1. Introduction

PHP is a scripting language through which you can generate web pages dynamically. PHP code is directly inserted in HTML documents through opportune TAGs declaring the code presence and then executed when a client demand the page. PHP is a server-side language, that's to say that PHP code is directly executed by the server, while the client receives processing results as an HTML document. This way of working is different from that of other scripting languages as JavaScript, whose code is first loaded and then executed by the client. PHP way of proceeding involves:

1. All compatibility problems existing between different browsers are completely solved. Client's browser, after the execution of a PHP code on the server, receives a common HTML page and so it is always able to process it correctly. This does not happen with scripting languages interpreted by the client's browser. In this last case the client downloads the script code and tries to process it on the local machine. This procedure works correctly only if the client is equipped with the right software.
2. The server side code processing protects the script code assuring that it is always not visible to clients. That prevents "thefts" of source code.
3. The server side code execution presupposes that your webserver has been well configured. It must be able to recognize HTML documents containing PHP code. In order to make this, it is necessary to install a PHP engine and to edit some lines of the webserver configuration file. We will see how to perform these operations, in the next chapter.
4. Server side code processing needs calculation resources (cpu time). An high number of clients' requests could overload the server.

To make the webserver able to recognise HTML documents containing PHP code, .php, .php4 or .phtml extensions are used in place of the .html ones. These extensions can change according to the webserver configuration.

Suppose for instance, that a client demands the following page (example1.php).

```
<HTML>
    <HEAD>
        <TITLE>Example 0.01</TITLE>
    </HEAD>
    <BODY>
        <?
        echo ("<H1>This is an example</H1>");
        ?>
    </BODY>
</HTML>
```

this page will be recognised, thanks to the extension from which it is characterised and it will be processed as a HTML document containing PHP code. The code is interpreted before the output's transmission to the client. The webserver passes the page containing the code to the PHP engine which process it. Once interpreted the PHP engine sends the resulting HTML page to the webserver which transmits it to the client. The client receives the following HTML document:

```
<HTML>
    <HEAD>
        <TITLE>Example 0.01</TITLE>
    </HEAD>
    <BODY>
        <H1>This is an example</H1>
    </BODY>
</HTML>
```

As you see, in the document sent from the web server to the client there is no sign of PHP code. The code has been interpreted and replaced with HTML lines. The client will never be able to deduce that the received HTML page is the result of a dynamic PHP code processing.

PHP instructions are placed inside of special the TAGs <?php e ?>. In this way PHP engine can distinguish the PHP syntax from the rest of the document. It is possible to use different TAGs editing the engine configuration file php.ini.

3. First steps

In this chapter we introduce the first script PHP that consists in the classic program that shows the " HELLO WORLD " line. Copy the following HTML code, paste and save it with the name helloworld.php in your webserver root directory (or in the webserver's php files directory), then start your browser and connect to:

http://127.0.0.1/helloworld.php

or similar, according to the way you have configurated your system.
If it works correctly you can proceed in the reading of this tutorial, otherwise return to the previous chapter and check you have carried out all the suggested configuration steps correctly. If you can't solve the problem, before abandoning, try to have a look to the software documentation.

```
<HTML>
    <HEAD>
        <TITLE>Example 3.01</TITLE>
    </HEAD>
    <BODY>
        <?
        echo ("<H1>Hello World!</H1>");
        ?>
    </BODY>
</HTML>
```

4. Variables

A variable is a block of memory, accessible through a name chosen by the software developer, in which a value is stored. This value, usually set up on a default value at the beginning of the application, can change during the execution of the program. PHP requires variables' names to begin with the dollar ("$") character. Variables' names can be composed by capital and lowercase letters, digits and underline characters ("_"). Instead, it is not possible to include spaces or other special or reserved characters to define the name of a variable. Remember that PHP is a case-sensitive language. This means it distinguishes between capital and lowercase letters. For instance, if we write:

```
<HTML>
    <HEAD>
        <TITLE>Example 4.01A</TITLE>
    </HEAD>
    <BODY>
        <?
        $VAR1 = 5;
        echo ($var1);
        ?>
    </BODY>
</HTML>
```

We'll get an error message because the PHP interpreter does not recognise the $var1 variable (lowercase written).

A script where a variable is declared and then printed, follows. Read it with attention because it contains some particularities we will discuss about.

```
<HTML>
<HEAD>
<TITLE>Example 4.02</TITLE>
</HEAD>
<BODY>
<?
$website = "http://www.bitafterbit.com";
echo ("<BR>Surf to: $website");
echo ('<BR>Surf to: $website');
?>
</BODY>
</HTML>
```

The program declares and prints the variable $website. Observe that in the echo() function, whose purpose is to write a string on the monitor, two different kinds of inverted commas are used: they are the single and double ones. In this tutorial we will refer to the double inverted commas (") with the name of inverted commas and to the single ones (') with the name of quotes. The main difference between these two types of syntax consists in the fact that the PHP interprets what is enclosed in the inverted commas, while everything appearing between quotes is considered a constant value and it is not interpreted. The example we have just proposed produces the output:

```
    Surf to: http://www.bitafterbit.com
    Surf to: $website
```

The text $website is in fact interpreted and replaced with the value of the correspondent variable only in the first echo() instruction. This because in the first instruction we have used the inverted commas to enclose the text to print. The result of this first instruction is:

```
    Surf to: http://www.bitafterbit.com
```

In according with what we have just said, in the second echo() instruction, where quotes have been used in place of inverted commas, the embedded text is not analysed, because considered as a constant. The output of the second echo() instruction therefore is:

```
Surf to: $website
```

It is important to make attention to this last characteristic of PHP. It is different from other programming languages where all what appears enclosed between inverted commas is considered a constant value. We will talk about strings accurately in next chapters when we'll analyse the PHP's datatypes.

In PHP some variables, holding particular information, exist. These variables stores information about the webserver, the operating system and the PHP engine configuration. We'll talk about these variables in next chapters.

5. if... then... else... instruction

The if... then... else... instruction, through which it is possible to manage the application's logic flow, has the following syntax in PHP:

```
    if (condition)
        istruction1;
    else
        istruction2;
```

condition is a boolean expression that generally consists in comparing a variable with a constant value or two variables between themselves. A boolean expression returns a true or false value. If a true value is returned, instruction1 is executed, otherwise, when a false value is returned, istruction2 is executed.

Look at the following example code:

```
<HTML>
    <HEAD>
        <TITLE>Example 5.01</TITLE>
    </HEAD>
    <BODY>
        <?
        $Lastname = "Bit";
        if ($Lastname == "Bit")
            echo ("Hello, Mr. Bit!");
        else
            echo ("Who are you?");
        ?>
    </BODY>
</HTML>
```

it will supply the output:

```
Hello, Mr. Bit!
```

The program checks the boolean expression (condition) $Lastname == "Bit", that turns out to be true; so the immediately successive echo() instruction is executed. The second echo() instruction, contained in the else branch, will not be executed.

To execute comparisons between variable and/or constant values, you can use the comparison operators. Their PHP syntax is shown in tab.5.1.

==	equal to
>	greater than
<	minor than
!=	not equal to
>=	greater or equal to
<=	minor or eqaul to

Tab.5.1: comparison operators

Note that the 'equal to' operator (==) is expressed by two symbols "=" (equal) placed side by side ("=="). One of the most common errors is to forget a symbol "=" while writing a boolean expression (comparing two variables, for instance). This kind of error seriously alters the logic flow of the program. Look at the following lines:

```
    if ($Lastname = = "Bit") echo "Hello, Mr. Bit!";
    $Lastname = "Bit";
```

The first instruction is processed as a comparison operation and it returns a true value if the variable $Lastname stores the value "Bit", false otherwise. The second instruction instead assigns the value "Bit" to the variable $Lastname.

PHP proposes some alternative ways to write an if... then... else... instruction. Here is the first one:

```
<? ...
if (condition) : ?>
    HTML instruction
<? else : ?>
    HTML instruction
<? endif ?>
```

The condition is followed by a first block of HTML instructions that will be executed only if the condition returns a true value. The else keyword is followed by a second HTML block that will be executed if the condition returns a false value. The if...then...else... instruction finishes with the endif keyword.

The following PHP code is functionally analogous to the first example of this chapter:

```
<HTML>
        <HEAD>
                <TITLE>Example 5.01</TITLE>
        </HEAD>
        <BODY>
                <?
                $Surname = "Bit";
                if ($Surname == "Bit"): ?>
                        Hello, Mr. Bit!
                <? else: ?>
                        Who are you?
                <? endif ?>
        </BODY>
</HTML>
```

The output will be obviously the same:

```
        Hello, Mr. Bit!
```

If we have more instructions as result of the if... then... else... structure, it is necessary to enclose them in the brackets { and } . The brackets task is to create a block where more instructions can be held together. Otherwise we can use the endif keyword as it follows:

```
if (condition)
        istruction1A;
        istruction1B;
        ...
        istruction1Z;
endif;
```

When we are interested in analysing more conditions before choosing the instruction to execute, we can create a if... then... else... chain structure. This allows us to specify a set of conditions that will be checked up sequentially. Each condition is checked up only if all the previous ones have returned a false value. When the first true condition is found, all the successive ones will not be processed. The syntax of an if... then... else... chain instruction is the following:

```
if (condition)
        istruction1;
elseif
        istruction2;
elseif
        istruction3;
        ...
else
        istruction;
```

The if... then... else... instruction is completed by the use of the logic operators AND, OR and NOT. Such operators are expressed in PHP through the syntax shown in table 5.2.

&&	AND
||	OR
^	XOR
!	NOT

Tab.5.2: logic operators

The use of these operators allows you to write sophisticated boolean expressions that will be estimated according to the truth tables (tab.5.3) that you probably already know.

a	b	a AND b	a OR b	NOT a
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Tab.5.3: Truth tables

6. switch instruction

When it is requested to execute a set of comparisons on the same variable, instead of writing an if...else...else... instruction, we are used to choose the switch one. Its syntax is:

```
switch ($VarName) {
    case V1: Istruction1;
    break;
    case V2: Istruction2;
    break;
    ...
    case VN: IstructionN;
    break;
    default: Istruction;
    break;
}
```

$VarName is the variable on which comparisons are performed, while V1, V2, V3 are variables or constant values to which $VarName is compared to. When one of the comparison operation results verified the corresponding instruction is executed. If no comparison are verified, the instruction held in the default branch is executed.

Suppose, for instance, that we need to compare the value of an integer variable named $var, with a collection of integer constants. We'll write:

```
<?
$var=2;
switch ($var) {
        case 1: echo ("\$var isn't 1");
        break;
        case 2: echo ("\$var isn't 2");
        break;
        case 5: echo ("\$var isn't 5");
        break;
        default: echo ("\$var value isn't 1, 2 or 5, but $var");
        break;
}
?>
```

We could get the same result by writing an if...else...else instruction as it follows:

```
<?
$var=2;
if ($var==1)
        echo ("\$var value isn't 1");
elseif ($var==2)
        echo ("\$var value isn't 2");
elseif ($var==1)
        echo ("\$var value isn't 3");
else
        echo("\$var value isn't 1, 2, or 5, but $var");
?>
```

observe that the escape character (" \ ") has been used before the name of the variable. Its purpose is to visualize the '$' character, avoiding that the $var variable is interpreted and replaced with its value.

7. for loops

Another useful instruction that well controls the logic flow of the program, is the for instruction. It allows software developers to demand the execution of one or more instructions more times, until a particular condition returns a true value. It's the condition of escape (or exit condition) of the loop. The PHP syntax of a for loop instruction is the following one:

```
for ($v=startValue; condition_on_v; istruction_on_v){
        LoopBody;
}
```

$v is the control variable of the cycle. $v is initialized to the value startValue when the loop begins. condition_on_v is the condition of control of the loop. The loop ends when condition_on_v returns a false value (that's to say when the condition_on_v is no more verified). The first check on the condition is carried out before the first iterance; therefore, it is possible that the instructions held in the body of the loop (LoopBody) are never executed. This happens when condtion_on_v immediately returns a false value. instruction_on_v is an instruction that modifies the value of the control variable $v. With this instruction v generally moves to a value that makes condition_on_v false; in this way the loop execution ends. This should happen after a certain number of iterances. If it is not so, the risk would be to generate an infinite loop, that to say a never ending loop.

Here are some examples in which the for instruction is used. The first example simply prints the numbers from one to ten. [Pending: example1]

The control variable of the loop is $k. $k is first declared and then initialised on a zero value at the cycle's beginning. The control condition of the cycle returns a true value until $k is smaller than 10. The instruction on $k consists in its increment at every iterance. The body of the loop is a echo() instruction. At the first iterance $k has been just initialised and its value is zero. The output of the first iterance will be ($k+1), that is 1. In the second iterance, $k has assumed value 1, therefore the output of the echo() instruction will be 2. The cycle repeats until $k does not assume value 10. At this step the condition of control (k < 10) turns out false and the cycle exits. The output of the program is:

```
1
2
3
4
5
```

The second example uses a for cycle to create an HTML table

```
<!--file: helloworld.html -->
<HTML>
    <HEAD>
        <TITLE>Example 7.01</TITLE>
    </HEAD>
    <BODY>
        <?
        echo ("<TABLE ALIGN=CENTER BORDER=1 CELLSPACING=5>");
        for ($j=1;$j<=5;$j++) {
            echo ("<TR>");
            for ($k=1;$k<=3;$k++)
                echo ("<TD> Line $j, Cell $k </TD>");
            echo("</TR>");
        }
        echo ("</TABLE>");
        ?>
    </BODY>
</HTML>
```

The code generates the output:

Line 1, Cell 1	Line 1, Cell 2	Line 1, Cell 3
Line 2, Cell 1	Line 2, Cell 2	Line 2, Cell 3
Line 3, Cell 1	Line 3, Cell 2	Line 3, Cell 3
Line 4, Cell 1	Line 4, Cell 2	Line 4, Cell 3
Line 5, Cell 1	Line 5, Cell 2	Line 5, Cell 3

8. while loops

A while loop is a little less sophisticated than a for loop, but it carries out approximately the same function. It is composed by a body containing some instructions and an exit condition. At the beginning of the cycle and every time that all the instructions in the body are executed, the exit condition, constituted by a boolean expression, is checked up. The loop ends when the condition returns a false value. The syntax of a while loop is the following:

```
while (condition){
    BodyOfLoop;
}
```

condition is the control condition of the loop. The first check of the condition's validity takes place at the beginning of the cycle, before the first iterance. Also in this case, it can happen that the instructions included in the loop's body are never executed. This happens when the condition immediately returns a false value. As it is for the for loops, also with while loops there is the danger to create a never ending loop. This happens when iterance after iterance, the exit condition never returns a false value.

The first example, that we are going to propose, supplies the same output of the first example proposed in the previous chapter: it prints the numbers from one to ten.
[Pending: example]

observe that in this case it is necessary to supply to the increment of the control variable $k, by adding an increase instruction into the loop's body. The output of the program is:

```
1
2
3
4
5
```

The same application seen in the previous chapter, that creates an HTML table, has been realised replacing the for instruction with the while one.
Fig.5.1 shows the program's output. It is perfectly equal to the one we saw in the previous chapter.

```
<HTML>
    <HEAD>
        <TITLE>Example 7.01</TITLE>
    </HEAD>
    <BODY>
        <?
        $j=1;
        echo ("<TABLE ALIGN=CENTER BORDER=1 CELLSPACING=5>");
        while ($j<=5) {
            echo ("<TR>");
            $k=1;
            while ($k<=3) {
                echo ("<TD> Line $j, Cell $k </TD>");
                $k++;
            }
            echo("</TR>");
            $j++;
        }
        echo ("</TABLE>");
        ?>
    </BODY>
</HTML>
```

the output will be:

Line 1, Cell 1	Line 1, Cell 2	Line 1, Cell 3
Line 2, Cell 1	Line 2, Cell 2	Line 2, Cell 3
Line 3, Cell 1	Line 3, Cell 2	Line 3, Cell 3
Line 4, Cell 1	Line 4, Cell 2	Line 4, Cell 3
Line 5, Cell 1	Line 5, Cell 2	Line 5, Cell 3

PHP proposes an alternative syntax for while loops, too. It is possible to avoid the use of the brackets "{" and "}" by enclosing the instructions that constitute the body of the loop with the endwhile keyword. The syntax of a while expression using the endwhile keyword follows:

```
while (condition)
    BodyOfLoop;
endwhile;
```

As it happens with the if... then... else... instruction, we can set up a while loop whose body is constituited by some HTML instructions. The syntax is:

```
<?
while (condition) : ?>
    HTML instructions
<? endwhile ?>
```

In table 5.2 for and while loops code is compared. Each couple of cycles executes the same operations and returns exactly the same output.

for loop	*while* loop

for ($i=0; $i<10; $i++) {
    echo("Iterazione n?$i<BR>");
}
	$i=0;
	while ($i<10) {
	    echo ("Iterazione n?$i<BR>");
	    $i++;
	}

Tab.8.2: Comparison between for and while loops

9. Funzioni

Functions are blocks of instructions used by the main program in order to get its goal. A function generally executes a basic, not complex operation. Every function has a name that identifies it. A function can get as input an arbitrary number of parameters (variables), according to which it executes some specific instructions. Then, if necessary, an output value is returned to the main program. This value can be saved and used when necessary. Functions are a basic instrument in every programming language. This allows software developers to share the application's goal in many blocks of elementary instructions. This simplifies the application's building, testing, debugging and editing process. This technique is the most popular between the "imperative programming languages" and it is called modular programming. The application demands the execution of every single function when a particular computation is needed. The progressive order in which the functions are executed can change according to the user's input data and/or according to the result of the functions already executed.
A function is constituted by a head and a body. In PHP the syntax is as it follows:

```
function FunctionName($par1, $par2, ..., $parN){
    FunctionBody;
}
```

FunctionName is the name of the function. $par1 , $par2 ..., $parN are the N parameters that the function receives in input when its execution is needed (N can be 0, too). The parameters, sent to a function, do not have to be necessarily elementary data types (integers, char, strings...), they can be every type of objects defined by the software developer. Between the brackets ("{" and "}") is the function's body, containing the instructions to be executed at the function's activation. Executed these instructions, a value, called function's output, is generally returned.

Now we want to define a function that visualizes some text in a HTML document using a paragraph TAG. This text is passed as parameter. We can write a function like the following one:

```
function writeString ($str){
    echo("<FONT FACE=\"Comic Sans MS\" SIZE=3><P ALIGN=\"JUSTIFY\">$str</P></FONT>");
}
```

To use the writeString() function, we will write the following line i nside the PHP code:
writeString("Text to be formatted and displayed by the function");
the complete HTML code of a page in which the writeString() function is defined and used follows.

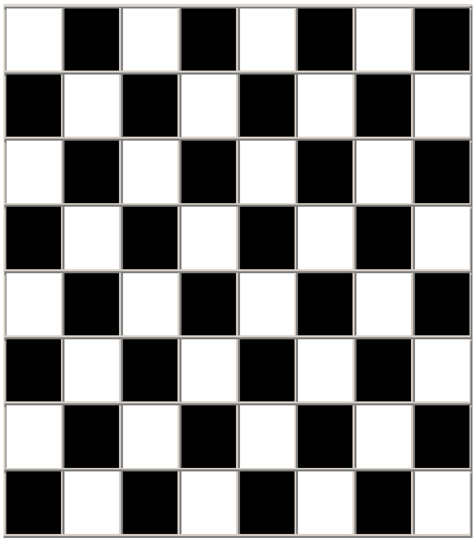
```
<HTML>
	<HEAD>
		<TITLE>Example 3.01</TITLE>
	</HEAD>
	<BODY>
		<?
		function writeString ($str){
			echo("<FONT FACE=\"Comic Sans MS\" SIZE=3><P ALIGN=\"JUSTIFY\">$str</P></FONT>");
		}

		writeString ("Text to be formatted and displayed by the function");
		?>
	</BODY>
</HTML>
```

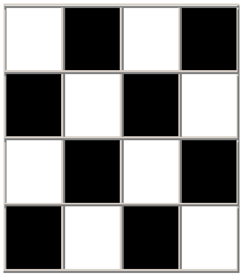
The code will generate the following output:

```
Text to be formatted and displayed by the function
```

The following PHP script is made up by a function that creates a chessboard. The chessboard's dimensions can be passed to the function as input parameters.

The output will be:





```
<HTML>
    <HEAD>
        <TITLE>Example 3.01</TITLE>
    </HEAD>
    <BODY>
        <?
        function createBoard ($lines, $cols){
            $j=1;
            echo ("<TABLE ALIGN=CENTER BORDER=1 CELLSPACING=0>");
            while ($j<=$lines) {
                echo ("<TR>");
                $k=1;
                while ($k<=$cols) {
                    if (($j+$k)%2>0)
                        echo ("<TD WIDTH=30 HEIGHT=30 BGCOLOR=#000000> </TD>");
                    else
                        echo ("<TD WIDTH=30 HEIGHT=30 BGCOLOR=#FFFFFF> </TD>");
                    $k++;
                }
                echo("</TR>");
                $j++;
            }
            echo ("</TABLE><BR><BR>");
        }

        createBoard(8,8);
        createBoard(4,4);
        ?>
    </BODY>
</HTML>
```

10. Datatype

To declare a variable it is sufficient to choose a name, through which the variable will be identified, and then initialize it. It is not necessary, as it happens for other programming languages, to declare the type of the variable. PHP is able to deduce the type of a variable from the value with which it has been initialized. For instance, if we write:

```
    $var1 = 7;
```

PHP will initialize a variable of integer type, called $var1, storing value 7.
If instead we write:

```
    $var2 = "This is a variable";
```

it will be initialized a variable of string type, called $var2, storing the value "This is a variable ".

We have already indicated how to proceed to choose a variable's name. The type of a variable indicates which kind of data the variable will store. The concept of type of data (datatype) is particularly important, not only in PHP, but all over the world of computer science. It is usual to distinguish between atomic types, or simple one and structured types. An analogous distinction could be made in PHP, too. Now we are going to list and to describe atomic data types.

In PHP the following atomic datatypes are admitted.

Integer;
Floating Point number;
String;

10.1. Datatypes: Integers

Variables of integer type represent positive and negative integers, or zero. The declaration and initialization of an integer variable has the following syntax:

$VarName = IntValue;

$varName is the name we have chosen for the variable, while IntValue has to be replaced with a datatype compatible number (a positive, negative or zero number).

In order to execute operations with integers we can use the classic math operators shown in tab.10.1.1 and the increase operators in tab.10.1.2.

Arithmetic operators	
\$a + \$b	Addition
\$a - \$b	Subtraction
\$a * \$b	Multiplication
\$a / \$b	Division
\$a % \$b	Module (division's rest)

Tab.10.1.1: Arithmetic operator

Increase operator	
\$a++	Return after increase
++\$a	Return before increase
\$a--	Return after decrease
--\$a	Return before decrease

Tab.10.1.2: Increase operator

To make comparisons between integer variables we can use the comparison operators shown in tab.10.1.2.

Comparison operators	
\$a == \$b	equal to
\$a != \$b	not equal to

$a < $b	Minor than
$a <= $b	Minor or equal to
$a > $b	Greater than
$a > $b	Greater or equal to

Tab.10.1.3: Comparison operators

10.2. Datatypes: floating point numbers

Variables of floating point datatype are used to store numbers with decimal part. The declaration and initialization of an integer variable has the following syntax:

    $varName = FloatValue;

$varName is the name we have chosen for the variable, while IntValue has to be replaced with a datatype compatible number.

    $x = 3.13
    $y = 0.143

In order to execute operations with floating point numbers we can use the classic math operators shown in tab.10.2.1 and the increase operators in tab.10.2.2.

Arithmetic operators	
$a + $b	Addition
$a - $b	Subtraction
$a * $b	Multiplication
$a / $b	Division

Tab.10.2.1: Arithmetic operator

Obviously the module operator (%) lacks, because the division between decimal number has no rest.

Increase operator	
$a++	Return after increase
++$a	Return before increase
$a--	Return after decrease
--$a	Return before decrease

Tab.10.2.2: Increase operator

To make comparisons between floating point numbers we can use the comparison operators shown in tab.10.2.2.

Comparison operators

$a == $b	equal to
$a != $b	not equal to
$a < $b	Minor than
$a <= $b	Minor or equal to
$a > $b	Greater than
$a > $b	Greater or equal to

Tab.10.2.3: Comparison operators

10.3. Datatypes: strings

The declaration of a string variable is analogous to the declaration of any other variable. The only difference that the value to store in a string variable must be enclosed between inverted commas or quotes. We have already talked about the existing difference between inverted commas and quotes in chapter 4. Here are two string variable declarations. The first one uses inverted commas, while in the second one quotes are used.

```
$str1 = "This is a string datatype variable";
$str2 = 'This is a string datatype variable, too';
```

In chapter 4 we said that PHP interprets all what is enclosed between inverted commas and carries out the needed substitutions; on the contrary, it leaves unchanged all what is enclosed between quotes. Let's analyze some examples that will help us to familiarize with PHP way of operating. Observe the following script and try to anticipate the output.

```
$Civicc = 8;
$Address = "Via Tespi, $Civic";
echo 'My address is $Address';
```

The script's output will be:

```
My address is $Address
```

If you get right it means you understood. If you have still some troubles to understand why PHP has printed the variable name instead of its value, the answer is simple: PHP does not interpret the content between quotes.

The second script is:

```
$Civic = 8;
$Address = "Via Tespi, $Civic";
echo "My address is $Address";
```

L'output di questo script ?il seguente:

```
My address is Via Tespi, 8
```

By replacing the quotes with the inverted commas at the third line, the PHP interprets the text before printing it. The $Address variable is replaced with its value.

At this point a doubt could born, In fact the described syntax does not allow to write everything. Suppose we need to write something like:

```
The value of $Address variable is Via Tespi, 8
```

It is not possible to use the inverted commas, because the script:

```
$Civic = 8; $Address = "Via Tespi, $Civic"; echo "The value of $Address variable is $Address";
```

would give the output:

```
The value of Via Tespi, 8 variable is Via Tespi, 8
```

At the same time, we could not use the quotes because the script:

```
$Civic = 8;
$Address = "Via Tespi, $Civic";
echo 'The value of $Address variable is $Address';
```

would give the output:

```
The value of $Address variable is $Address
```

Something like the following script could instead work correctly:

```
$Civic = 8;
$Address = "Via Tespi, $Civic";
echo 'The value of $Address variable';
echo " is $Address";
```

that in fact supplies the right output:

```
The value of $Address variable is Via Tespi, 8
```

However it looks like a little bit complicated. An easier way exists. It is possible to use between inverted commas the escape character ("\"), that allows to print some special characters. This solves our problem, too.

```
$Civic = 8; $Address = "Via Tespi, $Civic"; echo "The value of \$Address variable is $Address"
```

it will print:

```
The value of $Address variable is Via Tespi, 8
```

Even if we have used inverted commas, by adding the escape character ("\") before the variable name, the variable has not been interpreted.

An analogous procedure allows us to construct a string containing the inverted commas character. Observe the following script:

```
$Civic = 8; $Address = "Via Tespi, $Civic"; echo "The value of $Address variable is \"$Address\""
```

It will visualise the line:

```
The value of $Address variable is "Via Tespi, 8"
```

Other escape codes used to represent special characters, are shown in table 10.3.1.

Codici di escape	
\"	inverted commas
\n	newline
\r	carriage return
\t	tab
\$	dollar
\\	backslash

Tab.10.3.1: Escape codes

11. Arrays

An Array is a sort of multi-valued variable. It stores one or more values, each one identified by a number representing its position in the array. When we write, as example, the line:

```
    $myArray[5]
```

we're referring to the sixth element (the first one is numbered 0) of the array called myArray. To assign the value 5 to the second element of the same array we'll write:

```
    $myArray[1] = 5;
```

At the same time to have the third value of the array stored in myVariable we write:

```
    $myVariable = $myArray[2];
```

Arrays are often used to collect related variables in the same object, and so with the some name.

The declaration of an array variable can be made by listing the elements that the array contains as it is shown below:

```
    $varArray = array (element1, element2, ..., elementN);
```

$varArray is the name we chose for the array variable, while element1, element2..., elementN are the values stored in the array in position 1, 2..., N. When we need to add a string variable into an array we'll use the inverted commas or quotes.

There is a second way to declare an array variable. It is possible to assign a value to each position of the array through the following syntax:

```
$varArray[0] = element1;
$varArray [1] = element2;
...
$varArray [2] = elementN;
```

The result is analogous to the previous one.

Remember that the first element of an Array is always identified by number zero (0). This means that if an array holds eight elements, they will be numbered from 0 to 7.

Adding elements to an array (append instruction) can be made through the following syntax:

```
$varArray[] = elementN1;
```

the instruction will append elementN1 to the $varArray array variable, in position N+1.

Using an Array data type could be useful during data processing operations, too. If we decide to save variables in the same array object it will be easier to manage them. This because it is possible to process (read/write/edit) all them by writing a cycle instruction (for or while). In fact the cycle's control variable can be used as the array's index.

Now we are going to define an array and print its content with a for loop. Then we'll add a new value to the array and finally we'll print the array again. This is the code:

```
<HTML>
    <HEAD>
        <TITLE>Example 11.01</TITLE>
    </HEAD>
    <BODY>
        <?
        $vv = array ("<BR>","Welcome to ", "Bit After Bit. ", "Loading, just ", 5, " minutes");
        for ($k=0;$k<6;$k++)
            echo ($vv[$k]);

        $vv[]= " or a little more!";

        for ($k=0;$k<7;$k++)
            echo ("$vv[$k]");
        ?>
    </BODY>
</HTML>
```

The script will generate an HTML document whose output is:

```
Welcome to Bit After Bit. Loading, just 5 minutes
Welcome to Bit After Bit. Loading, just 5 minutes or a little more!
```

The array variables we've described since now are the scalar arrays. Working with scalar arrays, you can access to any elements through an index. PHP allows to define a second kind of arrays called the associative arrays. An associative array is an array in which the elements can be accessed through a keyword instead of the index. Each element, in an associative array, is characterized by a keyword. This means we'll no more use indexes to read/write/edit elements, because they have been replaced by keywords. The associative arrays is constituted by couples "key - element".

The syntax we'll use to define an associative array is the same we have used to define scalar array. We just need to replace indexes with keywords:

```
    $varArray = array (
    "key1" => element1;
    "key2" => element2;
    ...
    "keyN" => elementN;
    );
```

varArray is the name of the array variable , key1, key2..., keyN are the N keyword through which we can access to the respective elements element1, element2, ..., elementN.
Look at the following example where associative arrays are used.

```
$website = array(
"name" => "Bit After Bit",
"URL" => "http://www.bitafterbit.com",
"email" => "webmaster@bitafterbit.com",
);
```

An associative array with three elements has been defined. The first element, accessible through the keyword "name" is of string type and contains the value "Bit After Bit". The second element is a string too, it is accessible through keyword "URL" and stores the value "http://www.bitafterbit.com". The third and last element are still a string containing the value "webmaster@bitafterbit.com".

Once defined this Array we could write the following script:

```
<?
echo ("Surf to the website $website[name] at $website[URL]!");
?>
```

in order to visualise the line.