, a concept was created that will allow you to return accurate search results even if the search term is pronounced in a similar tone. PHP 5 has a function that can determine matching strings based on something called a *soundex key*. The function `soundex()` has the goal of identifying a match based on pronunciation. The prototype for the function is as follows:

```
string soundex ( string str )
```

The Code

```
<?php

  echo soundex ("Apress") . "<br />";
  echo soundex ("ahhperess") . "<br />";

  echo soundex ("Lee") . "<br />";
  echo soundex ("lhee") . "<br />";

  echo soundex ("babin") . "<br />";
  echo soundex ("bahbeen") . "<br />";
```

```
  //Now, say I wanted to buy a xylophone online but had no idea how to spell it.
  echo soundex ("xylophone") . "<br />";

  //Here is a common misspelling no doubt.
  echo soundex ("zilaphone");
  //Note, how the end 3 numbers are the same? That could be used to perform a match!

?>
```

```
A162
A162
L000
L000
B150
B150
X415
Z415
```

How It Works

As you can see, similar-sounding pronunciations can result in similar (if not exact) results. The first character returned is the first letter used in the query, and the next set of three numerical values is the soundex key that is based on how the word sounds. By integrating this sort of functionality into your search engines, you can return a set of potential results with much greater accuracy than if you were using exact matches.

Project: Creating and Using a String Class

It is certainly one thing to show how string functions could be used but quite another to apply them to a real-world example. String manipulation is a common solution to many programming dilemmas, and sometimes the ability to put string functionality to use on the fly can mean the difference between a botched project and a fully functional web solution. In the next example, we have created an actual real-world project that draws on string functionality to process a wide range of applications.

6-14. Using a Page Reader Class

One of the more amusing algorithms that you can use is a web page reader, more commonly referred to as a *spider*. Basically, the point of the `pagereader` class is to read a web page that is located somewhere on the Internet and then parse it for appropriate or interesting information.

The next class's intent is to read a page and uncover a listing of all links, e-mails, and words contained within a given web page. The same sort of functionality is applied to many modern-day, large-scale operations including web search engines and, sadly, spam e-mail collectors. The following class will show you the basics of using a wide variety of string functions to process an effective application.

The Code

```
<?php

  //Class to read in a page and then output various attributes from said page.
  class pagereader {

    protected $thepage;

    //The constructor function.
    public function __construct (){

      $num_args = func_num_args();

      if($num_args > 0){
        $args = func_get_args();
        $this->thepage = $args[0];
      }
    }

    //Function to determine the validity of a file and then open it.
    function getfile () {
      try {
        if ($lines = file ($this->thepage)){
          return $lines;
        } else {
          throw new exception ("Sorry, the page could not be found.");
        }
      } catch (exception $e) {
        echo $e->getmessage();
      }
    }

    //Function to return an array of words found on a website.
    public function getwords (){
      $wordarray = array ();
      $lines = $this->getfile ();
      //An array of characters we count as an end to a word.
      $endword = array ("\"","<",">"," ",";","(",")","}","{");
      //Go through each line.
      for ($i = 0; $i < count ($lines); $i++){
        $curline = $lines[$i];
        $curline = str_split ($curline);
        for ($j = 0; $j < count ($curline); $j++){
          //Then start counting.
          $afterstop = false;
          $afterstring = "";
          $counter = 0;
```

```
      for ($k = $j; $k < count ($curline); $k++){
        $counter++;
        if (!$afterstop){
          if (!in_array ($curline[$k],$endword)){
            $afterstring = $afterstring . $curline[$k];
          } else {
            $afterstop = true;
            //Set j to the next word.
            $j = $j + ($counter - 1);
          }
        }
      }
      if (trim ($afterstring) != ""){
        $wordarray[] = $afterstring;
      }
    }
  }
  return $wordarray;
}

//Function to deliver an array of links from a website
public function getlinks (){
  //Read the file.
  $lines = $this->getfile ();
  $impline = implode ("", $lines);
  //Remove new line characters.
  $impline = str_replace ("\n","",$impline);
  //Put a new line at the end of every link.
  $impline = str_replace("</a>","</a>\n",$impline);
  //Then split the impline into an array.
  $nlines = split("\n",$impline);

  //We now have an array that ends in an anchor tag at each line.
  for($i = 0; $i < count($nlines); $i++){
    //Remove everything in front of the anchor tag.
    $nlines[$i] = eregi_replace(".*<a ","<a ",$nlines[$i]);
    //Grab the info in the href attribute.
    eregi("href=[\"']{0,1}([^\"'> ]*)",$nlines[$i],$regs);
    //And put it into the array.
    $nlines[$i] = $regs[1];
  }

  //Then we pass back the array.
  return $nlines;
}

//Function to deliver an array of e-mails from a site.
```

```
    public function getemails (){
      $emailarray = array ();
      //Read the file.
      $lines = $this->getfile ();
      //Go through each line.
      for ($i = 0; $i < count ($lines); $i++){
        //Then, on each line, look for a string that fits our description.
        if (substr_count ($lines[$i],"@") > 0){
          //Then go through the line.
          $curline = $lines[$i];
          //Turn curline into an array.
          $curline = str_split ($curline);
          for ($j = 0; $j < count ($curline); $j++){
            if ($curline[$j] == "@"){
              //Then grab all characters before and after the "@" symbol.
              $beforestring = "";
              $beforestop = false;
              $afterstring = "";
              $afterstop = false;
              //Grab all instances after the @ until a blank or tag.
              for ($k = ($j + 1); $k < count ($curline); $k++){
                if (!$afterstop){
                  if ($curline[$k] != " " && $curline[$k] != "\""  ➥
&& $curline[$k] != "<"){
                    $afterstring = $afterstring . $curline[$k];
                  } else {
                    $afterstop = true;
                  }
                }
              }
              //Grab all instances before the @ until a blank or tag.
              for ($k = ($j - 1); $k > 0; $k--){

                if (!$beforestop){
                  if ($curline[$k] != " " && $curline[$k] != ">"  ➥
&& $curline[$k] != ":"){
                    $beforestring = $beforestring . $curline[$k];
                  } else {
                    $beforestop = true;
                  }
                }
              }
              //Reverse the string since we were reading it in backwards.
              $beforestring = strrev ($beforestring);
              $teststring = trim ($beforestring) . "@" . trim ($afterstring);
              if (preg_match("/^([a-zA-Z0-9])+([.a-zA-Z0-9_-])*@([a-zA-Z0-9_-])➥
+(.[a-zA-Z0-9_-]+)+[a-zA-Z0-9_-]$/",$teststring)){
```

```
                //Only include the e-mail if it is not in the array.
                if (!in_array ($teststring,$emailarray)){
                  $emailarray[] = $teststring;
                }
              }
            }
          }
        }

      }
      //Then we pass back the array.
      return $emailarray;
    }

  }

  $myreader = new pagereader ("http://www.apress.com");

  //None found ;).
  ?><p style="font-weight: bold;">Emails:</p><?php
  print_r ($myreader->getemails ());
  //Whoa, a few links.
  ?><p style="font-weight: bold;">Links:</p><?php
  print_r ($myreader->getlinks ());
  //Hold on to your hats, this will take a while...
  ?><p style="font-weight: bold;">Words:</p><?php
  print_r ($myreader->getwords ());

?>
```

How It Works

The `pagereader` class's core functionality is based around reading a web page on the Internet and then performing operations on the read. Therefore, it comes with the validated method `getfile()`, whose sole purpose is to attempt to read in a web page using the `file()` function. If the function receives a valid read, then you can begin work on the received information.

The class has three main functions, and they all perform somewhat differently to accomplish their goals. The `getwords()` method is perhaps the simplest of the three merely because of its somewhat global goal. The purpose of the `getwords()` method is to collect an array filled with all words contained on a website. The problem is, what constitutes a word? The answer to such a question will probably vary from user to user, so an array filled with characters that will be omitted when determining the end of a word has been instantiated. By changing the values contained within this array, you can determine what constitutes a word and thus change the way the script reads in a word list.

The way it works after that is quite simple. The script takes in each line of the received file individually and then splits it into an array. It then parses through the array and waits until it finds a character that is not in the current array of end characters. After it finds such a character,

it loops through the string of characters found after the start character and waits until it finds another character in the array, adding to a final string as it goes. Once it reaches a final character, it stores the "word" into an array to be returned when the script has finished processing.

The `getemails()` method works similarly to the `getwords()` method, except it bases everything upon the `@` symbol. So, although it also goes through each line received from the file and breaks it down, it instead breaks it down according to the `@` symbol. When a valid symbol has been found, it cycles through all characters before and after the symbol and quits cycling once an end character has been found. Once an end character has been found before and after the `@` symbol, a full string is concocted and compared against a valid e-mail string using the `preg_match()` function. (For more information, see Nathan A. Good's Chapter 9.) If a valid match is received, the e-mail is returned in an array filled with e-mail addresses.

The last method in this class also differs the most. It combines string functionality with regular expressions to create a link targeting script. Basically what this method does is break the received lines down into a single line and kill off all new line characters. Then, it searches for any instances of an anchor tag and places a new line character after the closing anchor tag. Then, with an array of anchor tags delimited by a new line character in place, it strips out everything from in front of the leading anchor tag and grabs all information from within the `href` argument. At this point, the data contained within is stored into an array for returning.

As you can see, you can perform a wide range of functionality using the received file information; what is shown here is only a small glimpse. With the wide range of functionality available in the form of string functions, anything is possible.

Summary

So, as you can see, strings will always be a rather important subject when dealing with a programming language such as PHP. Thankfully, because of PHP 5's new class functionality, it is becoming easier to take matters into your own hands and concoct some truly powerful classes to help make your life just a little easier. It is important to experiment and use the tools available to you. As you can see from the real-world example in this chapter of a `pagereader` class, the ability to use string functionality on the fly is a learned and highly appreciated skill.

Looking Ahead

In the next chapter, we will go through the ins and outs of working with your current file system. This is a handy set of functionality that will likely serve you well in your quest for the perfect web application. While operating systems and server configurations may differ, the ability to react to such changes, with a bit of help from this book, will define you as the master programmer that you are.