# Introduction to PHP

PHP (standing for *PHP Hypertext Preprocessor*) is a **server-side scripting programming language** which is embedded in HTML and allows web developers to create dynamic content that interacts with databases. It is one of the most popular languages for developing **web-based software applications**.

PHP syntax is similar to C, and its OOP is similar to C#. However, because PHP is a scripting language, there is **no separate compilation step** for PHP scripts.

# Coding Convention

# Syntax Overview

## Escaping to PHP

The PHP parser needs a way to differentiate PHP code from other elements (mostly HTML tags) in the page. The mechanism for doing so is known as 'escaping to PHP'. There are four ways to do this:

**Canonical PHP tags**

<?php

...

?>

This is the most universal, popular and effective option.

**Short-open (SGML-style) tags**

<?

...

?>

This is the shortest option, but you must do one of two things to enable PHP to recognize the tags:

* Choose the --enable-short-tags configuration option when you're building PHP.
* Set the short\_open\_tag setting in your php.ini file to on. This option must be disabled to parse XML with PHP because the same syntax is used for XML tags.

**ASP-style tags**

<%

...

%>

To use this option, you'll need to set the configuration option in your php.ini file.

**HTML script tags**

<script language = "PHP">

...

</script>

## Commenting

There are two commenting formats in PHP:

**Single-line comments**

<?php

   # This is a comment

   // This is a comment too. Each style comments only

   print("An example with single line comments");

?>

**Multi-lines comments**

<?php

   /\* This is a comment with multiline

      Author : Mohammad Mohtashim

      Purpose: Multiline Comments Demo

      Subject: PHP

   \*/

   print("An example with multi line comments");

?>

## Output to Screen

In PHP, there are two basic ways to get output: echo and print. These statements can be used with or without parentheses: echo or echo(), print or print().

They are **more or less the same**. The differences are small:

* echo has no return value while print has a return value of 1 so it can be used in expressions.
* echo can take multiple parameters (although such usage is rare) while print can take one argument.
* echo is marginally faster than print.

Example 1: We can replace echo by print in all below cases, except the last one.

<?php

echo "<h2>PHP is Fun!</h2>";

echo "Hello world!<br>";

echo "I'm about to learn PHP!<br>";

# Only echo can do this:

echo "This ", "string ", "was ", "made ", "with multiple parameters.";

?>

Example 2: We can replace echo by print in all below cases

<?php

$txt1 = "Learn PHP";

$txt2 = "W3Schools.com";

$x = 5;

$y = 4;

echo "<h2>" . $txt1 . "</h2>";

echo "Study PHP at " . $txt2 . "<br>";

echo $x + $y;

?>

## Others

* PHP is whitespace insensitive.
* PHP is case sensitive.
* Statements must be terminated by semicolons (;).
* Braces ( {…} ) make blocks.

# Variables and Data Types

## Variables

Here are the most important things to know about variables in PHP:

* All variables are denoted with a **leading dollar sign** ($).
* PHP is **dynamic typed**. So, a variable doesn't need to know in advance whether it will be used to store a number or a string of characters.
* PHP does a good job of automatically converting types from one to another when necessary.
* Variables used before they are assigned always have **default values**.

*You might not know!*

In PHP 7, type declarations were added. This gives an option to specify the data type expected when declaring a function, and by enabling the strict requirement, it will throw a "Fatal Error" on a type mismatch.

### Scope

PHP variables can be one of four scope types:

* [Local variables](https://www.tutorialspoint.com/php/php_local_variables.htm): Declared in the function. Can be accessed in that function only.
* [Function parameters](https://www.tutorialspoint.com/php/php_function_parameters.htm): Declared after the function name and inside parentheses. Can be accessed in that function only.
* [Global variables](https://www.tutorialspoint.com/php/php_global_variables.htm): Declared outside all functions. Can be accessed in any part of the program. Have to use keyword GLOBAL to modify global variables inside functions.
* [Static variables](https://www.tutorialspoint.com/php/php_static_variables.htm): Declared in the function. Will lose its scope when the function exits, but will NOT lose its value (meaning the variable will still hold that value once the function is called again). Have to use keyword STATIC to declare static variables.

### Pre-Defined Variables

<https://www.tutorialspoint.com/php/php_predefined_variables.htm>

## Data Types

PHP has a total of 8 data types which we use to construct variables:

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

### Integers

They correspond to whole numbers, both positive and negative.

<?php

$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.

### Doubles

By default, doubles print with the minimum number of decimal places needed. For example:

<?php

   $many = 2.2888800;

   $many\_2 = 2.2111200;

   $few = $many + $many\_2;

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

?>

Output:

2.28888 + 2.21112 = 4.5

### Boolean

PHP provides a couple of constants especially for use as Booleans: TRUE and FALSE. For example:

<?php

if (TRUE) {

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

  }

  else {

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

  }

?>

We can **interpret 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 0 and TRUE otherwise.
* If the value is a string, it is FALSE if the string is empty or if the string is "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.

For example:

<?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

To give a variable the NULL value, simply assign it like this:

<?php

$my\_var = NULL;

?>

or

<?php

$my\_var = null;

?>

**Note**: If a variable is created without a value, it is automatically assigned a value of NULL.

### Strings

#### Concept and Syntax

They are sequences of characters. For example:

<?php

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

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

$string\_3 = "This string has thirty-nine characters";

$string\_4 = "";     // An empty string

?>

**Note**: Singly-quoted strings are treated almost literally, whereas doubly-quoted strings replace variables with their values as well as specially interpreting certain character sequences. For example:

<?php

   $variable = "name";

   $literally = 'My $variable will not print.';

   print($literally);

   print("<br>");

   $literally = "My $variable will print.";

   print($literally);

?>

Output:

My $variable will not print.

My name will print.

Strings that are delimited by double quotes are preprocessed in both the following two ways:

* 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 (\)

#### String Concatenation Operator

To concatenate two string variables, use the dot (.) operator:

<?php

   $string1 = "Hello";

   $string2 = "World";

   echo $string1 . " " . $string2;

   echo $string1 . $string2;

?>

Output:

Hello World

HelloWorld

#### String Built-In Functions

**strlen()**

Find the length of a string.

For example:

<?php

   echo strlen("Hello world!");

?>

Output:

12

**strpos()**

Search for position (start from 0) of a string or character within a string. If a match is found, it returns the position of the first match. If no match is found, it returns FALSE.

For example:

<?php

   echo strpos("Hello world!", "world");

?>

Output:

6

**For a complete list of string built-in functions, check** [**here**](https://www.w3schools.com/php/php_ref_string.asp)**.**

### Arrays

The concept and use of arrays in PHP are similar as in C/C++. However, there are some small differences in syntax – we have used array() function to create array.

In PHP, there are three different types of arrays:

* **Numeric array** − An array with a numeric index. Values are stored and accessed in linear fashion.
* **Associative array** − An array with strings as index. Each value is stored in association with a unique key rather than in a strict linear index order.
* **Multidimensional array** − An array containing one or more arrays. Values are accessed using multiple indices.

#### Numeric Arrays

These arrays can store numbers, strings and any object. But their index must be represented by numbers. By default, array index starts from 0.

Example:

<?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>";

    }

?>

Output:

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

Associate arrays in PHP share the same concept as std::map in C++ or dictionary in Python. They all have their index as string and establish a strong association between *keys* and *values*.

Example:

<?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>";

?>

Output:

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

In 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 indices.

Example:

<?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>";

?>

Output:

Marks for Mohammad in Physics: 35

Marks for Qadir in Maths: 32

Marks for Zara in Chemistry: 39

### Objects

Check [this section](#_Creating_Objects).

### Resources

The special resource type is not an actual data type. It is the storing of a reference to functions and resources external to PHP.

A common example of using the resource data type is a database call.

# Constants

The concept and use of constants are similar to other languages. However, the way it's defined is a bit different:

* To define a constant in PHP, we can use either the const keyword or the define() function.
* To retrieve the value of a constant, we can either simply specify its name or use the constant() function.

Unlike variables, constant don't need a leading $.

## Example

<?php

    const MINSIZE = 50;

    echo MINSIZE . "<br>";

    define('MAXNSIZE', 10);

    echo constant('MAXNSIZE') . "<br>";

?>

Output:

50

100

* **const vs define()**
* The const keyword defines constants at **compile time**, whereas define() defines constants at **run time**.
* We can't use the const keyword to declare constant in conditional blocks, while with define() we can achieve that.

<?php

    if() {

        const VAR = 'FOO';      // invalid

    }

    if() {

        define('VAR', 'FOO');   // valid

    }

?>

* The const keyword accepts a static scalar (number, string or other constants like TRUE, FALSE, NULL, \_\_FILE\_\_, etc.), whereas define() takes any expression.
* The const keyword are always **case-sensitive**, whereas define() allows you to define case-insensitive constants by passing true as the third argument.
* The const keyword can be **utilized within a class or interface** to declare a class constant or interface constant, while define() can't be utilized for this reason.

<?php

    class MyClass {

        const VAR = 2;      // valid

        echo VAR;           // valid

    }

    // but

    class MyClass {

        define('VAR', 2);   // invalid

        echo VAR;           // invalid

    }

?>

## PHP Magic Constants

PHP provides a large number of predefined constants. These are case-insensitive. Some of them are:

|  |  |
| --- | --- |
| **Name** | **Description** |
| \_\_LINE\_\_ | The current line number of the file. |
| \_\_FILE\_\_ | The absolute path and filename of the file. If used inside an include, the name of the included file is returned. |
| \_\_FUNCTION\_\_ | The function name. |
| \_\_CLASS\_\_ | The class name. |
| \_\_METHOD\_\_ | The class method name. |

# Operators

PHP supports following types of operators.

* Arithmetic operators
* Comparison operators
* Logical operators
* Assignment operators
* Conditional operators

There syntax and use are similar to as in C/C++.

More details: <https://www.tutorialspoint.com/php/php_operator_types.htm>

# Decision Making

## The 'if' Statement

**Syntax:**

if (condition) {

    // code to be executed if condition is true

}

**Example:**

<?php

    $a = 1;

    if ($a == 1) {

       echo "Value is 1!";

    }

?>

Output:

Value is 1

## The 'if...else' Statement

**Syntax:**

if (condition) {

    // code to be executed if condition is true;

}

else {

    // code to be executed if condition is false;

}

**Example:**

<?php

    $d = date("D");

    if ($d == "Fri") {

        echo "Have a nice weekend!";

    }

    else {

        echo "Have a nice day!";

    }

?>

Output:

Have a nice weekend!

## The 'elseIf' Statement

**Syntax:**

if (condition1) {

    // code to be executed if condition1 is true

}

elseif (condition2) {

    // code to be executed if condition1 is true

}

else {

    // code to be executed if all conditions are false

}

**Example:**

<?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!";

    }

?>

Output:

Have a nice weekend!

## The 'switch' Statement

**Syntax:**

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 not label1 and label2

}

**Example:**

<?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?";

    }

?>

Output:

Today is Friday

# Loops

## The 'for' Loop

Same as in C/C++.

More details: <https://www.tutorialspoint.com/php/php_loop_types.htm>

## The 'while' Loop

Same as in C/C++.

## The 'do…while' Loop

Same as in C/C++.

## The 'foreach' Loop

**Syntax:**

foreach (<array-name> as <array-item>) {

    // code to be executed;

}

**Example:**

<?php

    $arr = array(1, 2, 3, 4, 5);

    foreach($arr as $value) {

        echo "Value is $value <br>";

    }

?>

Output:

Value is 1

Value is 2

Value is 3

Value is 4

Value is 5

## The 'break' and 'continue' Statements

Same as in C/C++.

# Functions

PHP functions are similar to other programming languages.

To define a function, we use keyword function and put all the code inside { … } braces.

## Function Without Parameters

For example:

<?php

    // Defining a function

    function writeMessage() {

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

    }

    // Calling a function

    writeMessage();

?>

Output:

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

## Functions With Parameters

**Passing by value:**

For example:

<?php

    function addFunction($num1, $num2) {

        $sum = $num1 + $num2;

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

    }

    addFunction(10, 20);

?>

Output:

Sum of the two numbers is: 30

**Passing by Reference:**

You can pass an argument by reference by adding an ampersand (&) to the variable name in either the function call or the function definition.

For example:

<?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>";

?>

Output:

Original Value is 10

Original Value is 16

## Default Values for Function Parameters

For example, 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();

?>

Output:

This is test

## Functions With Return Value

A function can return a value using the return statement in conjunction with a value or object. This 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).

For example:

<?php

    function addFunction($num1, $num2) {

        $sum = $num1 + $num2;

        return $sum;

    }

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

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

?>

Output:

Returned value from the function: 30

## 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.

For example:

<?php

    function sayHello() {

        echo "Hello<br>";

    }

    $function\_holder = "sayHello";

    $function\_holder();

?>

Output:

Hello

# OOP

## Classes

### Example

Here is an example which defines a class of Books using keyword class:

class Books {

    // Member variables

    var $title;              # Note the keyword "var"

    var $price;

    // Member functions

    function setTitle($par) {

        $this->title = $par;    # Note the variable "$this"

    }

    function getTitle() {

        echo $this->title ." <br/>";

    }

    function setPrice($par) {

        $this->price = $par;

    }

    function getPrice() {

        echo $this->price ."<br/>";

    }

}

**Notes:**

* Class member variables must be declared with a special keyword var.
* The variable $this is a special variable which refers to the object itself. Note that PHP also have a keyword self to refer to the class itself (it's usually used with [static class members](#_Static_Keyword)).

### Creating Objects

Once you defined your class, then you can create as many objects as you like of that class type. These objects are independent of each other and thus, have their existence separately.

Following is an example of how to create object using the new operator.

$physics = new Books;

$maths = new Books;

### Calling Member Functions

After creating your objects, you will be able to call member functions related to that object using operator ->.

Following example shows how to set title for physics and maths books by calling member functions:

$physics->setTitle("Mechanics");

$maths->setTitle("Algebra");

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

$physics->getTitle();

$maths->getTitle();

This will produce the following result:

Mechanics

Algebra

### Constructor

Constructor is a special function which is called automatically whenever an object is created. We can take full advantage of this behavior, by initializing many things through the constructor.

In PHP, a constructor must be named as \_\_construct(). You can pass as many arguments as you like into it.

Following example creates one constructor for Books class and initializes title and price for the book at the time of object creation.

function \_\_construct($par1, $par2) {

    $this->title = $par1;

    $this->price = $par2;

}

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

$physics = new Books("Mechanics", 10);

$maths = new Books ("Algebra", 7);

// Get those set values

$physics->getTitle();

$maths->getTitle();

$physics->getPrice();

$maths->getPrice();

Output:

Mechanics

Algebra

10

7

### Destructor

You can define a destructor function using function \_\_destruct(). You can release all the resources with-in a destructor.

## Properties

### Inheritance

Just like C#, inheritance in PHP allows us to define a class that inherits all the methods and properties from another class.

* *Parent class* is the class being inherited from, also called base class.
* *Child class* is the class that inherits from another class, also called derived class.

Following example inherit Books class and adds more functionality using keyword extends:

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 have two additional member functions:

$novel = new Novel();

// Use methods of the parent class

$novel->setTitle("Harry Potter");

$novel->getTitle();

// Use methods of itself

$novel->setPublisher("Bloomsbury");

$novel-> getPublisher();

Output:

Harry Potter

Bloomsbury

*You might not know!*

**PHP** **only supports single inheritance**: a child class can inherit only from one single parent.

So, what if a class needs to inherit multiple behaviors? PHP provides feature called [traits](https://www.w3schools.com/php/php_oop_traits.asp) solve this problem.

### Polymorphism

Just like C#, polymorphism in PHP allows us to override methods of parent classes to have a different functionality in your child classes.

In the following example, getPrice() and getTitle() functions are overridden to return some values:

function getTitle(){

    echo $this->title . "<br/>";

    return $this->title;

}

function getPrice() {

    echo $this->price . "<br/>";

    return $this->price;

}

* **Calling parent constructors**

In some cases, we need to override the constructor of the parent class in a child class. Instead of writing an entirely new constructor for the child, we can write it by calling the parent's constructor explicitly and then doing whatever is necessary.

For example:

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 FullName extends Name {

    var $middleName;

    function FullName($first\_name, $middle\_name, $last\_name) {

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

        $this->middleName = $middle\_name;

    }

    function toString() {

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

    }

}

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

### Encapsulation

Just like C#, encapsulation in PHP allows us to make object's attributes and methods 'invisible' outside the class definition. We have three types of *access modifiers*.

* **Public**: By default, member variables and methods of classes are public. That means they can be accessed from anywhere.
* **Private**: Member variables and methods are limited to the class in which they're declared. They cannot be accessed from outside the class or from classes that inherit the class in which it 's declared. To make members private, add keyword private before its declaration.
* **Protected**: Member variables and methods are not available outside of the class. But they are accessible in the class in which it is declared, as well as in classes that extend that class. To make members protected, add keyword protected before its declaration.

Example 1: Private

class MyClass {

    private $car = "skoda";

    $driver = "SRK";

    function myPublicFunction() {

       return("I'm visible!");

    }

    private function myPrivateFunction() {

       return("I'm  not visible outside!");

    }

}

Example 2: Protected

class MyClass {

    protected $car = "skoda";

    $driver = "SRK";

    function myPublicFunction() {

       return("I'm visible!");

    }

    protected function myPrivateFunction() {

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

    }

}

### Abstraction

## Abstract Classes

Just like C#, an abstract class in PHP is one that cannot be instantiated, only inherited.

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.

To create an abstract class, we use keyword abstract:

abstract class MyAbstractClass {

    abstract function myAbstractFunction() {

    }

}

Note: 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.

## Interfaces

Just like C#, interfaces in PHP are defined to provide a common layer to the implementers. Different implementers can implement those interfaces according to their requirements.

To create an interface, we use keyword interface:

interface Mail {

    public function sendMail(); // Cannot have definition inside an interface

}

To implement an interface, we use keyword implements:

class Report implements Mail {

    // Definition for sendMail() goes here

}

## Static Members

Just like C#, declaring class members as static makes them accessible without needing an instantiation of the class.

**Note**: A variables declared as static cannot be accessed with an instantiated class object (though a static method can).

For example:

class Foo {

    public static $my\_static = 'foo';

    public function staticValue() {

        return self::$my\_static; // Cannot use $this because we're referring to

// class itself, not object itself.

    }

}

// Error

echo Foo:$my\_static;

// valid

echo Foo::staticValue();

// Also valid

$foo = new Foo();

echo $foo->staticValue();

## Final Members

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

Following example results in *Fatal error: Cannot override final method BaseClass::moreTesting()*:

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>";

    }

}

## Namespace

Just like C#, namespaces in PHP are qualifiers that solve two different problems:

* They allow for better organization by grouping classes that work together to perform a task.
* They allow the same name to be used for more than one class.

Namespaces must be declared at the beginning of a file using the namespace keyword.

### Example

The following example declares a namespace called Html. All classes, variables, constants, functions declared in this file will belong to the Html namespace:

<?php

    namespace Html;

    class Table {

        public $title = "";

        public $numRows = 0;

        public function message() {

            echo "<p>Table '{$this->title}' has {$this->numRows} rows.</p>";

        }

    }

    $table = new Table();

    $table->title = "My table";

    $table->numRows = 5;

?>

<!DOCTYPE html>

<html>

<body>

<?php

    $table->message();

?>

</body>

</html>

Output:

Table 'My table' has 5 rows.

**Notes:**

* A namespace declaration **must be the first thing in the PHP file**. The following code is invalid:

<?php

    echo "Hello World!";

    namespace Html;

    ...

?>

* Any code that follows a namespace declaration is operating inside the namespace, so classes that belong to the namespace can be instantiated without any qualifiers. However, **to access classes from outside a namespace, the class needs to have the namespace attached to it**.

The following example creates two objects from class Table and Row. Assume these classes are not defined in the same file which declares the namespace Html.

$table = new Html\Table()

$row = new Html\Row();

### Nested Namespaces

For further organization, it is possible to have nested namespaces.

The following example declares a namespace called Html inside a namespace called Code:

<?php

    namespace Code\Html;

    ...

?>

### Namespace Alias

It can be useful to give a namespace or class an alias to make it easier to write. This is done with the use keyword:

For example:

use Html as H;

$table = new H\Table();

# Exception Handling

PHP 5 has an exception model similar to that of other programming languages. Exceptions provides a better control over [error handling](#_Error_Handling).

## Syntax

* **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, 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 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

<?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(), "<br>";

   }

   // Continue execution

   echo 'Hello World';

?>

Output:

Caught exception: Always throw this error

Hello World

In the above example, the getMessage() method from the Exception class is used to get error message. All methods include:

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

# Error 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.

## 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 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 "Opened file successfully";

   }

   // Test of the code here.

?>

Using 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 2 parameters (error level and error message), but can accept up to 5 parameters (optionally: file, line-number, and error context).

### Syntax

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

Where:

* 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 file name 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 values can be used in combination using | operator:

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

All the above error level can be set using following [PHP built-in error handling functions](https://www.tutorialspoint.com/php/php_error_handling_functions.htm) where level can be any of the value defined in above table.

int error\_reporting ( [int $level] )

Following is the way you can create one 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();

?>

# File Inclusion

You can include the content of a PHP file into another PHP file before the server executes it. There are two PHP functions which can be used to included one PHP file into another PHP file.

## The include() Function

This function copies all the text in a specified file into the file that uses the function. **If there is any problem in loading a file, the include() function generates a warning (E\_WARNING) but the script will continue execution**.

**Example**: Assume you want to create a common menu for your website. You create a file menu.php with the following content:

<a href="http://www.tutorialspoint.com/index.htm">Home</a> -

<a href="http://www.tutorialspoint.com/ebxml">ebXML</a> -

<a href="http://www.tutorialspoint.com/ajax">AJAX</a> -

<a href="http://www.tutorialspoint.com/perl">PERL</a> <br>

Now create as many pages as you like and include this file to create header. Assume your test.php file has following content:

<html>

   <body>

    <?php

        include("menu.php");

    ?>

    <p>This is an example to show how to include PHP file!</p>

   </body>

</html>

Output:



If menu.php does not exist, you will get the below warning and the script will keep executing:

This is an example to show how to include PHP file!

PHP Warning: include(menu.php): failed to open stream: No such file or directory in /home/cg/root/9578026/main.php on line 5

## The require() Function

This function copies all the text in a specified file into the file that uses the function. **If there is any problem in loading a file then, the require() function generates a fatal error (E\_COMPILE\_ERROR) and halt the execution of the script**.

So, there is no difference between require() and include() except they handle error conditions. It is **recommended to use the require()** function instead of include(), because scripts should not continue executing if files are missing or misnamed.

You can try using above example with require() function and it will generate same result. But if menu.php does not exist, you will get the below error but the script will stop executing:

PHP Warning: require(menu.php): failed to open stream: No such file or directory in /home/cg/root/9578026/main.php on line 5

**NOTE**: You may get plain warning messages or fatal error messages or nothing at all. This depends on your PHP Server configuration.

# Form

<https://www.tutorialspoint.com/php/php_form_introduction.htm>

<https://hocwebchuan.com/tutorial/php/php_form_handing.php>

<https://hocwebchuan.com/tutorial/php/php_form_validate.php>

# GET and POST Methods

There are two ways the **browser client can send information to the web server**: the GET method and the POST method.

## Similarity

With two methods, the information is encoded using a scheme called "URL encoding" before it's sent to the server.

In this scheme:

* Name/value pairs are joined with equal signs and different pairs are separated by the ampersand (&).
* Spaces are removed and replaced with the + character.
* Any other nonalphanumeric characters are replaced with hexadecimal values.

For example: name=John&age=23&location=New+York

Both GET and POST create an array – which is: array(name => "John", age => 23, location => "New York")

## Differences

|  |  |  |
| --- | --- | --- |
|  | **GET** | **POST** |
| **Principle** | Sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character. | Transfers information via **HTTP headers**. |
| **Security** | Sent data is **visible to everyone** (all variable names and values are displayed in the URL).  🡪 Never use GET method if you have password or other sensitive information to be sent to the server. | Sent data goes is **invisible to others** (all names/values are embedded within the body of the HTTP request).  🡪 Security depends on HTTP protocol. By using Secure HTTP, you can make sure that your information is secure. |
| **Data length** | Has **limits** on the amount of information to send. Most browsers support URLs up to around 2000 characters, while Apache has a default of 8000. | Has **no limits** on data size to be sent. |
| **Data type** | Can't be used to send binary data, like images or word documents, to the server. | Can be used to send ASCII as well as binary data. |
| **Array** | PHP provides $\_GET associative array to access all the sent information using GET method. | PHP provides $\_POST associative array to access all the sent information using POST method. |
|  | Developers prefer POST for sending form data. | |

## Examples

**Example 1: GET**

In view.php:

<form action = "get\_post.php" method = "GET">

    Name: <input type = "text" name = "name" />

    Age: <input type = "text" name = "age" />

    <input type = "submit" />

</form>

In get\_post.php:

<?php

   if( $\_GET["name"] || $\_GET["age"] ) {

      echo "Welcome ". $\_GET['name']. "!<br>";

      echo "You are ". $\_GET['age']. " years old.";

   }

?>

Enter name and age as below:



Then press "Submit" button.

Output:

The URL change from: [localhost/get\_post.php](https://www.tutorialspoint.com/php/get_post.php) to: [localhost/get\_post.php?name=John&age=23](https://www.tutorialspoint.com/php/get_post.php?name=John&age=23)

And you get following output:

Welcome John!

You are 23 years old.

**Example 2: POST**

In view.php:

<form action="get\_post.php" method="POST">

     Username: <input type="text" name="user"><br>

     Password: <input type="password" name="password"><br>

     <input type="submit"/>

</form>

In get\_post.php:

<?php

   if( $\_GET["user"] || $\_GET["password"] ) {

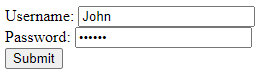
      echo "Welcome ". $\_GET['user']. "!<br>";

      echo "Your password is ". $\_GET['password']. ".";

   }

?>

Enter user and password as below:



Then press "Submit" button.

Output:

The URL does not change.

And you get following output:

Welcome John!

Your password is abc123.

# PHP and MySQL

<https://www.tutorialspoint.com/php/php_and_mysql.htm>

<https://hocwebchuan.com/tutorial/php/php_mysql_connect.php>

<https://www.w3schools.com/php/php_mysql_intro.asp>

# PHP and AJAX

<https://www.tutorialspoint.com/php/php_and_ajax.htm>

<https://www.w3schools.com/php/php_ajax_intro.asp>