

Appendix F: More About PHP

This Appendix clarifies the approach taken with PHP in this book, provides additional PHP references, and includes a list of useful functions that will help you to expand your use of this language. Refer to the book Web site for the latest additions. (NOTE: refer to Appendix D for help **debugging** your PHP code).

Useful PHP References

Once you are comfortable with the basics of PHP, you will want a detailed reference to PHP syntax and functions. There are many good books and online resources available. For a complete PHP reference consult the PHP home page:

<http://www.php.net/>

For a very good PHP tutorial and easy-to-use reference, go to:

<http://www.w3schools.com/php/default.asp>

Use PHP print or echo?

In addition to the **print** statement that has been applied throughout this book, PHP also offers the **echo** statement. The two are almost identical in operation, and most programmers simply choose one over the other.

Multiple PHP sections

The book examples almost always uses a single PHP section within a document that also includes HTML tags. In fact, a PHP file can include as many PHP sections as the programmer desires, combined with any number of HTML sections. A PHP file may consist **entirely** of PHP code, and may even contain **no** PHP code (only HTML code).

Often a working program contains many sections of PHP code interspersed with HTML code. This can be useful to avoid using the escape characters needed to display quotes inside character strings produced by print statements inside PHP sections. The only character strings that **must** be generated inside PHP section are those that include values from PHP variables, functions, or expressions.

Should I use .html or .php files?

Many examples in this book use HTML forms stored in **.html** files, to submit user input to run PHP programs, stored as **.php** files. There is no requirement that documents containing only HTML tags **must** be saved with the .html extension. PHP programmers often save **all** code files as .php files, including files that contain only HTML markup and no PHP.

Consistent file names

This book takes the approach of using the same file name for the page containing the HTML form and the file containing the PHP code that processed the form (for example, **wage1.html** and **wage1.php**). Once again this was simply a design decision to simplify the approach and make the files in the samples folder easy to identify. There is no requirement that these files should share the same name.

Program variables and the \$_POST array

Similarly the samples usually created PHP variables with the same name as the names submitted from the HTML forms, for example: **\$hourlyWage = \$_POST['hourlyWage'];** There is no requirement that the variable should have the same name as the key field of the array. For example, **\$wage = \$_POST['hourlyWage'];** would be equally acceptable.

There is also no requirement that the values from the \$_POST array should be assigned to other variables in order to be used. The program could simply use **\$_POST['hourlyWage']** throughout the code rather than assign this value to **\$hourlyWage** and then use that variable. Many programmers follow the practice outlined in this book since it often makes the code more readable

Form-processing best practices

Unlike the examples in this book, PHP programmers often create a single file that combines an HTML form with the PHP code that processes the form input. The program uses an IF..ELSE structure to determine whether, each time the file is accessed, the code should **generate** the form or **process** the form.

The IF..ELSE structure is controlled by a call to the **isset()** function. Each time the file is processed, the **isset()** function tests whether the \$_POST array contains an expected input element. If it does, it means that a form has been submitted and so the form input is processed. If it does not, it means that no form was submitted so the program displays the form to the user.

Your **samples** folder includes **addTwoNumbers.html** and **addTwoNumbers.php**. These follow the standard approach taken in this book: providing an HTML form in an HTML file and the code to process the form in a PHP document. You will also find a file named **addTwoImproved.php**, which takes the approach described above. Study the code in this file to see how the form and the form-processing code are combined. This design can be applied to many different purposes.

Camelback notation

The book uses **camelback** notation to name variables, and also to name files. This approach was taken only to develop familiarity with this naming convention since it is followed in many current languages, and also encourages the habit of naming files without spaces in the file names. PHP programmers often use underscores when naming variables (for example **\$hourly_wage**) and there are no conventions for naming files.

More about PHP functions

PHP provides a very large set of useful standard functions, for many different purposes. For a list of ALL standard PHP functions (this may be overwhelming), see:

<http://www.php.net/quickref.php>

Here are a few general-purpose functions that may be especially useful as you build on what you have learned in this book:

empty() Checks whether a variable is empty, useful for validating form input.
Example: `if (empty($someVariable))`

is_numeric() Checks whether a variable contains a number.
Example: `if (is_numeric ($someVariable))`

file_exists() Checks whether a file exists before trying to open a file.
Example:

```
if ( file_exists($someFileName) )
{
    ..process the file..
}
else
    print("FILE $someFileName NOT FOUND");
```

define() Defines a constant variable (a variable that cannot be changed).
Example: `define("COMPANY_NAME", "XYZ Company");`
Example: `define("TAX_RATE", 0.07);`

phpinfo() Displays in-depth information about your PHP installation. Try it!

isset() Checks whether a variable contains a value.
Example: `if (isset($myAge))`

unset() Destroys a variable.
Example: `unset($myAge);`

Standard PHP array functions

Here are some commonly used array-processing functions:

array_shift() removes the first element from an array
Example: `array_shift($someArray)`

array_unshift() adds an element with a new value to the start of an array (the previous first element becomes the 2nd element, and so on).
Example: `array_unshift($someArray, $value)`

- array_pop()** removes the last element from an array
Example: array_pop(\$someArray)
- array_push()** adds an element with a new value to the end of an array
Example: array_push(\$someArray, \$value)
- array_sum()** sums the values in a numerical array
Example: array_sum(\$someArray)
- max()** Returns the highest value in a numerical array
Example: max(\$someArray)
- min()** Returns the lowest value in a numerical array
Example: min(\$someArray)
- sort()** sorts the values in alphabetical or numerical order
Example: sort(\$someArray)
- asort()** sorts the values in an associative array based on the stored **values**, while preserving the key/value relationships
Example: asort(\$someArray)
- ksort()** sorts the values in an associative array based on the **keys**, while preserving the key/value relationships
Example: ksort(\$someArray)

PHP data types

Variables may be used to store data of various **data types**. Each data type is stored in a different manner and allows specific operations. PHP supports the following simple data types:

Integer: A whole number (**int** data type), such as -100, -1, 0, 1, 25, or 7388.

Floating point number: A decimal number (**float** data type), such as 5.24 or 123.456789.

Character string: A sequence of 0 or more characters (**string** data type), such as "Joe Smith", or "What is your name?" or "123 Main Street" or "<p>This is a paragraph.</p>".

Boolean: A TRUE or FALSE value.

Unlike most languages, PHP allows you to use variables without first defining their data type. PHP evaluates each expression and determines the appropriate data types at the time a value is assigned, and when variables are used in expressions. For example:

```
$hoursWorked = 30;
$hourlyWage = 15.75;
$wage= $hoursWorked * $hourlyWage;
```

The first statement stores 30 as an integer. The next statement stores 15.75 as a float. The last statement converts \$hoursWorked to a float and then performs the multiplication and stores the result as a float.

To ensure that a value is being handled as a specific data type you can **type cast** the variable, for example:

```
$hourlyWage = (float) 15;
```

This statement stores 15 as a float

```
$wage = (string) $hourlyWage;
```

This statement converts the value stored in \$hourlyWage to a string and stores the string in \$wage.

Note that if you cast a **float** value as an **int**, PHP truncates the value in the same way as the **floor()** function. For example **\$value = (int) 33.75** will store **33** in **\$value**.

You can use the **gettype()** function to obtain the data of a variable. For example **gettype(\$hoursWorked)** would return "integer".