# "Hello World" Script in PHP

To get a feel of PHP, first start with simple PHP scripts. Since "Hello, World!" is an essential example, first we will create a friendly little "Hello, World!" script.

As mentioned earlier, PHP is embedded in HTML. That means that in amongst your normal HTML (or XHTML if you're cutting-edge) you'll have PHP statements like this:

<html>

<head>

<title>Hello World</title>

<body>

<?php echo "Hello, World!";?>

</body>

</html>

It will produce the following result:

Hello, World!

If you examine the HTML output of the above example, you'll notice that the PHP code is not present in the file sent from the server to your Web browser. All of the PHP present in the Web page is processed and stripped from the page; the only thing returned to the client from the Web server is pure HTML output.

PHP code must be included inside one of the three special markup tags ate are recognized by the PHP Parser.

<?php PHP code goes here ?>

<? PHP code goes here ?>

<script language="php"> PHP code goes here </script>

Most common tag is the <?php...?> and we will also use the same tag in our tutorial.

From the next chapter, we will start with PHP Environment Setup on your machine and then we will dig out almost all concepts related to PHP to make you comfortable with the PHP language.

# PHP ─ Syntax Overview

## Commenting PHP Code

A *comment* is the portion of a program that exists only for the human reader and stripped out before displaying the programs result. There are two commenting formats in PHP:

**Single-line comments:** They are generally used for short explanations or notes relevant to the local code. Here are the examples of single line comments.

<?php

# This is a comment, and

# This is the second line of the comment

// This is a comment too. Each style comments only print "An example with single line comments";

?>

**Multi-lines printing:** Here are the examples to print multiple lines in a single print statement:

<?php

# First Example

print <<<END

This uses the "here document" syntax to output

multiple lines with $variable interpolation. Note

that the here document terminator must appear on a

line with just a semicolon no extra whitespace!

END;

# Second Example

print "This spans multiple lines. The newlines will be

output as well";

?>

**Multi-lines comments:** They are generally used to provide pseudocode algorithms and more detailed explanations when necessary. The multiline style of commenting is the same as in C. Here is the example of multi lines comments.

<?php

/\* This is a comment with multiline

Author : Joko

Purpose: Multiline Comments Demo

Subject: PHP

\*/

print "An example with multi line comments";

?>

## PHP is whitespace insensitive

Whitespace is the stuff you type that is typically invisible on the screen, including spaces, tabs, and carriage returns (end-of-line characters).

PHP whitespace insensitive means that it almost never matters how many whitespace characters you have in a row.one whitespace character is the same as many such characters.

For example, each of the following PHP statements that assigns the sum of 2 + 2 to the variable $four is equivalent:

<?php

$four = 2 + 2; // single spaces

$four <tab>=<tab2<tab>+<tab>2 ; // spaces and tabs

$four = 2+

2; // multiple lines

?>

## PHP is case sensitive

Yeah it is true that PHP is a case sensitive language. Try out the following example:

<html>

<body>

<?php

$capital = 67;

print("Variable capital is $capital<br>");

print("Variable CaPiTaL is $CaPiTaL<br>");

?>

</body>

</html>

This will produce the following result:

Variable capital is 67

Variable CaPiTaL is

## Statements are expressions terminated by semicolons

A *statement* in PHP is any expression that is followed by a semicolon (;). Any sequence of valid PHP statements that is enclosed by the PHP tags is a valid PHP program. Here is a typical statement in PHP, which in this case assigns a string of characters to a variable called $greeting:

<?

$greeting = "Welcome to PHP!";

?>

## Expressions are combinations of tokens

The smallest building blocks of PHP are the indivisible tokens, such as numbers (3.14159), strings (.two.), variables ($two), constants (TRUE), and the special words that make up the syntax of PHP itself like if, else, while, for and so forth

## Braces make blocks

Although statements cannot be combined like expressions, you can always put a sequence of statements anywhere a statement can go by enclosing them in a set of curly braces.

Here both statements are equivalent:

if (3 == 2 + 1)

print("Good - I haven't totally lost my mind.<br>");

if (3 == 2 + 1)

{

print("Good - I haven't totally");

print("lost my mind.<br>");

}

## Running PHP Script from Command Prompt

Yes, you can run your PHP script on your command prompt. Assuming you have the following content in test.php file:

<?php

echo "Hello PHP!!!!!";

?>

Now run this script as command prompt as follows:

$ php test.php

It will produce the following result

Hello PHP!!!!!

# PHP ─ Variable Types

The main way to store information in the middle of a PHP program is by using a variable. Here are the most important things to know about variables in PHP.

* + All variables in PHP are denoted with a leading dollar sign ($).
  + The value of a variable is the value of its most recent assignment.
  + Variables are assigned with the = operator, with the variable on the left-hand side and the expression to be evaluated on the right.
  + Variables can, but do not need, to be declared before assignment.
  + Variables in PHP do not have intrinsic types - a variable does not know in advance whether it will be used to store a number or a string of characters.
  + Variables used before they are assigned have default values.
  + PHP does a good job of automatically converting types from one to another when necessary.
  + PHP variables are Perl-like.

PHP has a total of eight data types which we use to construct our variables:

* + **Integers:** are whole numbers, without a decimal point, like 4195.
  + **Doubles:** are floating-point numbers, like 3.14159 or 49.1.
  + **Booleans:** have only two possible values either true or false.
  + **NULL:** is a special type that only has one value: NULL.
  + **Strings:** are sequences of characters, like 'PHP supports string operations.'
  + **Arrays:** are named and indexed collections of other values.
  + **Objects:** are instances of programmer-defined classes, which can package up both other kinds of values and functions that are specific to the class.
  + **Resources:** are special variables that hold references to resources external to PHP (such as database connections).

The first five are *simple types*, and the next two (arrays and objects) are compound - the compound types can package up other arbitrary values of arbitrary type, whereas the simple types cannot.

We will explain only simile data type in this chapters. Array and Objects will be explained separately.

## Integers

They are whole numbers, without a decimal point, like 4195. They are the simplest type they correspond to simple whole numbers, both positive and negative. Integers can be assigned to variables, or they can be used in expressions, like so:

$int\_var = 12345;

$another\_int = -12345 + 12345;

Integer can be in decimal (base 10), octal (base 8), and hexadecimal (base 16) format. Decimal format is the default, octal integers are specified with a leading 0, and hexadecimals have a leading 0x.

For most common platforms, the largest integer is (2\*\*31 . 1) (or 2,147,483,647), and the smallest (most negative) integer is . (2\*\*31 . 1) (or .2,147,483,647).

## Doubles

They like 3.14159 or 49.1. By default, doubles print with the minimum number of decimal places needed. For example, the code:

<?php

$many = 2.2888800;

$many\_2 = 2.2111200;

$few = $many + $many\_2;

print(.$many + $many\_2 = $few<br>.);

?>

It produces the following browser output:

2.28888 + 2.21112 = 4.5

## Boolean

They have only two possible values either true or false. PHP provides a couple of constants especially for use as Booleans: TRUE and FALSE, which can be used like so:

<?php

if (TRUE)

print("This will always print<br>"); else

print("This will never print<br>");

?>

**Interpreting other types as Booleans**

Here are the rules for determine the "truth" of any value not already of the Boolean type:

* + If the value is a number, it is false if exactly equal to zero and true otherwise.

If the value is a string, it is false if the string is empty (has zero characters) or is the string "0", and is true otherwise.

* + Values of type NULL are always false.
  + If the value is an array, it is false if it contains no other values, and it is true otherwise. For an object, containing a value means having a member variable that has been assigned a value.
  + Valid resources are true (although some functions that return resources when they are successful will return FALSE when unsuccessful).
  + Don't use double as Booleans.

Each of the following variables has the truth value embedded in its name when it is used in a Boolean context.

<?php

$true\_num = 3 + 0.14159;

$true\_str = "Tried and true"

$true\_array[49] = "An array element";

$false\_array = array();

$false\_null = NULL;

$false\_num = 999 - 999;

$false\_str = "";

?>

## NULL

NULL is a special type that only has one value: NULL. To give a variable the NULL value, simply assign it like this:

$my\_var = NULL;

The special constant NULL is capitalized by convention, but actually it is case insensitive; you could just as well have typed:

$my\_var = null;

A variable that has been assigned NULL has the following properties:

* + It evaluates to FALSE in a Boolean context.
  + It returns FALSE when tested with IsSet() function.

## **Strings**

They are sequences of characters, like "PHP supports string operations". Following are valid examples of string:

$string\_1 = "This is a string in double quotes";

$string\_2 = "This is a somewhat longer, singly quoted string";

$string\_39 = "This string has thirty-nine characters";

$string\_0 = ""; // a string with zero characters

Singly quoted strings are treated almost literally, whereas doubly quoted strings replace variables with their values as well as specially interpreting certain character sequences.

<?php

$variable = "name";

$literally = 'My $variable will not print!\\n';

print($literally);

$literally = "My $variable will print!\\n";

print($literally);

?>

This will produce the following result:

My $variable will not print!\n My name will print

There are no artificial limits on string length - within the bounds of available memory, you ought to be able to make arbitrarily long strings.

Strings that are delimited by double quotes (as in "this") are preprocessed in both the following two ways by PHP:

* + Certain character sequences beginning with backslash (\) are replaced with special characters
  + Variable names (starting with $) are replaced with string representations of their values.

The escape-sequence replacements are:

* + \n is replaced by the newline character
  + \r is replaced by the carriage-return character
  + \t is replaced by the tab character
  + \$ is replaced by the dollar sign itself ($)
  + \" is replaced by a single double-quote (")
  + \\ is replaced by a single backslash (\)

## Here Document

You can assign multiple lines to a single string variable using **here document**:

<?php

$channel =<<<\_XML\_

<channel>

<title>What's For Dinner<title>

<link>http://menu.example.com/<link>

<description>Choose what to eat tonight.</description>

</channel>

\_XML\_;

echo <<<END

This uses the "here document" syntax to output

multiple lines with variable interpolation. Note

that the here document terminator must appear on a

line with just a semicolon. no extra whitespace!

<br />

END;

print $channel;

?>

This will produce the following result:

This uses the "here document" syntax to output multiple lines with variable interpolation. Note that the here document terminator must appear on a line with just a semicolon. no extra whitespace!

<channel>

<title>What's For Dinner<title>

[<link>ht](http://menu.example.com/)tp://[menu.example.com/](http://menu.example.com/)<link>

<description>Choose what to eat tonight.</description>

## Variable Naming

Rules for naming a variable is:

* + Variable names must begin with a letter or underscore character.
  + A variable name can consist of numbers, letters, underscores but you cannot use characters like + , - , % , ( , ) . & , etc

There is no size limit for variables.

# PHP – Variables

Scope can be defined as the range of availability a variable has to the program in which it is declared. PHP variables can be one of four scope types:

* + Local variables
  + Function parameters
  + Global variables
  + Static variables

## PHP Local Variables

A variable declared in a function is considered local; that is, it can be referenced solely in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function:

<?php

$x = 4;

*function* assignx () {

$x = 0;

print "\$x inside function is $x. ";

}

assignx();

print "\$x outside of function is $x. ";

?>

This will produce the following result.

$x inside function is 0.

$x outside of function is 4.

## PHP Function Parameters

PHP Functions are covered in detail in PHP Function Chapter. In short, a function is a small unit of program which can take some input in the form of parameters and does some processing and may return a value.

Function parameters are declared after the function name and inside parentheses. They are declared much like a typical variable would be:

<?php

// multiply a value by 10 and return it to the caller function

multiply ($value) {

$value = $value \* 10; return $value;

}

$retval = multiply (10);

Print "Return value is $retval\n";

?>

This will produce the following result.

Return value is 100

## PHP Global Variables

In contrast to local variables, a global variable can be accessed in any part of the program. However, in order to be modified, a global variable must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword **GLOBAL** in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example:

<?php

$somevar = 15;

*function* addit() {

GLOBAL $somevar;

$somevar++;

print "Somevar is $somevar";

}

addit();

?>

This will produce the following result.

Somevar is 16

## PHP Static Variables

The final type of variable scoping that I discuss is known as static. In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable will not lose its value when the function exits and will still hold that value should the function be called again.

You can declare a variable to be static simply by placing the keyword STATIC in front of the variable name.

<?php

*function* keep\_track() { STATIC $count = 0;

$count++; print $count;

print "

";

}

keep\_track();

keep\_track();

keep\_track();

?>

This will produce the following result.

1

2

3

# PHP ─ Constants

A constant is a name or an identifier for a simple value. A constant value cannot change during the execution of the script. By default, a constant is case-sensitive. By convention, constant identifiers are always uppercase. A constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. If you have defined a constant, it can never be changed or undefined.

To define a constant you have to use define() function and to retrieve the value of a constant, you have to simply specifying its name. Unlike with variables, you do not need to have a constant with a $. You can also use the function constant() to read a constant's value if you wish to obtain the constant's name dynamically.

**constant() function**

As indicated by the name, this function will return the value of the constant.

This is useful when you want to retrieve value of a constant, but you do not know its name, i.e., it is stored in a variable or returned by a function.

**constant() example**

<?php

define("MINSIZE", 50);

echo MINSIZE;

echo constant("MINSIZE"); // same thing as the previous line

?>

Only scalar data (boolean, integer, float and string) can be contained in constants.

**Differences between constants and variables are**

* + There is no need to write a dollar sign ($) before a constant, where as in Variable one has to write a dollar sign.
  + Constants cannot be defined by simple assignment, they may only be defined using the define() function.
  + Constants may be defined and accessed anywhere without regard to variable scoping rules.
  + Once the Constants have been set, may not be redefined or undefined.

**Valid and invalid constant names**

<?php

// Valid constant names

define("ONE", "first thing"); define("TWO2",  "second thing"); define("THREE\_3", "third thing")

// Invalid constant names

define("2TWO",  "second thing"); define(" THREE ", "third value");

?>

## PHP Magic constants

PHP provides a large number of predefined constants to any script which it runs.

There are five magical constants that change depending on where they are used. For example, the value of \_\_LINE depends on the line that it's used on in your script. These special constants are case-insensitive and are as follows:

The following table lists a few "magical" PHP constants along with their description:

|  |  |
| --- | --- |
| **Name** | **Description** |
| LINE | The current line number of the file. |
| FILE | The full path and filename of the file. If used inside an include, the name of the included file is returned. Since PHP 4.0.2, FILE always contains an absolute path whereas in older versions it contained relative path under some circumstances. |
| FUNCTION | The function name. (Added in PHP 4.3.0) As of PHP 5 this constant returns the function name as it was declared (case-sensitive). In PHP 4 its value is always lowercased. |
| CLASS | The class name. (Added in PHP 4.3.0) As of PHP 5 this constant returns the class name as it was declared (case-sensitive). In PHP 4 its value is always lowercased. |
| METHOD | The class method name. (Added in PHP 5.0.0) The method name is returned as it was declared (case-sensitive). |

# PHP ─ Operator Types

**What is Operator?** Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. PHP language supports following type of operators.

* + Arithmetic Operators
  + Comparison Operators
  + Logical (or Relational) Operators
  + Assignment Operators
  + Conditional (or ternary) Operators Let’s have a look on all operators one by one.

## Arithmetic Operators

The following arithmetic operators are supported by PHP language: Assume variable A holds 10 and variable B holds 20 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiply both operands | A \* B will give 200 |
| / | Divide the numerator by denominator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | Increment operator, increases integer value by one | A++ will give 11 |
| -- | Decrement operator, decreases integer value by one | A-- will give 9 |

**Example**

Try the following example to understand all the arithmetic operators. Copy and paste following PHP program in test.php file and keep it in your PHP Server's document root and browse it using any browser.

<html>

<head><title>Arithmetical Operators</title><head>

<body>

<?php

$a = 42;

$b = 20;

$c = $a + $b;

echo "Addition Operation Result: $c <br/>";

$c = $a - $b;

echo "Subtraction Operation Result: $c <br/>";

$c = $a \* $b;

echo "Multiplication Operation Result: $c <br/>";

$c = $a / $b;

echo "Division Operation Result: $c <br/>";

$c = $a % $b;

echo "Modulus Operation Result: $c <br/>";

$c = $a++;

echo "Increment Operation Result: $c <br/>";

$c = $a--;

echo "Decrement Operation Result: $c <br/>";

?>

</body>

</html>

This will produce the following result:

Addition Operation Result: 62 Subtraction Operation Result: 22 Multiplication Operation Result: 840 Division Operation Result: 2.1 Modulus Operation Result: 2 Increment Operation Result: 42 Decrement Operation Result: 43

## Comparison Operators

There are following comparison operators supported by PHP language. Assume variable A holds 10 and variable B holds 20 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes, then condition becomes true. | (A == B) is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal, then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes, then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes, then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes, then condition becomes true. | (A <= B) is true. |

**Example**

Try the following example to understand all the comparison operators. Copy and paste the following PHP program in test.php file and keep it in your PHP Server's document root and browse it using any browser.

<html>

<head><title>Comparison Operators</title><head>

<body>

<?php

$a = 42;

$b = 20;

if( $a == $b ){

echo "TEST1 : a is equal to b<br/>";

} else {

echo "TEST1 : a is not equal to b<br/>";

}

if( $a > $b ){

echo "TEST2 : a is greater than b<br/>";

} else {

echo "TEST2 : a is not greater than b<br/>";

}

if( $a < $b ){

echo "TEST3 : a is less than  b<br/>";

} else {

echo "TEST3 : a is not less than b<br/>";

}

if( $a != $b ){

echo "TEST4 : a is not equal to b<br/>";

} else {

echo "TEST4 : a is equal to b<br/>";

}

if( $a >= $b ){

echo "TEST5 : a is either greater than or equal to b<br/>";

} else {

echo "TEST5 : a is neither greater than nor equal to b<br/>";

}

if( $a <= $b ){

echo "TEST6 : a is either less than or equal to b<br/>";

}else {

echo "TEST6 : a is neither less than nor equal to b<br/>";

}

?>

</body>

</html>

This will produce the following result:

TEST1 : a is not equal to b

TEST2 : a is greater than b

TEST3 : a is not less than b

TEST4 : a is not equal to b

TEST5 : a is either greater than or equal to b

TEST6 : a is neither less than nor equal to b

## Logical Operators

The following logical operators are supported by PHP language. Assume variable A holds 10 and variable B holds 20 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Called Logical AND operator. If both the operands are true, then condition becomes true. | (A and B) is true. |
| or | Called Logical OR Operator. If any of the two operands are non-zero, then condition becomes true. | (A or B) is true. |
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is true. |
| || | Called Logical OR Operator. If any of the two operands are non-zero, then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | !(A && B) is false. |

**Example**

Try the following example to understand all the logical operators. Copy and paste the following PHP program in test.php file and keep it in your PHP Server's document root and browse it using any browser.

<html>

<head><title>Logical Operators</title><head>

<body>

<?php

$a = 42;

$b = 0;

if( $a && $b ){

echo "TEST1 : Both a and b are true<br/>";

}else{

echo "TEST1 : Either a or b is false<br/>";

}

if( $a and $b ){

echo "TEST2 : Both a and b are true<br/>";

}else{

echo "TEST2 : Either a or b is false<br/>";

}

if( $a || $b ){

echo "TEST3 : Either a or b is true<br/>";

}else{

echo "TEST3 : Both a and b are false<br/>";

}

if( $a or $b ){

echo "TEST4 : Either a or b is true<br/>";

}else{

echo "TEST4 : Both a and b are false<br/>";

}

if( $a or $b ){

echo "TEST4 : Either a or b is true<br/>";

}else{

echo "TEST4 : Both a and b are false<br/>";

}

$a = 10;

$b = 20;

if( $a ){

echo "TEST5 : a is true <br/>";

}else{

echo "TEST5 : a is false<br/>";

}

if( $b ){

echo "TEST6 : b is true <br/>";

}else{

echo "TEST6 : b is false<br/>";

}

if( !$a ){

echo "TEST7 : a is true <br/>";

}else{

echo "TEST7 : a is false<br/>";

}

if( !$b ){

echo "TEST8 : b is true <br/>";

}else{

echo "TEST8 : b is false<br/>";

}

?>

</body>

</html>

This will produce the following result:

TEST1 : Either a or b is false TEST2 : Either a or b is false TEST3 : Either a or b is true TEST4 : Either a or b is true TEST5 : a is true

TEST6 : b is true TEST7 : a is false TEST8 : b is false

## Assignment Operators

PHP supports the following assignment operators:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign the value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C  = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C  = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |

**Example**

Try the following example to understand all the assignment operators. Copy and paste the following PHP program in test.php file and keep it in your PHP Server's document root and browse it using any browser.

<html>

<head><title>Assignment Operators</title><head>

<body>

<?php

$a = 42;

$b = 20;

$c = $a + $b; /\* Assignment operator \*/ echo "Addition Operation Result: $c <br/>";

$c += $a; /\* c value was 42 + 20 = 62 \*/

echo "Add AND Assignment Operation Result: $c <br/>";

$c -= $a; /\* c value was 42 + 20 + 42 = 104 \*/

echo "Subtract AND Assignment Operation Result: $c <br/>";

$c \*= $a; /\* c value was 104 - 42 = 62 \*/

echo "Multiply AND Assignment Operation Result: $c <br/>";

$c /= $a; /\* c value was 62 \* 42 = 2604 \*/

echo "Division AND Assignment Operation Result: $c <br/>";

$c %= $a; /\* c value was 2604/42 = 62\*/

echo "Modulus AND Assignment Operation Result: $c <br/>";

?>

</body>

</html>

This will produce the following result:

Addition Operation Result: 62

Add AND Assignment Operation Result: 104 Subtract AND Assignment Operation Result: 62 Multiply AND Assignment Operation Result: 2604 Division AND Assignment Operation Result: 62 Modulus AND Assignment Operation Result: 20

## Conditional Operator

There is one more operator called the conditional operator. It first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ? : | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |

Try the following example to understand the conditional operator. Copy and paste the following PHP program in test.php file and keep it in your PHP Server's document root and browse it using any browser.

<html>

<head><title>Arithmetical Operators</title><head>

<body>

<?php

$a = 10;

$b = 20;

/\* If condition is true then assign a to result otherwise b \*/

$result = ($a > $b ) ? $a : $b;

echo "TEST1 : Value of result is $result<br/>";

/\* If condition is true then assign a to result otherwise b \*/

$result = ($a < $b ) ? $a : $b;

echo "TEST2 : Value of result is $result<br/>";

?>

</body>

</html>

This will produce the following result:

TEST1 : Value of result is 20 TEST2 : Value of result is 10

**Operators Categories**

All the operators we have discussed above can be categorized into the following categories:

* Unary prefix operators, which precede a single operand.
* Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
* The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
* Assignment operators, which assign a value to a variable.

## Precedence of PHP Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator: For example, x = 7 + 3 \* 2; Here x is assigned 13, not 20 because operator \* has higher precedence than + so it first gets multiplied with 3\*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Unary | ! ++ -- | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= | Right to left |

# PHP ─ Decision Making

The if, elseif ...else and switch statements are used to take decision based on the different condition.

You can use conditional statements in your code to make your decisions. PHP supports the following three decision making statements:

* **if...else statement** - use this statement if you want to execute a set of code when a condition is true and another if the condition is not true
* **elseif statement** - is used with the if...else statement to execute a set of code if

**one** of several condition is true

* **switch statement** - is used if you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.

## The If. .Else Statement

If you want to execute some code if a condition is true and another code if a condition is false, use the if....else statement.

*Syntax*

<?php

if (condition)

code to be executed if condition is true;

else

code to be executed if condition is false;

?>

**Example**

The following example will output "Have a nice weekend!" if the current day is Friday, otherwise it will output

<html>

<body>

<?php

$d=date("D");

if ($d=="Fri")

echo "Have a nice weekend!";

else

echo "Have a nice day!";

?>

</body>

</html>

It will produce the following result:

Have a nice weekend!

</html>

## The ElseIf Statement

If you want to execute some code if one of the several conditions is true, then use the elseif statement.

**Syntax**

<?php

if (condition)

code to be executed if condition is true;

elseif (condition)

code to be executed if condition is true;

else

code to be executed if condition is false;

?>

**Example**

The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday. Otherwise it will output "Have a nice day!":

<html>

<body>

<?php

$d=date("D");

if ($d=="Fri")

echo "Have a nice weekend!";

elseif ($d=="Sun")

echo "Have a nice Sunday!";

else

echo "Have a nice day!";

?>

</body>

</html>

It will produce the following result:

Have a nice weekend!

</html>

## The Switch Statement

If you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.

**Syntax**

<?php

switch (expression)

{

case label1:

code to be executed if expression = label1;

break;

case label2:

code to be executed if expression = label2;

break;

default:

code to be executed if expression is different from both label1 and label2;

}

?>

**Example**

The *switch* statement works in an unusual way. First it evaluates the given expression, then seeks a label to match the resulting value. If a matching value is found, then the code associated with the matching label will be executed. If none of the labels match, then the statement will execute any specified default code.

<html>

<body>

<?php

$d=date("D");

switch ($d)

{

case "Mon":

echo "Today is Monday";

break;

case "Tue":

echo "Today is Tuesday";

break;

case "Wed":

echo "Today is Wednesday";

break;

case "Thu":

echo "Today is Thursday";

break;

case "Fri":

echo "Today is Friday";

break;

case "Sat":

echo "Today is Saturday";

break;

case "Sun":

echo "Today is Sunday";

break;

default:

echo "Wonder which day is this ?";

}

?>

</body>

</html>

It will produce the following result:

Today is Friday

</html>

# PHP ─ Loop Types

Loops in PHP are used to execute the same block of code a specified number of times. PHP supports following four loop types.

* + **for -** loops through a block of code a specified number of times.
  + **while -** loops through a block of code if and as long as a specified condition is true.
  + **do...while -** loops through a block of code once, and then repeats the loop as long as a special condition is true.
  + **foreach -** loops through a block of code for each element in an array.

We will discuss about **continue** and **break** keywords used to control the loops execution.

## The for loop statement

The for statement is used when you know how many times you want to execute a statement or a block of statements.

**Syntax**

<?php

for (initialization; condition; increment)

{

code to be executed;

}

?>

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it $i.

**Example**

The following example makes five iterations and changes the assigned value of two variables on each pass of the loop:

<html>

<body>

<?php

$a = 0;

$b = 0;

for($i=0; $i<5; $i++ )

{

$a += 10;

$b += 5;

}

echo ("At the end of the loop a=$a and b=$b" );

?>

</body>

</html>

This will produce the following result:

At the end of the loop a=50 and b=25

## The while loop statement

The while statement will execute a block of code if and as long as a test expression is true. If the test expression is true, then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.

**Syntax**

<?php

while (condition) {

code to be executed;

}

?>

**Example**

This example decrements a variable value on each iteration of the loop and the counter increments until it reaches 10 when the evaluation becomes false and the loop ends.

<?php

$i = 0;

$num = 50;

while( $i < 10)

{

$num--;

$i++;

}

echo ("Loop stopped at i = $i and num = $num");

?>

This will produce the following result:

Loop stopped at i = 10 and num = 40

## The do. .while loop statement

The do...while statement will execute a block of code at least once - it will then repeat the loop as long as a condition is true.

**Syntax**

<?php

do

{

code to be executed;

} while (condition);

?>

**Example**

The following example will increment the value of i at least once, and it will continue incrementing the variable **i** as long as it has a value of less than 10:

<html>

<body>

<?php

$i = 0;

$num = 0;

do

{

$i++;

} while( $i < 10 );

echo ("Loop stopped at i = $i" );

?>

</body>

</html>

This will produce the following result:

Loop stopped at i = 10

## The foreach loop statement

The foreach statement is used to loop through arrays. For each pass the value of the current array element is assigned to $value and the array pointer is moved by one and in the next pass next element will be processed.

**Syntax**

<?php

foreach (*array* as value)

{

code to be executed;

}

?>

**Example**

Try out the following example to list out the values of an array.

<html>

<body>

<?php

$array = array( 1, 2, 3, 4, 5);

foreach( $array as $value )

{

echo "Value is $value <br />";

}

?>

</body>

</html>

This will produce the following result:

Value is 1

Value is 2

Value is 3

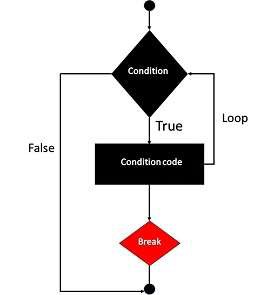
Value is 4

Value is 5

## The break statement

The PHP **break** keyword is used to terminate the execution of a loop prematurely.

The **break** statement is situated inside the statement block. If gives you full control and whenever you want to exit from the loop you can come out. After coming out of a loop immediate statement to the loop will be executed.



**Example**

In the following example, the condition test becomes true when the counter value reaches 3 and loop terminates.

<?php

$i = 0;

while( $i < 10)

{

$i++;

if( $i == 3 )

break;

}

echo ("Loop stopped at i = $i" );

?>

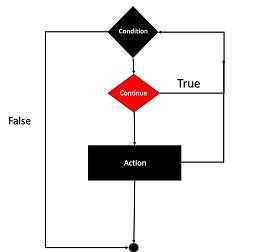
This will produce the following result:

Loop stopped at i = 3

## The continue statement

The PHP **continue** keyword is used to halt the current iteration of a loop but it does not terminate the loop.

Just like the **break** statement the **continue** statement is situated inside the statement block containing the code that the loop executes, preceded by a conditional test. For the pass encountering **continue** statement, rest of the loop code is skipped and next pass starts.



**Example**

In the following example, the loop prints the value of array, but when the condition becomes true, it just skips the code and next value is printed.

<?php

$array = array( 1, 2, 3, 4, 5);

foreach( $array as $value )

{

if( $value == 3 )

continue;

echo "Value is $value \n";

}

?>

This will produce the following result:

Value is 1

Value is 2

Value is 4

Value is 5

# PHP ─ Arrays

An array is a data structure that stores one or more similar type of values in a single value. For example, if you want to store 100 numbers, then instead of defining 100 variables, it is easy to define an array of 100 lengths.

There are three different kind of arrays and each array value is accessed using an ID c which is called array index.

* + **Numeric array** - An array with a numeric index. Values are stored and accessed in linear fashion
  + **Associative array** - An array with strings as index. This stores element values in association with key values rather than in a strict linear index order.
  + **Multidimensional array** - An array containing one or more arrays and values are accessed using multiple indices

**NOTE:** Built-in array functions is given in function reference [PHP Array Functions](http://www.tutorialspoint.com/php/php_array_functions.htm)

## Numeric Array

These arrays can store numbers, strings and any object but their index will be represented by numbers. By default, the array index starts from zero.

**Example**

The following example demonstrates how to create and access numeric arrays.

Here we have used **array()** function to create array. This function is explained in function reference.

<html>

<body>

<?php

/\* First method to create array. \*/

$numbers = array( 1, 2, 3, 4, 5);

foreach( $numbers as $value )

{

echo "Value is $value <br />";

}

/\* Second method to create array. \*/

$numbers[0] = "one";

$numbers[1] = "two";

$numbers[2] = "three";

$numbers[3] = "four";

$numbers[4] = "five";

foreach( $numbers as $value )

{

echo "Value is $value <br />";

}

?>

</body>

</html>

This will produce the following result:

Value is 1

Value is 2

Value is 3

Value is 4

Value is 5 Value is one Value is two Value is three Value is four Value is five

## Associative Arrays

The associative arrays are very similar to numeric arrays in term of functionality but they are different in terms of their index. Associative array will have their index as string so that you can establish a strong association between key and values.

To store the salaries of employees in an array, a numerically indexed array would not be the best choice. Instead, we could use the employee’s names as the keys in our associative array, and the value would be their respective salary.

**NOTE:** Don't keep associative array inside double quote while printing, otherwise it would not return any value.

**Example**

<html>

<body>

<?php

/\* First method to associate create array. \*/

$salaries = array(

"mohammad" => 2000,

"qadir" => 1000,

"zara" => 500

);

echo "Salary of mohammad is ". $salaries['mohammad'] . "<br />";

echo "Salary of qadir is ". $salaries['qadir']. "<br />";

echo "Salary of zara is ".  $salaries['zara']. "<br />";

/\* Second method to create array. \*/

$salaries['mohammad'] = "high";

$salaries['qadir'] = "medium";

$salaries['zara'] = "low";

echo "Salary of mohammad is ". $salaries['mohammad'] . "<br />"; echo "Salary of qadir is ".  $salaries['qadir']. "<br />";

echo "Salary of zara is ".  $salaries['zara']. "<br />";

?>

</body>

</html>

This will produce the following result:

Salary of mohammad is 2000 Salary of qadir is 1000 Salary of zara is 500 Salary of mohammad is high Salary of qadir is medium Salary of zara is low

## Multidimensional Arrays

A multi-dimensional array each element in the main array can also be an array. And each element in the sub-array can be an array, and so on. Values in the multi-dimensional array are accessed using multiple index.

*Example*

In this example, we create a two dimensional array to store marks of three students in three subjects:

This example is an associative array; you can create numeric array in the same fashion.

<html>

<body>

<?php

$marks = array(

"mohammad" => *array* ("physics" => 35, "maths" => 30, "chemistry" => 39),

"qadir" => *array* ("physics" => 30, "maths" => 32, "chemistry" => 29),

"zara" => *array* ( "physics" => 31, "maths" => 22, "chemistry" => 39 )

);

/\* Accessing multi-dimensional array values \*/

echo "Marks for mohammad in physics : " ;

echo $marks['mohammad']['physics'] . "<br />";

echo "Marks for qadir in maths : ";

echo $marks['qadir']['maths'] . "<br />";

echo "Marks for zara in chemistry : " ;

echo $marks['zara']['chemistry'] . "<br />";

?>

</body>

</html>

This will produce the following result:

Marks for mohammad in physics : 35

Marks for qadir in maths : 32

Marks for zara in chemistry : 39

chemistry : 39

# PHP ─ Functions

PHP functions are similar to other programming languages. A function is a piece of code which takes one more input in the form of parameter and does some processing and returns a value.

You already have seen many functions like **fopen()** and **fread()** etc. They are built-in functions but PHP gives you option to create your own functions as well.

There are two parts which should be clear to you:

* Creating a PHP Function
* Calling a PHP Function

In fact, you hardly need to create your own PHP function because there are already more than 1000 of built-in library functions created for different area and you just need to call them according to your requirement.

Please refer to PHP Function Reference for a complete set of useful functions.

## Creating PHP Function

It is very easy to create your own PHP function. Suppose you want to create a PHP function which will simply write a simple message on your browser when you will call it.

The following example creates a function called writeMessage() and then calls it just after creating it.

Note that while creating a function its name should start with keyword **function** and all the PHP code should be put

<?php

/\* Defining a PHP Function \*/

*function* writeMessage()

{

echo "You are really a nice person, Have a nice time!";

}

/\* Calling a PHP Function \*/

writeMessage();

?>

This will display the following result:

You are really a nice person, Have a nice time!

## PHP Functions with Parameters

PHP gives you option to pass your parameters inside a function. You can pass as many as parameters your like. These parameters work like variables inside your function. The following example takes two integer parameters and adds them together and then prints them.

<?php

*function* addFunction($num1, $num2)

{

$sum = $num1 + $num2;

echo "Sum of the two numbers is : $sum";

}

addFunction(10, 20);

?>

This will display the following result:

Sum of the two numbers is : 30

## Passing Arguments by Reference

It is possible to pass arguments to functions by reference. This means that a reference to the variable is manipulated by the function rather than a copy of the variable's value.

Any changes made to an argument in these cases will change the value of the original variable. You can pass an argument by reference by adding an ampersand to the variable name in either the function call or the function definition.

The following example depicts both the cases.

<?php

*function* addFive($num)

{

$num += 5;

}

*function* addSix(&$num)

{

$num += 6;

}

$orignum = 10; addFive( &$orignum );

echo "Original Value is $orignum<br />";

addSix( $orignum );

echo "Original Value is $orignum<br />";

?>

This will display the following result:

Original Value is 10 Original Value is 16

## PHP Functions returning value

A function can return a value using the **return** statement in conjunction with a value or object. return stops the execution of the function and sends the value back to the calling code.

You can return more than one value from a function using **return array(1,2,3,4)**.

The following example takes two integer parameters and add them together and then returns their sum to the calling program. Note that **return** keyword is used to return a value from a function.

<?php

*function* addFunction($num1, $num2)

{

$sum = $num1 + $num2; return $sum;

}

$return\_value = addFunction(10, 20);

echo "Returned value from the function : $return\_value";

?>

This will display the following result:

Returned value from the function : 30

## SetTing Default Values for Function Parameters

You can set a parameter to have a default value if the function's caller doesn't pass it. The following function prints NULL in case use does not pass any value to this function.

<?php

*function* printMe($param = NULL)

{

print $param;

}

printMe("This is test"); printMe();

?>

This will produce the following result:

This is test

## Dynamic Function Calls

It is possible to assign function names as strings to variables and then treat these variables exactly as you would the function name itself. The following example depicts this behavior.

<?php

*function* sayHello()

{

echo "Hello<br />";

}

$function\_holder = "sayHello";

$function\_holder();

?>

This will display the following result:

Hello

# PHP ─ Coding Standard

Every company follows a different coding standard based on their best practices. Coding standard is required because there may be many developers working on different modules. If they start inventing their own standards, then the source will become very un- manageable and it will become difficult to maintain that source code in future.

Here are several reasons why to use coding specifications:

* Your peer programmers have to understand the code you produce. A coding standard acts as the blueprint for all the team to decipher the code.
* Simplicity and clarity achieved by consistent coding saves you from common mistakes.
* If you revise your code after some time, then it becomes easy to understand that code.

There are a few guidelines which can be followed while coding in PHP.

* **Indenting and Line Length** - Use an indent of 4 spaces and don't use any tab because different computers use different setting for tab. It is recommended to keep lines at approximately 75-85 characters long for better code readability.
* **Control Structures** - These include if, for, while, switch, etc. Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls. You are strongly encouraged to always use curly braces even in situations where they are technically optional.

**Example**

<?php

if ((condition1) || (condition2)) {

action1;

} elseif ((condition3) && (condition4)) {

action2;

} else {

default action;

}

?>

You can write switch statements as follows:

<?php

switch (condition) {

case 1:

action1;

break;

case 2:

action2;

break;

default:

defaultaction;

break;

}

?>

**Function Calls** - Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon. Here's an example:

$var = foo($bar, $baz, $quux);

**Function Definitions** - Function declarations follow the "BSD/Allman style":

*function* fooFunction($arg1, $arg2 = '')

{

if (condition)

{

statement;

}

return $val;

}

**Comments** - C style comments (/\* \*/) and standard C++ comments (//) are both fine. Use of Perl/shell style comments (#) is discouraged.

**PHP Code Tags** - Always use <?php ?> to delimit PHP code, not the <? ?> shorthand. This is required for PHP compliance and is also the most portable way to include PHP code on differing operating systems and setups.

**Variable Names:**

* Use all lower case letters
* Use '\_' as the word separator.
* Global variables should be prepended with a 'g'.
* Global constants should be all caps with '\_' separators.
* Static variables may be prepended with 's'.

**Make Functions Reentrant** - Functions should not keep static variables that prevent a function from being reentrant.

**Alignment of Declaration Blocks** - Block of declarations should be aligned.

**One Statement Per Line** - There should be only one statement per line unless the statements are very closely related.

**Short Methods or Functions** - Methods should limit themselves to a single page of code. There could be many more points which should be considered while writing your PHP program. Over all intension should be to be consistent throughout of the code programming and it will be possible only when you will follow any coding standard. You can device your own standard if you like something different.

# PHP ─ Object Oriented Programming

We can imagine our universe made of different objects like sun, earth, moon etc. Similarly, we can imagine our car made of different objects like wheel, steering, gear etc. In the same way, there are object oriented programming concepts which assume everything as an object and implement a software using different objects.

## Object Oriented Concepts

Before we go in detail, let’s define important terms related to Object Oriented Programming.

* + - **Class:** This is a programmer-defined datatype, which includes local functions as well as local data. You can think of a class as a template for making many instances of the same kind (or class) of object.
    - **Object:** An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as **instance**.
    - **Member Variable:** These are the variables defined inside a class. This data will be invisible to the outside of the class and can be accessed via member functions. These variables are called **attribute** of the object once an object is created.
    - **Member function:** These are the function defined inside a class and are used to access object data.
    - **Inheritance:** When a class is defined by inheriting existing function of a parent class, then it is called inheritance. Here child class will inherit all or few member functions and variables of a parent class.
    - **Parent class:** A class that is inherited from by another class. This is also called a base class or super class.
    - **Child Class:** A class that inherits from another class. This is also called a subclass or derived class.
    - **Polymorphism:** This is an object oriented concept where the same function can be used for different purposes. For example, function name will remain same but it may take different number of arguments and can do different task.
    - **Overloading:** a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly, functions can also be overloaded with different implementation.
    - **Data Abstraction:** Any representation of data in which the implementation details are hidden (abstracted).
    - **Encapsulation:** refers to a concept where we encapsulate all the data and member functions together to form an object.
    - **Constructor**: refers to a special type of function which will be called automatically whenever there is an object formation from a class.
    - **Destructors:** refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.

## Defining PHP Classes

The general form for defining a new class in PHP is as follows:

<?php

*class* phpClass{

*var* $var1;

*var* $var2 = "constant string";

*function* myfunc ($arg1, $arg2) {

[..]

}

[..]

}

?>

Here is the description of each line:

* + - The special form class, followed by the name of the class that you want to define.
    - A set of braces enclosing any number of variable declarations and function definitions.
    - Variable declarations start with the special form var, which is followed by a conventional $ variable name; they may also have an initial assignment to a constant value.
    - Function definitions look much like standalone PHP functions but are local to the class and will be used to set and access object data.

*Example*

Here is an example which defines a class

<?php

*class* Books{

/\* Member variables \*/ *var* $price;

*var* $title;

/\* Member functions \*/

*function* setPrice($par){

$this->price = $par;

}

*function* getPrice(){

echo $this->price ."/n";

}

*function* setTitle($par){

$this->title = $par;

}

*function* getTitle(){

echo $this->title ." /n";

}

}

?>

The variable **$this** is a special variable and it refers to the same object, i.e., itself.

## Creating Objects in PHP

Once you defined your class, then you can create as many objects as you like of that class type. Following is an example of how to create object using **new** operator.

$physics = new *Books*;

$maths = new *Books*;

$chemistry = new *Books*;

Here we have created three objects and these objects are independent of each other and they will have their existence separately. Next, we will see how to access member function and process member variables.

## Calling Member Functions

After creating your objects, you will be able to call member functions related to that object. One-member function will be able to process member variable of related object only.

Following example shows how to set title and prices for the three books by calling member functions.

$physics->setTitle( "Physics for High School" );

$chemistry->setTitle( "Advanced Chemistry" );

$maths->setTitle( "Algebra" );

$physics->setPrice( 10 );

$chemistry->setPrice( 15 );

$maths->setPrice( 7 );

Now you call another member functions to get the values set by in above example:

$physics->getTitle();

$chemistry->getTitle();

$maths->getTitle();

$physics->getPrice();

$chemistry->getPrice();

$maths->getPrice();

This will produce the following result:

Physics for High School Advanced Chemistry Algebra

10

15

7

## Constructor Functions

Constructor Functions are special type of functions which are called automatically whenever an object is created. So we take full advantage of this behavior, by initializing many things through constructor functions.

PHP provides a special function called **construct()** to define a constructor. You can pass as many as arguments you like into the constructor function.

Following example will create one constructor for Books class and it will initialize price and title for the book at the time of object creation.

*Function* \_construct( $par1, $par2 ){

$this->price = $par1;

$this->title = $par2;

}

Now we don't need to call set function separately to set price and title. We can initialize these two member variables at the time of object creation only. Check following example below:

$physics = new *Books*( "Physics for High School", 10 );

$maths = new *Books* ( "Advanced Chemistry", 15 );

$chemistry = new *Books* ("Algebra", 7 );

/\* Get those set values \*/

$physics->getTitle();

$chemistry->getTitle();

$maths->getTitle();

$physics->getPrice();

$chemistry->getPrice();

$maths->getPrice();

This will produce the following result:

Physics for High School Advanced Chemistry Algebra

10

15

7

## Destructor

Like a constructor function you can define a destructor function using function \_**destruct()**. You can release all the resources with-in a destructor.

## Inheritance

PHP class definitions can optionally inherit from a parent class definition by using the extends clause. The syntax is as follows:

*class* Child extends *Parent* {

<definition body>

}

The effect of inheritance is that the child class (or subclass or derived class) has the following characteristics:

* Automatically has all the member variable declarations of the parent class.
* Automatically has all the same member functions as the parent, which (by default) will work the same way as those functions do in the parent.

Following example inherit Books class and adds more functionality based on the requirement.

*class* Novel extends *Books*{

*var* publisher;

*function* setPublisher($par){

$this->publisher = $par;

}

*function* getPublisher(){

echo $this->publisher. "<br />";

}

}

Now apart from inherited functions, class Novel keeps two additional member functions.

## Function Overriding

Function definitions in child classes override definitions with the same name in parent classes. In a child class, we can modify the definition of a function inherited from parent class.

In the following example, getPrice and getTitle functions are overridden to return some values.

*function* getPrice(){

echo $this->price . "<br/>";

return $this->price;

}

*function* getTitle(){

echo $this->title . "<br/>";

return $this->title;

}

## Public Members

Unless you specify otherwise, properties and methods of a class are public. That is to say, they may be accessed in three possible situations:

* From outside the class in which it is declared
* From within the class in which it is declared
* From within another class that implements the class in which it is declared

Till now we have seen all members as public members. If you wish to limit the accessibility of the members of a class, then you define class members as **private** or **protected**.

## Private members

By designating a member private, you limit its accessibility to the class in which it is declared. The private member cannot be referred to from classes that inherit the class in which it is declared and cannot be accessed from outside the class.

A class member can be made private by using **private** keyword in front of the member.

*class* MyClass {

private $car = "skoda";

$driver = "SRK";

*function* construct($par) {

// Statements here run every time an instance of the class is created.

}

*function* myPublicFunction() {

return("I'm visible!");

}

private *function* myPrivateFunction() {

return("I'm not visible outside!");

}

}

When *MyClass* class is inherited by another class using extends, myPublicFunction() will be visible, as will $driver. The extending class will not have any awareness of or access to myPrivateFunction and $car, because they are declared private.

## Protected members

A protected property or method is accessible in the class in which it is declared, as well as in classes that extend that class. Protected members are not available outside of those two kinds of classes. A class member can be made protected by using **protected** keyword in front of the member.

Here is different version of MyClass:

*class* MyClass {

protected $car = "skoda";

$driver = "SRK";

*function* construct($par) {

// Statements here run every time

// an instance of the class

// is created.

}

*function* myPublicFunction() {

return("I'm visible!");

}

protected *function* myPrivateFunction() {

return("I'm visible in child class!");

}

}

## Interfaces

Interfaces are defined to provide a common function names to the implementors. Different implementors can implement those interfaces according to their requirements. You can say, interfaces are skeletons which are implemented by developers.

As of PHP5, it is possible to define an interface, like this:

*interface* Mail {

public *function* sendMail();

}

Then, if another class implemented that interface, like this:

*class* Report implements *Mail* {

// sendMail() Definition goes here

}

## Constants

A constant is somewhat like a variable, in that it holds a value, but is really more like a function because a constant is immutable. Once you declare a constant, it does not change. Declaring one constant is easy, as is done in this version of MyClass:

*class* MyClass {

const requiredMargin = 1.7;

*function* construct($incomingValue) {

// Statements here run every time

// an instance of the class

// is created.

}

}

In this class, requiredMargin is a constant. It is declared with the keyword const, and under no circumstances can it be changed to anything other than 1.7. Note that the constant's name does not have a leading $, as variable names do.

## Abstract Classes

An abstract class is one that cannot be instantiated, only inherited. You declare an abstract class with the keyword **abstract**, like this:

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same visibility.

abstract *class* MyAbstractClass {

abstract *function* myAbstractFunction() {

}

}

Note that the function definitions inside an abstract class must also be preceded by the keyword abstract. It is not legal to have abstract function definitions inside a non-abstract class.

## Static Keyword

Declaring class members or methods as static makes them accessible without needing an instantiation of the class. A member declared as static cannot be accessed with an instantiated class object (though a static method can).

Try out the following example:

## Final Keyword

PHP 5 introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final, then it cannot be extended.

<?php

*class* Foo

{

public static $my\_static = 'foo';

public *function* staticValue() {

return *self*::$my\_static;

}

}

print Foo::$my\_static . "\n";

$foo = new *Foo*();

print $foo->staticValue() . "\n";

?>

The following example results in Fatal error: Cannot override final method BaseClass::moreTesting()

<?php

*class* BaseClass {

public *function* test() {

echo "BaseClass::test() called<br>";

}

final public *function* moreTesting() {

echo "BaseClass::moreTesting() called<br>";

}

}

*class* ChildClass extends *BaseClass* {

public *function* moreTesting() {

echo "ChildClass::moreTesting() called<br>";

}

}

?>

### Calling parent constructors

Instead of writing an entirely new constructor for the subclass, let's write it by calling the parent's constructor explicitly and then doing whatever is necessary in addition for instantiation of the subclass. Here's a simple example:

<?php

*class* Name

{

*var* $\_firstName;

*var* $\_lastName;

*function* Name($first\_name, $last\_name)

{

$this->\_firstName = $first\_name;

$this->\_lastName = $last\_name;

}

*function* toString() {

return($this->\_lastName .", " .$this->\_firstName);

}

}

*class* NameSub1 extends *Name*

{

*var* $\_middleInitial;

*function* NameSub1($first\_name, $middle\_initial, $last\_name) {

Name::Name($first\_name, $last\_name);

$this->\_middleInitial = $middle\_initial;

}

*function* toString() {

return(Name::toString() . " " . $this->\_middleInitial);

}

}

?>

In this example, we have a parent class (Name), which has a two-argument constructor, and a subclass (NameSub1), which has a three-argument constructor. The constructor of NameSub1 functions by calling its parent constructor explicitly using the :: syntax (passing two of its arguments along) and then setting an additional field. Similarly, NameSub1 defines its non-constructor toString() function in terms of the parent function that it overrides.

**NOTE**: A constructor can be defined with the same name as the name of a class. It is defined in above example

# PHP ─ Error and Exception Handling

Error handling is the process of catching errors raised by your program and then taking appropriate action. If you would handle errors properly, then it may lead to many unforeseen consequences. It is very simple in PHP to handle errors.

## Using die() function

While writing your PHP program you should check all possible error condition before going ahead and take appropriate action when required.

Try the following example without having **/tmp/test.xt** file and with this file.

<?php if (!file\_exists("/tmp/test.txt")) {

die("File not found");

} else {

$file=fopen("/tmp/test.txt", "r");

print "Opend file sucessfully";

}

// Test of the code here.

?>

You can thus write an efficient code. Using the above technique, you can stop your program whenever it errors out and display more meaningful and user-friendly message.

## Defining Custom Error Handling Function

You can write your own function to handling any error. PHP provides you a framework to define error-handling function.

This function must be able to handle a minimum of two parameters (error level and error message) but can accept up to five parameters (optionally: file, line-number, and the error context):

**Syntax**

error\_function(error\_level,error\_message, error\_file,error\_line,error\_context);

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| error\_level | Required - Specifies the error report level for the user-defined error. Must be a value number. |
| error\_message | Required - Specifies the error message for the user-defined error |
| error\_file | Optional - Specifies the filename in which the error occurred |
| error\_line | Optional - Specifies the line number in which the error occurred |
| error\_context | Optional - Specifies an array containing every variable and their values in use when the error occurred |

### Possible Error levels

These error report levels are the different types of error the user-defined error handler can be used for. These values cab used in combination using **|** operator

|  |  |  |
| --- | --- | --- |
| **Value** | **Constant** | **Description** |
| 1 | E\_ERROR | Fatal run-time errors. Execution of the script is halted |
| 2 | E\_WARNING | Non-fatal run-time errors. Execution of the script is not halted |
| 4 | E\_PARSE | Compile-time parse errors. Parse errors should only be generated by the parser. |
| 8 | E\_NOTICE | Run-time notices. The script found something that might be an error, but could also happen when running a script normally |
| 16 | E\_CORE\_ERROR | Fatal errors that occur during PHP's initial startup. |
| 32 | E\_CORE\_WARNING | Non-fatal run-time errors. This occurs during PHP's initial startup. |
| 256 | E\_USER\_ERROR | Fatal user-generated error. This is like an E\_ERROR set by the programmer using the PHP function trigger\_error() |
| 512 | E\_USER\_WARNING | Non-fatal user-generated warning. This is like an E\_WARNING set by the programmer using the PHP function trigger\_error() |
| 1024 | E\_USER\_NOTICE | User-generated notice. This is like an E\_NOTICE set by the programmer using the PHP function trigger\_error() |
| 2048 | E\_STRICT | Run-time notices. Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code. |
| 4096 | E\_RECOVERABLE\_ERROR | Catchable fatal error. This is like an E\_ERROR but can be caught by a user defined handle (see also set\_error\_handler()) |
| 8191 | E\_ALL | All errors and warnings, except level E\_STRICT (E\_STRICT will be part of E\_ALL as of PHP 6.0) |

All the above error level can be set using the following PHP built-in library function where level cab be any of the value defined in above table.

int error\_reporting ( [int $level] )

Here is how you can create an error handling function:

<?php

*function* handleError($errno, $errstr, $error\_file, $error\_line)

{

echo "<b>Error:</b> [$errno] $errstr - $error\_file:$error\_line";

echo "<br />";

echo "Terminating PHP Script";

die();

}

?>

Once you define your custom error handler, you need to set it using PHP built-in library **set\_error\_handler** function. Now let’s examine our example by calling a function which does not exist.

<?php

error\_reporting( E\_ERROR );

*function* handleError($errno, $errstr, $error\_file, $error\_line)

{

echo "<b>Error:</b> [$errno] $errstr - $error\_file:$error\_line";

echo "<br />";

echo "Terminating PHP Script";

die();

}

//set error handler

set\_error\_handler("handleError");

//trigger error

myFunction();

?>

## Exceptions Handling

PHP 5 has an exception model similar to that of other programming languages. Exceptions are important and provides a better control over error handling.

Let’s now explain the new keyword related to exceptions.

* Try - A function using an exception should be in a "try" block. If the exception does not trigger, the code will continue as normal. However, if the exception triggers, an exception is "thrown".
* Throw - This is how you trigger an exception. Each "throw" must have at least one "catch".
* Catch - - A "catch" block retrieves an exception and creates an object containing the exception information.

When an exception is thrown, the code following the statement will not be executed, and PHP will attempt to find the first matching catch block. If an exception is not caught, a PHP Fatal Error will be issued with an "Uncaught Exception ...

* An exception can be thrown, and caught ("catched") within PHP. Code may be surrounded in a try block.
* Each try must have at least one corresponding catch block. Multiple catch blocks can be used to catch different classes of exceptions.
* Exceptions can be thrown (or re-thrown) within a catch block.

**Example**

Copy and paste the following piece of code into a file and verify the result.

<?php

try {

$error = 'Always throw this error';

throw new *Exception*($error);

// Code following an exception is not executed.

echo 'Never executed';

} catch (*Exception* $e) {

echo 'Caught exception: ', $e->getMessage(), "\n";

}

// Continue execution

echo 'Hello World';

?>

In the above example, $e->getMessage function is used to get error message. The following functions can be used from **Exception** class.

* getMessage()- message of exception
* getCode() - code of exception
* getFile() - source filename
* getLine() - source line
* getTrace() - n array of the backtrace()
* getTraceAsString() - formated string of trace

### Creating Custom Exception Handler

You can define your own custom exception handler. Use the following function to set a user-defined exception handler function.

string set\_exception\_handler ( callback $exception\_handler )

Here **exception\_handler** is the name of the function to be called when an uncaught exception occurs. This function must be defined before calling set\_exception\_handler().

**Example**

<?php

*function* exception\_handler($exception)

{

echo "Uncaught exception: " , $exception->getMessage(), "\n";

}

set\_exception\_handler('exception\_handler');

throw new *Exception*('Uncaught Exception');

echo "Not Executed\n";

?>

Check the complete set of error handling functions at PHP Error Handling Functions.