**OOP’s concepts :- \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Constructor Functions

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

$this->title = $par1;

$this->price = $par2;

}

Destructor Functions

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

}

Inheritance

class Child extends Parent {

<definition body>

}

Public Members

* 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

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.

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.

Interfaces

Interfaces are defined to provide a common function names to the implementers. 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 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 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 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.