How to install Xampp:-

Xampp download.
↓
choose New Xampp with Php versions.
↓
Download the latest

Xampp download
↓
Download Xampp-Apache friends
↓

Static website → It is one whose content doesnot change.
Dynamic website → It is one whose page content changes each time a visitor enters to site or reloads/refreshes a page from the site.

There are a variety of languages available via which dynamic page content can be served to a Browser.

PHp is an open source hypertext preprocessor i.e a server side, programming (Scripting) language extensively used at the web seever to process data provided by an HTML form which was used to capture & validate such data.

Note:- embedded PHP code in an HTML document

Embedding PHP in HTML can generate dynamic web pages.

The dynamic content is processed at the web server and then sent to the requester's Browser as pure HTML.

<?php
?>

PHP code is separated from HTML code with the symbols <?php (Less-Than Sign followed by a question mark and the word php) and ?> (a question mark followed by a Greater-Than Sign)

<?php indicates the start of a PHP block or code & tells the web server that all statements that follow until ?> are PHP statements.

Valid ways (to indicate the start & end of a PHP code block).

<?php
?>

or

<SCRIPT LANGUAGE = "php">
</SCRIPT>

PHP runs at the web server,
The web server executes the embedded PHP code first and then sends the output of the PHP code, as pure HTML code back to the browser that requested it.
The result is a web page with dynamic content

microsoft edge
↓
localhost/learn_php

Note → the above can be verified by viewing the source code of the HTML page when it is displayed in the Browser. This can be done using the view source option. This source code will contain no PHP code segments at all. It is now pure HTML code

Comments

PHP provides the following ways to set comments.
// at the beginning of each line. (single line comment)
/* and */ to comment out several lines (i.e a multi-line comment)
(i.e a single line comment)

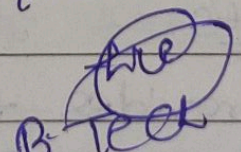
· Print ("Hello world");
It is the command which tells the PHP interpreter what to do.

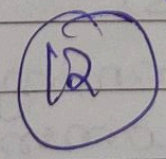
echo "Hello";
echo is another command which tells the PHP interpreter what to do.

Note →
Create the file using any ASCII editor & save it as myfirst.php.
Save the file in the root folder i.e. htdocs.

Browser: localhost/mypage.php

The web server runs the code & delivers the output the Browser





Variables

Datatypes

1. String.
2. Integer.
3. Float.
4. Boolean.
5. Array.
6. Object
7. Resource.

Variables

Variables are nothing but identifiers which point to a memory location (RAM) in which data is stored.

PHP variables are weakly typed. This means that when a PHP variable is being defined their data type is not declared prior their use.

Note :- PHP is case sensitive.

php is called hypertext preprocessor as its libraries are already compiled. when any person request for any php page in the address bar of the browser that request is first sent to the server then server interpret php files and return back response in form of HTML

Extension

→ .php
.phps
.phtml

Note → PHP is compatible with almost all webservers used today (Apache, IIS & so on)

XAMPP

→ cross-platform, Apache, mySQL, PHP and Perl.

(able to use on different types of computers or with different software packages)

XAmpp is one of the widely used cross-platform web servers which helps developers to create and test their programs on a local webserver.

PHP attributes are:

PHP stands for Hypertext Preprocessor.

→ It is a web server-side scripting language

→ PHP scripts are executed at the web server. Their output, which is pure HTML is returned to the requesting Browser.

→ PHP can connect natively to many database engines such as MySQL, oracle, postgre, sqlite and so on.

→ It is open source software.

→ PHP is totally free of cost to download & use.

17. THE BASICS OF PHP

All programming environments are built around syntax, some of which is used to declare and initialize variables. Variables are user defined containers which are used to store user defined values for later use. PHP permits the creation and use of different variables.

For example, the passport number of a client may have to be referenced often by program code, it would be nice to define **ppno** (i.e. passport number) as a variable once, load it with a value and just refer to its contents, whenever required. Now any time the client's passport number needs to be referenced simply refer to the variable **ppno** which will return the value it stores.

DATA TYPES

PHP has several types of variables. All hold a specific class of information, **except one**.

The PHP variable types are:

- **String** - Strings are a sequence of characters that are always internally NULL terminated. String variables hold characters. Usually a set of characters such as "S", "Ivan", "PHP is my favorite language" and so on. Strings can be as short or as long as desired. *There is **no limit** to size*
- **Integer** - Integer variables hold whole numbers, either positive or negative such as 1, -50, 22937932 and so on. *There is a **maximum limit** to the size of integers*. Any numbers lower than -2147483647 and higher than 2147483647 are automatically converted to floats as this data type can hold a much larger range of values
- **Float** - Float variables hold fractional numbers as well as very high integer numbers such as 3.5, 1.00000001, 2147483647000 and so on
- **Boolean** - Boolean variables hold **TRUE** or **FALSE**. Behind the scenes these are just integers. PHP considers the number **0** to be **FALSE** and **everything else** to be **TRUE**
- **Array** - Arrays are a special variable type, which hold multiple values. They are a great way of storing variables that are related such as colors, days of the week, members of a team or items in a shopping basket
- **Object** - Object variables are complex variables that have their own functions (i.e. methods) for accessing and/or manipulating their content
- **Resource** - Resource variables are variables that hold anything that is **not** PHP data. This could be picture data loaded from a file, the result of an SQL query and so on. A Resource type variable is used like any other variable with the key difference being that they should be freed up by the user when not required

VARIABLES

Variables are nothing but identifiers which point to a memory location (RAM) in which data is stored. Variables in PHP are quite different from compiled languages such as C and Java. This is because PHP variables are **weakly typed**. This means that when a PHP variable is being defined their data type is not declared prior their use. A PHP variable takes its data type from the user defined value that is being stored in it. As a result, a PHP variable can change its type (based on the user value, data type), **as often as needed**.

A PHP variable must be named / declared starting with the **$** character followed by a letter. The rest of the variable name can consist of a mix of both alphabets and numbers.

The following are three examples of valid names for variables:

- **$**city
- **$**address2
- **$**age_30

The _ (underscore) character can also be used in variable names. It is used as a **replacement for space**. Space is a **character** that **cannot** be used when naming a PHP variable.

The following characters are **not allowed** in a variable name and cause errors if used:

*	(asterix)
+	(plus)
#	(hash)
@	(at the rate)
-	(minus)
&	(ampersand)
£	(pound)

Tip

There is no limit on the size of variables in PHP.

Once a variable is named, **it is empty** until assigned a value. To assign a value to variable:

```
$city = 'Mumbai';
```

Everything inside the single quotes will be assigned to the variable named **city**. The **named variable** is on the **left** side of = (i.e. the assignment operator) and the **value** to be held by the variable is on its **right**.

Tip

PHP does not require variables to be declared before being initialized. They can simply be populated with a value and put into action whenever and whereever required.

Similarly, numeric values can be assigned.

```
$age = 24;
```

Here, **$age** is assigned a value of **24.** The only difference is that the value 24 is passed **without** quotation marks.

NOTE

Variables declared **without a dollar sign will not work**. This is a common mistake by new PHP programmers!

Different types of values can be assigned to variables. Integer and String types have already been demonstrated.

Irrespective of the value a variable holds in the future, PHP takes care of integer and string conversions **on the fly**. Hence, the same variable can hold different types of user values, at different instances in time.

One thing that causes many hours of hair pulling and anguish is case sensitivity **as PHP is case sensitive**.

Example:

```
<?php
    $Name = "Ivan Bayross";
    echo $name;
?>
```

In the above example a variable named **$Name** is declared. The value **Ivan Bayross** has been assigned but while accessing the value held by the variable, the variable name is **misspelled** (i.e. is in lowercase). This example when run displays the following error on the VDU screen:

```
Notice: Undefined variable: name on line 3
```

Numbers

It's really simple to deal with numbers in PHP. Just use them as required. All the normal rules about number precedence apply.

```
$x = 5;
$y = 10;
$z = 2 + 3 * $x + 5 * $y; // This is evaluated as 2+(3*5)+(5*10) = 67
echo $z;
```

When a variable is referenced by its name, PHP knows that it's the value held in the variable that must be processed and not the variable itself. This is **automatically** done **before** a new value is stored in the variable on the **left** hand side of the assignment operator. Hence, something like this can be done:

```
$x = 8;
$y = 4;
$x = (2 * $x + 5)/$y;     // Evaluated as $x = (2* 8 + 5)/4 = 5.25
```

Strings

When assigning a String value to a variable it **must be** enclosed in quotes. Either single quotes (') or double quotes (") can be used. There is a **vital** difference between the two types of quotes.

The following code demonstrates this difference:

```
$firstName = 'Ivan';
$wishing = "Hello everyone, my first name is $firstName. <BR />";
echo $wishing;
$lastName = 'Bayross';
$greeting = ' Hello everyone, my last name is $lastName.';
echo $greeting;
```

This code produces the following output:

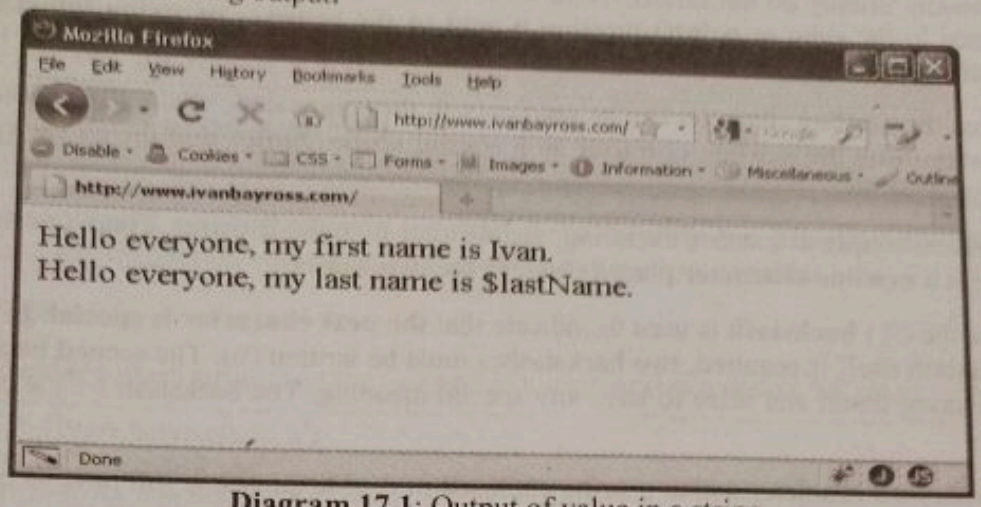


Diagram 17.1: Output of value in a string

When **double quotes** are used, PHP performs **variable expansion**, this means that PHP substitutes the value of **$firstName** wherever a reference to **$firstName** is encountered.

The result is that the string stored in the variable **$wishing** is **Hi, my first name is Ivan**.

When a value is assigned to the variable **$greeting** and **single quotes** are used then PHP does not perform any variable expansion and hence **$greeting** ends up with **Hello, my last name is $lastName**. This means that anything enclosed in **single quotes** is treated as a **string constant** and is not changed in anyway by PHP when processed.

Double quotes are also used for expanding other **special characters**, such as the **newline** character (**\n**).

The following code:

```
<?php
    echo "Ivan\nBayross";
?>
```

Will look like this when the source of the web page is displayed: (Refer to diagram 17.1.1 and diagram 17.1.2)

The following diagrams depict the new line character application at runtime along with the source code.

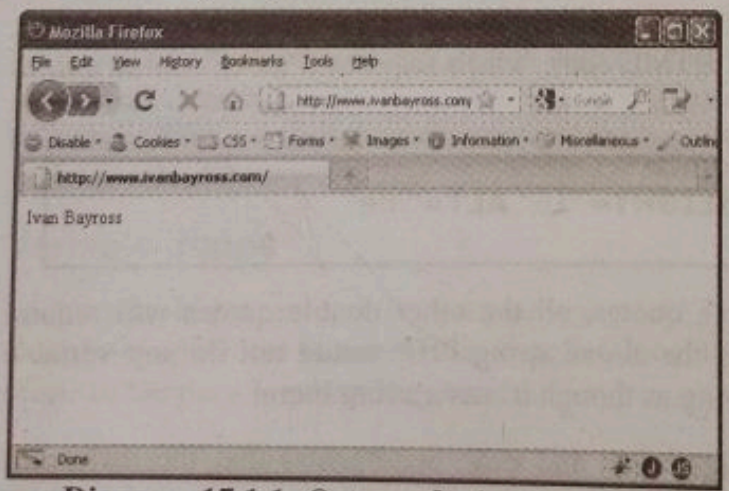


Diagram 17.1.1: Output of echo.php in Web browser

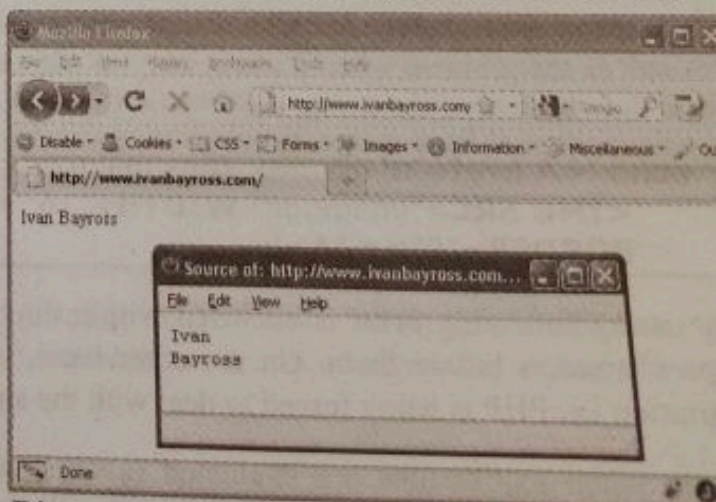


Diagram 17.1.2: Output of echo.php in web bro and in View source

However, browsers usually do not render the newline character while displaying the output. The newline character passed to the **echo** or **print()** function **is sent to the browser as string information and not formatting information**. Hence **
** tag should be used instead to create a line break in HTML.

This means that the **newline** character can be seen only in the source code of the PHP based page because the Web browser treats the newline character as a normal space. **Notice that there's no space before** or **after the \n character**.

The **echo** statement simply dispatches the string, visible after it, to the Browser. There's no trace of the PHP code, but there is a **newline character** placed between the two words.

This is because the (\ ) **backslash** is used to indicate that the **next character is special.** In PHP code spec if a single backslash itself is required, two backslashes must be written (\\). The second backslash will then be treated as a string literal and seize to have any special meaning. The backslash (\ ) is called an **escape character**.

Similarly to display a **$** character within a string enclosed in double quotes, it will have to be preceded by a backslash. This is because PHP looks for a variable, since all variables start with a **$**.

This string will give a little problem:

```
$name = "Ivan Bayross is a Professor" residing in Mumbai";
```

If tried out, an error pops up. That's because the parser expects the string to stop after the second double quote, just after the word **Professor**. The rest of the line is ignored by the PHP parser. The following string is correct:

```
$name = "Ivan Bayross is a \"Professor \" residing in Mumbai";
echo $name;
```

This code produces the following output:

```
Ivan Bayross is a "Professor" residing in Mumbai
```

The same applies if single quotes were used throughout the string.

Another way to resolve this issue is to enclose the string within **single quotes** and then use as many double quotes within the string as required.

Or

Do the inverse, enclose the string in double quotes and use as many single quotes in the string as required. This solves the problem when it's necessary to output HTML code, which requires a lot of double quotes within the structure of the string:

```
echo '<A href="mailto:ivan@ivanbayross.com">
    <IMG SRC="image.gif" WIDTH="16" HEIGHT="16" ALT="me"
    BORDER="0"></A>';
```

If the string following **echo** is enclosed within double quotes, all the other double quotes will require escape characters before them. On the other hand, in the above string PHP would not do any variable substitution i.e. PHP is being forced to deal with the string as though it was a string literal.

It's only when expressions are evaluated (as in assignments and echo statements) that the difference between single and double quotes come into the picture. After the assignment is done, no one can tell how the string was produced.

Consider the following example:

```
$fullname = 'Ivan Bayross is a ';
$profession = "Programmer";
$var1 = "$fullname $profession ";
$var2 = 'Ivan Bayross is a Programmer';
```

Variables **$var1** and **$var2** will both contain the string **Ivan Bayross is a Programmer** after evaluation, PHP will not make any distinction between the values stored in the two variables.

Joining Strings

To concatenate (add together) strings use the (.) period character (i.e. a dot **or** full stop).

```
$fullname = 'Ivan Bayross is a';
$profession = 'Programmer';
$data = $fullname.$profession;
```

Now **$data** will contain the string **Ivan Bayross is aProgrammer**. That's probably not what is wanted. It would be nice with a space between the words **a** and **Programmer**. To do this, execute the following code:

```
$data = $fullname . ' ' . $profession;
```

Notice that the **space** between the variable **$name** and the dot is ignored, it's the **string with the space (within single quotes) that's important**.

This could be solved using variable substitution. It is often easier to read, just remember to use double quotes around the string. This PHP code gives the same results as concatenating two variables using a period:

```
$data = "$fullname $profession ";
```

When using variable substitution a tiny problem will be encountered. The problem occurs with substitution of a variable in a string, which requires having text immediately after the variable. The variable name has to be **delimited** somehow.

Using curly braces is the solution as shown:

```
$noun = 'car';
echo "A $noun, two ${noun}s";
```

The above code shows playing around with English words in plural form. The first occurrence of **$noun** is substituted correctly, even when it is followed immediately by a comma because a comma cannot be a part of a name. But certainly there can be a variable called $nouns, so curly braces are used to indicate that the variable called **$noun** is being referenced and not some nonexistent variable called **$nouns**.

Variable Scope

Each variable has an area (in program memory) in which it exists, known as its scope, which means a domain in which the variable (and therefore its value) can be accessed. In other words, variables have the scope of the page in which they reside.

For example if a variable named **$myVar** is defined on a page, any code spec belonging to the rest of the page can access **$myVar**, but other pages code spec cannot.

All of this becomes interesting when user defined functions are used. Functions have their own scope, which means that variables used within a function are not available outside the run of the function. Variables defined outside a function are in turn not accessible to function code spec. For this reason, a variable inside a function can have the same name as one outside it and PHP will treat both as different variables and they can each hold different values. This is puzzling for most first time programmers.

It is technically possible for a PHP script to have several variables called **$a** in existence during program execution, however there's only one **$a** active at any specific instance in time.

The reason why the scope of a variable is so important is because it prevents crucial variables from being manipulated unexpectedly. If a variable is available in the main part of the program and accidentally the same variable is modified inside a function, the program may not run the way it is expected to. Here's how to correctly use the scoping rules in PHP which in turn controls the visibility of PHP variables.

All variables set **outside** a function or an object are considered **global**. This means, they are accessible from anywhere in the PHP script.

Consider the following example:

```
<?php
    function function()
    {
        global $myVar;
    }
    $myVar = 35;
    // Function call.
    function();
?>
```

In the above example, **$myVar** inside the function is the same as **$myVar** outside it.

This further means that the function **$myVar** already has a value of 35 and if that value changes inside the function, the external **$myVar**'s value will also change.

Example:

```
<?php
    // A global scope variable
    $MyGlobalScpVar = 35;
    // Defining a function
    function function()
    {
        echo $MyGlobalScpVar;
    }
    // Calling a function
    function();
?>
```

The above example when put into action will not produce any output because the **echo** statement refers to the local version of the **$MyGlobalScpVar** variable and it has not been assigned a value within this scope.

This behavior is very much different from **C** where global variables are automatically available to functions unless specifically overridden by a local definition. This usually causes problems when programmers inadvertently change a global variable. In PHP global variables must be declared **global inside a function** if they are going to be used in that function.

Example:

```
<?php
    $var1 = 85;
    $var2 = 50;

    function MySum()
    {
        global $var1, $var2;
        $var2 = $var1 + $var2;
    }
    MySum();
    echo "The Sum is:", $var2;
?>
```

Output: (Refer to diagram 17.2)

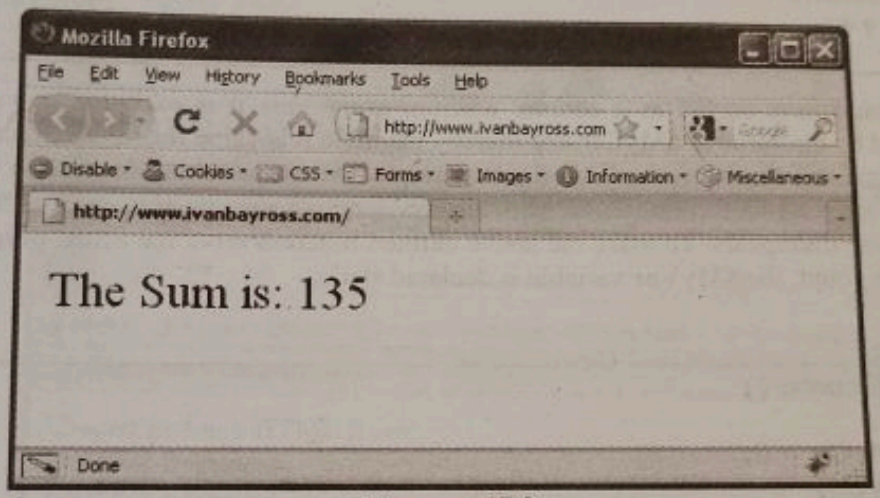


Diagram 17.2

When run the above example will display the result as **135**. This is because **$var1** and **$var2** are now declared global within the function. All references made to either of these variables will point to the global version.

NOTE

There is no limit to the number of global variables that can be manipulated by a function.

Alternatively the same can be achieved by using the special PHP defined **$GLOBALS** array.

Example:
The previous example can be rewritten as:

```
<?php
    $var1 = 85;
    $var2= 50;

    function MySum()
    {
        $GLOBALS["var2"] =
                $GLOBALS["var1"] + $GLOBALS["var2"];
    }
```

```
    MySum();
    echo "The Sum is:", $var2;
?>
```

The **$GLOBALS** array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element.

Another important feature of variable scoping is the static variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope.

Example:

```
function MyCounter()
{
    $MyCtr = 0;
    echo $MyCtr;
    $MyCtr++;
}
```

This function should have served as a counter wherein every time it is executed it will increment the counter value held by the variable **$MyCtr** and thereby maintain this value for future executions. However, currently every time this function is called, it sets the value of the variable **$MyCtr** to **0** and displays 0 (zero). The **$MyCtr++;** which increments the variable serves no purpose since as soon as the function exits, the **$MyCtr** variable disappears. To make the above defined function serve the actual purpose and not lose track of the current count, the **$MyVar** variable is declared static.

Example:

```
function MyCounter()
{
    static $MyCtr = 0;
    echo $MyCtr;
    $MyCtr++;
}
```

Now, every time the **MyCounter()** function is called it will print the value of **$MyCtr** and increment it.

Variable Variables

Variable variables allow accessing the contents of a variable without knowing its name directly. This is like indirectly referencing a variable.

Consider the following code spec:

```
<?php
    $book = PHP For Beginners;
    $name = "book";
?>
```

There are two ways to retrieve the value of **$book:**

- Use **print $book**, which is quite straightforward

Or

- Use **print $$name** (Note the use of **two dollar** signs when using the variable variables technique)

By using **$$name**, PHP will look up the contents of **$name**, convert it to a string, then look up the variable of the same name and return its value.

In the above example, **$name** contains the string **book** and so PHP will look up the variable named **$book** and output its value in this case will be **5.1**.

Example:

```
<?php
    $book = "PHP For Beginners";
    $name = "book";
    $php = "name";
    $language = "php";
    print "Chapter in 4 in 1 HTML Book<BR />";
    print $book. "<BR />";
    print $$name . "<BR />";
    print $$$php . "<BR />";
    print $$$$language . "<BR />";
?>
```

Output: (Refer to diagram 17.3)

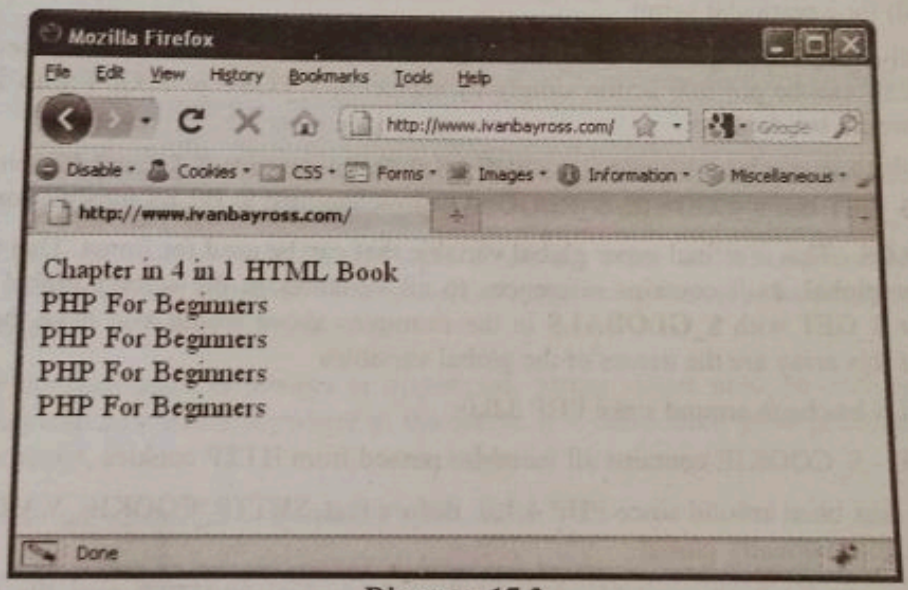


Diagram 17.3

PHP Superglobals

Superglobals are variables that are automatically available throughout all program code, in all scopes. These variables require no declaration they can simply be accessed. Super global variables provide:

- Useful information about the environment
- Allow access to HTML form variables or parameters
- Access to cookies stored on a client
- Keeping track of sessions and file uploads

They are as follows:

- **$_GET** - $_GET is used for data passed using the HTTP GET method, which includes variables passed as part of the URL

  For example: www.sharanamshah.com/index.php?book=PHP&cat=linux

  The GET method is conventionally used when the processing script has no lasting observable effect, such as changing a value in a database table. $_GET has been around since PHP 4.1.0. Before that, $HTTP_GET_VARS was used though this was not automatically global
- **$_POST** - The HTTP POST method is very similar to the $_GET. It is conventionally used when the contents of an HTML form are going to change values in a database table or make some other permanent change

  $_POST has been around since PHP 4.1.0. Before that, $HTTP_POST_VARS was used though this was not automatically global
- **$_REQUEST** - $_REQUEST holds variables provided to the script via the GET, POST and COOKIE input mechanisms. The presence and the order of variable inclusion in this array are defined according to the PHP **variables_order** configuration directive. It is preferable to use this specific super global variable, if it is not known how the variables are being passed (i.e. which method GET or POST is being used) for a particular script

  Additionally, this super global variable also contains all the information contained in $_COOKIE. $_REQUEST can be put into action simply by replacing $_POST or $_GET with $_REQUEST and the result would be identical

  $_REQUEST has also been around since PHP 4.1.0. Before version 4.3.0, in addition to the contents of $_POST, $_GET and $_COOKIE, $_REQUEST also contained $_FILES information
- **$_GLOBALS** - This is a final super global variable that can be used for forms. This would mean it is a super super global, as it contains references to all variables in the script's global scope. Replacing $_POST or $_GET with **$_GLOBALS** in the examples above would also have the identical results. The keys of this array are the names of the global variables

  $_GLOBALS has been around since PHP 3.0.0
- **$_COOKIE** - $_COOKIE contains all variables passed from HTTP cookies

  $_COOKIE has been around since PHP 4.1.0. Before that, $HTTP_COOKIE_VARS was used though this was not automatically global
- **$_FILES** - $_FILES holds variables provided to the script via HTTP post file uploads to provide feedback to the script

  $_FILES has also been around since PHP 4.1.0. Before that, $HTTP_POST_FILES was used though this was not automatically global

- **$_ENV** - $_ENV holds variables provided to the script via the environment. Analogous to the old $HTTP_ENV_VARS array, which is still available, **but deprecated**
- **$_SESSION** - $_SESSION holds variables which are currently registered to a script's session

  $_SESSION, as with most of the super global variables, has been around since PHP 4.1.0. Before that, $HTTP_SESSION_VARS was used though this was not automatically global
- **$_SERVER** - Variables set by the web server or otherwise directly related to the execution environment of the current script

  $_SERVER has also been around since PHP 4.1.0. Before that, $HTTP_SERVER_VARS was used though this was not automatically global

CONSTANTS

A constant, like variable, is a temporary placeholder in memory that holds a value. Unlike variables, the value of a constant never changes.
For example **PI** (3.14) or the value of **midnight** (12:00 am) are examples of constants applicable in a real life scenario.

When a constant is declared, the **define()** function is used and requires the name of the constant and the value that is required to be assigned to that constant. The names for constants have the same rules as variables except that they **do not** have the leading dollar sign.

Unlike variables, constants, once defined are globally accessible. Constants need not be declared again and again in each new function and PHP file.

After a constant is created, using that constant is similar to using a variable. However, unlike variables, a script cannot change the value of a constant. Attempting to do so will generate an error.

Constants make it easy to write or maintain the code. Since the value of the constant is set when it is created, the same can be modified if desired. The rest of the script using the value held by the constants will not need any modifications to reflect the new value.

By convention, the constant name is **always in uppercase**. String values must be enclosed in quotation marks. Although they can be created anywhere in the script, it is considered good practice to put them at the beginning of the script.

A constant may be a string, an integer or a floating point value.

Syntax:

```
define("<ConstantName>", <Expr> [, <CaseSensitive>])
```

Where,

- **ConstantName** is the name of the constant
- **Expr** is any valid PHP expression excluding arrays and objects
- **CaseSensitive** is a Boolean (TRUE/FALSE) and is **optional**. The default is **TRUE**

Example:

```
<?php
    define("STRCONSTANT", "This is a String Constant");
    define("INTCONSTANT", 2000);
    define("FLTCONSTANT", 3500.25);

    echo STRCONSTANT;
    echo '<BR />';
    echo INTCONSTANT;
    echo '<BR />';
    echo FLTCONSTANT;
    echo '<BR />';
?>
```

Output: (Refer to diagram 17.4)

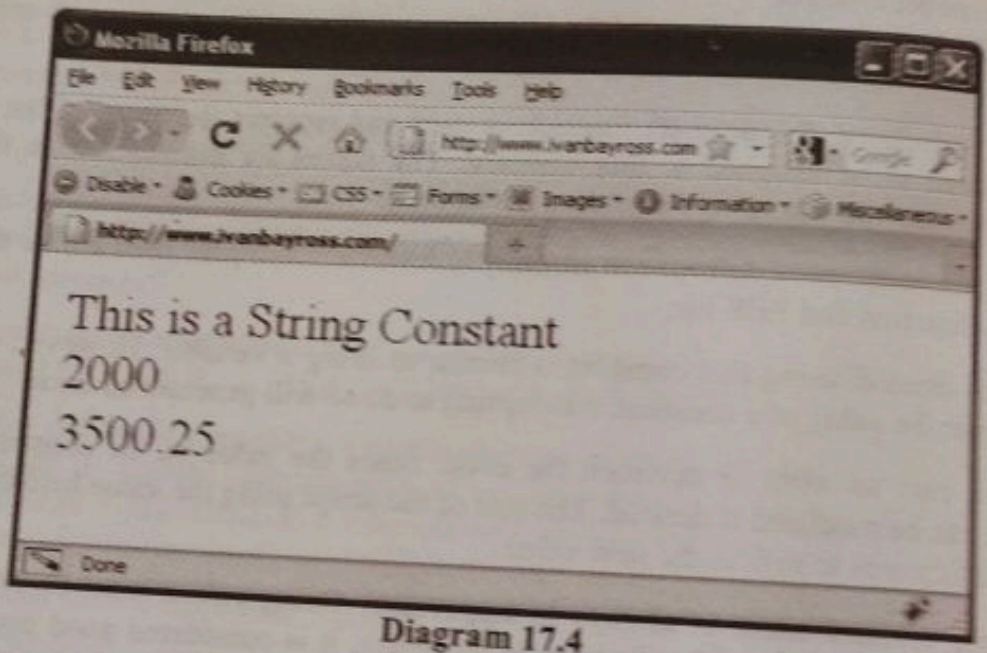


Diagram 17.4

Since constants are never preceded by a **$** (dollar sign), PHP cannot determine when a constant is to be used. Hence, it is not possible to embed a constant within a string.

Example:

```
define(AUTHORS, "Ivan Bayross");
echo "This book is authored by AUTHORS";
```

Output:

```
This book is authored by AUTHORS
```

Now separate the constant from the string with the concatenation operator.

Example:

```
define("AUTHORS", "Ivan Bayross");
echo "This book is authored by " . AUTHORS;
```

Output:

```
This book is authored by Ivan Bayross
```

PHP is bundled with some system defined constants that can be put to use in any PHP script. The most commonly used are:

__FILE__	The filename of the current script
__LINE__	The current line number
PHP_VERSION	The PHP version number
PHP_OS	The operating system running PHP
TRUE	The value TRUE (1)
FALSE	The value FALSE (0 or an empty string)
NULL	No value

HERE DOCUMENTS

HERE documents use a special form of I/O redirection to feed a command list to an interactive program or command.

They are a way of including large blocks of text (code spec) instead of using multiple statements.

The following example uses several **echo** statements.

Example:

```
<?php
    echo "Ivan Bayross has authorized the following books:" ;
    echo "<OL>";
    echo "<LI>The Chronicles Of Java On Linux";
    echo "<LI>Four in One HTML Book";
    echo "<LI>Moving From Windows To Linux";
    echo "<LI>Using L.A.M.P.P. (Linux Apache MySQL PHP PERL) On Linux";
    echo "</OL>";
?>
```

The following example using **here documents** technique clubs all the echo statements.

Example:

```
<?php
$hereMessage = <<<EOF
Ivan Bayross has authorized the following books:
<OL>
    <LI>The Chronicles Of Java On Linux
    <LI> Four in One HTML Book
    <LI> Moving From Windows To Linux
    <LI> Using L.A.M.P.P. (Linux Apache MySQL PHP PERL) On Linux
</OL>
EOF;
echo ($hereMessage);
?>
```

Output: (Refer to diagram 17.5)

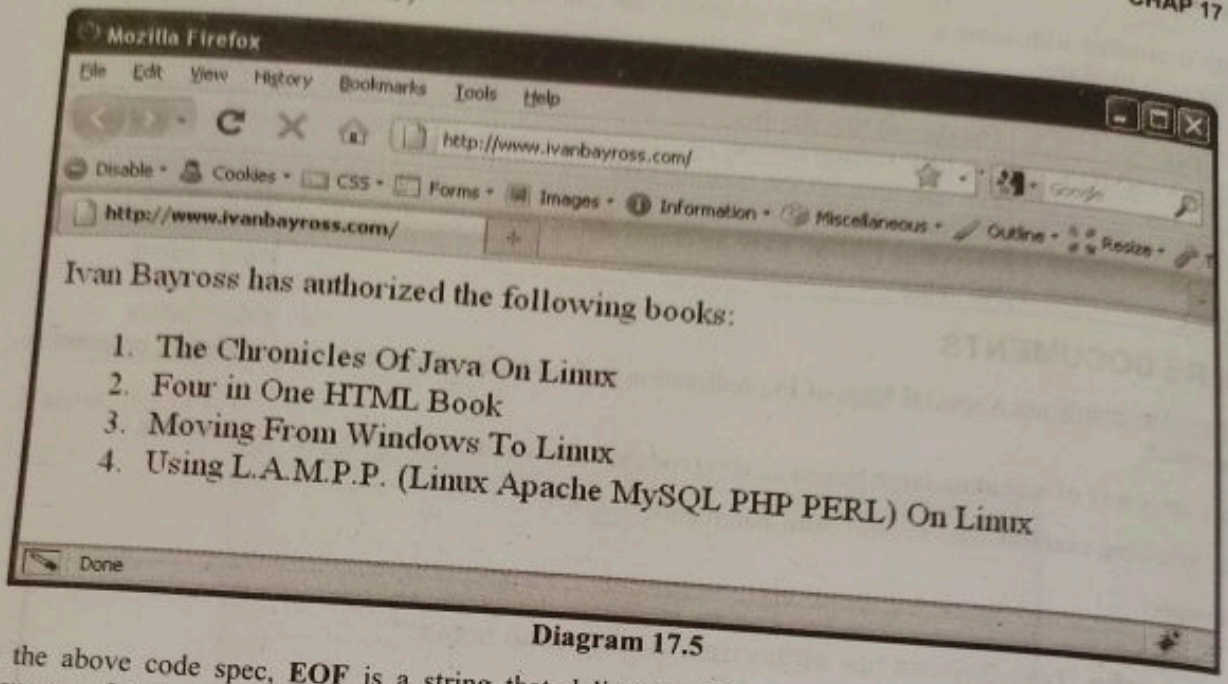


Diagram 17.5

In the above code spec, **EOF** is a string that delineates (frames) the command list. This can be any sequence of alphanumeric characters and/or underscores, although it **must not begin** with a number or an underscore. The special symbol <<< designates the limit string.

The text of the **here** document then begins on the next line. To tell PHP that it has reached the end of the here document, simply include a line beginning with the final delimiter which was declared at the beginning (in this case **EOF**).

TIP

The text within a here document is interpreted according to the same substitution rules as a double-quoted string, so variables and escape characters can also be included.

OPERATORS

An operator is any symbol used to perform an operation on a value. In PHP, operators can easily be grouped by function.

Operators are indispensable components of PHP. They are used for simple purposes such as assigning values to variables, working with strings, working with numbers, controlling program flow and so on.

PHP operators can be classified into the following areas:

- Unary operators
- Binary operators
- Ternary operator

Unary Operators

Unary Operators work on only one operand