PHP and HTML Text

3.1.1. Embedding PHP code in HTML

You can embed PHP inside HTML tags. In the following example 3.1 only HTML tags are there.

Example 3-1 is a simple HTML file that we'll use for an example.

Example 3-1. All you need to start with PHP is a simple HTML document

```
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <p>This is without any PHP code.</p>
    </body>
</html>
```

Nothing is special here; just your plain HTML file. However, you can enter PHP right into this file; for example, let's use PHP's `echo` command to output some text in Example 3-2.

Example 3-2. Adding some PHP code to the HTML file

```
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        echo("<p>This is with PHP code.</p>");
    </body>
</html>
```

3.1.1.1. Separating PHP from HTML

This looks pretty simple, but there are some problems. There's no way to tell in this file which part is standard HTML and which part is PHP and therefore must be handled differently. To fix this, surround your PHP code in `<?php` and `?>` tags.

When you start writing PHP code, you'll be working with plain vanilla text files that contain PHP and HTML code. HTML is a simple markup language that designates how your page looks in a browser, but it is simply that: text only. The server doesn't have to process HTML files before

sending them to the user's browser. Unlike HTML code, PHP code must be interpreted before the resulting page is sent to the browser. Otherwise, it will be one big mess on the user's screen.

To set apart the PHP code to inform the web server what needs to be processed, the PHP code is placed between formal or informal tags mixed with HTML. Example 3-3 uses <?php ?> to place the php commands inside. In this example `echo` and `print` commands are kept within <?php ?> tags.

Example 3-3. Calling echo and print

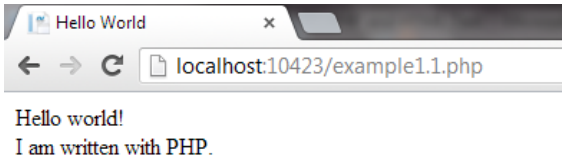
```
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <?php

        echo ("Hello world!<br />");
        print ('I am written with PHP.<br />');
        ?>
    </body>
</html>
```

When a browser requests this file, PHP interprets it and produces HTML markup. Example 3-4 is the HTML that is produced from the code in Example 3-3.

Example 3-4. The HTML markup produced by the PHP code

```
<html>
<head>
        <title>Hello World</title>
    </head>
    <body>
        Hello world!<br />I am written with PHP.<br />
    </body>
</html>
```

Figure 3-1. The output as it appears in the web browser



While writing PHP code, it's better to add comments so that the code is easier to read and understand. PHP supports two styles of comments single-line comments and multiple-line comments.

Example 3-5. Using comments to make your code easier to read

```
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <?php

        // A single line comment could say that we are going to
        // print hello world.

        /* This is how to do a
           multi-line comment and could be used comment out a block
           of code */

        echo("Hello world!<br />");
        print('Goodbye.<br />');

        ?>
    </body>
</html>
```

In Example 3-5, two comment styles are used: `//` for single-line comments and `/*` ... `*/` for multiline comments. Keep in mind that if you want to place a comment in HTML markup, you need to use the open comment `<!` and close comment `>` tags.

A semicolon (`;`) ends all code statements in PHP. (Because of this, semicolons can't be used in names.) It's good style as well as practical to also start a new line after your semicolon so the code is easier to read.

3.2. Variables

A variable stores a value, such as the text string "Hello World!" or the integer value 1. A variable can then be reused throughout your code, instead of having to type out the actual value over and over again for the entire life of the variable, which can be frustrating and tedious.

In PHP, you define a variable by preceding $ symbol with user defined name like:

```
$variable_name = value;
```

The dollar sign (`$`) must always fill the first space of the variable. The first character after the dollar sign must be a letter or underscore. It can't under any circumstances be a number; otherwise, your code will not execute, so watch those typos!

- PHP variables may only be composed of alphanumeric characters and underscores; for example, `a-z, A-Z, 0-9`, and `_`.
- Variables in PHP are case sensitive. This means that `$variable_name` and `$Variable_Name` are different.
- Variables with more than one word should be separated with underscores; for example, `$test_variable`.
- Variables can be assigned values by using the equals sign (`=`).
- Always end with a semicolon (`;`) to complete the assignment of the variable.

If you were to assign a new value to a variable with the same name, as happens in Example 3-6, the old name would be overwritten and you would have a potential logic error in your code.

Example 3-6. Assigning a variable

```
<?php
 $age = 30;
  echo $age;
?>
```

The value of `$age` is

```
30
```

3.2.1.1. Reading a variable's value

To access the value of a variable that's already been assigned, simply specify the dollar sign (`$`) followed by the variable name, and use it as you would the value of the variable in your code.

For example, the code echo in Example 3.6 displays the value of `$age`.

3.2.1.2. Variable types

We have already seen two types of data:string and numerical data types that can be stored in variables. Other data types are available namely:

Numerical data – floating point and integer type
Example : $age=30
Booleans - true or false value

*String- for non-numeric data type, that can hold letters, special characters, numbers etc.*String values must be enclosed in either single quotes or double quotes.
Example $name=”welcome”

- You may also encounter the NULL data type, which is a “special” data type first introduced in PHP 4. NULLs are used to represent “empty” variables in PHP; a variable

Other than initial assignment PHP automatically picks a data variable based on the value assigned to it on the right hand side.

In situations where a specific type of data is required, such as the mathematical division operation, PHP attempts to convert the data types automatically. If you have a string with a single "2," it will be converted to an integer value of 2. This conversion is nearly always exactly what you want PHP to do and makes coding seamless for you.

3.2.1.3. Variable scope

PHP helps keep your code organized by making sure that, if you use code that someone else wrote (and you very likely will), the names of the variables in your code don't clash with other previously written variable names. For example, if you're using a variable called `$name` that has a value of `Bill`, and you use someone else's code that also has a variable called `$name` but uses it to keep track of the filename log.txt, your value could get overwritten. Your code's value for `$name` of `Bill` will be replaced by `log.txt`, and your code will say `"Hello log.txt"` instead of `"Hello Bill"`, which would be a big problem.

To solve this problem, PHP organizes code into functions. Functions allow you to group a chunk of code together and execute that code by its name. To keep variables in your code separate from variables in functions, PHP provides separate storage of variables within each function. This separate storage space means that the scope, or where a variable's value can be accessed, is the local storage of the function.

Example 3-7 shows how the variable you use outside of the function isn't changed by the code within the function.

.

Example 3-7. The default handling of variable scope

```
<?php

// define a function
function setname(){
    // Set name to ram
    $name = "ram";
}

// Set name to krishnan
$name = "krishnan";

// Call the function
setname();

// Display the age
echo $name;

?>
```

This displays:

```
krishnan
```

Although calling the function setname assigns “ram” to the variable `$name`, it's not accessing the same variable that was defined on the main level of the program. Therefore, when you print `$name`, you see the original value of “krishnan”. The bolded part of the code is what is seen when `$name =` is printed, because `$name` in setname is a separate variable.

All of this is great, but if you really want to access or change the variable `$name` that was created outside the setname function, you would use a global variable.

3.2.1.3.1. Global variables

Global variables allow you to cross the boundary between separate functions to access a variable's value. The `global` statement specifies that you want the variable to be the same variable everywhere, or globally. Figure 3-4 shows how a global variable is accessible to everything.

Figure 3-4. The global keyword creates one global variable called $age

Example 3-8 shows that use of a global variable can result in a change.

Example 3-8. Using a global variable changes the result

```
<?php

// Define a function
function birthday(){
    // Define age as a global variable
    global $age;

    // Add one to the age value
    $age = $age + 1;
}

// set age to 30
$age = 30;

// Call the function
birthday();

// Display the age
echo $age;

?>
```

This displays:

```
31
```

Global variables should be used sparingly, since it's easy to accidentally modify a variable by mistake. This kind of error can be very difficult to locate. Additionally, when we discuss functions in detail, you'll learn that you can send in values to functions when you call them and get values returned from them when they're done. That all boils down to the fact that you really don't have to use global variables.

If you want to use a variable in a specific function without losing the value each time the function ends, but you don't want to use a global variable, you would use a static variable.

3.2.1.3.2. Static variables

Static variables provide a variable that isn't destroyed when a function ends. You can use the static variable value again the next time you call the function.

The easiest way to think about this is to realize that the variable is a global to just that function. In Example 3-9, we use the `static` keyword to define these function variables.

Example 3-9. A static variable remembering its last value

```
<?php

// Define the function

function birthday(){
    // Define age as a static variable
    static $age = 0;

    // Add one to the age value
    $age = $age + 1;

    // Print the static age variable
    echo "Birthday number $age<br />";
}

// Set age to 30
$age = 30;

// Call the function twice
birthday();
birthday();

// Display the age
echo "Age: $age<br />";

?>
```

This displays:

```
Birthday number 1
Birthday number 2
Age: 30
```

The value of `$age` is now retained each time the `birthday` function is called. The value will stay around until the program quits. We've discussed two types of variables, but there's one more to discuss: super global.

3.2.1.3.3. Super global variables

PHP uses special variables called super globals to provide information about the PHP script's environment. These variables don't need to be declared as global; they are automatically available and provide important information beyond the script's environment, such as values from a user input.

Since PHP 4.01, the super globals are defined in arrays. Table 3-1 shows the existing arrays since PHP 4.01.

Table 3-1. PHP super globals

Variable array name	Contents
$GLOBALS	Contains any global variables that are accessible for the local script. The variable names are used to select which part of the array to access.
$_SERVER	Contains information about the web server environment.
$_GET	Contains information from `GET` requests (a form submission).
$_POST	Contains information from `POST` requests (another type of form submission).
$_COOKIE	Contains inform from HTTP cookies.
$_FILES	Contains information from `POST` file uploads.
$_ENV	Contains information about the environment (Windows or Mac).
$_REQUEST	Contains information from user inputs. These values should not be trusted.
$_SESSION	Contains information from any variables registered in a session.

An example of a super global is `PHP_SELF`. This variable contains the name of the running script and is part of the `$_SERVER` array, as shown in Example 3-10.

Example 3-10. PHP_SELF being used with a file called test.php

```
<?php
echo $_SERVER["PHP_SELF"];
?>
```

This outputs:

```
/test.php
```

This variable is especially useful, as it can be used to call the current script again when processing a form. Super global variables provide a convenient way to access information about a script's environment from server settings to user inputted data.

3.2.2. Strings

Variables can hold more than just numbers. They can hold characters and strings, or an ordered list of characters. Figure 3-6 demonstrates how an ordered list of characters becomes a string.

A string can be used directly in a function call or it can be stored in a variable. In Example 3-11, we create the exact same string twice: first we store it in a variable, and then we place the string directly into a function.

Example 3-11. Working with strings

```
<?php
$my_string = "Employee Paybill Application";
echo " Employee Paybill Application ";
?>
```

In Example 3-12, the first string is stored in the variable `$my_string`, while the second string is used in the `echo` function and isn't stored. Remember to save your strings into variables if you plan on using them more than once!

Strings are flexible. You can even insert variables into string definitions when using double quotes to start and end your string, as shown in Example 3-12. Using a single quote to start and end your string does not allow a variable to be placed in the string.

Example 3-12. Using a variable in a string definition

```
<?php
$my_string = " XYZ Company Employee Paybill Application";
echo "welcome to  $my_string";
?>
```

This example displays "welcome to XYZ Company Employee Paybill Application" Double quotes are used in the above string, but single quotes or apostrophes can both be used as long as you won't be inserting variable values; see Example 3-13.

Example 3-13. Single quotes used in a string assignment

```
<?php
$my_string = 'Margaritaville - Suntan Oil Application!';
echo $my_string;
?>
```

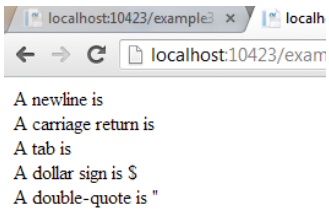
Remember, if you want to use a single quote within a string marked with single quotes, you have to escape the single quote with a backslash (\). Double quotes allow the use of many special escaped characters that you can't use with a single quote string, such as an apostrophe. If you had escaped an apostrophe with a backslash in a double-quoted string, the backslash would show up when you output the string.

3.2.2.1. Special characters in strings

Tab, newline, and carriage returns are all examples of extra, yet ignorable, whitespace (see Example 3-14). If you are writing to a file, a valuable tool is an escaped character. One downside of using the apostrophe to start and end a string is that you can't include a variable. This leads us to be careful about using HTML markup or any other string that includes quotes.

Example 3-14. Various special characters in string assignments

```php
<?php
$newline = "A newline is \n";
$return = "A carriage return is \r";
$tab = "A tab is \t";
$dollar = "A dollar sign is \$";
$doublequote = "A double-quote is \"";
echo $newline,"<br>";
echo $return,"<br>";
echo $tab,"<br>";
echo $dollar,"<br>";
echo $doublequote,"<br>";
?>
```



The `echo` function uses quotes to define the start and end of a string, so you must use one of the following tactics if your string contains quotations:

- Don't use quotes inside your string.
- Escape quotes within the string with a slash. To escape a quote, just place a slash directly before the quotation mark; i.e., `\"`.
- Use single quotes (apostrophes) for quotes inside your string.
- Start and end your string with apostrophes.

In Examples 3-15 and 3-16, the wrong and right use of the `echo` function is demonstrated.

Example 3-15. Using echo with special characters

```
<?php
// This won't work because of the quotes around specialH2!
echo "<h2 class="specialH2">Margaritaville!</h2>";
?>
specialH2
```

In the first `echo` example, we forgot to escape the double quotes that surround the `specialH2`, which is HTML text. Attempting to display this page produces the error:

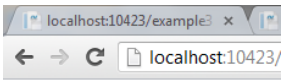
```
Parse error: parse error, unexpected T_STRING, expecting ','
or ';' in /home/www/html/oreilly/ch3/parse.php on line 3
```

If you see that error, start by checking your single and double quotes to make sure they all match up correctly, as in Example 3-16.

Example 3-16. Correct escaping of special characters

```
<?php
// OK because we used single quotes
echo "<h2 class=\"specialH2\">Margaritaville!</h2>";
echo '<h2 class="specialH2">Margaritaville!</h2>';
?>
```

Output



Margaritaville!

Margaritaville!

Example 3-16 escapes quotations by placing a slash in front of each one (`\"`). The slash tells PHP that you want the quotation to be used within the string and not ending `echo`'s string. You can also use an apostrophe (`'`) to mark the beginning and end of a string.

> If you use an apostrophe or single quote to define your string, double quotes don't need to be escaped. However, you can't include variables when using single quotes.

3.2.2.2. Comparing strings

PHP has functions to compare strings that aren't exactly alike. For example, you may want to consider "Bill" to be the same as "BILL," ignoring the case of the string.

Use `strcmp (string1, string2)` to compare two strings including the case. The return value is 0 if the two strings have the same text. Any nonzero value indicates they are not the same.

Use `strcasecmp (string1, string2)` to compare two strings without comparing the case. The return value is 0 if the two strings have the same text. Any nonzero value indicates they are not the same.

Example 3-17 compares "Bill" to "BILL" without considering the case.

Example 3-17. Using strcasecmp to compare two strings

```
<?php

$name1 = "Bill";
$name2 = "BILL";

$result = strcasecmp($name1, $name2);

if (!$result){
   echo "They match.";
}

?>
```

This returns:

```
They match.
```

3.2.3. Concatenation

Concatenation combines one or more text strings and variables, as shown in Example 3-18. When performing this combination, you save yourself the hassle of creating numerous `echo` statements, or in other words, you build up a string and use it.

Example 3-18. Concatenating strings together

```
<?php
$my_string = "Hello Max. My name is: ";
$newline = "<br />";
echo $my_string . "Paula" . $newline;
echo "Hi, I'm Max. Who are you? " . $my_string . $newline;
```

```
echo "Hi, I'm Max. Who are you? " . $my_string . "Paula";
//The last line is the same as echo "Hi, I'm max. Who are you?
$my_string Paula";
?>
```

Variables and text strings are joined together with a period (.).

Since your time is finite, tying strings and variables together helps you create dynamic web sites faster.

3.2.3.1. Combining strings with other types

If you combine a string with another data type, such as a number, the result is also a string, as shown in Example 3-19.

Example 3-19. Combining a string and a number

```
<?php
$str = "This is an example of ". 3 ." in the middle of a string.";
echo $str;
?>
```

This displays:

```
This is an example of 3 in the middle of a string.
```

`$str` contains a string even though a number was inserted into the middle.

3.2.4. Constants

You can define `constants` in your program. A constant, like its name implies, cannot change its value during the execution of your program. It's defined using the `define` function, which takes the name of the constant as the first parameter and the values as the second parameter. The definition of a constant is global and can be defined as any simple (scalar) data type such as a string or a number. You can get the value of a constant by simply specifying its name; see Example 3-20. Unlike how you handle variables, you should not put the dollar sign (`$`) before a constant. A cool thing you do with constants is using the function `constant(`*`name`*`)` to return a constant's value when the constant's name is determined dynamically. Or you could use `get_defined_constants` to return a list (as an array) of all your defined constants. If you're unsure about the arguments to a function, you can search the PHP site at http://www.php.net to find function parameters and return values.

These are the differences between constants and variables:

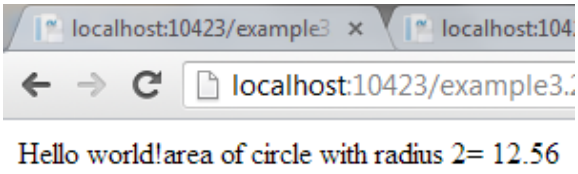
- Constants do not have a dollar sign (`$`) before them.
- Constants can only be defined using the `define` function, not by simple assignment.
- Constants are defined and accessed globally.
- Constants cannot be redefined or undefined once they have been set.
- Constants can only evaluate to scalar values.

Example 3-20 demonstrates how to use a constant in your program.

Example 3-20. Using a constant in your program

```
<?php
define("HELLO", "Hello world!");
define("PI",3.14);
echo HELLO; // outputs "Hello world."
echo "area of circle with radius 2= ", PI*2*2;
?>
```

This outputs as:



Constants are useful for values that you need to make sure don't change, such as a configuration file location.

If you use an undefined constant, PHP assumes that you mean the name of the constant itself, just as if you called it as a stringfor example, `CONSTANT` as opposed to `"CONSTANT"`. If the define line of Example 3-20 is commented out, the output becomes:

```
HELLO
```

3.2.4.1. Predefined constants

PHP provides a few constants that are predefined similar to the way we have some super globals. Examples of these include `__FILE__`, which returns the name of the PHP file that's being executed, and `__LINE__`, which returns the line number in that file. They can be handy for generating an error message as they tell you where in your code the error occurred, as shown in Example 3-21.

Example 3-21. Echoing the line and file predefined constants for a script called predefined_constants.php

```
<?php
echo "Executing line ".  __LINE__ . " of PHP script " . __FILE__ .
'.';
?>
```

This returns:

```
Executing line 2 of PHP script
/home/www/html/oreilly/ch3/predefined_constants.php.
```

The path to your script may be different than the example. On Windows, it's likely to be C:\Program Files\Apache Group\htdocs\c3.

3.2.5. Doing Math

Variables can hold numbers, too, and it's useful to perform mathematical operations on those numbers. All fundamental mathematical functions are available using PHP. You may feel like you're back in middle school algebra, but the basic functions are just like they were then: adding, subtracting, multiplying, and dividing. In Example 3-22, the divide (`/`) operator calculates the average from $sum and 10.

Example 3-22. PHP mathematical function usage

```
<?php>
$sum=200;
$avg=$sum/10;
echo $avg;
?>
```

This outputs as

20

In Figure 3-9, the 82 percent outcome from our example code displays in your browser window.

PHP also supports the mathematical operations listed in Table 3-2.

Table 3-2. The basic mathematical operators

Mathematical operator	Name	Example	Result
+	Addition	2+2	4
-	Subtraction	2-1	1
*	Multiplication	2*2	4
/	Division	2/2	1
%	Modulo (remainder)	2%1	0

The operators can take whole numbers or decimal numbers as their input.

3.2.5.1. Combined assignment

Combined assignment operators provide a shortcut for performing two common tasks at the same time. They combine reading a variable, performing an operation on it, and placing the result back in the same variable. The operations are mostly mathematical but can also include other operators like concatenation.

Combined assignment operators take the form the arithmetic operator directly followed by an equals sign (=). For example, the statement:

```
$counter=$counter+1;
```

is equivalent to:

```
$counter+=1;
```

Which is shorthand for taking the value in `$counter`, adding one to it, and then saving the result back in `$counter`.

Table 3-3 lists the most common combined assignment operators.

Table 3-3. Combined assignment operators

Combined operation	Operation	Produces
$num+=y	Addition	$num=$num+y
$num -=y	Subtraction	$num=$num-y
$num *=y	Multiplication	$num=$num*y

$num /=y	Division	$num=$num/y
$num.= "y"	Concatenation	$string=$string."y"

You'll find that these operators are very handy when creating your dynamic web pages. They'll also be used frequently in our examples. They have the added benefit of reducing the chance that you'll have a typo in your variable name, since you need to specify the variable name once only.

Along the same lines as combined operators comes a shorthand method for adding one or subtracting one from a variable.

3.2.5.2. Autoincrement and autodecrement

It's very common when writing your code to ether increment or decrement a variable by one. It's so common that PHP has a special shortcut for doing it. The autoincrement operator is `++` and is used like this:

```
$counter++;
```

What we did was:

```
$counter+=1;
```

Example 3-23 adds one to `$counter`.

Example 3-23. Using autoincrement to add to a variable

```
<?php
$counter=1;
$counter++;
echo $counter
?>
```

This produces:

```
2
```

The same concept applies to the automatic decrement operator, `--`.

Example 3-24 subtracts one from `$counter`.

Example 3-24. Using the autodecrement operator

```
<?php
$counter=1;
$counter--;
echo $counter
?>
```

This produces:

```
0
```

This notation is used frequently when doing repetitive tasks to keep track of how many times you've done them.

3.2.5.3. Pre-increment and -decrement

If you're incrementing or decrementing at the same time that you're also comparing the value of the variable, such as in a `for` or `while` loop, a pre-increment or -decrement can affect the value that's used for the comparison. When using the pre- operations, the value changes before the comparison, which is different than the de facto post-processing.

For example:

```
--$counter;
```

or:

```
++$counter;
```

Both of the operators still change the value of the counter variable, but they change the value sooner. If you are using that variable in a test, you'll see the current value before the change. We'll talk more about testing the values of variable executing blocks of code repetitively in the next chapter. Example 3-25 shows how these operators work.

Example 3-25. Using pre- and post-increment

```
<?php
$test=1;
echo "Preincrement: ".(++$test);
echo "<BR>";
echo "Value afterwords: ".$test;
```

```
echo "<BR>";
$test=1;
echo "Postincrement: ".($test++);
echo "<BR>";
echo "Value afterwords: ".$test;
?>
```

This produces:

```
Preincrement: 2
Value afterwords: 2
Postincrement: 1
Value afterwords: 2
```

Notice that in Example 3-25, the value after a post- or pre-increment is always 2. When using the pre-increment, the value is 2 in the `echo` statement that contains the combined operator.

In this chapter, you've learned about the basic concepts for writing PHP scripts. You've introduced variables that can remember information while our scripts execute. You know how to store values in variables and access those values. You don't have to worry about specifying data types, because PHP attempts to convert types automatically. You've also learned how to do basic mathematical operations and the shortcuts for the most common combined assignment operators.

These concepts will form the basis for the rest of what you learn about PHP programming, including building expressions.

The next chapter will introduce more complicated PHP code such as arrays, including looping and conditional logic. After that, we'll be able to jump into MySQL and how it operates as a database.