Chapter 16

More About PHP

Intended Learning Outcomes

After completing this chapter, you should be able to:

- Access textbook and global PHP help resources.
- Distinguish between conventions used in this textbook and PHP requirements and best practices.
- Choose between the PHP print and echo statements.
- Choose between single and double quotes when enclosing strings.
- Use the PHP shortcut operators when incrementing, decrementing, or accumulating values.
- Recognize when the PHP concatenation operator must be used in strings.
- Use the PHP SWITCH statement instead of chained IF..ELSE statements where this is an option.
- Utilize the PHP DO..WHILE statement instead of a WHILE statement when an event-driven loop is required to iterate at least once..
- Create and process numerically indexed multi-dimensional arrays.
- Create and process ragged multi-dimensional arrays..
- Create and process an associative array that contains any number of numerically indexed arrays.
- Describe the basic data types and convert values to different data types using the cast operator.
- Use some common PHP date and time functions.
- Utilize a broad range of standard PHP functions.

Introduction

Congratulations! By now you should have a good grasp of basic program logic and design, as well as some understanding of HTML, CSS, MySQL, and PHP. The purpose of this chapter is twofold: first to identify some of the conventions used throughout the book that should not be confused with actual requirements of the PHP language; and second, to extend your knowledge of PHP by describing additional shortcut operators and control structures, date and time functions, PHP tools for data validation and sanitizing, and PHP data types. The chapter ends with an extensive (but by no means complete) list of standard PHP functions.

Visit the textbook Web site for the latest version of the appendices, for corrections, and for additional learning materials:

```
https://www.mikeokane.com/textbooks/wbip/
```

You will also want a standard reference to PHP syntax and functions. For a complete PHP reference consult the PHP home page:

```
https://www.php.net/
```

There are also many excellent forums, books, and online references available. For a very good online PHP tutorial and easy-to-use reference, try:

```
https://www.w3schools.com/php/default.asp
```

Textbook Conventions

A major challenge writing this textbook has been to keep the focus on the fundamentals of program logic and design, while at the same time providing the valuable practical experience that comes with hands-on coding and learning a specific language. In order to balance these goals in a manner suitable for beginning programmers, decisions were made concerning the level and type of detail to include. In some cases, some procedures were neglected to avoid introducing too much syntactical detail, or too many options at once, that might distract from achieving key learning outcomes and fundamental concepts. The danger of this approach is that the reader might easily confuse the author's strategic design decisions with actual requirements of the PHP language. With this in mind, the initial sections in this chapter identify some of the practices that were followed in this textbook, that are **not** required PHP procedures and do **not** necessarily represent best practices when developing PHP applications.

Code Comments

The textbook examples include comment sections at the top of each file, but for the most part do not include line by line comments. This was to make it easier to read the examples in the book but is not a good approach in practice. It is more standard to provide a small comment at the beginning of each code section to explain the purpose, or a longer comment when needed to describe a specific methodology, explain an approach, or indicate a section that needs testing or review. This is an important and expected procedure: it encourages a thoughtful approach to the coding process itself; provides a guide to another programmer who might be reviewing or testing (or even inheriting) your code. Well commented code also serves your own needs: code that you are familiar with today might not look at all familiar when you return to it at a later date.

(Remember that PHP Permits two type of comments: **Multi-line comments** begin with /* and end with */, while **line comments** begin with // and extend to the end of the current line.)

Location of Curly Braces

Curly braces { and } are used to indicate which statements belong to a particular block of code, for example to indicate the block of statements in the IF or ELSE sections of a selection structure, or the block of statements in a loop structure. When curly braces appear in the code examples in this textbook, the left and right braces are both aligned below the headings of the structure, and the statements enclosed in each block are indented, for example:

```
if ($hoursWorked > 40)
{
  $bonus = 50;
  print ("<p>You receive a $50.00 bonus this week.</p>");
}
else
{
  $bonus = 0;
  print ("<p>No bonus this week.<p>");
}
```

While this layout is used by many programmers, a more common approach is to type the beginning { curly brace on the same line as the heading, as follows:

```
if ($hoursWorked > 40) {
  $bonus = 50;
  print ("<p>You receive a $50.00 bonus this week.</p>");
}
else {
  $bonus = 0;
  print ("<p>No bonus this week.</p>");
}
```

Since the processor ignores white-space, either approach is acceptable. The decision was made to use the less common convention simply because aligning the { and } braces at the same indent level, below the block heading, seemed to provide beginning students with a clearer picture of the overall structure. It is important to be familiar with both conventions since the second approach will be seen more often when looking at code examples, not just in PHP but most languages.

HTML and PHP Files

Most of the applications in the samples and coursework folders were developed using pairs of .html files (to contain HTML forms) and .php files (to process the form input using a mix of HTML and PHP code). This was just a useful convention, used to more easily identify the names of related files in these folders, and to reduce any file-name ambiguity when completing exercises. There is no need for files that work together to be related by name, furthermore a file that contains only HTML code can be named with a .php extension, and many PHP coders follow this approach.

A single Web page may contain multiple forms, and each of these may reference a different .php file to process the form input (the .php file that is actually executed will depend on which form is submitted by the user). Furthermore, the use of two separate files to display and process a form is not a best practice for PHP coding. A better approach is to combine the code to display the form and the code to process the form in a single .php file. You can find a detailed explanation of this procedure in the "Web Sessions" section of Chapter 12.

Most real world PHP applications will consist of code that is assembled from any number of files with significant sections of code provided by include files (see Chapter 13). This reduces duplication of code since the code in include files can be included in any number of different PHP applications.

Multiple PHP Sections

The textbook PHP code examples almost all follow a format where a .php file contains an initial HTML section, followed by a PHP section, followed by a final HTML section. In fact, a PHP file can include any combination of HTML and PHP sections in any order. A PHP file may consist entirely of PHP code, and may even consist entirely of HTML code (no PHP code at all).

Usually a working program contains many small sections of PHP code interspersed with HTML code. This minimizes the need for the escape characters needed to display quotes and other special characters when the HTML code is generated by PHP print or echo statements. The only HTML text that must be generated inside PHP sections is text that includes values derived from PHP variables, functions, or expressions.

Program Variables and the $_POST Array

The textbook examples also followed a convention of creating a variable with the same name as an HTML form input box or other input element, if the variable was used to receive the value from the form element, for example: $hourlyWage = $_POST[‘hourly-Wage'];

This was just a textbook convention, to make it easier to see the relationship between the variable and the user’s form input; there is no PHP requirement to use the same name, for example, $wage = $_POST[‘hourlyWage']; is perfectly acceptable.

Furthermore it is not necessary to always assign the value from the $_POST array to a new variable. After all $_POST[‘hourlyWage'] is itself a variable (it is an element of the $_POST associative array). However it is fairly standard to assign $_POST array values to individual variables in order improve readability.

Naming Files, Variables, and Constants

When naming files for use with this textbook, the author has used all lower-case letters combined with numbers, and hyphens are used to represent spaces between English words.

The book uses camel case, or camelback, notation to name variables (for example $hourlyWage). This is the convention followed by programmers working in most current languages. Some PHP programmers prefer to use underscores instead of camelback notation when naming variables (for example $hourly_wage).

The usual convention when naming constant values is to use all upper-case letters, with underscores to represent spaces between English words.

Choosing Between print and echo

The textbook uses the PHP print statement throughout to generate HTML output. The PHP echo statement can be used just as well for this purpose. The two are almost identical in operation, and most programmers simply choose one over the other. For example:

```
print("Hello, ". $firstName." how are you today?");
```

could be written:

```
echo "Hello, ".$firstName." how are you today?";
```

Either **print** or **echo** statements can be used with or without parentheses. Users of the **echo** statement usually leave out the opening and closing parentheses since parentheses do not work when concatenation is included in an echo statement. Although parentheses are also not required when using the **print** statement, the textbook examples have followed this convention in order to more closely reflect the use of functions in most other languages to deliver output (functions always require parentheses).

Unlike print statements, the echo statement also allows a list of arguments (like a function), separated by commas, which means that a series of strings can be published in sequence, but this is not very different from using the concatenation operator to join strings and then publish them (there can some improved efficiency in cases of very large numbers of iterations).

Single or Double Quotes in PHP

Throughout this textbook, character strings are surrounded in **double quotes**. PHP allows either **double** or **single** quotes to be used around strings, as long as the same quote is used to indicate the start and end of each string, in other words "test" and 'test' are both acceptable but "test' is not. It can be useful to choose one of the other to avoid escaping quotes within a string, for example **"She's coding right now."**, or '** He said "I like to code". **'.

There are some significant differences to be aware of when choosing between single and double quotes. For example if a **variable** is included in a string enclosed in **double** quotes, the **value** of the variable will be used in the string. However if a variable is included in a string enclosed in **single** quotes, the variable **name** will be used in the string. Another difference applies when escape characters are included within a string: only the \' and \\ **escape characters** can be used in strings enclosed in single quotes, whereas **any** escape characters can be included in strings enclosed in double quotes. Here's an example to show the difference (assuming **$age** contains the value **24**):

```
print ("<p>He said \"yes, I'm $age years old\".</p>");
```

The use of **double** quotes generates <**p**>**He said "yes, I'm 24 years old**".</**p**>

print (' He said \"yes, I'm $age years old\". ');

The use of **single** quotes generates <**p**>**He said "yes, I'm $age years old**".</**p**>

Beware of "Smart" Quotes!

Note that, typographically, quotes can be "simple" (or "straight") quotes, where opening and closing quotes are the same (as used "here"), or "smart" quotes (as used "here"), where the opening and closing quotes are different from one another. **Text editors** use **straight** quotes but **word processors** by default usually use **smart** quotes, This is important because the PHP processor only recognizes **straight** quotes, so you will have problems if your code includes smart quotes. This can happen, for example, if you **paste** text that contains smart quotes from a word-processing document or some other source into your text editor.

The PHP Concatenation Operator

Unlike most programming languages, PHP requires the $ symbol as the first character of each variable name. This allows some unusual practices that would not work in other languages. For example, a PHP programmer can include direct references to variables inside character strings, using statements such as: **"You are $age years old."**.

The reference to $age within the character string is possible because the $ symbol indicates that $age is a variable that contains a value, and not simply the word "age". Other languages **always** require concatenation when a value stored in a variable is to be included in a character string. Concatenation of a simple variable is optional in PHP, but required when the string includes PHP expressions, function calls, and objects. For example **"You have 65 - $age years until retirement."** will not work, and should be written using the concatenation operator. **"You have ".(65 - $age)." years until retirement."** Similarly **print(" Your wages are number_format($wage, 2) ");** will not work and should be written as **print(" Your wages are ".number_format($wage, 2)." ");**

Some programmers prefer to **always** use the concatenation operator to combine variable values with strings since it makes code easier to understand and follows a similar syntax to other languages. For example, **"You are ".$age." years old." or "Your pay is $".$weeklyPay[$id]."."**

(Note that although concatenation is a standard feature of most languages, the period is not always used as the concatenation operator. Some languages use the + character as the concatenation operator, so the same string in, for example, a Java statement would be written as **"You are " + age + " years old."** Notice also that variables do not begin with a $ character in Java.)

PHP Shortcut Operators

In addition to the standard arithmetic operators, most programming languages provide **shortcut operators**. The operators are very efficient and commonly used when an operation is intended to increment, decrement, add to, subtract from, multiply, or divide the numeric value currently stored in a variable, with the result replacing the previous value in the variable. Chapter 5 introduced the increment and decrement operators, here is the complete list:

Instead of:	**Use the shortcut operator:**
`$count = $count + 1;`	`$count++;`
`$count = $count - 1;`	`$count—;`
`$value = $value + $number;`	`$value += $number;`
`$value = $value - $number;`	`$value -= $number;`
`$value = $value * $number;`	`$value *= $number;`
`$value = $value / $number;`	`$value /= $number;`
`$value = $value % $number;`	`$value %= $number;`

The PHP SWITCH Statement

Most languages provide a SWITCH statement that can be used as an alternative selection structure to chained IF..ELSE statements in cases where the possible actions are all based upon the value of a single variable.

For example a SWITCH structure could be used instead of multiple chained IF..ELSE structures to determine which month name to display based on the numeric value stored in the variable $month:

```
switch ($month)
{
  case 1: print("January"); break;
  case 2: print("February"); break;
  case 3: print("March"); break;
  case 4: print("April"); break;
  case 5: print("May"); break;
  case 6: print("June"); break;
  case 7: print("July"); break;
  case 8: print("August"); break;
  case 9: print("September"); break;
```

```
  case 10: print("October"); break;
  case 11: print("November"); break;
  case 12: print("December"); break;
  default: print("ERROR!"); break;
}
```

The variable to be tested (**$month**) appears in the switch statement heading, enclosed in parentheses. Each possible value of this variable is handled by a case statement inside the SWITCH structure. The processor tests the value in each **case** against the value stored in the variable. When a match is found, the statements following the relevant **case** are executed.

Each case may include any number of program statements and a **break** statement is added to break out of the switch statement and move on to the next code statement. If the **break** statement is **not** included at the end of a case, the statements in subsequent case statements will also be processed until a **break** statement is found. This feature allows multiple cases to use the same statements, for example:

```
switch ($month)
{
  case 2: print("This month has 28 or 29 days"); break;
  case 4:
  case 6:
  case 9:
  case 11: print("This month has 30 days"); break;
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12: print("This month has 31 days"); break;
  default: print("ERROR!"); break;
}
```

In this case, if **$month** has the value 2, only the **first** case is executed. If **$month** has the value 4, 6, 9, or 11, the **fifth** case is executed. If **$month** has the value 1, 3, 5, 7, 8, 10, or 12, the **twelfth** case is executed. If **$month** has any other value the **thirteenth** case is executed.

Another useful characteristic of omitting break statements from some cases is that this can allow a **cascade** of case actions to be executed since the statements in every case below the case that contained the correct value will be executed until a break is encountered. Here's an example where the cases are based on a movie rating between 5 and 1:

```
switch ($movieRating)
{
  case 5: print("TOP RATING!");
  case 4:
  case 3: print("Worth seeing!"); break;
  case 2:
  case 1: print("Don't bother!"); break;
}
```

Note that the first two cases do **not** include break statements. If the rating is 5, **"TOP RATING!"** will be printed, followed by **"Worth Seeing!"** since the first break occurs at the end of case 3. If the rating is 4 or 3,**"Worth Seeing!"** will be printed. If the rating is 2 or 1,**"Don't bother!"** will be printed. Note that in this example the **default** option is not included. This option is not needed if it is known that the variable being tested does not contain any value others than those listed.

In the previous examples, each case only required a **single** statement to be executed. In fact each case can include any number of statements and even other control structures. Here is another example where a switch statement is used to determine discounts and shipping costs as part of a customer billing calculation:

```
$unitCode = "12345";
$unitCost = 4.75;
$numItems = 10;
$subtotal = $numItems * $unitCost;

switch ($numItems)
{
case 1: $discountRate = 0;
  $shipping = 1.50;
  break;
case 2: $discountRate = 0;
  $shipping = 2.50;
  break;
case 3:
case 4: $discountRate = 0.05;
  $shipping = 3.50;
  break;
default: $discount = 0.1;
  if ($subTotal > 100)
    shipping = 0;
  else
     shipping = 3.50 + numItems - 4 * 0.50;
}
```

```
$discount = $subTotal * $discountRate;
$total = $subTotal - $discount + $shipping;
```

The PHP DO..WHILE Loop

Most languages provide an additional loop structure that was not described in the two loop chapters. This is the DO..WHILE or REPEAT—UNTIL loop (the specific structure depends on the programming language).

Do..WHILE and REPEAT..UNTIL loops are event-controlled loops, just like the WHILE loop. However the test that controls a WHILE loop appears at the **start** of the loop structure, and so a WHILE loop may execute 0 times. The test for DO..WHILE and REPEAT..UNTIL loops appears at the **end** of the loop structure, and so the instructions in these loops will always execute at least once.

Here is a real world example of a DO..WHILE loop:

```
DO
  Instruct your dog to "sit"
WHILE the dog refuses to sit
Give the dog a treat
```

Here the test is "the dog refuses to sit" and the loop repeats as long as this test is true. Since the test is at the end of the loop, the loop instruction will always be executed at least once. In this case you want to instruct the dog to sit at least once, so this is an more appropriate loop than a WHILE loop.

Here is the same example using a REPEAT..UNTIL loop:

```
REPEAT
  Instruct your dog to "sit"
UNTIL the dog sits
Give the dog a treat
```

The REPEAT..UNTIL structure is basically the same as the DO..WHILE except that the logic of the test is reversed. Instead of repeating **while** a condition is true, the loop repeats **until** a condition becomes true.

PHP uses the DO..WHILE statement structure rather than the REPEAT..UNTIL structure. Here is a PHP example of a DO..WHILE loop used to display the value of 2 raised to exponents between 0 and 5:

```
$exponent = 0;
$value = 2;
do
  $result = pow($value, $exponent);
  print("$value to the power of $exponent is $result<br>");
  $exponent = $exponent + 1;
while ($exponent <= 5);
```

Note that the test of this loop statement must be followed by a semi-colon, to indicate the end of the structure.

Note also that braces are **not** required between the **do** and the **while** since these two words are sufficient to tell the processor where the loop instructions begin and end.

Multi-Dimensional Arrays

Chapter 11 focused on the use of **one-dimensional** numerically-indexed arrays. Arrays can include any number of dimensions. For example, a **two-dimensional** array could store 3 scores for each of 5 students:

```
$scores[0] = array(90, 80, 92);
$scores[1] = array(82, 81, 83);
$scores[2] = array(87, 90, 84);
$scores[3] = array(78, 69, 73);
$scores[4] = array(89, 91, 92);
```

Here, the $scores array is an array that contains five elements indexed from 0 to 4. Each of these elements of the $scores array is **also** an array, each containing three **elements**, indexed from 0 to 2. To obtain the **second** score of the **third** student we must reference element 1 of the $scores[2] array as follows: $scores[2][1].

If you wish to use a **FOR** loop to find the average score of the fifth student, you can simply refer to the array named $scores[4] in your loop, and use the loop variable to reference the index of each element of this array:

```
$totalScore = 0;
for ($i = 0; $i < sizeof($scores[4]); $i = $i + 1)
{
  $totalScore = $totalScore + $scores[4][$i];
}
```

This loop will repeat for index positions 0 through 2 to add the values of each element of the array named $scores[4].

If you wish to use a FOR loop to find the average score for each of the five students, you can use a FOR loop nested inside **another** FOR loop:

```
for ($i = 0; $i < sizeof($scores); $i = $i + 1)
{
  $totalScore = 0;
  for ($j = 0; $j < sizeof($scores[$i]); $j = $j + 1)
  {
    $totalScore = $totalScore + $scores[$j][$i];
  }
  $averageScore = $totalScore / sizeof($scores[$i]);
  print("Average score for student $i is $averageScore");
}
```

Consider this code carefully. First note that the two loops must use different variables to count through the array indexes so that there is no conflict when the loops are processed. Here the outer loop uses **$i** as a counting variable, and the inner loop uses **$j**.

Next note that the outer loop is controlled by **$i** < **sizeof**(**$scores**). That's because we want this loop to repeat for each of the five elements of the $scores array.

Since the inner loop is located inside the outer loop, the inner loop will be processed **entirely** for every repetition of the outer loop. The inner loop is controlled by **$j** < **sizeof**(**$scores**[**$i**]). That's because we want this loop to repeat for each of the three elements in the array stored in $scores[$i]. Does that make sense?

Inside the inner loop, the **$totalScore** variable accumulates the values from each element of $scores[$i] and the average score for that student is calculated and displayed. Note that the $totalScore must be reset to **0** for each student each time the outer loop repeats, otherwise it would add the next student's scores to the previous total. Similarly, the average score must be calculated and displayed for each student before the outer loop moves on the next student. If you instead wanted to obtain the **overall** average of **all** the scores of **all** the students, you would initialise $totalScores to 0 **before** the outer loop, and only calculate and display the average **after** the outer loop has completed (the total would need to be divided by the total number of elements in all the arrays).

Ragged Arrays

Each element of a multi-dimensional array such as $scores contains an array. These inner arrays can actually contain different numbers of elements. These are then known as **ragged** arrays. For example consider a 2-dimensional array that contains donations that have been received by five different participants in a fund-raiser:

```
$donations[0] = array(25.00, 35.00, 25.00, 15.00);
$donations[1] = array(45.00, 55.00, 75.00, 25.00, 25.00,
35,00, 50.00);
$donations[2] = array(25.00, 45.00);
$donations[3] = array(15.00);
$donations[4] = array(65.00, 35.00, 35.00, 20.00, 30.00);
```

Here is another example that demonstrates the usefulness of the **sizeof**() function, since this will tell us the number of elements in any array. If you needed to find the total of all donations, you can use the same pair of nested loops as in the previous example:

```
$total = 0;
for ($i = 0; $i < sizeof($donations); $i = $i + 1)
{
  for ($j = 0; $j < sizeof($donations[$i]); $j = $j + 1)
  {
    $total = $total + $donations[$j][$i];
  }
}
```

The outer loop will repeat 5 times since that is the number of elements in the $donations array. However the inner loop will repeat a different number of times for each repetition of the outer loop. That's because the inner loop is controlled by sizeof($donations[$i]), and this value will be different for each element of $donations.

Multi-Dimensional Associative Arrays

Associative arrays can also be multi-dimensional, and we can even create multi-dimensional arrays that combine associative arrays with numerically-indexed arrays. For example a 2-dimensional array could be used to store the number of cars sold by an auto sales business every quarter for 3 years:

```
$carSales["2005"] = array (121, 174, 165, 112);
$carSales["2006"] = array (134, 143, 146, 121);
$carSales["2007"] = array (101, 203, 149, 101);
```

Here, the **$carSales** array is an associative array that contains three elements ("2005", "2006", and "2007"). Each of these elements contains a numerically-indexed array of four numbers. To obtain the number of cars sold in the 3rd quarter of 2006 we can reference the $carSales["2006"] array as follows: **$carSales["2006"]**[2].

If you wish to use a FOR loop to find the total sales for 2007, you can simply refer to that array in your loop:

```
$total2007 = 0;
for ($i = 0; $i < sizeof($carSales["2007"]); $i = $i + 1)
{
  $total2007 = $total2007 + $carSales["2007"][$i];
}
```

If you wish to use a FOR loop to find the total sales for ALL THREE years, you can use a FOR loop nested inside a FOREACH loop (the use of FOREACH loops with associative arrays is covered in Chapter 12):

```
$total = 0;
foreach ($carSales as $nextYear)
{
  for ($i = 0; $i < sizeof($nextYear); $i = $i + 1)
  {
    $total = $total + $nextYear[$i];
  }
}
```

In this case, each time the FOREACH loop repeats, the next element of $carSales is assigned to $nextYear. Each of these elements is an array, so the nested FOR loop processes the array of the $carSales element that is currently stored in $nextYear, adding every value to the total. When the FOR loop completes, control returns to the FOREACH loop which assigns the next element to $nextYear, and so on.

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 data types:

Integer: A whole number (**int** data type), such as -100, -1, 0, 1, 25, or 7388. Note that integers can be expressed in **decimal** (base 10), **octal** (base 8), **hexadecimal** (base 16), or **binary** (base (2)

Floating point number: A decimal number (**float** data type), such as 5.24 or 123.456789. Float values can also be expressed as **exponents**, for example 1.4e27.

Boolean: A TRUE or FALSE value.

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 " This is a paragraph. ".

Object: An object data type is an instance of an OO class.

Array: An array is a variable that contains multiple values, referenced by an index.

Null: a simple data type with only one value (NULL), which is no value. It can be used to de-value a variable of any data type, or to test if a variable of any data type has been has not been assigned a value.

Resource: this special data type acts as a reference to an external source of data such as a file or database. For example **fopen**() returns a Resource data type that references the file stream. Similarly **mysqli_connect**() returns a Resource data type that references the link to the SQL database. In both cases the reference that is returned is assigned to a variable that is then included in subsequent function calls that are intended to work with the data source, for example fgets() and mysqli_query().

Unlike most languages, PHP allows you to use variables without first defining their data type. PHP evaluates each expression by **context**, and determines which data type is appropriate at the time a value is assigned, and again when a value is used in an expression.

To ensure that a value is being handled as a specific data type you can **type cast** the variable. For example you can use the **int** cast operator to cast a string or float value as an integer (note that if a float value is converted to an integer it will be truncated, so any decimal part will be lost). Here is a simple example, where we cast the value stored in $someValue to an integer value and store this in **$intValue**:

```
$intValue = (int) $someValue;
```

You need to be careful to choose between typecasting a single value or the value of an expression. For example compare these two statements:

```
$result = (int) ($num1 * $num2);
$result = (int) $num1 * (int) num2;
```

In the first case the result of the entire expression is cast as an integer, so if **$num1** contains 2.5 and **$num2** contains 4.5, these two values are multiplied to obtain **11.25**, and then this value is cast as an integer to obtain **11**, which is stored in **$result**.

In the second case, the values of **$num1** and **$num2** are both cast as integers **before** the expression is evaluated, so if $num1 contains 2.5 and $num2 contains 4.5, the value of $num1 is cast as an integer to obtain 2, then the value of $num2 is cast as an integer to obtain **4**, and then these values are multiplied to obtain **8**, which is stored in **$result**.

In some circumstance you may need to ensure that a variable that contains a string version of a number will treat it as a numeric value, for example if an hourly wage of 15.75 was read from a file and stored as "15.75", you can ensure that it will stored as a float value as follows:

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

And to convert a float value to a string:

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

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

PHP allows the following casts: (**integer**) or (**int**), (**boolean**) or (**bool**), (**float**) or (**double**), (**string**), (**array**), (**object**), or (**unset**), which casts to NULL.

PHP Tools to Validate and Sanitize Data

Data sanitizing and validation are critical components of any real world application that must work with data, whether directly received from the user or from files, databases, or other sources. **Sanitizing** refers to the process of **changing** data (removing or rendering harmless) in order to protect against corrupt or malicious content, while **validation** refers to **testing** data, to be sure that input meets requirements. All input data should be sanitized and validated, in that order.

Appendix E provides an important introduction to security and validation, and is a **must-read** if you are planning to develop your applications online. The appendix describes the use of a number of PHP functions that will help to keep your applications secure, and to validate your application input. This appendix also includes a brief introduction to regular expressions (**regex**).

PHP Date and Time Functions

The need to process or display dates and times is an important requirement for many applications. Your computer stores dates and times in a special **timestamp** format, which defines a specific time as the number of seconds since midnight Greenwich Mean Time on January 1, 1970 (known as the beginning of the Unix epoch). Fortunately PHP provides objects and functions that make it easy to work with timestamps, using our usual measuring system (years, months, days, hours, minutes, and seconds). This short introduction will provide a few examples just to get you started.

The PHP **date**() function will return a date and time string, derived from any timestamp, formatted to your specific requirements. The function has two parameters: the first is **required** and consists of a character string that combines different character codes to define the required date format; the second is an **optional** timestamp that specifies a specific time (if a timestamp is not included, the date() function will use the timestamp of the current moment). To start with an example, **date("Y:m:d")** contains the format string **"Y:m:d"**, which uses three format codes: **'Y'** indicates the **year**, expressed as 4 digits, **'m'** indicates a 2-digit **month** number, and **'d'** a 2-digit **day** number. The two **colons** are also to be included in the formatted date string. Since there is no second parameter, the current time stamp is assumed. If today's date is **April 14, 2021**, **date("Y:m:d")** would return **"2021:04:14"**.

The format string can be constructed with any combination of format codes, in any order, to construct the desired output, and can also include any of the following separators: ';', ':', '/', '.', ',', '-', '(', ')', as well as simple **spaces** ' ':

d - Day of the month as 2 digits (01, 02,)
j - Day of the month as 1 or 2 digits (1, 2, .., 10, 11, ..)
D – Three-letter abbreviation of the day ("SUN", "MON", ..)
S - The English suffix ("st", "nd", "rd", or "thv). This can used after j, for example to create "1st" or "14th")
l (lowercase 'L') the full day name ("Sunday", "Monday"..)
F – The full month name ("January", "February", ..)
m – The 2-digit month number (01, 02, ..)
n – The 1-digit month number (1, 2, .., 10, 11, ..)
M - Three-letter abbreviation of the month ("JAN", "FEB",)
L – returns 1 if it is a leap year, 0 if it is not
Y – The year as 4 digits (2021, 2022, ..)
y - The year as 2 digits (21, 22, ..)
a - Lowercase am or pm
A - Uppercase AM or PM
g or h - 12-hour format of an hour (1, .., 12)
G or H - 24-hour format of an hour (0, .., 23)
i – 2 digit minutes (00, 01, ..)
s – 2-digit seconds (00, 01, ..)

The following examples demonstrate just a few ways in which these codes can be combined for different purposes. Your **samples** folder includes **date-time.php**, which includes these and other examples.

To display the time of day (hour and minutes) in 12-hour format followed by "am" or "pm" (for example **"The time is 7:45am"**):

```
print("<p>The time is ".date("g:ia")."</p>");
```

To display the day of the week (for example **"Today is Wednesday"**):

```
print("<p>Today is ".date("l")."</p>");
```

To provide a morning wake up message consisting of the time, day of the week, name of the month, followed by the day and a suitable suffix ("st", "rd", or "th"), and the year (for example **"Good morning! It's 7:45am, Wednesday April 14th, 2021"**):

```
print("<p>Good morning! It's ". date("g:ia, l F jS,
  Y")."</p>");
```

A **copyright** notice is often a necessary feature on a Web page. A string returned by the date() function can be concatenated with other text to ensure that the notice always displays the current year, for example **"Copyright © 2021"**. In the example below the **"©"** part of the string is an HTML **entity** that will display the copyright symbol, entities are explained in chapter 17.

```
print("<p>Copyright &copy; ".date("Y")."</p>");
```

Here's another interesting use of a date string. It is often useful to include a date as part of the name of a data file (such as those rainfall files you worked with in Chapter 6). This is useful for reference, and a **"year-month-day"** format will also ensure that similarly named files will be listed in date order when sorted by name. Here is a statement that creates a file name by concatenating a formatted date with a filename that begins **"rainfall-"** and ends with the extension **".txt"**, for example **"rainfall-2021-04-21.txt"**:

```
$fileName = "rainfall-".date("Y-m-d").".txt";
print("<p>The filename is $fileName</p>");
```

One way to create any timestamp is to use the PHP **mktime**() function, which takes 6 parameters (hour, minute, second, month, day, year). To create a timestamp for **April 14, 2021**:

```
mktime (0,0,0,4,14,2021);
```

You could use this for example to check the **day** of your birthday:

```
$birthday = mktime (0,0,0,4,14,2021);
print("<p>My birthday will be on a ".date("l",
  $birthday)."</p>");
```

You can easily improve this code so it will continue to work over time, for any year, by first using the **date**() function to obtain the current year, then including this as the year parameter in the call to **mktime**():

```
$year = date("Y");
$birthday = mktime (0,0,0,4,14,$year);
print("<p>My birthday will be on a ".date("l",
$birthday)."</p>");
```

You will want to extend your familiarity with PHP date and time functions, and also calendar functions. The **w3schools** site provides a useful introduction and code samples.

Standard PHP Functions

PHP provides many useful functions for a wide range of common programming tasks. We have used a few of these in this book. For an exhaustive list of PHP functions see: **https://php.net/manual/en/funcref.php**.

This list is quite overwhelming! It is often more useful to see a list of functions, organized by category (for example math functions, string functions, array functions, etc). A list of this kind can be found at the **w3schools** site (see the additional reference section at the end of this chapter).

To get you started, on the following pages you will find lists of the more commonly used PHP functions, by category. These include the functions that have been introduced in this book as well as many more, with short descriptions. You are encouraged to go online and research the use of any of these functions that interest you or that meet your requirements.

First, 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
is_numeric()	checks whether a variable contains a number
file_exists()	checks whether a file exists before trying to open a file
define()	defines a constant variable (a variable that cannot be changed)
phpinfo()	displays in-depth information about your PHP installation

isset() checks whether a variable contains a value

unset() destroys a variable

Standard PHP Array Functions

Here are some commonly used array-processing functions:

array() creates a new array

array_fill() fills an array with values

array_key_exists() checks if a certain key exists in the array

array_keys() returns the key values of an array

array_merge() merges one or more arrays into a single array

array_pop() removes the last element from an array

array_push() adds one or more elements to the end of an array

array_reverse() returns an array with the values in reverse order

array_search() searches an array for a value and returns the key

array_shift() removes the first element from an array, and returns the value of the removed element

array_slice() returns specific parts of an array

array_sum() returns the sum of the values in an array

array_unique() removes duplicate values from an array

array_unshift() adds one or more elements to the beginning of an array

arsort() sorts an associative array in descending order, based on the values

asort() sorts an associative array in ascending order, based on the values

count() returns the number of elements in an array, same as sizeof()

krsort() sorts an associative array in descending order, based on the key values

ksort() sorts an associative array in ascending order, based on the key values

list() assigns values to variables from an array

rsort() sorts values in an indexed array in descending order

sizeof() returns the number of elements in an array, same as count()

sort() sorts values in an indexed array in ascending order

Standard PHP File Functions

Be careful how you use these file functions. Some have the ability to overwrite or delete files or folders!

basename() returns the filename from a complete path name
copy() copies a file
dirname() returns the folder name from a complete path name
disk_free_space() returns the available space
disk_total_space() returns the total size
fclose() closes a file
feof() tests for end-of-file (EOF)
fflush() flushes buffered output to an open file
fgets() returns a string containing the text from the next line in a file
file_exists() checks whether or not a file or folder exists
file_get_content() reads an entire file into a string
file_put_contents writes a string to a file
fopen() opens a file or url
fputs() writes a string to a file, same as fwrite()
fwrite() writes a string to a file, same as fputs()
is_executabl() checks if a file is executable
mkdir() creates a new folder
rename() renames a file or folder
rmdir() removes an folder (if it is empty)
stat() returns information about a file
umask() changes file permissions
unlink() deletes a file

Standard PHP Math Functions

abs() returns the absolute value of a number
acos() returns the arccosine of a number
asin() returns the arcsine of a number
atan() returns the arctangent of a number
bindec() converts a binary number to a decimal number

ceil()	returns the value of a number rounded upwards to the nearest integer
cos()	returns the cosine of a number
decbin()	converts a decimal number to a binary number
dechex()	converts a decimal number to a hexadecimal number
decoct()	converts a decimal number to an octal number
deg2rad()	converts a degree to a radian number
floor()	returns the value of a number rounded downwards to the nearest integer
hexdec()	converts a hexadecimal number to a decimal number
hypot()	returns the length of the hypotenuse of a right-angle triangle
is_finite()	returns true if a value is a finite number
is_infinite()	returns true if a value is an infinite number
is_nan()	returns true if a value is not a number
log()	returns the natural logarithm (base e) of a number
log10()	returns the base-10 logarithm of a number
max()	returns the number with the highest value of two specified numbers
min()	returns the number with the lowest value of two specified numbers
octdec()	converts an octal number to a decimal number
pi()	returns the value of pi
pow()	returns the value of x to the power of y
rad2deg()	converts a radian number to a degree
rand()	returns a random integer
round()	rounds a number to the nearest integer
sin()	returns the sine of a number
sinh()	returns the hyperbolic sine of a number
sqrt()	returns the square root of a number
tan()	returns the tangent of an angle

Standard PHP String Functions

bin2hex()	converts a string of ascii characters to hexadecimal values
chr()	returns a character from a specified ascii value
count_chars()	counts the number of times an ascii character occurs within a string
echo()	outputs strings

explode() parses a string into an array of substrings
fprintf() writes a formatted string to an output stream
html_entity_decode() converts html entities to characters
htmlentities() converts characters to html entities
htmlspecialchars_decode() converts specific html entities to characters
htmlspecialchars() converts specific characters to html entities
ltrim() removes whitespace from the left side of a string
money_format() returns a string formatted as currency
number_format() formats a number according to a format specification
ord() returns the ascii value of the first character in a string
print() outputs a string
printf() outputs a formatted string
rtrim() removes whitespace from the right side of a string
str_word_count() counts the words in a string
strcasecmp() compares two strings (case-insensitive)
strcmp() compares two strings (case-sensitive)
strlen() returns the length of a string
strrev() reverses a string
substr() returns a part of a string
substr_count() counts the number of times a substring occurs in a string
substr_replace() replaces part of a string with another string
trim() removes whitespace from both sides of a string
wordwrap() wraps a string to a specified number of characters

A lengthy list of standard PHP mysql functions can be found at

```
https://www.w3schools.com/php/php_ref_mysqli.asp
```

A list of standard PHP date and time functions and constants can be found at:

```
https://www.w3schools.com/php/php_ref_date.asp
```