Introduction to PHP

PHP commands are inserted right into HTML documents. But how does the web server can tell the PHP code to be executed embedded into the HTML document? Actually, it's very easy: Special tags are used to indicate the beginning and the end of the PHP code. Everything outside of these tags is treated as HTML code.

Most often, the beginning and the end of PHP commands are marked in the following way:

```
<?php
 PHP code
?>
```

Everything between the tags the web server treats as PHP code and processes it accordingly. Everything outside of these tags is treated as HTML code and is sent to the client as is and is processed in the browser.

This is the most preferential format of the PHP code delimiting tags; using it you can be certain that it will processed by the server correctly. There also are other currently supported PHP code delimiting formats. Despite this, you should realize that only the tags are guaranteed support in all PHP versions. Use of other PHP code delimiting tags may be discontinued or limited in any future version of the language. This may lead to the problem of having to rewrite all the scripts in which discontinued delimiters are used. It's no big deal if you have only a few of them, but if you have a big site with lots of PHP scripts, then you've got a problem.

Another, the shortest, PHP code delimiter is the following:

```
<?
PHP Code
?>
```

For the server to recognize this format, its support must be enabled. This can be done by either compiling the PHP interpreter with the –-enable-short-tags or setting the value of the short_open_tag parameter in the php.ini file to on. The former option is preferable, because it is better to disable the short_open_tag parameter. Having it enabled may cause problems with recognizing XML, because some XML tags may create conflicts. This PHP code delimiting format is seldom used by programmers, and some PHP syntax-highlighting editors do not recognize it.

I do not recommend using the delimiter tags, because they were borrowed from SGML and may cause lots of conflicts. Moreover, your scripts may execute properly on the local computer, but not when uploaded to the remote server, unless the server supports this format. I even recommend that you disable this format on the local server, so that an error message would be issued should you accidentally use it.

Yet another PHP code delimiting format looks as follows:

```
<%
PHP Code
%>
```

This version is a bit simpler than the first one, but it is also fraught with problems as it is used in ASP. Thus the server may attempt to process the same code using different interpreters (i.e., ASP and PHP), which, most likely, will also produce undesirable effects.

The lengthiest PHP code delimiting format is the following:

```
<SCRIPT LANGUAGE="php">
PHP code
</script>
```

This format is similar to the HTML conventions, but it is too bulky, is more difficult to read, and can also cause conflicts.Here, conflicts can be caused if a programmer writes LANGUAGE="VBScript" instead of LANGUAGE="php" by mistake. In this case the code will be taken for a VB script and passed to the client computer, where the browser will attempt to handle it.

Let's try to write a PHP program doing something. The easiest thing to start with is to display some text and obtain information about the PHP interpreter installed. This is done by creating a file information.php:

Displaying information about the PHP interpreter

```
<HTML>
<HEAD>
<TITLE> Information </TITLE>
</HEAD>

<BODY>
<?php
print("<p>This is the information about PHP</p>");
phpinfo();
?>
</BODY>
</HTML>
```

Load this file to your remote web server or place it into the server folder on the local disk if you are using a local server. Now load this file into the browser. If the file was placed into the root folder of a remote server, enter the following URL into the browser's address field: www.server.com/information.php, where server.com is the name of the particular remote web server. For the local server, the URL is specified as: 127.0.0.1/information.php. The results of the script execution are shown in the next fig:

The information.php file execution results.

This script is often used to test server operability, PHP installation, and the installation parameters. Let's consider the contents of the script. The following two command strings are enclosed between the delimiting tags:

```
print("<p>This is the information about PHP</p>");
phpinfo();
```

The first command line calls the print function. This function displays text in the browser window. As you already know, simple text can be displayed in browsers with the help of HTML, but in this case I simply wanted to show how to do this using PHP functions. We will explore this function in more detail further on, but basic information about it will suffice for now.

Any PHP function can receive parameters, which are specified in the parenthesis after the name of the function. The parameter for the print function is the text to be displayed in the client's browser. The text has to be included into single or double quotation marks. Look at the text that was displayed in our example. You can see that it ends in the <P> HTML tag, which means end of paragraph. Thus, HTML tags can be inserted in the displayed text to format it as necessary.

The second command line calls the phpinfo function. It displays in the browser the information about the PHP interpreter installed in the system. This function takes no parameters, so none is specified in the parenthesis.

You can alternate PHP and HTML code within a page at will. To be more exact, you can insert PHP code into any location of an HTML document, as is shown in the next listing.

An example of alternating HTML and PHP codes

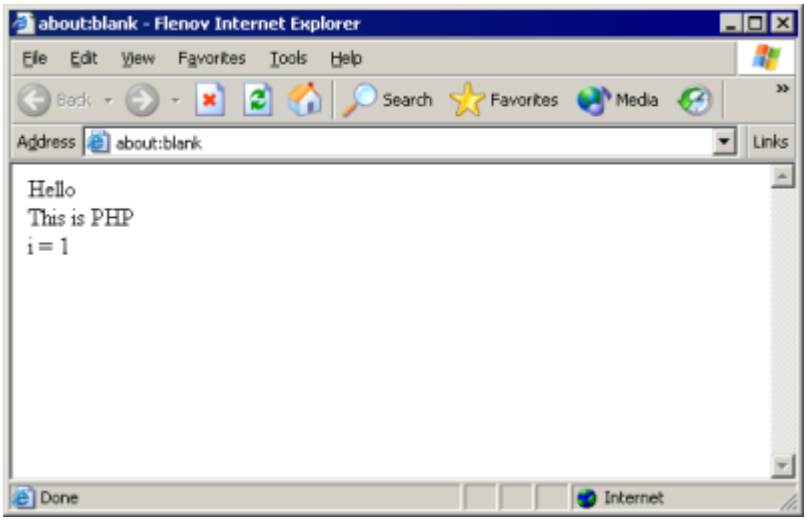
```
<HTML>
<HEAD>
<TITLE> Vision </TITLE>
</HEAD>
<BODY>
<P> Hello
<P> <?php $i =1; print("This is PHP");?>
```

```
<P> i = <?php print($i) ?>
</BODY>
<HTML>
```

The example in above contains two lines of PHP code, with the rest of the code being HTML. PHP code can be inserted into HTML code as often as necessary.

There is more to is, though. In the first PHP code line a $i variable was declared and set to 1. What is a variable? For the time being it will suffice to view a variable as a memory location (either a single cell or a block of cells) for storing certain values (e.g., numbers, strings, etc.), which can then be manipulated in different ways. A variable name starts with the $ sign, followed by the name proper. Thus, our memory cell will be named $i and hold the value of 1.

The second piece of code print out the contents of the memory cell named $i. Running this code will produce the results shown in the next fig.



Outputting the contents of a variable

As you can see, the value of the memory cell $i did not disappear anywhere and still equals 1, despite the fact that the variable was declared in one piece of PHP code and used in another. Thus, variable retain their values throughout the entire document. We will examine variables in more detail later on.

Including files

Every programmer tries to reuse existing code and any programming language tries to provide support for this. When we develop a new project, we don't feel at all as solving the same problems that had been solved in previous projects. Being able to reuse already developed code allows us to avoid doing the same mundane routines over and over again.

You have probably heard a lot about Windows dynamic libraries. These libraries store various different resources (images, icons, dialog window forms, menus, etc.) and/or program code. Any program can load this library and use the resources stored in it. For example, the OpenGL graphics library stores functions for creating 3D graphics of practically any complexity. Any programmer can load this library into the computer memory and use its resources in his or her own projects. Accordingly, it is not necessary to write graphic functions code for each new program, but use code already developed by other programmers.

Dynamic libraries not only allow a programmer use his or her own code in different projects, but also share it with other programmers, as we saw in the example with the OpenGL library.

Being able to reuse code is even more important when programming web pages. Your site may contain hundreds of files doing the same thing. Writing the same piece of code in each of them is a tedious time-consuming task. Moreover, this results in bloated, slow-executing files.

The code reusing function in PHP is implemented by connecting files. File connection can be considered as another way for embedding PHP code into web pages. This is done by the include and require commands, each of which has two versions:

```
include('/filepath/filename');
include_once('/filepath/filename');

require('/filepath/filename');
require_once('/filepath/filename');
```

All of the commands above connect the file specified in the parenthesis. In previous PHP versions there was a little difference in the execution speed of the include/include_once and require/require_once functions. In the present version of the language, the difference between them is only the error they generate. When an error is encountered (e.g., the file to be connected is missing), the include functions issue a warning message and continue executing. The require functions in this case generate a critical error
and terminate the execution.

Thus, the difference between the include/require and include_once/require_once function pairs is that the first pair guarantees that the specified file will be connected to the current file only once. It is sometimes necessary not to connect twice a file containing critical code of which there has to be only one instance. An example of such code is PHP functions (which we will consider later), which must be declared only once. When an attempt to connect a file already connected is made using the include_once or require_once function, a fatal error will be generated.

It may seem at first that the include_once and require_once functions are not necessary if pains are taken not to connect the same file twice in the same script. That's not so. A file may have been connected in another script. Suppose that in your script you want connect files 1.php and 2.php. Each file contains different code. But what is the 1.php file connects the 2.php file? In this case, connecting 1.php you automatically connect 2.php also (from 1.php); connecting 2.php again (from the master script) will produce an error. This error can be easily caught using the include_once and require_once functions. In this case, you will only have to connect 1.php, and 2.php will be connected from it.

When is each function to be used? I recommend using require and require_once to connect files containing program code. In this case, you will be able to react timely to an incorrect connection error. Some programmers don't want users to see any errors, but they are wrong here. It will not be any better if your code works incorrectly because of such an error.

The include function is better to use to connect document parts. For example, of late, it has become common to divide a page into three parts: the header, the body, and the footer. The header contains the upper part of the page (the logo, menu, etc.) that is the same for all site documents. The footer also contains information that does not change (owner information and the like). The body is different for each page. Thus, so as not to create the header and the footer in each file, they are removed into a separate file and are connected to each page as needed. It is better to connect headers and footers using the include function.

Let's consider an example of a site built using three-part pages. The contents of the header file, which is most often called header.inc, may look as shown in the next listing.

```
<HTML>
<HEAD>
<TITLE> Test page </TITLE>
</HEAD>
<BODY>
 <CENTER><H1> Welcome to my home page. </H1></CENTER>
 <!-- Here you can insert page menu, links -->
 <p><a href="http://www.cydsoft.com/">Home</a>
    <a href="http://www.cydsoft.com/products">Products</a>
    <a href="http://www.cydsoft.com/register.php">Purchase</a>
    <a href="MailTo:noreply@cydsoft.com">Contuct Us</a>
 <hr/>
```

The contents of the footer file, which is most often called footer.inc, may look as the following:

```
<P><HR>
 <CENTER><I> Copyright Flenov Mikhail</I></CENTER>
</BODY>
<HTML>
```

The source code for, for example, a news page, can look as shown in the next listing.

```
<?php include('header.inc'); ?>

CyD Network Utilities (Security Tools) 2009 Beta 1
is now available. New version has a new interface
and an improved security test module. Download
the beta version CyD Network Utilities (Security Tools).
We will glad to hear any comments. Do not hesitate.
Try and use our next-generation network and security
tools.

<?php include('footer.inc'); ?>
```

Now we no longer have to write the header and footer code for each page. All we have to do is to connect them to the necessary page as needed. This not only makes it easier creating files, but also maintaining them. Suppose that your site consists of 100 files, and you have to change one menu item. Using static HTML, you would have to change all 100 file and load all of them on the remote server. Imagine the workload and the traffic? When using dynamically loaded header and footer, all you will have to do is to change and upload only the header file and this file will be used for each page on your site. Neat, ain't it?

Note that we wrote all of the connected files in simple HTML. If you thought that these files had to contain PHP code, you were wrong. By default connected files are assumed to be HTML and to use PHP commands in them, the PHP code delimiting tags (i.e.,) also have to be used in them.

If the connected file has to contain only PHP code, the opening tag should be placed at the very beginning of the file, and the closing tag at the very end of it. Pay attention not to have any blank lines or spaces before and after the PHP code. As you know, everything outside of the PHP code delimiting tags is interpreted as HTML, and extra space or blank line may be interpreted as formatting. This may cause you many wasted hours looking what causes those wide open spaces in your pages.

Connecting file can simplify site development substantially. Many sites take advantage of this feature to a lesser or greater extent. But this feature may be a security problem. Suppose that you created a site using the three-part concept for its pages. Whereas only one file is needed for each header and footer, there will be many files for the page body parts stored in a separate directory. Which one of them to display depends on the current choice of the user. You have probably seen sites in the Internet whose URL looks like this:

http://www.sitename.com/index.php?file=main.html

Everything that goes after the question sign are parameters. In this case, the main.html file is sent as a parameter and it is this file that will be displayed in the page body for this URL. But what if a hacker manages to change this parameter to /etc/shadow, the Linux password file? This will display it on the hacker's computer. And even though the passwords are encrypted, there are tools to recover at least some of them. Some times, we will consider a site on which no checks for such substitutions were performed, and I managed to view all of the system files using this trick.

Outputing information on the page

We already used the print function to observe the results of script execution. We have to explore the available output functions more closely, because if you can`t see the results of your scripts` work, you may have problems digesting the material presented. The best way to learn anything is to personally touch and see every detail.

In addition to the print function, information can be displayed by the echo function. There are two ways of calling it. These are the following:

```
echo("Hello, this is text");
echo "Hello, this is text";
```

In the first case, the text to be displayed in placed into parenthesis, and in the other, it is simply placed after the function name after a space. In either case, the output text is enclosed into quotation marks. Both calling method are equivalent to each other, and you can use the one you feel more comfortable with. You can output several strings with the echo function by separating them with a comma. For example:

```
echo("Hello, this is text", "This is text too");
echo "Hello, this is text", "This is text too";
```

One of the differences between the print and the echo function is that the former can return a value indicating the success of the output operation. If the function returns 1, the output was successful; otherwise, the function returns 0. Another difference is that the print function can only output one string, that is, you cannot specify two strings for output by delimiting them with a comma.

The following is an example of the function`s use:

```
print("Hello, this is text.");
```

Comments in PHP

The format of PHP comments is similar to that of C/C++ and Java, which is another indication of these languages being related. What is a comment as related to a programming language? This is supplementary information given in the code that has no effect on the program execution. For example, you may want to explain how a particular piece of code works. Naturally, you don't

want this explanation to execute or to show in the browser. Such explanation is inserted as a comment in the vicinity of the code it explains.

There are single-line and multi-line comments. A single line comment starts with two slashes (like in C++) or with the pound sign (like in Linux). Everything following these characters is disregarded by the interpreter and treated as a comment. For example:

```
<?php
 # This is a comment
 //This is also a comment
 This is code//But this is a comment
?>
```

As you can see in the third line of the code above, a comment does not have to be placed on an individual line, but can follow a PHP command. In this way you can explain a single code line, should there be such a need. I will be using comments to explain as much material as possible throughout the book. This will help you with understanding the code examples.

Comments can take more than one line, such as, for example, an explanation describing module's features. This type of comments starts the /* character sequence and end with the */ character sequence. In practice, this type of comments looks like the following:

```
<?php
Code
/* This is a comment
This is also a comment
And this is a comment
*/ But this is not a comment; this can only be code.
Code
?>
```

Note that this type of comments can only be made in the PHP mode. A comment like this made in an HTML code section will be displayed in the browser, because comment rules are different for HTML.

Make it a habit to comment code in your programs. Most beginning and even experience programmers think that they can always remember what their code does. This is only true during the development stage. When a month later you decide to do some minor modification, you will be surprised to discover to have forgotten how the code works and will have to spend your time remembering this. To avoid this development, comment your code from the very beginning. Believe me, a few minutes spent commenting your code now will return dividends in hours saved later.

Sensitivity to spaces, carriage returns, and tabs

PHP is not sensitive to spaces, carriage returns, and tabs. This means that you can break one command into several lines, or delimit variables, values, or operators with different number of spaces. For example, as the following:

```
<?php
 $index = 10;
 $index   =  10  +   20;
 $index=10+10;
 $index=
 10
 +
```

```
 10;
?>
```

All of the code above is correct and will work without a hitch.

Each PHP command is terminated with a colon (;). In this way, the interpreter separates one command from another. Do not forget to use this character where it is required, because failing to do so may cause unpredictable errors, which usually cannot be attributed to a missing command separator.

The command separator makes it possible to write one command on several lines or place several commands in one line. The interpreter will be able to tell the commands apart thanks to the command separator.

What PHP is sensitive to is the character case. This means that variable names written in uppercase and lowercase will be treated as different. Many beginner programmers often make many mistakes because of this circumstance. Consider the following example:

```
<?php
 $index = 10;
 $Index = 20;
 print($index);
 print($Index);
?>
```

The piece of code above will display 10 and 20, because $index and $Index are treated as different variables due to their first letter being of different case. If PHP were not case-sensitive, both variables would point to the same memory location. In this case, the first command would write 10 to this memory location, and the second would overwrite the previous value with 20. The next two print commands would display 20 twice.

Thus, pay attention to the case in your variable names. The following code will not produce the result that you might expect:

```
<?php
 $index = 10;
 print($Index);
?>
```

In the code above we are assigning a value, 10 in the given case, to the variable $index, but outputting the $Index variable. Naturally, nothing will be output to the screen, because the $Index variable has no value.

To avoid making mistakes in your programs, always use the same case for your variable names. I recommend using the lowercase for all letters. This will reduce the possibility of the code being interpreted incorrectly to practically zero.

But only variable names are case sensitive in PHP. PHP statements can be written in any case. Let's consider this using the if..else conditional branch statement as an example. This statement has the following format: if (condition) action1 else action2. If the condition evaluates to true, action1 is executed; otherwise, action2 is. We will consider other control statements later, but for the time being we just want to verify that they can be written in any case. Consider the following code:

```
<?php
 $index=1;
 if ($index==1)
  print('true');
 else
  print('false');
?>
```

In the code above, the if…else statement is written in lowercase, but it can also be written in uppercase or even using a mixture of cases as in the following piece of code:

```
<?php
 $index=1;
 If ($index==1)
  print('true');
 ElsE
  print('false');
?>
```

Here the keywords if and else are written using both uppercase and lowercase, and this is perfectly alright. In PHP, only variable and function names are case sensitive, but not statements or keywords. Despite statements and keywords not being case sensitive, I recommend writing them in lowercase: The situation may change in the future PHP versions and they may become case sensitive. In that case, no pun intended, you will have a bit of work to do hunting down all your mixed case statements and keywords and converting them to the proper case.

Variables in PHP

If you have experience programming in high-level languages (e.g., C++, Delphi), you must know that all variables must be of a strictly defined type, and the code must follow quite stringent syntax rules. PHP is more flexible and does not impose strict rules. This flexibility, however, comes at the price of greater chances of erroneous script execution and being much more difficult to provide proper security. Lots of break-ins have been perpetrated exactly because variables in PHP are not assigned a specific type when declared.

Suppose a hacker passes a database query string through a parameter that the programmer intended to be used to pass a numerical value. (By the way, this is how the PHPNuke site management system was cracked.) If the parameter variable were defined of a specific type, such action would return an error, because the string could not be converted into a number. Because PHP is weak-typed language, programmers must implement data type checks and handle incorrect data type errors themselves.

If you have programmed in C, Java, or Perl before, many concepts in PHP will be familiar to you, because PHP is very similar to C/C++.

We have already touched slightly on variables and know that a variable is a memory area in which values can be stored and can be referenced by a name. We don't care where exactly in the memory our variables are stored, because their values can always be retrieved or changed by referencing their names. PHP variables have the following properties:

- A variable name must start with the dollar sign ($). This character tells the interpreter that the next sequence of characters is a variable. The

dollar sign is followed by any combination of alphanumerical characters and the underline character. However, the first character after the dollar sign must be a letter.

- The variable value is the latest value written to it.

  If you worked with C++ or Delphi programming language before, you must know that a variable must be declared before it can be used. In PHP, it is not necessary to declare a variable beforehand. A variable starts existing from the moment it is assigned a value or used for the first time. If a variable is used before it was assigned a value explicitly, it will contain the default value.

- A variable is not assigned a specific type. The variable type is determined by its value and the current operation.

Try to give variables meaningful names. If you name your variable as $param1, $param2, $param3, and so on, some time later you will have problems remembering what the $param2 variable stores and how it is used.

So, if you have to store, for example, a sum, name the corresponding variable $sum. If you need a counter (which we will consider in the loop section), a good choice for the corresponding variable name is $i. It is a common counter variable name. I guess, the i stands for iterations.

A variable is assigned a value using the equal sign. The variable name goes to the left side of the equal sign, and the value assigned to it goes to the right:

```
$var_name = var_value;
```

PHP has three main variable types: numerical, string, and Boolean. I believe the first two types are self-explanatory. A numerical type variable holds a number; we already used such variables. Values of string variables are enclosed into single or double quotation marks, as follows:

```
$str = 'This is a string';
$str = "This is a string";
```

What is the difference between these two ways of assigning string values to variables? Suppose that you want to display the value of the $index variable. You also want to know what this value is, that is, give it a description. This can be done with the following piece of code:

```
$index = 10;
$str = 'Index = $index';
print ($str);
$str = "Index = $index";
print ($str);
```

The first print command will display "Index = $index," and the second "Index = 10."Thus, in strings enclosed in double quotes, the interpreter looks for variables and outputs their values; in strings in single quotes it interprets all enclosed text as a string. In the latter case, the text is displayed slightly faster rapidly, because there is not extra processing performed. Therefore, if you don't have to output a variable value in a string, enclose it in single quotes, so that the interpreter would not parse the string needlessly. Doing this may raise the page display efficiency. Don't use double quotes everywhere just because they used universally.

A string can be broken into several lines, if for some reason you don't want to display it on one line in the editor. This is done as follows:

```
$str = "This is a string.
   PHP is the next generation of WEB programming.
```

```
   You will like this.";
```

As you can see, you simply hit the carriage return key and continue on the new line. The PHP interpreter will determine the beginning and the end of a multi-line string by the quotation marks enclosing it.

I recommend that when typing multi-line strings you indent all continuation lines to distinguish them from new code lines. This will make the program code easier to read.

A Boolean type variable takes on only two values: true (any value greater than 1) and false (a value equaling 0). Variables of this type are not used explicitly, but implicitly in logical operations. For example, the if…else operation, which we already considered, returns a logical true value if the condition evaluated is met, and false otherwise.

If a variable is used before is its value was set, it will be assigned the default value. How can we determine the default value of a variable without knowing its type? The type can be determined from the code context. For example, if a mathematical operation involving using a number is performed on a variable, the variable will be set to 0. If this is a string operation, the result will be an empty string.

How can we tell whether a variable has been set? This is done by the IsSet(Var_Name) function. The function is passed the variable in question as the parameter. If the variable is set, the function return true; otherwise, false is returned. Consider the next example:

```
<?php
 if (IsSet($index))
   print('The variable is set');     // The variable is set
 else
   print('The variable is not set'); // The variable is not set

 $index = 1;

 if (IsSet($index))
   print(The variable is set');   // The variable is set
 else
   print('The variable is not set'); // The variable is not set
?>
```

The first check tells us that the $index variable is not set. We then write a value to the variable and do another check, which shows that this time the variable is set.

Because variables in PHP are not assigned a specific type, we can set them to value of any type, leaving it to the system to sort it out which type assign to the variable. For example:

```
$index = 1;            // An integer
$fl = 3.14;            // A floating number (a.k.a. double)
$str = 'This is a string'; // A string
```

The variable type is determined by the context in which it is used. Consider, for example, the following piece of code:

```
$str = '10';
```

```
$index = 2 * $str;
print("Result = $index");
```

At first, you may say that the operation in the second line cannot be performed. The first command declared a variable $str and assigned it a string value of '10.' The second command multiplies the string held in the $str variable by number 2. This operation cannot be done in most programming languages. If the PHP interpreter, however, sees that the $str variable holds a string that can easily be converted into a number, it does so and assigns the $index variable the value of 20.

The last command converts the numerical value of the $index variable back to string and displays it to the screen: "Result = 20."

As you can see, the variable type does not depend on its value, but is determined by the context in which the variable is used. If the contents are invalid in a certain context, the variable is set to 0. Consider, for example, the following piece of code:

```
$str = 'r10';
$index = 2 * $str;
print("Result is $index");
```

In this case, the $str variable cannot be converted into a number during the multiplication operation because of the letter r in its string value. Thus, it will be set to 0, and 2*0=0.

What do you think the following code will produce?

```
print(3*"hello"+2+TRUE);
```

It is difficult to see right away how PHP will process this operation. Let's do this in stages. First number 3 is multiplied by the string "hello." Because the string cannot be converted into a number, 0 will be used instead. Three times zero is zero. To this result 2 is added, producing 2. Finally, true as added to 2. What is the numerical value of true? As you should remember, it is 1 or greater. The PHP interpreter assigns true the numerical value of 1. So, it is this number that will be added to the result of the previous operations, producing 2+1=3. Try to do this operation on your computer.

This implicit type conversion, called casting in computerese, makes, on one hand, writing code easier, but, on the other hand, its debugging more difficult. It is easy to make a type conversion error that will be very difficult to locate. Be careful when working with variables, especially with those provided by users. The subject of user-provided variables will be discussed further on.

PHP Main Operations

Variables are created to perform some operations on them. At present, we will consider only the following simple mathematical operations:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)

As is common in mathematics, mathematical operations in PHP are performed in the order of operator precedence. Multiplication and division are performed before addition and subtraction. Consider the following classical example:

$index = 2 + 2 * 2;

If you ask, for example, a third-grader to evaluate the above expression, the most likely answer will be 8. But those with further advances in math will not fall into the sequential evaluation trap. Their answer will be 6, because pursuant to the mathematical operator precedence order, first the multiplication operation is performed, yielding 4, which is then added to 2, thus producing 6.

The operator precedence is controlled by parenthesis, with operations in parenthesis having higher precedence than those outside. The result of the following example:

$index = (2 + 2) * 2;

will be 8.

The same operator precedence rules apply to operations inside parenthesis. Consider the following example:

$index = (2 + 2 * 2) * 3;

Here, first, the operations inside the parenthesis are performed, and of those, the multiplication is done first: 2*2=4, and then the addition: 2+4=6. This result is then multiplied by 3 to produce 6*3=18.

PHP has a shortcut notation for increasing or decreasing a variable value by 1, which results in easier to read code that is also more elegant. A variable value is increased and decreased as follows: $var_name++ and $var_name-- respectively. For example:

```
$index = 2;
$sum = $index++;
```

The result of this code execution will be the variable $sum set to 3.

I always try to increase or decrease variable values in this way. It is not that difficult to get used to it, and I advise you to switch to it from the cumbersome $var_name = $var_name+1 method. These operations are most often used in loops, which will be considered further.

In PHP, the string concatenation operator is the period character, and not the plus sign, as is the case in other high-level programming languages. For example:

```
$str1 = "This is a test ";
$str2 = "string.";
$str3 = $str1.str2;
```

In the preceding code, first two variables – $str1 and $str2 – are created and assigned values. Next, a third variable – $str3 – is created and assigned the concatenated values of the first two variables. The result is the following string: "This is a test string."

Constants in PHP

Constants are similar to variables in that they are named memory locations holding certain values; unlike variables, however, once a constant was assigned a value at its declaration, it cannot be changed during script execution.

Constants are used to store some frequently used numbers or strings. For example, your site may be programmed for 640-pixel wide pages and you want to switch to using 800-pixel wide pages. If you used number 640 explicitly in your code, you will have to find all instances it was used in the code and change it. Even though this task can be automated, there is no guarantee that you will find all the numbers that need to be changed or not change a number 640 referring to something other than the page width. Instead of using the number 640 explicitly, you can declare a constant, for example $PgWdth, at the beginning of the file, set it to 640, and then use the constant throughout the file wherever you need to use number 640. Then, if you need to change 640 to 800, all you need to do is to reassign the value of the constant $PgWdth to 800 once at the beginning of the file.

I recommend always using constants or at least variables if a number or a string is used more than once in the code. These constants and variables can be stored in a separate file, which can then be included into the PHP files using these constants or variables. Based on my personal experience, I can tell that using constants can make software maintenance and modification significantly easier.

Storing all text messages as variables in a separate file, you will be able to save your time and make software localization an easier task. For example, one file can store Russian language messages, and English messages can be stored in another file. The program language can be changed by simply using the necessary file, depending on the user's choice.

Constant names, unlike variable names, do not start with the dollar sign. To make constants stand our, their names are written in all uppercase letters. PHP has many constants making programming easier. We will be getting to know them in the course of the book and as a need arises, but for the time being, we will learn how user constants are declared. Constants are declared using the define() function, to which two parameters are passed. The first parameter is the constant name and the second is the constant's value. For example, a constant with the value of Pi can be created by the following piece of code:

```
define('PI', 3.14);
$index = 10 * 3,14;
print($index);
```

In the first line, a new constant, named PI, is declared and its value set to 3.14.In the second line, the value of the constant PI is multiplied by 10 and stored in the variable $index. In the last line, the value of the variable $index is displayed on the screen.

As already mentioned, a constant value cannot be changed; therefore, the following piece of code will produce an error message.

```
define('PI', 3.14);
PI = 10 * 3,14;
```

Controlling Program Execution

It is a rare program that simply executes from the beginning to the end, because in most cases there are some conditions that can change the program execution flow. Thus, these conditions have to be checked and reacted to in one way or another. Let's consider an example of a site's

main page. When a user visits the site for the first time, he or she can be shown some additional information or greeted with some funny presentation to get him interested in the site. For succeeding visits by the same user, the presentation is no longer shown. The script logic for these actions will be something like the following:

- If visiting for the first time, show the presentation before showing the main page.
- Otherwise, show the main page right away.

As another example, we have to do numerous checks to ensure that a script is reliable and secure. For example, if a script is intended to send a mail message, it is a good idea to check whether the address is specified correctly before mailing the message. Here, the logic can be the following:

- If the address format is valid, mail the message.
- Otherwise, don't and issue an error message.

As you can see, the logic comes down to simple testing: If the condition is true, do one thing; otherwise do another thing. In PHP, this logic is implemented in the following way:

```
if (condition)
 Action_1;
else
 Action_2;
```

If the condition specified in the parenthesis evaluates to true, Action_1 is performed; otherwise, Action_2 is. For example:

```
$index = 0;
if ($index > 0)
 print("Index > 0");
else
 print("Index = 0");
```

In this piece of code, the following testing is performed: If the $index variable is greater than zero, the first message is displayed; otherwise, the second message is displayed with the print command after the else keyword.

Don't forget that only one command will be performed. To execute two commands, they have to be combined into a block using curly brackets {}. For example, the following code is incorrect:

```
$index = 0;
if ($index > 0)
 print("Index > 0");
 $index = 0;
else
 print("Index = 0");
```

If the $index variable is greater than zero, we want to display a message and change the variable value to zero. But this involves executing two commands, whereas only one can be executed. The correct way of doing this is the following:

```
$index = 0;
if ($index > 0)
 {
  print("Index > 0");
  $index = 0;
```

```
 }
else
 print("Index = 0");
```

The same as after the if keyword, only one command can also be executed after the else keyword. To execute more than one command, all of them have to be included into curly brackets. For example:

```
$index = 0;
if ($index > 0)
 {
  print("Index > 0");
  $index = 0;
 }
else
 {
  print("Index = 0");
  $index = 1;
 }
```

If an action is to be taken only when the condition is met, but none when it is not, the keyword else is not used. For example:

```
$index = 0;
if ($index > 0)
 print("Index > 0");
```

In this case, the message will be printed only if the $index variable is greater than zero. In either case, the program flow resumes after the print command.

Note the way the if…else code is formatted. The if statement is written at the left margin. The command to be executed if the condition is satisfied is written on the next line indented one space. The if keyword is written with the same indent as the corresponding if, at the left margin in this case. Finally, the command to be executed in the else part, is written on a new line indented as the command in the if part. This formatting makes it easy to see which else belongs to which if. This formatting is very handy when several nested if…else statements are used. Consider the following example:

```
$index = 0;
if ($index > 0)
 {
  if ($index > 10)
   $index = 10;
  else
   $index = 0;
 }
else
 {
  print("Index = 0");
  $index = 1;
 }
```

The formatting makes it easy to see to where individual code fragments belong.

Once I had to edit a Delphi program. The source code was over 3,000 lines, but this was not the problem. The problem was that all these lines were on the same level and without any spaces between code blocks. It was impossible to work with this code and I had to format it by making appropriate line indents and inserting line spaces between logical code blocks. As an example, try to follow the logic in the following code:

```
$index = 0;
if ($index > 0)
{
if ($index > 10)
$index = 10;
else
$index = 0;
}
else
{
print("Index = 0");
$index = 1;
}
```

Not that easy, eh? And this is a simplest piece of code. Imagine the difficulties trying to make out about 10 if…else statements, with some of them nested to boot?

PHP has many comparison operators. These are shown in the next table

Operation	Result
Parameter 1 > Parameter 2	Returns true if the first parameter is greater than the second one.
Parameter 1 >= Parameter 2	Returns true if the first parameter is greater than or equal to the second one.
Parameter 1 < Parameter 2	Returns true if the first parameter is less than the second one.
Parameter 1 <= Parameter 2	Returns true if the first parameter is less than or equal to the second one.
Parameter 1 == Parameter 2	Returns true if the first parameter is equal to the second one.
Parameter 1 === Parameter 2	Returns true if the first parameter is equal to the second one, and both parameters are of the same type.
Parameter 1 != Parameter 2	Returns true if the first parameter is not equal to the second one.

Two conditions can be tested simultaneously by using logical operators to combine comparison operators. These are shown:

Condition 1 and Condition 2 (Condition 1 && Condition 2)	Returns true if both conditions return true.
Condition 1 or Condition 2 (Condition 1 || Condition 2)	Returns true if at least one of the conditions returns true.
Condition 1 xor Condition 2	Returns true if only one condition returns true.
!Condition	Inverts the condition evaluation results; that is, converts true to false and vice versa.

Let's consider a double comparison example in the following piece of code:

```
$index1 = 0;
$index2 = 1;
if ($index1 > index2 and $index2 == 1)
 print("Index1 greater than Index2 and Index2 is equal to 1");
else
 print("Index1 is equal to Index2 and Index2 is not equal to 1");
```

And in the following example, a variable is checked for falling into the 1 to 10 range.

```
$index = 0;
if ($index >= 1 and $index <= 10)
 print("Index is greater than 1 and is less than 10");
```

In the preceding examples, we compared only integers, but strings and fractions can also be compared. Try to compare several variables yourself using different methods and observe which result is displayed. Only by getting hands-on experience will you be able to see the difference between different comparison operators. We will be using the if…else conditional statement repeatedly in the future material, so if you don't quite get it now, you will master it with time.

There is another interesting way to evaluate a condition. Suppose that you have to compare two variables and store the larger of them into the result variable. This can be done by the following line of code:

```
$result = $index1 > $index2  ? index1 : index2;
```

How does this code work? Let's write it in a general format as follows:

```
Condition ? Action_1 : Action_2;
```

If Condition evaluates to true, Action_1 is performed; otherwise, Action_2 is. Thus, if in our example, the $index1 variable is greater than the $index2 variable, its value will be stored in the $result variable; otherwise, the value of the $index2 variable will. This is a quite handy technique, but most beginning programmers have problems orienting in this type of code.

Another comparison task is to compare a variable with several values. For example, the value of the $day variable is tested for being one of the numbers from 1 to 7 and depending on the result the day of the week is printed out. This can be done with the help of the code shown in the next code:

```
$day = 2;
if ($day == 1)
 print("Sunday");
else
 if ($day == 2)
  print("Monday");
 else
  if ($day == 3)
   print("Tuesday");
  else
```

```
   if ($day == 4)
    print("Wednesday");
   else
    if ($day == 5)
     print("Thursday");
    else
     if ($day == 6)
      print("Friday");
    else
     if ($day == 7)
      print("Saturday");
```

Even though this code does the job, it is not very readable and is too bulky; without the indents, it would be even more difficult to follow. It can be rewritten using the if...elseif statement as shown in the next code:

```
$day = 2;
if ($day == 1)
 print("Sunday");
elseif ($day == 2)
 print("Monday");
elseif ($day == 3)
 print("Tuesday");
elseif ($day == 4)
 print("Wednesday");
elseif ($day == 5)
 print("Thursday");
elseif ($day == 6)
 print("Friday");
elseif ($day == 7)
 print("Satruday");
```

This code is much easier to follow. The general format of the if...elseif statement is as follows:

```
if (Condition_1)
 Action_1;
elseif (Condition_2)
 print(Action_2);
...
```

If Condition_1 evaluates to true, Action_1 is performed and no further check is performed; otherwise, Condition_2 is evaluated.If Condtion_2 is true, Action_2 is performed.

Another variation of the if statement is the following:

```
if (Condition):
 Statement 1;
 STatement 2;
endif
```

In the piece of code above, numerous statements that are executed are not enclosed into curly brackets, because if the condition is true, everything following it will be executed until the keyword endif is encountered. In a way, the colon after the if statement plays the role of the opening curly bracket, and the endif serves as the closing bracket.

Another program flow control statement is the switch statement. The general format of this statement is the following:

```
switch (Variable)
{
 case Value_1:
   Statement_Block_1;
   break;
 case Value_2:
   Statement_Block_2;
   break;
 [default: Statement]
}
```

In this instance, the variable is sequentially compared with the values after each case keyword. Once a match is found, all of the statements in the corresponding case block are executed until the keyword break is encountered. If none of the case values matches the switch variable, the statement following the default keyword will be executed. This keyword is optional. You can use it if you want to do something if no case match is encountered.

The next code shows how to solve the day of the week problem using the switch statement.

```
$day = 4;
switch ($day)
{
 case 1:
   print("Sunday");
   print("It's back to the salt pits tomorrow...");
   break;
 case 2:
   print("Monday");
   print("Boy, do I hate Mondays!");
   break;
case 3:
   print("Tuesday");
   print("Just another day at the office.");
   break;
case 4:
   print("Wednesday");
   print("Over the hump.");
   break;
case 5:
   print("Thursday");
   print("Four down, one to go.");
   break;
case 6:
   print("Friday");
   print("T.G.I.F!");
   break;
case 7:
   print("Saturday");
   print("Whoopee! Party time!");
   break;
}
```

Note the way the code is formatted. Commands in each case block are indented for better code readability. This way each code block stands out.

Note that the same as in the if...endif statement, commands in case blocks do not have to be enclosed in curly brackets. Here, the break keyword plays the role of the closing bracket. At first this solution may seem too bulky, but believe me, it is much easier to read. Moreover, the code can be written more briefly, as shown below.

```
$day = 4;
switch ($day) {
 case 1: print("Sunday"); break;
 case 2: print("Monday"); break;
 case 3: print("Tuesday"); break;
 case 4: print("Wednesday"); break;
 case 5: print("Thursday"); break;
 case 6: print("Friday"); break;
 case 7: print("Saturday"); break;
 default: print("Error");
}
```

This code is quite compact and, at the same time, is easy to read. Code readability is a very important factor affecting debugging, modification, and maintenance of the script.

The keyword break is mandatory. If it is omitted, the code will continue executing, which may produce undesired results. However, the break keyword can be omitted on purpose. Suppose you have to develop universal code to raise any number to the power from 1 to 5. This can be realized by the following piece of code:

```
$sum = 1;
$i = 3;
switch ($i) {
 case 5: $sum = $sum * $i;
 case 4: $sum = $sum * $i;
 case 3: $sum = $sum * $i;
 case 2: $sum = $sum * $i;
 case 1: $sum = $sum * $i;
 default: print($sum);
}
```

I set the initial value of the $i variable to 3.We will calculate the power of three. Here is how the code will do it. The case 5 and case 4 statements will not be triggered; the first case to be triggered will be case 3. Here, the value of the $i variable (i.e., 3) is multiplied by the value of the $sum variable (i.e., 1), and the result is stored in the $sum variable. Because the code of the case 3 statement is not terminated with the break keyword, the code following it will be executed, that is, the code of the case 2 statement. Here, the value of the $sum variable (i.e., 3) is multiplied by 3, with the result, 9, stored in the $sum variable. There is no break after the case 2 code, so the code for the case 1 statement is executed, and the value of the $sum variable (which by now equals 9) is multiplied by 3. The result of the last multiplication is 27, which is also stored in the $sum variable. Because the case 1 statement is also missing the break keyword, the code for the default block is executed, that is, the value of the $sum variable displayed.

Of course, the described example is not efficient, because the concept itself is impractical. A better way to solve this

problem is to use loop controls, which will be considered shortly. But under certain circumstances, you may have to execute code from more than one case statements. For this occasion omitting the break operator will make you job easier.

When developing your scripts, choose the testing method most suitable for the task on hands. The execution speed of all of the described methods is the same, but when multiple comparisons are performed, using the switch statement makes the code much more readable.

Loops

Loops are important program flow control. For example, the problem of raising a number to a power that we used as an example when considering the switch statement, can be solved much easier and more efficient using one of the loop statements. A number is raised to a certain power by multiplying it by itself this number of times. For example, the operation of raising 2 to the power of 3 can be written as follows: 2*2*2. But what if a number has to be raised to the power of 100? This task is somewhat more difficult. Even more difficult is the problem when the power is not known in advance. Here is where loops come to the rescue.

The most often used loop is the for loop. It is also the easiest to understand, so we start our study of loops with it. In the general format it looks as the following:

```
for (start counter value; end counter value; counter step)
 Statements
```

Let's use the for loop to raise a number to a power. The code for this may look like the following:

```
<?php
$nbr = 3; // the number to raise to the power
$raiseTo=4;// the power to raise to
$pwr = 1; // the number raised to the power
for ($i=1; $i<=$raiseTo; $i++) // run the loop four times
   $pwr = $pwr * $nbr;
$i--
print("$nbr to the power of $i is $pwr");
print("<p>");
?>
```

The initial value of the iteration counter, which is what the $i variable is, is 1.Before the statements in the loop are executed, the value of the iteration counter checked. If it is less than or equal to 4, the statements in the loop body are executed. If the counter is greater than 4, the loop is exited and the result is printed with the print() function following the loop. After the loop body is executed, the value of the iteration counter is increased by 1 with the $i++ operation.

At each loop iteration, only one code line is executed: $pwr = $pwr * nbr. Because the final counter value is 4, this line will be executed 4 times.

After the loop completes, the final result is printed out. If more than one statement has to be executed in the loop body, they are enclosed in curly brackets. Thus, the previous example, but with each intermediate result printed out, looks as follows:

```
<?php
$nbr = 3;
$raiseTo=4;
```

```
$pwr = 1;
for ($i=1; $i<=$raiseTo; $i++)
{
  $pwr = $pwr * $nbr;
  print("$nbr to the power of $i is $pwr");
  print("<p>");
}
?>
```

The execution results of this code look as follows:

```
3 to the power of 1 is 3
3 to the power of 2 is 9
3 to the power of 3 is 27
3 to the power of 4 is 81
```

Several variables can be specified as the initial values. For example, the declaration of the $pwr variable can be made in the parameter list of the for statement as follows:

```
<?php
$nbr = 3;
$raiseTo=4;

for ($pwr=1, $i=1; $i<=$raiseTo; $i++)
{
  $pwr = $pwr * $nbr;
  print("$nbr to the power of $i is $pwr");
  print("<p>");
}
?>
```

All initial values are listed delimited with a comma.

Similarly, different conditions for the loop termination can be specified. For example, the following loop executes while the value of the $i variable is less than or equal to 3, or while the value of the $sum variable is less than 100:

```
for ($sum=1, $i=1; $i<=3, $sum<100; $i=$i+1)
 {
  $sum = $sum * 3;
  print("Sum = $sum, Counter = $i <BR>");
 }
```

As you can see, two conditions are checked for in the loop termination parameter: $i<=3 and $sum<100.The comma between the two items in the third parameter is the same as the logical operator or. Thus, the statement can also be written as follows:

```
for ($sum=1, $i=1; $i<=3 or $sum<100; $i=$i+1)
```

If the loop has to be terminated when one of the terminating conditions is met, the conditions are joined with the and operator as follows:

```
for ($sum=1, $i=1; $i<=3 and $sum<100; $i=$i+1)
```

There is, however, a flaw in this code. Executing this code as is, you will see that when it terminates, the sum will exceed 100. This happens because the check is performed before the exponentiation operation. Three to the power of 4 is 81. This is less than 100, the instructions in the loop are executed again, and the result, 81*3=243, is displayed. Only the next check will see that the number has exceeded the allowed value. This flaw is fixed by moving the exponentiation operation into the counter step area as follows:

```
for ($sum=1, $i=1; $i<=3, $sum<100; $i=$i+1, $sum = $sum * 3)
 print("Sum = $sum, Counter = $i <BR>");
```

Now, the value of $sum will increase before going into the loop body and the code will produce the correct results.

The while Loop

The while loop is executed while a certain condition holds. In the general format, it looks as follows:

```
while (condition)
 command;
```

To execute more than one command in the while loop, they have to be enclosed into curly brackets. Consider the following example of raising 3 to the power of 3 using the while loop:

```
$i = 1;
$sum = 1;
while ($i<=3)
{
 $sum = $sum * 3;
 $i = $i + 1;
}
```

In this case, the counter is incremented within the loop body, along with the multiplication operation. As soon as the condition no longer evaluates to true, the loop execution is terminated. There may be situations when the loop body has to be executed at least once? This can be done using the following variety of the while loop:

```
do command
while (condition);
```

In this case, the loop body is executed first and the condition is evaluated second. Thus, the loop body will be executed at least once even if the condition is not met to start with. For example, as in the following code:

```
$i = 1;
$sum = 1;
do
 {
  $sum = $sum * 3;
  $i = $i + 1;
 }
while ($i<=3)
```

The following is yet another variation of the while loop:

```
while (condition)
 Command 1;
 Command 2;
endwhile
```

In the piece of code above, numerous statements that are executed are not enclosed into curly brackets, because if the condition is true, everything following it will be executed until the keyword endwhile is encountered.

Endless Loops

In certain circumstances, we may need a loop to execute endlessly. In this case, we can use a while loop with such condition that always evaluates to true. For example, as follows:

```
while (TRUE)
 {
 }
```

Here, the condition used is true from the start and simply cannot change during the execution, thus, the loop will be executing endlessly.

The for loop can also be made to execute endlessly. This is done as follows:

```
for (;;)
 {
 }
```

Here, there is no condition nor counter increments at all, so the loop will execute endlessly.

I recommend against using endless loops. Even though we will learn further on how to control loops and interrupt their execution when necessary, endless loops, and especially needles branching, put a great workload on the system.

Actually, loops in PHP are not really endless. By default, execution time of any script is limited to 30 seconds. You can change this value to infinity, but this entails a risk of an actually endless loop overloading the system and making your site unavailable.

Controlling Loops

Sometimes it is necessary to interrupt a loop or change the its execution flow.

Loop execution can be interrupted using the break command. For example, as in the following code:

```
$index=1;
while ($index<10)
{
 print("$index <BR>");
 $index++;
 if ($index==5)
  break;
```

```
}
```

According to the while condition, the loop is supposed to execute while $index is less than 10. However, there is an if statement in the loop body that breaks the loop when the value of $index becomes 5. Thus, regardless that according to the while condition the loop could be executed four more times, it is interrupted after only five iterations.

There can also be situations when execution of the loop body of a part of it has to be skipped for certain values of the loop counter. This can be done using the continue statement. The following code shows an example of using the continue statement:

```
$index=0;
while ($index<10)
{
 $index++;
 if ($index==5)
  continue;
 print("$index <BR>");
}
```

This code displays numbers from 1 to 9 skipping 5. The starting value of the $index variable is set to 0. This is done because the first command in the loop body increments $index by 1 before the latter is printed. The $index variable will not be printed when its value is 5, because in this case the continue statement is executed, which passes the execution control to the start of the loop.

Be careful when using the continue statement. See if you can find an error in the following code:

```
$index=1;
while ($index<10)
{
 if ($index==5)
  continue;
 print("$index <BR>");
 $index++;
}
```

There are no syntax errors in the code, and at glance, it should do the same thing as the previous code. The initial value of the $index variable is set to 1, because this time the counter is incremented after the print statement, and everything seems to be alright. When the loop goes through the fifth iteration, the continue statement passes the control to the while statement and the current counter value is not printed. And this is where the error lies. The $index counter was not incremented and remains 5 and once the if statement is reached, the continue statement will be executed again. Thus, the loop hangs between the while and the continue statement.

This problem does not arise with the for loop. Here the counter is incremented when a new iteration is started. Thus the following code will execute without problems.

```
for ($index=1; $index<10; $index++)
{
 if ($index==5)
  continue;
```

```
 print("$index <BR>");
}
```

There is also another way to increment the counter value in the for loop: do it yourself, as in the following code:

```
for ($index=1; $index<10; $index++)
{
 if ($index==5)
  $index++;
 print("$index <BR>");
}
```

But this is only a special case of starting a new iteration, when a certain counter value has to be skipped. The problem or condition may change, and may not see the error then. For example, you may want to skip several iterations in a row. In such a case, it is difficult to determine the new value to which the counter has to be set. For real life programming I recommend that you don't increase loop counters manually and use the continue statement instead.