**Object Oriented Concepts:**

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. Same way there is 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, lets define important terms related to Object Oriented Programming.

## Introduction

Starting with PHP 5, the object model was rewritten to allow for better performance and more features. This was a major change from PHP 4. PHP 5 has a full object model.

Among the features in PHP 5 are the inclusions of [visibility](http://www.php.net/manual/en/language.oop5.visibility.php), [abstract](http://www.php.net/manual/en/language.oop5.abstract.php) and [final](http://www.php.net/manual/en/language.oop5.final.php) classes and methods, additional [magic methods](http://www.php.net/manual/en/language.oop5.magic.php), [interfaces](http://www.php.net/manual/en/language.oop5.interfaces.php), [cloning](http://www.php.net/manual/en/language.oop5.cloning.php) and [typehinting](http://www.php.net/manual/en/language.oop5.typehinting.php).

* **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 same function can be used for different purposes. For example function name will remain same but it make 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 of Books type:

|  |
| --- |
| <?php  class Books{  /\* Member variables \*/  var $price;  var $title;  /\* Member functions \*/  function setPrice($par){  $this->price = $var;  }  function getPrice(){  echo $this->price ."<br/>";  }  function setTitle($par){  $this->title = $par;  }  function getTitle(){  echo $this->title ." <br/>";  }  }  ?> |

The variable **$this** is a special variable and it refers to the same object ie. 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 existance 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 follwoing 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 behaviour, 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 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 resourceses 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 follwoing example getPrice and getTitle functions are overriden to retrun 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 infront of the member.

|  |
| --- |
| class MyClass {  private $car = "skoda";  $driver = "vettel";  function \_\_construct($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 infront 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 theri requirements. You can say, interfaces are skeltons 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 visibillity.

|  |
| --- |
| abstract class MyAbstractClass {  abstract function myAbstractFunction() {  }  } |

Note that 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 can not be accessed with an instantiated class object (though a static method can).

Try out following example:

|  |
| --- |
| <?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"; |

**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.

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:

|  |
| --- |
| 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 nonconstructor 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.

**The Basics**

**class**

Basic class definitions begin with the keyword *class*, followed by a class name, followed by a pair of curly braces which enclose the definitions of the properties and methods belonging to the class.

The class name can be any valid label which is not a PHP [reserved word](http://www.php.net/manual/en/reserved.php). A valid class name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: *[a-zA-Z\_\x7f-\xff][a-zA-Z0-9\_\x7f-\xff]\**.

A class may contain its own [constants](http://www.php.net/manual/en/language.oop5.constants.php), [variables](http://www.php.net/manual/en/language.oop5.properties.php) (called "properties"), and functions (called "methods").

**Example #1 Simple Class definition**

<?php  
class SimpleClass  
{  
    // property declaration  
    public $var = 'a default value';  
  
    // method declaration  
    public function displayVar() {  
        echo $this->var;  
    }  
}  
?>

The pseudo-variable *$this* is available when a method is called from within an object context. *$this* is a reference to the calling object (usually the object to which the method belongs, but possibly another object, if the method is called [statically](http://www.php.net/manual/en/language.oop5.static.php) from the context of a secondary object).

**Example #2 Some examples of the *$this* pseudo-variable**

<?php  
class A  
{  
    function foo()  
    {  
        if (isset($this)) {  
            echo '$this is defined (';  
            echo get\_class($this);  
            echo ")\n";  
        } else {  
            echo "\$this is not defined.\n";  
        }  
    }  
}  
  
class B  
{  
    function bar()  
    {  
        // Note: the next line will issue a warning if E\_STRICT is enabled.  
        A::foo();  
    }  
}  
  
$a = new A();  
$a->foo();  
  
// Note: the next line will issue a warning if E\_STRICT is enabled.  
A::foo();  
$b = new B();  
$b->bar();  
  
// Note: the next line will issue a warning if E\_STRICT is enabled.  
B::bar();  
?>

The above example will output:

$this is defined (A)

$this is not defined.

$this is defined (B)

$this is not defined.

**new**

To create an instance of a class, the *new* keyword must be used. An object will always be created unless the object has a [constructor](http://www.php.net/manual/en/language.oop5.decon.php) defined that throws an [exception](http://www.php.net/manual/en/language.exceptions.php) on error. Classes should be defined before instantiation (and in some cases this is a requirement).

If a [string](http://www.php.net/manual/en/language.types.string.php) containing the name of a class is used with *new*, a new instance of that class will be created. If the class is in a namespace, its fully qualified name must be used when doing this.

**Example #3 Creating an instance**

<?php  
$instance = new SimpleClass();  
  
// This can also be done with a variable:  
$className = 'Foo';  
$instance = new $className(); // Foo()  
?>

In the class context, it is possible to create a new object by *new self* and *new parent*.

When assigning an already created instance of a class to a new variable, the new variable will access the same instance as the object that was assigned. This behaviour is the same when passing instances to a function. A copy of an already created object can be made by [cloning](http://www.php.net/manual/en/language.oop5.cloning.php) it.

**Example #4 Object Assignment**

<?php  
  
$instance = new SimpleClass();  
  
$assigned   =  $instance;  
$reference  =& $instance;  
  
$instance->var = '$assigned will have this value';  
  
$instance = null; // $instance and $reference become null  
  
var\_dump($instance);  
var\_dump($reference);  
var\_dump($assigned);  
?>

The above example will output:

NULL

NULL

object(SimpleClass)#1 (1) {

["var"]=>

string(30) "$assigned will have this value"

}

PHP 5.3.0 introduced a couple of new ways to create instances of an object:

**Example #5 Creating new objects**

<?php  
class Test  
{  
    static public function getNew()  
    {  
        return new static;  
    }  
}  
  
class Child extends Test  
{}  
  
$obj1 = new Test();  
$obj2 = new $obj1;  
var\_dump($obj1 !== $obj2);  
  
$obj3 = Test::getNew();  
var\_dump($obj3 instanceof Test);  
  
$obj4 = Child::getNew();  
var\_dump($obj4 instanceof Child);  
?>

The above example will output:

bool(true)

bool(true)

bool(true)

**extends**

A class can inherit the methods and properties of another class by using the keyword *extends* in the class declaration. It is not possible to extend multiple classes; a class can only inherit from one base class.

The inherited methods and properties can be overridden by redeclaring them with the same name defined in the parent class. However, if the parent class has defined a method as [final](http://www.php.net/manual/en/language.oop5.final.php), that method may not be overridden. It is possible to access the overridden methods or static properties by referencing them with [parent::](http://www.php.net/manual/en/language.oop5.paamayim-nekudotayim.php).

When overriding methods, the parameter signature should remain the same or PHP will generate an **E\_STRICT** level error. This does not apply to the constructor, which allows overriding with different parameters.

**Example #6 Simple Class Inheritance**

<?php  
class ExtendClass extends SimpleClass  
{  
    // Redefine the parent method  
    function displayVar()  
    {  
        echo "Extending class\n";  
        parent::displayVar();  
    }  
}  
  
$extended = new ExtendClass();  
$extended->displayVar();  
?>

The above example will output:

Extending class

a default value

**Properties**

Class member variables are called "properties". You may also see them referred to using other terms such as "attributes" or "fields", but for the purposes of this reference we will use "properties". They are defined by using one of the keywords *public*, *protected*, or *private*, followed by a normal variable declaration. This declaration may include an initialization, but this initialization must be a constant value--that is, it must be able to be evaluated at compile time and must not depend on run-time information in order to be evaluated.

See [Visibility](http://www.php.net/manual/en/language.oop5.visibility.php) for more information on the meanings of *public*, *protected*, and *private*.

**Note**:

In order to maintain backward compatibility with PHP 4, PHP 5 will still accept the use of the keyword *var* in property declarations instead of (or in addition to) *public*, *protected*, or *private*. However, *var* is no longer required. In versions of PHP from 5.0 to 5.1.3, the use of *var* was considered deprecated and would issue an **E\_STRICT** warning, but since PHP 5.1.3 it is no longer deprecated and does not issue the warning.

If you declare a property using *var* instead of one of *public*, *protected*, or *private*, then PHP 5 will treat the property as if it had been declared as *public*.

Within class methods non-static properties may be accessed by using *->* (Object Operator): *$this->property* (where *property* is the name of the property). Static properties are accessed by using the *::* (Double Colon): *self::$property*. See [Static Keyword](http://www.php.net/manual/en/language.oop5.static.php) for more information on the difference between static and non-static properties.

The pseudo-variable *$this* is available inside any class method when that method is called from within an object context. *$this* is a reference to the calling object (usually the object to which the method belongs, but possibly another object, if the method is called [statically](http://www.php.net/manual/en/language.oop5.static.php) from the context of a secondary object).

**Example #1 property declarations**

<?php  
class SimpleClass  
{  
   // invalid property declarations:  
   public $var1 = 'hello ' . 'world';  
   public $var2 = <<<EOD  
hello world  
EOD;  
   public $var3 = 1+2;  
   public $var4 = self::myStaticMethod();  
   public $var5 = $myVar;  
  
   // valid property declarations:  
   public $var6 = myConstant;  
   public $var7 = array(true, false);  
  
   // This is allowed only in PHP 5.3.0 and later.  
   public $var8 = <<<'EOD'  
hello world  
EOD;  
}  
?>

**Autoloading Classes**

Many developers writing object-oriented applications create one PHP source file per class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).

In PHP 5, this is no longer necessary. You may define an [\_\_autoload()](http://www.php.net/manual/en/function.autoload.php) function which is automatically called in case you are trying to use a class/interface which hasn't been defined yet. By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error.

**Tip**

[spl\_autoload\_register()](http://www.php.net/manual/en/function.spl-autoload-register.php) provides a more flexible alternative for autoloading classes. For this reason, using [\_\_autoload()](http://www.php.net/manual/en/function.autoload.php) is discouraged and may be deprecated or removed in the future.

**Note**:

Prior to 5.3.0, exceptions thrown in the \_\_autoload function could not be caught in the [catch](http://www.php.net/manual/en/language.exceptions.php) block and would result in a fatal error. From 5.3.0+ exceptions thrown in the \_\_autoload function can be caught in the [catch](http://www.php.net/manual/en/language.exceptions.php) block, with 1 provision. If throwing a custom exception, then the custom exception class must be available. The \_\_autoload function may be used recursively to autoload the custom exception class.

**Note**:

Autoloading is not available if using PHP in CLI [interactive mode](http://www.php.net/manual/en/features.commandline.php).

**Note**:

If the class name is used e.g. in [call\_user\_func()](http://www.php.net/manual/en/function.call-user-func.php) then it can contain some dangerous characters such as *../*. It is recommended to not use the user-input in such functions or at least verify the input in [\_\_autoload()](http://www.php.net/manual/en/function.autoload.php).

**Example #1 Autoload example**

This example attempts to load the classes *MyClass1* and *MyClass2* from the files *MyClass1.php* and *MyClass2.php* respectively.

<?php  
function \_\_autoload($class\_name) {  
    include $class\_name . '.php';  
}  
  
$obj  = new MyClass1();  
$obj2 = new MyClass2();   
?>

**Example #2 Autoload other example**

This example attempts to load the interface *ITest*.

<?php  
  
function \_\_autoload($name) {  
    var\_dump($name);  
}  
  
class Foo implements ITest {  
}  
  
/\*  
string(5) "ITest"  
  
Fatal error: Interface 'ITest' not found in ...  
\*/  
?>

**Constructors and Destructors**

**Constructor**

void **\_\_construct** ([ [mixed](http://www.php.net/manual/en/language.pseudo-types.php#language.types.mixed) $args [, $... ]] )

PHP 5 allows developers to declare constructor methods for classes. Classes which have a constructor method call this method on each newly-created object, so it is suitable for any initialization that the object may need before it is used.

**Note**: Parent constructors are not called implicitly if the child class defines a constructor. In order to run a parent constructor, a call to **parent::\_\_construct()** within the child constructor is required. If the child does not define a constructor then it may be inherited from the parent class just like a normal class method (if it was not declared as private).

**Example #1 using new unified constructors**

<?php  
class BaseClass {  
   function \_\_construct() {  
       print "In BaseClass constructor\n";  
   }  
}  
  
class SubClass extends BaseClass {  
   function \_\_construct() {  
       parent::\_\_construct();  
       print "In SubClass constructor\n";  
   }  
}  
  
class OtherSubClass extends BaseClass {  
    // inherits BaseClass's constructor  
}  
  
// In BaseClass constructor  
$obj = new BaseClass();  
  
// In BaseClass constructor  
// In SubClass constructor  
$obj = new SubClass();  
  
// In BaseClass constructor  
$obj = new OtherSubClass();  
?>

For backwards compatibility, if PHP 5 cannot find a [\_\_construct()](http://www.php.net/manual/en/language.oop5.decon.php#object.construct) function for a given class, and the class did not inherit one from a parent class, it will search for the old-style constructor function, by the name of the class. Effectively, it means that the only case that would have compatibility issues is if the class had a method named [\_\_construct()](http://www.php.net/manual/en/language.oop5.decon.php#object.construct) which was used for different semantics.

Unlike with other methods, PHP will not generate an **E\_STRICT** level error message when [\_\_construct()](http://www.php.net/manual/en/language.oop5.decon.php#object.construct) is overridden with different parameters than the parent [\_\_construct()](http://www.php.net/manual/en/language.oop5.decon.php#object.construct) method has.