1

The Basics of PHP

PHP is the most popular web-development language in the world. According to estimates compiled in April 2004, there are over fifteen million unique domains—and almost two million unique IPs—on the World Wide Web that reside on servers where PHP is supported and used.

The term "PHP" is actually a "recursive acronym" that stands for *PHP: Hypertext Preprocessor*. It might look a bit odd, but it is quite clever, if you think of it. PHP is a "scripting language"—a language intended for interpretation and execution rather than compilation such as, for example, C.

The fact that PHP is interpreted and not compiled, however, does not mean that it is incapable of meeting the demands of today's highly intensive web environments—in fact, a properly written PHP script can deliver incredibly high performance and power.

Terms You'll Need to Understand

- Language and Platform
- Language construct
- Data type
- Opening and closing tags
- Expression
- Variable
- Operation and operator precedence
- Conditional structures
- Iteration and Loops
- Functions
- Variable variables and variable functions

Techniques You'll Need to Master

- Creating a script
- Entering PHP mode
- Handling data types
- Type casting and type juggling
- Creating statements
- Creating operations and expressions
- Writing functions
- Handling conditional statements
- Handling loops

Language and Platform

The two biggest strengths of PHP are its simplicity and the incredible set of functionality that it provides. As a language, it incorporates C's elegant syntax without the hassle of memory and pointer management, as well as Perl's powerful constructs—without the complexity often associated with Perl scripts.

As a platform, PHP provides a powerful set of functions that cover a multitude of different needs and capabilities. Programmers who work on commercial platforms such as Microsoft ASP often marvel at the arsenal of functionality that a PHP developer has at his fingertips without the need to purchase or install anything other than the basic interpreter package. What's more, PHP is also *extensible* through a set of well-defined C APIs that make it easy for anyone to add more functionality to it as needed.

You have probably noticed that we have made a distinction between "language" and "platform." By the former, we mean PHP proper—the body of syntactical rules and constructs that make it possible to create a set of commands that can be executed in a particular sequence. The latter, on the other hand, is a term that we use to identify those facilities that make it possible for the language to perform actions such as communicating with the outside, sending an email, or connecting to a database.

The certification exam verifies your knowledge on both the language and the platform—after all, a good programmer needs to know how to write code *and* how to use all the tools at his disposal. Therefore, it is important that you acquaint yourself with both aspects of PHP development in order to successfully pass the exam.

Getting Started

The basic element of a PHP application is the *script*. A PHP script contains a number of commands that the PHP interpreter reads, parses, and executes.

Because PHP is designed to manipulate text files—such as HTML documents—and output them, the process of mixing hard-coded textual content and PHP code is facilitated by the fact that unless you tell it otherwise, the PHP interpreter considers the contents of the script file as plain text and outputs them as they are.

It's only when you explicitly indicate that you are embedding PHP code inside a file that the interpreter goes to work and starts parsing and executing commands. This is done by using a special set of *opening* and *closing tags*. In part because of historical reasons and in order to promote maximum flexibility, PHP supports three different sets of tags:

- PHP opening (`<?php`) and closing (`?>`) tags
- HTML-style tags (`<script language="php">` and `</script>`)
- "Short" tags: `<?` and `?>`
- "ASP-style" tags: `<%` and `%>`

The full PHP tags are always available to a script, whereas short tags and ASP-style tags might or might not be available to your script, depending on how the particular installation of the PHP interpreter used to execute it is configured. This is made necessary by the fact that short tags can interfere with XML documents, whereas ASP-style tags can interfere with other languages that can be used in conjunction with PHP in a chain of preprocessors that manipulate a file multiple times before it is outputted.

Let's take a look at the following sample PHP script:

```
<html>
<head>
<title>
    This is a sample document
</title>
<body>
    <?php
        echo 'This is some sample text';
    ?>
</body>
</html>
```

As you can see, this document looks exactly like a normal HTML page until the interpreter hits the `<?php` tag, which indicates that text following the tag should be interpreted as PHP commands and executed.

Right after the opening tag, we see a line of PHP code, which we'll examine in detail later on, followed by the `?>` closing tag. After the interpreter sees the closing tag, it stops trying to parse PHP commands and simply outputs the text as it appears without any change. Note that, as long as your copy of PHP has been configured to support more than one type of opening and closing tags, you *can* mix and match opening and closing tags from different families—for example, `<?php echo 'a' %>` would be a valid script. From a practical perspective, however, doing so would be pointless and confusing—definitely not a good programming practice.

Naturally, you can switch between plain-text and PHP execution mode at any point during your script and as long as you remember to balance your tags—that is, to close any tags you open, you can switch an arbitrary number of times.

The special <?= ?> tags

A special set of tags, `<?=` and `?>`, can be used to output the value of an expression directly to the browser (or, if you're not running PHP in a web environment to the standard output). They work by forcing PHP to evaluate the expression they contain and they output its value. For example,

```
<?= "This is an expression" ?>
```

Scripts and Files

It's important to note that there isn't necessarily a one-to-one correspondence between scripts and files—in fact, a script could be made up of an arbitrary number of files, each containing one or more portions of the code that must be executed. Clearly, this means that you can write portions of code so that they can be used by more than one script, such as library, which makes a PHP application even more flexible.

The inclusion of external files is performed using one of four different language constructs:

- `include`, which reads an external file and causes it to be interpreted. If the interpreter cannot find the file, it causes a warning to be produced and does *not* stop the execution of the script.
- `require`, which differs from `include` in the way it handles failure. If the file to be included cannot be found, `require` causes an error and stops the script's execution.
- `require_once` and `include_once`, which work in a similar way to `require` and `include`, with one notable difference: No matter how many times you include a particular file, `require_once` and `include_once` will only read it and cause it to be interpreted once.

The convenience of `require_once` and `include_once` is quite obvious because you don't have to worry about a particular file being included more than once in any given script—which would normally cause problems because everything that is part of the file would be interpreted more than once. However, generally speaking, situations in which a single file is included more than once are often an indicator that something is not right in the layout of your application. Using `require_once` or `include_once` will deprive you of an important debugging aid because you won't see any errors and, possibly, miss a problem of larger magnitude that is not immediately obvious. Still, in some cases there is no way around including a file more than once; therefore, these two constructs come in very handy.

Let's try an example. We'll start with a file that we will call `includefile.php`:

```
<?php

echo 'You have included a file';

?>
```

Next, we'll move on to `mainfile.php`:

```
<?php

include 'includefile.php';

echo 'I should have included a file.';

?>
```

If you make sure to put both files in the same directory and execute `mainfile.php`, you will notice that `includefile.php` is included and executed, causing the text `You have included a file` to be printed out.

Note that if the two files are not in the same folder, PHP will look for `includefile.php` in the *include path*. The include path is determined in part by the environment in which your script is running and by the php.ini settings that belong to your particular installation.

Manipulating Data

The manipulation of data is at the core of every language—and PHP is no exception. In fact, handling information of many different types is very often one of the primary tasks that a script must perform; it usually culminates with the output of some or all the data to a device—be it a file, the screen, or the Internet.

When dealing with data, it is often very important to know what *type* of data is being handled. If your application needs to know the number of times that a patient has visited the hospital, you want to make sure that the information provided by the user is, indeed, a number, and an integer number at that because it would be difficult for anybody to visit the hospital 3.5 times. Similarly, if you're asking for a person's name, you will, at the very least, ensure that you are not being provided with a number, and so on.

Like most modern languages, PHP supports a variety of data types and is capable of operating them in several different ways.

Numeric Values

PHP supports two numeric data types: integer and real (or floating-point). Both types correspond to the classic mathematical definition of the types—with the exception that real numbers are stored using a mechanism that makes it impossible to represent certain numbers, and with a somewhat limited precision so that, for example, 2 divided by 3 is represented as 0.66666666666667.

Numeric values in base 10 are represented only by digits and (optionally) a dot to separate units from fractional values. The interpreter does not need commas to group the integer portion of the value in thousands, nor does it understand it, producing an error if you use a format such as 123,456. Similarly, the European representation (comma to separate the fractional part of the value from the integer one) is not supported.

As part of your scripts, you can also enter a value in hexadecimal (base 16) representation—provided that it is prefixed by `0x`, and that it is an integer. Both uppercase and lowercase hexadecimal digits are recognized by the interpreter, although traditionally only lowercase ones are actually used.

Finally, you can represent an integer value in octal (base 8) notation by prefixing it with a single zero and using only digits between 0 and 7. Thus, the value `0123` is *not* the same as `123`. The interpreter will parse `0123` as an octal value, which actually corresponds to 83 in decimal representation (or `0x53` in hexadecimal).

String Values

Although we often think of strings as pieces of text, a string is best defined as a collection of bytes placed in a specific order. Thus, a string *could* contain text—say, for example, a user's first and last name—but it could also contain arbitrary binary data, such as the contents of a JPEG image of a MIDI file.

String values can be declared using one of three methods. The simplest one consists of enclosing your string in single quotes:

```
'This is a simple string'
```

The information between the quotes will be parsed by the interpreter and stored without any modification in the string. Naturally, you can include single quotation marks in your string by "escaping" them with a backslash:

```
'He said: \'This is a simple string\''
```

And this also means that, if you want to include a backslash, you will have to escape it as well:

```
'The file is in the c:\\test directory'
```

Another mechanism used to declare a string uses double quotation marks. This approach provides a bit more flexibility than the previous one, as you can now include a number of special characters through specific *escape sequences*:

- `\n`—A line feed
- `\r`—A carriage return
- `\t`—A horizontal tab
- `\\`—A backslash
- `\"`—A double quote
- `\`*nnn*—A character corresponding to the octal value of *nnn* (with each digit being between 0 and 7)
- `\x`*nn*—A character corresponding to the hexadecimal value of *nn*

Double-quote strings can also contain carriage returns. For example, the following strings are equivalent:

```
"This\nis a string"
"This
is a string"
```

The final method that you can use to declare a string is through the *heredoc* syntax:

```
<<<ENDOFTEXT
My text goes here.
More text can go on another line.
You can even use escape sequences: \t
ENDOFTEXT;
```

As you can see, the `<<<` heredoc tag is followed by an arbitrary string of text (which we'll call the *marker*) on a single line. The interpreter will parse the contents of the file as a string until the marker is found, on its own, at the beginning of a line, followed by a semicolon. Heredoc strings can come in handy when you want to embed large amounts of text in your scripts—although you can sometimes achieve a similar goal by simply switching in and out of PHP mode.

Boolean Values

A Boolean value can only be `True` or `False`. This type of value is generally used in Boolean expressions to change the flow of a script's execution based on certain conditions.

Note that, although PHP defines `True` and `False` as two valid values when printed, Boolean values are always an empty string (if false) or the number 1 (if true).

Arrays

Arrays are an *aggregate* value—that is, they represent a collection of other values. In PHP, arrays can contain an arbitrary number of elements of heterogeneous type (including other arrays). Each element is assigned a *key*—another scalar value used to identify the element within the array. You'll find this particular data type discussed in greater detail in Chapter 4, "Arrays."

Objects

Objects are self-contained collections of code and data. They are at the core of object-oriented programming and can provide a very valuable tool for creating solid, enterprise-level applications. They are described in Chapter 2, "Object-Oriented PHP."

The NULL Data Type

It is sometimes important to indicate that a datum has "no value". Computer languages need a special value for this purpose because even zero or an empty string imply that a value and a type have been assigned to a datum.

The NULL value, thus, is used to express the absence of *any* type of value.

Resources

A resource is a special type of value that indicates a reference to a resource that is external to your script and, therefore, not directly accessible from it.

For example, when you open a file so that you can add contents to it, the underlying code actually uses the operating system's functionality to create a file descriptor that can later be used for manipulating the file. This description can only be accessed by the functionality that is built into the interpreter and is, therefore, embedded in a resource value for your script to pass when taking advantage of the proper functionality.

Identifiers, Constants, and Variables

One of the most important aspects of any language is the capability to distinguish between its various components. To ensure that the interpreter is capable of recognizing each token of information passed to it properly, rules must be established for the purpose of being capable to tell each portion apart from the others.

In PHP, the individual tokens are separated from each other through "whitespace" characters, such as the space, tab, and newline character. Outside of strings, these characters have no semantic meaning—therefore, you can separate tokens with an arbitrary number of them. With one notable exception that we'll see in the next section, all tokens are not case sensitive—that is, `echo` is equivalent to `Echo`, or even `eCHo`.

Identifiers, which, as their name implies, are used as a label to identify data elements or groups of commands, must be named as follows:

- The first character can either be a letter or an underscore.
- Characters following the second can be an arbitrary combination of letters, digits, and underscores.

Thus, for example, the following are all valid identifiers:

- `__anidentifier`
- `yet_another_identifier___`
- `_3_stepsToSuccess`

Variables

As you can imagine, a language wouldn't be very useful if all it could deal with were immediate values: Using it would be a bit like buying a car with no doors or windows—sure, it can run fast, but to what purpose?

Similar to almost every computer language, PHP provides a facility known as a "variable" capable of containing data. PHP variables can contain one value at a time (although that value could, for example, be an array, which itself is a container for an arbitrary number of other values).

Variables are identifiers preceded by a dollar sign (`$`). Therefore, they must respect all the rules that determine how an identifier can be named. Additionally, variable names are case sensitive, so `$myvar` *is* different from `$MyVar`.

Unlike other languages, PHP does not require that the variables used by a script be declared before they can be used. The interpreter will create variables as they are used throughout the script.

Although this translates in a high degree of flexibility and generally nimbler scripts, it can also cause plenty of frustration and security issues. A simple spelling mistake, for example, could turn a reference to `$myvar` to, say, `$mvvar`, thus causing your script to reference a variable that doesn't exist. Similarly, if the installation of PHP that you are running has `register_globals` set to true, a malicious user will be able to set variables in your script to arbitrary values unless you take the proper precautions—more about that later in this chapter.

Variable Substitution in Strings

Both the double-quote and heredoc syntaxes support the ability to embed the value of a variable directly in a string:

```
"The value of \$a is $a"
```

In the preceding string, the second instance of `$a` will actually be replaced by the value of the `$a` variable, whereas the first instance will not because the dollar sign is escaped by a backslash.

For those cases in which this simple syntax won't work, such as when there is no whitespace between the name of the variable whose value you want to extract and the remainder of the string, you can forcefully isolate the data to be replaced by using braces:

```
<?

$thousands = 100;

echo "There are {$thousands}000 values";

?>
```

Statements

A statement corresponds to one command that the interpreter must execute. This could be an expression, a call to another block of code, or one of several constructs that PHP defines. For example, the `echo` construct causes the value of an expression to be sent to the script's output device.

Statements always end in a semicolon—if they don't, the system will output a parsing error.

Constants

As their name implies, constants are data holders whose type and value doesn't change.

A constant is create by using the `define()` construct. Here's an example:

```
<?php

define ("SOME_CONSTANT", 28);

echo SOME_CONSTANT;

?>
```

As you can see, `define()` takes two parameters; the first, a string, indicates the name of the constant, whereas the second indicates its value. After you have defined a constant, you can use it directly from your code, as we have done here. This means that although, in theory, you can define a constant with an arbitrary name, you will only be able to use it if that name follows the identifier naming rules that we discussed in the previous sections.

Operators

Variables, constants, and data types are not very useful if you can't combine and manipulate them in a variety of ways. In PHP, one of these ways is through *operators*.

PHP recognizes several classes of operators, depending on what purpose they are used for.

The Assignment Operator

The assignment operator `=` is used to assign a value to a variable:

```
$a = 10;
$c = "Greetings Professor Faulken";
$test = false;
```

It's very important to understand that, by default, variables are assigned *by value*. This means that the following

```
$a = $b
```

Assigns the *value* of `$b` to `$a`. If you change `$b` after the assignment has taken place, `$a` will remain the same. This might not always be what you actually want to happen—you might need to link `$a` and `$b` so that a change to the latter is also reflected in the latter. You can do so by assigning to `$a` *a reference* to `$b`:

```
$a = &$b
```

Any change to `$b` will now also be reflected in `$a`.

Arithmetic Operators

Perhaps the class of operators that most newcomers to PHP most readily identify with is the one that includes arithmetic operations. These are all part of binary operations (meaning that they always include two operators):

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)

Operations are written using the infix notation that we are all used to. For example,

```
5 + 4
2 * $a
```

Keep in mind that the modulus operation works a bit different from the typical mathematical operation because it returns a signed value rather than an absolute one.

PHP also borrows four special incrementing/decrementing operators from the C language:

- The prefix incrementing operator ++ increments the value of the variable that succeeds it, and then returns its new value. For example, `++$a`
- The postfix incrementing operator ++ returns the value of the variable that precedes it, and then increments its value. For example, `$a++`
- The prefix decrementing operator – decrements the value of the variable that succeeds it, and then returns its new value. For example, `–$a`
- The postfix decrementing operator – returns the value of the variable that precedes it, and then decrements its value. For example, `$a–`

The difference between the prefix and postfix version of the operators is sometimes difficult to grasp, but generally speaking is quite simple: The prefix version changes the value of the variable first, and then returns its value. This means that if the value of `$a` is `1`, `++$a` will first increment `$a` by one, and then return its value (which will be `2`). Conversely, the postfix version returns the value first and then modifies it—so, if `$a` is `1`, `$a++` will first return 1 and then increment `$a` to `2`.

Unary incrementing and decrementing operations can be extremely helpful because they enable for the modification of a variable in an atomic way and can easily be combined with other operations. However, this doesn't mean that they should be abused, as they can make the code more difficult to read.

Bitwise Operators

This class of operators manipulates the value of variables at the bit level:

- The bitwise AND (&)operation causes the value of a bit to be set if it is set in both the left and right operands. For example, `1 & 1 = 1`, whereas `1 & 2 = 0`.
- The bitwise OR (|) operation causes the value of a bit to be set if it is set in either the left or right operands. For example, `1 | 1 = 1` and `1 | 2 = 3`.
- The bitwise XOR (^) operation causes the value of a bit to be set if it is set in either the left or right operands, but not in both. For example, `1 ^ 1 = 0`, `1 ^ 0 = 1`.
- The bitwise NOT (~)operation causes the bits in its operand to be reversed—that is, set if they are not and unset otherwise. Keep in mind that if you're dealing with an integer number, *all* the bits of that integer number will be reversed providing a value that you might not expect. For example, on a 32-bit IBM platform, where each integer is represented by a single 32-bit value, `~0 = -1`, because the integer is signed.
- The bitwise >)>>)>left-shift (<<) and right-shift (>>) operators actually shift the bits of the left operands left or right by the number of positions specified by the right operand. For example, `4 >> 1 = 2`, whereas `2 << 1 = 4`. On integer values, shifting bits to the left by *n* positions corresponds to multiplying the left operand by 2^n, whereas shifting them right by *n* position corresponds to dividing the left operand by 2^n.

Remember that bitwise operations can only be performed on integer values. If you use a value of a different type, PHP will convert it for you as appropriate or output an error if it can't.

Error-control Operators

PHP is normally very vocal when it finds something wrong with the code it's interpreting and executing, outputting verbose and helpful error messages to mark the occasion. Sometimes, however, it's practical to ensure that no error be reported, even if an error condition occurs.

This can be accomplished by using the error-suppression operator `@` in front of the operation you want to perform. For example, the following would normally print an error because the result of a division by zero is infinity—a number that cannot be represented by any of the PHP data types. With the `@` operator, however, we can prevent the error from being printed out (but *not* from occurring):

```
<?php

@$a = 1 / 0;

?>
```

This operator can be very dangerous because it prevents PHP from notifying you that something has gone wrong. You should, therefore, use it only whenever you want to prevent errors from propagating to a default handler because you have a specialized code segment that you want to take care of the problem. Generally speaking, it's a bad idea to use this approach simply as a way to "silence" the PHP interpreter, as there are better ways to do so (for example, through error logging) without compromising its error reporting capabilities.

Note that not all types of errors can be caught and suppressed using the `@` operator. Because PHP first parses your script into an intermediate language that makes execution faster and then executes it, it won't be capable of knowing that you have requested error suppression until the parsing phase is over and the execution phase begins. As a result, syntax errors that take place during the parsing phase will always result in an error being outputted, unless you have changed your php.ini settings in a way that prevents all errors from being outputted independently from your use of the `@` operator.

String Operators

When it comes to manipulating strings, the only operator available is the concatenation operator, identified by a period (`.`). As you might imagine, it concatenates two strings into a third one, which is returned as the operation's result:

```
<?php

$a = 'This is string ';
$b = $a . "is complete now.";

?>
```

Comparison Operators

Comparison operators are used to determine the relationship between two operands. The result of a comparison is always a Boolean value:

- The == operator determines if two values are equal. For example, `10 == 10`
- The `!=` operator determines if two values are different. For example, `10 != 11`
- The < operator determines whether the left operand's value is less than the right operand's.
- The > operator)>)>) operator>determines whether the left operand's value is greater than the right operand's.
- The <= operator determines whether the left operand's value is less than or equal to the right operand's.
- The >= operator=)>=)>=) operator> determines whether the left operand's value is greater than the right operand's.

To facilitate the operation of comparing two values, PHP will "automagically" perform a set of conversions to ensure that the two operands being compared will have the same type.

Thus, if you compare the number 10 with the string `"10"`, PHP will first convert the string to an integer number and then perform the comparison, whereas if you compare the integer 10 to the floating-point number 11.4, the former will be converted to a floating-point number first.

For the most part, this feature of PHP comes in very handy. However, in some cases it opens up a few potentially devastating pitfalls. For example, consider the string `"test"`. If you compare it against the number 0, PHP will first try to convert it to an integer number and, because `test` contains no digits, the result will be the value `0`. Now, it might not matter that the conversion took place, but if, for some reason, you really needed the comparison to be between two numbers, you will have a problem: `"11test"` compared against `11` will return `True`—and that might not exactly be what you were expecting!

Similarly, a `0` value can give you trouble if you're comparing a number against a Boolean value because `False` will be converted to 0 (and vice versa).

For those situations in which both the type and the value of a datum are both relevant to the comparison, PHP provides two "identity" operators:

- The `===` operator determines whether the value *and the type* of the two operands is the same.
- The `!==` operator determines whether either the value *or the type* of the two operands is different.

Thus, while `10 == "10"`, `10 !== "10"`.

Logical Operators

Logical operators are often used in conjunction with comparison operators to create complex decision mechanisms. They also return a Boolean result:

- The AND operator (indicated by the keyword `and` or by `&&`) returns `True` if both the left and right operands cannot be evaluated to `False`
- The OR operator (indicated by the keyword `or` or by `||`) returns `True` if either the left or right operand cannot be evaluated as `False`
- The XOR operator (indicated by the keyword `xor`) returns `True` if either the left or right operand can be evaluated as `True`, but not both.
- The unary NOT operator (indicated by `!`) returns `False` if the operand can be evaluated as `True`, and `True` otherwise.

Note that we used the term "can be evaluated as" rather than "is." This is because, even if one of the operands is not a Boolean value, the interpreter will try to convert it and use it as such. Thus, any number different from 0 is evaluated as `True`, as is every string that is not empty or is not `'0'`.

Typecasting

Even though PHP handles data types automatically most of the time, you can still force it to convert a particular datum to a specific type by using a typecasting operator. These are

- `(int)` to cast a value to its integer equivalent
- `(float)` to cast a value to its floating-point equivalent
- `(string)` to cast a value to its string equivalent
- `(array)` to force the conversion of the operand to an array if it is not one already
- `(object)` to force the conversion of the operand to an object if it is not one already

Keep in mind that some of these conversions fall prey to the same pitfalls that we discussed earlier for automatic conversions performed during comparisons.

Combined Assignment Operators

A particular class of operators combines the assignment of a value with another operation. For example, `+=` causes the left-hand operator to be added to the right-hand operator, and the result of the addition stored back in to the left-hand operator (which must, therefore, be a variable). For example,

```
<?php

$a = 1;

$a += 5;

?>
```

At the end of the previous script's execution, `$a` will have a value of `6`. All the binary arithmetic and bitwise operators can be part of one of these combined assignment operations.

Combining Operations: Operator Precedence and Associativity

Operator precedence determines in what order multiple combined operations that are part of the same expression are executed. For example, one of the basic rules of arithmetic is that multiplications and divisions are executed before additions and subtractions. With a large number of types of operations available, things get a bit more complicated, but are by no means complex.

When two operations having the same precedence must be performed one after the other, the concept of *associativity* comes in to play. A left-associative operation is executed from left to right. So, for example, `3 + 5 + 4 = (3 + 5) + 4`. A right-associative

operation, on the other hand, is executed from right to left: `$a += $b += 10` is equivalent to `$a += ($b += 10)`. There are also some non-associative operations, such as comparisons. If two non-associative operations are on the same level of an expression, an error is produced. (If you think about it, an expression such as `$a <= $b >= $c` makes no sense in the context of a PHP script because the concept of "between" is not defined in the language. You would, in fact, have to rewrite that as `($a <= $b) && ($b >= $c)`.) Table 1.1 shows a list of operator precedences and associativity. Note that some of the operators will be introduced in Chapters 2 and 4.

Table 1.1 **Operator Precedence**

Associativity	Operator
right	`[`
right	`! ~ ++ – (int) (float) (string) (array) (object) @`
left	`* / %`
left	`<< >>`
non-associative	`< <= > >=`
non-associative	`== != === !==`
left	`&`
left	`^`
left	`|`
left	`&&`
left	`||`
left	`? :`
right	`= += -= *= /= .= %= &= |= ^= <<= >>=`
right	`print`
left	`and`
left	`xor`
left	`or`
left	`,`

As you have probably noticed, the logical operators `&&` and `||` have a different precedence than `and` and `or`. This is an important bit of information that you should keep in mind while reading through PHP code.

Operator precedence can be overridden by using parentheses. For example,

```
10 * 5 + 2 = 52
10 & (5 + 2) = 70
```

Parentheses can be nested to an arbitrary number of levels—but, of course, the number of parentheses in an expression must be balanced.

Conditional Structures

It is often necessary, at some point, to change the execution of a script based on one or more conditions. PHP offers a set of structures that can be used to control the flow of execution as needed.

The simplest such structure is the `if-then-else` statement:

```
if (condition1)
    code-block-1
[else
    code-block-2...]
```

The series of commands `code-block-1` is executed if `condition1` can be evaluated to the Boolean value `True`, whereas `code-block-2` is executed if `condition1` can be evaluated to `False`. For example,

```
<?php

$a = 10;

if ($a < 100)
    echo 'Less than 100';
else
    echo 'More than 100';

?>
```

In this case, the value of `$a` is obviously less than one hundred and, therefore, the first block of code will be executed, outputting `Less than 100`.

Clearly, if you could only include one instruction in every block of code, PHP would be extremely inefficient. Luckily, multiple instructions can be enclosed within braces:

```
<?php

$a = 10;

if ($a < 100)
{
    echo 'Less than 100';
    echo "\nNow I can output more than one line!";
}
else
    echo 'More than 100';

?>
```

`if-then-else` statements can be nested to an arbitrary level. PHP even supports a special keyword, `elseif`, that makes this process easier:

```
<?php

$a = 75;

if ($a > 100)
{
    echo 'More than 100';
    echo "Now I can output more than one line!";
}
elseif ($a > 50)
    echo 'More than 50';
else
    echo "I don't know what it is";

?>
```

In this case, the first condition `($a > 100)` will not be satisfied. The execution point will then move on to the second condition, `($a > 50)`, which *will* be satisfied, causing the interpreter to output `More than 50`.

Alternative `if-then-else` Syntax

As an alternative to the `if-then-else` syntax described in the previous section, which is what you will see in most modern PHP programs, PHP supports a different syntax in which code blocks start with a semicolon and end with the keyword `endif`:

```
<?php

$a = 10;

if ($a < 100):
    echo 'Less than 100';
    echo "Now I can output more than one line!";
elseif ($a < 50):
    echo 'Less than fifty';
else:
    echo "I don't know what it is";
endif

?>
```

Short-form `if-then-else`

A simple `if-then-else` statement can actually be written using a ternary operator (and, therefore, inserted directly into a more complex operation):

```
<?

$n = 15;

$a = ($n % 2 ? 'odd number' : 'even number');

echo $a;

?>
```

As you can see, the ternary operator's syntax is

```
(condition ? value_if_true : value_if_false)
```

In the specific case here, the `value_if_true` is returned by the expression if `condition` evaluates to `True`; otherwise, `value_if_false` is returned instead.

The `case` Statement

A complex `if-then-else` statement, composed of an arbitrary number of conditions all based on the same expression being compared to a number of immediate values, can actually be replaced by a `case` statement as follows:

```
<?php

$a = 10;

switch ($a)
{
    case    '1':

        echo '1';
        break;

    case    '5':

        echo 'Five';
        break;

    case    'Ten':

        echo 'String 10';
        break;

    case    10:

        echo '10';
        break;
```

```
    default:

        echo 'I don\'t know what to do';
        break;
}
?>
```

When the interpreter encounters the `switch` keyword, it evaluates the expression that follows it and then compares the resulting value with each of the individual `case` conditions. If a match is found, the code is executed until the keyword `break` or the end of the switch code block is found, whichever comes first. If no match is found and the `default` code block is present, its contents are executed.

Note that the presence of the `break` statement is essential—if it is not present, the interpreter will continue to execute code in to the next `case` or `default` code block, which often (but not always) isn't what you want to happen. You can actually turn this behavior to your advantage to simulate a logical `or` operation; for example, this code

```
<?php

if ($a == 1 || $a == 2)
{
    echo 'test one';
}
else
{
    echo 'test two';
}

?>
```

Could be rewritten as follows:

```
<?php

switch ($a)
{
    case    1:
    case    2:

        echo 'test one';
        break;

    default:

        echo 'test two';
        break;
}

?>
```

Once inside the `switch` statement, a value of `1` or `2` will cause the same actions to take place.

Iteration and Loops

Scripts are often used to perform repetitive tasks. This means that it is sometimes necessary to cause a script to execute the same instructions for a number of times that might—or might not—be known ahead of time. PHP provides a number of control structures that can be used for this purpose.

The `while` Structure

A `while` statement executes a code block until a condition is set:

```
<?php

$a = 10;

while ($a < 100)
{
    $a++;
}

?>
```

Clearly, you can use a condition that can never be satisfied—in which case, you'll end up with an *infinite loop*. Infinite loops are usually not a good thing, but, because PHP provides the proper mechanism for interrupting the loop at any point, they can also be useful. Consider the following:

```
<?php

$a = 10;
$b = 50;

while (true)
{
    $a++;

    if ($a > 100)
    {
        $b++;
        if ($b > 50)
        {
            break;
        }
    }
}

?>
```

In this script, the `(true)` condition is always satisfied and, therefore, the interpreter will be more than happy to go on repeating the code block forever. However, inside the code block itself, we perform two if-then checks, and the second one is dependent on the first so that the `$b > 50` will only be evaluated after `$a > 100`, and, if both are true, the `break` statement will cause the execution point to exit from the loop into the preceding scope. Naturally, we could have written this loop just by using the condition `($a <= 100 && $b <= 50)` in the `while` loop, but it would have been less efficient because we'd have to perform the check twice. (Remember, `$b` doesn't increment unless `$a` is greater than 100.) If the second condition were a complex expression, our script's performance might have suffered.

The `do-while` Structure

The big problem with the `while()` structure is that, if the condition never evaluates to `True`, the statements inside the code block are never executed.

In some cases, it might be preferable that the code be executed at least once, and then the condition evaluated to determine whether it will be necessary to execute it again. This can be achieved in one of two ways: either by copying the code outside of the `while` loop into a separate code block, which is inefficient and makes your scripts more difficult to maintain, or by using a `do-while` loop:

```
<?php

$a = 10;

do
{
    $a++;
}
while ($a < 10);

?>
```

In this simple script, `$a` will be incremented by one once—even if the condition in the `do-while` statement will never be true.

The `for` Loop

When you know exactly how many times a particular set of instructions must be repeated, using `while` and `do-while` loops is a bit inconvenient. For this purpose, `for` loops are also part of the arsenal at the disposal of the PHP programmer:

```
<?php

for ($i = 10; $i < 100; $i++)
{
```

```
    echo $i;
}

?>
```

As you can see, the declaration of a `for` loop is broken in to three parts: The first is used to perform any initialization operations needed and is executed only once *before* the loop begins. The second represents the condition that must be satisfied for the loop to continue. Finally, the third contains a set of instructions that are executed once at the end of every iteration of the loop before the condition is tested.

A `for` loop could, in principle, be rewritten as a `while` loop. For example, the previous simple script can be rewritten as follows:

```
<?php

$i = 10;

while ($i < 100)
{
    echo $i;
    $i++;
}

?>
```

As you can see, however, the `for` loop is much more elegant and compact.

Note that you can actually include more than one operation in the initialization and end-of-loop expressions of the `for` loop declaration by separating them with a comma:

```
<?php

for ($i = 1, $c = 2; $i < 10; $i++, $c += 2)
{
    echo $i;
    echo $c;
}

?>
```

Naturally, you can also create a `for` loop that is infinite—in a number of ways, in fact. You could omit the second expression from the declaration, which would cause the interpreter to always evaluate the condition to true. You could omit the third expression and never perform any actions in the code block associated with the loop that will cause the condition in the second expression to be evaluated as true. You can even omit all three expressions using the form `for(;;)` and end up with the equivalent of `while(true)`.

Continuing a Loop

You have already seen how the `break` statement can be used to exit from a loop. What if, however, you simply want to skip until the end of the code block associated with the loop and move on to the next iteration?

In that case, you can use the `continue` statement:

```
<?php

for ($i = 1, $c = 2; $i < 10; $i++, $c += 2)
{
    if ($c < 10)
        continue;

    echo 'I\'ve reached 10!';
}

?>
```

If you nest more than one loop, you can actually even specify the number of loops that you want to skip and move on from:

```
<?php

for ($i = 1, $c = 2; $i < 10; $i++, $c += 2)
{
    $b = 0;

    while ($b < 199) {
        if ($c < 10)
            continue 2;

        echo 'I\'ve reached 10!';
    }
}

?>
```

In this case, when the execution reaches the inner `while` loop, if `$c` is less than `10`, the `continue 2` statement will cause the interpreter to skip back two loops and start over with the next iteration of the `for` loop.

Functions and Constructs

The code that we have looked at up to this point works using a very simple top-down execution style: The interpreter simply starts at the beginning and works its way to the end in a linear fashion. In the real world, this simple approach is rarely practical; for example, you might want to perform a certain operation more than once in different portions of your code. To do so, PHP supports a facility known as a *function*.

Functions must be declared using the following syntax:

```
function function_name ([param1[, paramn]])
```

As you can see, each function is assigned a name and can receive one or more parameters. The parameters exist as variables throughout the execution of the entire function.

Let's look at an example:

```
<?php

function calc_weeks ($years)
{
    return $years * 52;
}

$my_years = 28;
echo calc_weeks ($my_years);

?>
```

The `$years` variable is created whenever the `calc_weeks` function is called and initialized with the value passed to it. The `return` statement is used to return a value from the function, which then becomes available to the calling script. You can also use `return` to exit from the function at any given time.

Normally, parameters are passed by value—this means that, in the previous example, a copy of the `$my_years` variable is placed in the `$years` variable when the function begins, and any changes to the latter are not reflected in the former. It is, however, possible to force passing a parameter *by reference* so that any changes performed within the function to it will be reflected on the outside as well:

```
<?php

function calc_weeks (&$years)
{
    $my_years += 10;
    return $my_years * 52;
}

$my_years = 28;
echo calc_weeks ($my_years);

?>
```

You can also assign a *default value* to any of the parameters of a function when declaring it. This way, if the caller does not provide a value for the parameter, the default one will be used instead:

```
<?php

function calc_weeks ($my_years = 10)
{

    return $my_years * 52;
}

echo calc_weeks ();

?>
```

In this case, because no value has been passed for `$my_years`, the default of `10` will be used by the interpreter. Note that you can't assign a default value to a parameter passed by reference.

Functions and Variable Scope

It's important to note that there is no relationship between the name of a variable declared inside a function and any corresponding variables declared outside of it. In PHP, variable scope works differently from most other languages so that what resides in the global scope is not automatically available in a function's scope. Let's look at an example:

```
<?php

function calc_weeks ()
{
    $years += 10;
    return $years * 52;
}

$years = 28;
echo calc_weeks ();

?>
```

In this particular case, the script assumes that the `$years` variable, which is part of the global scope, will be automatically included in the scope of `calc_weeks()`. However, this does not take place, so `$years` has a value of `Null` inside the function, resulting in a return value of `0`.

If you want to import global variables inside a function's scope, you can do so by using the `global` statement:

```
<?php

function calc_weeks ()
{
    global $years;
```

```
    $years += 10;
    return $years * 52;
}

$years = 28;
echo calc_weeks ();

?>
```

The `$years` variable is now available to the function, where it can be used and modified. Note that by importing the variable inside the function's scope, any changes made to it will be reflected in the global scope as well—in other words, you'll be accessing the variable itself, and not an ad hoc copy as you would with a parameter passed by value.

Functions with Variable Parameters

It's sometimes impossible to know how many parameters are needed for a function. In this case, you can create a function that accepts a variable number of arguments using a number of functions that PHP makes available for you:

- `func_num_args()` returns the number of parameters passed to a function.
- `func_get_arg($arg_num)` returns a particular parameter, given its position in the parameter list.
- `func_get_args()` returns an array containing all the parameters in the parameter list.

As an example, let's write a function that calculates the arithmetic average of all the parameters passed to it:

```
<?php

function calc_avg()
{
    $args =  func_num_args();

    if ($args == 0)
        return 0;

    $sum = 0;

    for ($i = 0; $i < $args; $i++)
        $sum += func_get_arg($i);

    return $sum / $args;
}

echo calc_avg (19, 23, 44, 1231, 2132, 11);

?>
```

As you can see, we start by determining the number of arguments and exiting immediately if there are none. We need to do so because otherwise the last instruction would cause a division-by-zero error. Next, we create a `for` loop that simply cycles through each parameter in sequence, adding its value to the sum. Finally, we calculate and return the average value by dividing the sum by the number of parameters. Note how we stored the value of the parameter count in the `$args` variable—we did so in order to make the script a bit more efficient because otherwise we would have had to perform a call to `func_get_args()` for every cycle of the `for` loop. That would have been rather wasteful because a function call is quite expensive in terms of performance and the number of parameters passed to the function does not change during its execution.

Variable Variables and Variable Functions

PHP supports two very useful features known as "variable variables" and "variable functions."

The former allows you use the value of a variable as the name of a variable. Sound confusing? Look at this example:

```
<?

$a = 100;
$b = 'a';

echo $$b;

?>
```

When this script is executed and the interpreter encounters the `$$b` expression, it first determines the value of `$b`, which is the string `a`. It then reevaluates the expression with `a` substituted for `$b` as `$a`, thus returning the value of the `$a` variable.

Similarly, you can use a variable's value as the name of a function:

```
<?

function odd_number ($x)
{
    echo "$x is odd";
}

function even_number ($x)
{
    echo "$x is even";
}

$n = 15;

$a = ($n % 2 ? 'odd_number' : 'even_number');
```

```
$a($n);

?>
```

At the end of the script, `$a` will contain either `odd_number` or `even_number`. The expression `$a($n)` will then be evaluated as a call to either `odd_number()` or `even_number()`.

Variable variables and variable functions can be extremely valuable and convenient. However, they tend to make your code obscure because the only way to really tell what happens during the script's execution is to execute it—you can't determine whether what you have written is correct by simply looking at it. As a result, you should only really use variable variables and functions when their usefulness outweighs the potential problems that they can introduce.

Exam Prep Questions

1. What will the following script output?

```
<?php

$x = 3 - 5 % 3;

echo $x;

?>
```

   A. 2
   B. 1
   C. Null
   D. True
   E. 3

   Answer **B** is correct. Because of operator precedence, the modulus operation is performed first, yielding a result of 2 (the remainder of the division of 5 by 2). Then, the result of this operation is subtracted from the integer 3.

2. Which data type will the `$a` variable have at the end of the following script?

```
<?php

$a = "1";

echo $x;

?>
```

A. (int) 1
B. (string) "1"
C. (bool) True
D. (float) 1.0
E. (float) 1

Answer **B** is correct. When a numeric string is assigned to a variable, it remains a string, and it is not converted until needed because of an operation that requires so.

3. What will the following script output?

```
<?php

$a = 1;

$a = $a– + 1;

echo $a;

?>
```

A. 2
B. 1
C. 3
D. 0
E. Null

Answer **A** is correct. The expression `$a–` will be evaluated after the expression `$a = $a + 1` but *before* the assignment. Therefore, by the time `$a + 1` is assigned to `$a`, the increment will simply be lost.