Object oriented programming

Object oriented programming is nothing but a technique to design your application. Application could be any type like it could be web based application, windows based application. OOP is a design concept. In object oriented programming, everything will be around the objects and class. By using OOP in php you can create modular web application. By using OOP in php we can perform any activity in the object model structure.

What is Object?

Any thing is the world is an object. Look around and you can find lots of object. Your laptop, pc, car every thing is an object. In this world every object has two thing *properties* and *behaviors*.

Your car has property (color, brand name) and behavior(it can go forward and backward). If you are able to find properties and behaviors of real object. Then it will be very easy for you to work with Object Oriented Programming.
In real world different objects has different properties and behaviors. For example your television has property size, color, and has behavior turn on, turn off. If you observe carefully then you can find that every object has some property and behavior from other object.

What is Class ?

Class is something which defines your object. For example your class is Car. And your Honda car is object of car class. Like object explanation, here we will take an example of the real word and then we will move further in programming definition.

Blueprint of the object is class. Class represents all properties and behaviors of object. For example your car class will define that car should have color, number of door andyour car which is an object will have color green and 2 doors. Your car is object of class car. Or in terms of programming we can say your car object is an instance of the car class. So structural representation (blueprint) of your object is class.

Advantage of Object Oriented Programming

There are various advantage of using OOP over the procedural or parallel programming. Following are some of the basic advantages of using oop techniques.

Re-Usability of your code: If you will use OOP technique for creating your application then it will gives you a greater re-usability. For example, if you have created database class at one place then you can use the same database class in your application.

Easy to Maintain : Application develop using oop technique are easier to maintain than normal programming. Again let us take an example of database class. Suppose you need to change the username and password. If we change username and password using connection class , this will change all part of database connection .

Good Level of Abstraction: Abstraction means making something hidden. By using oop technique you are abstracting your business logic from implementation. It will provide you greater ease. For example using oop you will only concern about creating object of database not more details about it.

Creating class and object in php:

we can use class(small letter) keyword to define class in php

syntax:

```
class className {

        //Properties

        //Methods

}
```

Suppose take an example:

So let us create a class for interest calculator and define its properties like rate, capital, duration and behavior like calculate interest.

```
class interestCalculator
{
var $rate;
var $duration;
var $capital;
function calculateInterest()
{
return ($this->rate*$this->duration*$->capital)/100;
}
}
```

Object in PHP:

As we have already discussed that object is an instance of any class. So we will take our interestCalculator class as an example. Creating object of the class is very easy in php. You can create object of class with the help of new keyword. Following is very basic example of creation of object of your class interest calculator:

```
$calculator = new interestCalculator()
```

In above declaration you are creating object of your class interestCalculator in variable $calculator. Now your variable $calculator is an object of class interestCalculator. Next step is to set property or variable of object calculator and perform calculation of interest.

```
$calculator = new InterestCalculator()
$calculator->rate = 3;
$calculator->duration =2;
$calculator->capital = 300;
echo $calculator->calculateInterest();
```

Here object of your class interestCalculator is your php variable $calculator. In next 3 lines of above code you are setting properties of class. You can access property of class with ->. So in above code rate property is set using $calculator->rate = 3;.

```
//Creating class interestCalculator
class interestCalculator
{
public $rate;
public $duration;
public $capital;
public function calculateInterest()
```

```
{
return ($this->rate*$this->duration*$this->capital)/100;
}
}
//Creating various object of class interestCalculator to calculate interest on various amount
$calculator1 = new InterestCalculator();
$calculator2 = new InterestCalculator();
$calculator1->rate = 3;
$calculator1->duration =2;
$calculator1->capital = 300;
$calculator2->rate = 3.2;
$calculator2->duration =3;
$calculator2->capital = 400;
$interest1 = $calculator1->calculateInterest();
$interest2 = $calculator2->calculateInterest();
echo "Your interest on capital $calculator1->capital with rate $calculator1->rate for
duration $calculator1->duration is $interest1 <br/> ";
echo "Your interest on capital $calculator2->capital with rate $calculator2->rate for
duration $calculator2->duration is $interest2 <br/> ";
```

$this keyword(variable)

$this is system defined object variable of the class. $this is object of self class in the current context. For the both object of interestCalculator class $this object is different.

Constructor of Classes and Objects

Constructor is nothing but a function defined in your php class. Constructor function automatically called when you will create object of the class. As soon as you will write $object = new yourClass() your constructor function of the class will be executed. You can't create constructor private.

you can also create constructor by defining magic function __construct

```
class interestCalculator {
```

```
        public $rate;
        public $duration;
        public $capital;
        //Constructor of the class
        public function __construct() {
                $this->rate = 3;
                $this->duration = 4;
        }
}
```

Restricting access to properties using 'access modifiers'

One of the fundamental principles in OOP is 'encapsulation'. The idea is that you create cleaner better code, if you restrict access to the data structures (properties) in your objects. You restrict access to class properties using something called 'access modifiers'.

There are 3 access modifiers:

- public
- private
- protected

Public is the default modifier.

when you declare a property as 'private', only the same class can access the property.

When a property is declared 'protected', only the same class and classes derived from that class can access the property - this has to do with inheritance ...more on that later.

Properties declared as 'public' have no access restrictions, meaning anyone can access them.

Reusing code the OOP way: inheritance

Inheritance is a fundamental capability/construct in OOP where you can use one class, as

the base/basis for another class ... or many other classes.

What is inheritance?

Inheritance is nothing but a design principle in oop. By implementing inheritance you can inherit(or get) all properties and methods of one class to another class. The class who inherit feature of another class known as **child class**. The class which is being inherited is know as **parent class**. Concept of the inheritance in oop is same as inheritance in real world. For example, child inherits characteristics of their parent. Same is here in oop. One class is inheriting characteristics of another class.

Syntax :

```
class ParentClass {
        // properties and methods here

}

class ChildClass extends ParentClass {
  // additional properties and methods here
}
```

Here we've created a class, ParentClass, then created another class,ChildClass, that inherits from ParentClass. ChildClass inherits all the properties and methods of ParentClass, and it can also add its own properties and methods.

Creating a parent object

Let's say that our intention is to create a storefront where we'll be selling cars. A simple shopping cart allows us to sell the cars and allows consumers to browse the products, and access specific information such as pricing and a description. With this in mind we know that regardless of what we're selling, all products have certain things in common, such as a name, description, price and photo. By inheriting the Product object we can share these common properties across unique child objects.

```
<?php
class Product {
                private $name;
        private $price;
        private $photo;
        private $description;

        public function Product() {}
```

```
        protected function setName($name) { $this-?>name = $name; }
        public function GetName() { return $this->name; }

        protected function setPrice($price = '0.00') { $this->price = $price; }
        public function GetPrice() { return $this->price; }

        protected function setPhoto($photo) { $this->photo = $photo; }
        public function GetPhoto() { return $this->photo; }

        protected function setDescription($description) { $this->description =
            $description;
      }
        public function GetDescription() { return $this->description; }

  }

?>
```

The Product object is quite simple, it allows us to define specific information such as a name, price, photo and description. The objects setters allow us to set these specific product properties, and by using the protected keyword we only allow these properties to be set by extending child objects. Also, the Products getters are public, which allows us to obtain specific data from an object with any PHP page that includes and instantiates it. The last thing to note about this object are the properties defined at the beginning of the class. We define these as private properties so they can't be changed directly unless using the setters provided by the Product object, this provides more security and reliability. Let's look at how to inherit the Product object.

Inheriting the parent object

Inheriting a parent object is probably one of the easiest concepts in object-oriented programming. All we need to do to inherit the Product object is add the extends keyword after the class name, followed by the object that we want to inherit, in this case the Product object.

```
<?php class Car extends Product

        {
            private $model;

            public function Car($name, $model, $price, $photo, $description)
            {
                parent::setName($name);
                $this->setModel($model);
                parent::setPrice($price);
                parent::setPhoto($photo);
                parent::setDescription($description);
            }
```

```
        private function setModel($model) { $this->model = $model; }
        public function GetModel() { return $this->model; }

    }

    ?>
```

By taking a look at the car object we see how simple it's become by creating the parent to handle all of the product details. Now we can use this object for custom properties that a car includes, which are not common to all products. As an example, we add a model property to specify the model of the car. Another benefit is being able to scale this application by adding new products that inherit the Product object or we could add more common properties to the Product object. The addition of properties will enhance all child objects without interfering with their existing ties.

One more example :

For example, say a web forum application has a Member class for forum members, containing methods such as createPost(), editProfile(),showProfile(), and so on. Since forum administrators are also members, you can also create a class called Administrator that is a child of the Member class. The Administrator class then inherits all of the properties and methods of the Member class, so an Administrator object behaves just like a Member object.

Then, you can add administrator-specific functionality to the Administrator class by adding extra methods such as createForum(), deleteForm() andbanMember(). Similarly, if you want different privilege levels for administrators then you can add an $admin Level property to the Administrator class.

In this way, you don't clutter up your Member class with administrator-specific methods that aren't appropriate for regular member. You also avoid having to copy and paste the Member class's properties and methods into the Administrator class. So inheritance gives you the best of both worlds.

Class for administrator and member :

```
class Member {

  public $username = "";
  private $loggedIn = false;

  public function login() {
```

```
    $this->loggedIn = true;
  }

  public function logout() {
    $this->loggedIn = false;
  }

  public function isLoggedIn() {
    return $this->loggedIn;
  }
}

// Administrator Class

class Administrator extends Member {

  public function createForum( $forumName ) {
    echo "$this->username created a new forum: $forumName<br>";
  }

  public function banMember( $member ) {
    echo "$this->username banned the member: $member->username<br>";
  }

}
```

As you can see, our Member class contains a public $username property, a private $loggedIn property, methods to log the member in and out, and a method to determine whether the member is logged in or not.

We then add an Administrator class as a child of the Member class.Administrator inherits all the properties and methods of the Member class. We also add a couple of extra admin-specific methods to the Administrator class:

```
// Create a new member and log them in
$member = new Member();
$member->username = "Fred";
$member->login();
echo $member->username . " is " . ( $member->isLoggedIn() ? "logged in" : "logged out"
) . "<br>";

// Create a new administrator and log them in
$admin = new Administrator();
$admin->username = "Mary";
$admin->login();
echo $admin->username . " is " . ( $member->isLoggedIn() ? "logged in" : "logged out" ) .
"<br>";
```

```
// Displays "Mary created a new forum: Teddy Bears"
$admin->createForum( "Teddy Bears" );

// Displays "Mary banned the member: Fred"
$admin->banMember( $member );
```

how the code works:

1.First we create a new Member object, give it a username of "Fred", log the member in, and display his logged-in status.
2.Then we create a new Administrator object. Since Administrator inherits from Member, we can use all the same properties and methods that we used for the Member object. We give the admin a username of "Mary", log her in, and display her logged-in status.
3.Now we call the admin's createForum() method, passing in the name of the forum to create ("Teddy Bears").
4.Finally we call the admin's banMember() method, passing in the member to ban (Fred).

Overriding parent class methods

As you've seen, when you create a child class, that class inherits all of the properties and methods of its parent. However, sometimes you might want an inherited method in a child class to behave differently to its parent's method.

When an administrator logs into the forum, you log them in just like a regular member, but you might also want to record the login event in a log file for security purposes.

By ***overriding*** the login() method in the Administrator class, you can redefine the method so that it also records the login event.

To override a parent class's method in a child class, you simply create a method in the child class with the same name as the parent class's method. Then, whenever the method is called for objects of the child class, PHP runs the child class's method instead of the parent class's method:

```
class Member {

  public $username = "";
  private $loggedIn = false;

  public function login() {
```

```
    $this->loggedIn = true;
  }

  public function logout() {
    $this->loggedIn = false;
  }
}

class Administrator extends Member {

  public function login() {
    $this->loggedIn = true;
    echo "Log entry: $this->username logged in<br>";
  }

}

// Create a new member and log them in
$member = new Member();
$member->username = "Fred";
$member->login();
$member->logout();

// Create a new administrator and log them in
$admin = new Administrator();
$admin->username = "Mary";
$admin->login();          // Displays "Log entry: Mary logged in"
$admin->logout();
```

we've redefined login() inside the Administrator class to display a mocked-up log entry, indicating that the admin has logged in.

We then create a regular Member object ("Fred"), and an Administrator object ("Mary"). When we call Fred's login() method, PHP calls Member::login()as usual. However, when we call Mary's login() method, PHP notices that we've overridden login() in the Administrator class, so it calls Administrator::login() instead. This displays "Log entry: Mary logged in" in the page.

On the other hand, since we haven't overridden the logout() method in theAdministrator class, Member:logout() is called for both the Member object and the Administrator object.

Calling a parent method from a child method

When you override a parent class's method in a child class, you don't always want to redefine the method entirely. Often, you still want to use the functionality of the parent

method, and merely add additional functionality in the child method.

To access a parent class's method from within a child class's method, you use theparent keyword, like this:

```
parent::myMethod();
```

For instance, in the code example in the previous section, we overrode the Memberclass's login() method in the Administrator class to display a log entry. However, we also duplicated the functionality of Member::login() insideAdministrator::login() when we set $this->loggedIn to true:

```
class Administrator extends Member {

  public function login() {
    parent::login();
    echo "Log entry: $this->username logged in<br>";
  }
}
```

Not only is this neater, but it's more future-proof. If, at a later date, you want to change the way members are logged in, you only have to change the code inMember::login(), and Administrator::login() will automatically run the new code.

Preventing inheritance with final methods and classes

Most of the time, allowing your classes to be extended using inheritance is a good thing. It's part of what makes object-oriented programming so powerful.

Occasionally, though, overriding certain methods of a class can cause things to break easily, create security issues, or make the resulting code overly complex. In these situations, you might want to prevent methods within the class — or even the entire class — from being extended.

To prevent a parent class's method from being overridden by any of its child classes, you add the final keyword before the method definition. For example, you might decide to prevent your Member class's login() method from being overridden for security reasons:

```
class Member {

  public $username = "";
```

```
  private $loggedIn = false;

  public final function login() {
    $this->loggedIn = true;
  }

  public function logout() {
    $this->loggedIn = false;
  }
}

class Administrator extends Member {

  public function login() {
    $this->loggedIn = true;
    echo "Log entry: $this->username logged in<br>";
  }


}
```

Now php will generate error when we try to override login method of member class.

We can also prevent class from inheriting from another class.

```
final class Member {
  // This class can't be extended at all

}
```

Working with abstract classes

An ***abstract class*** is a special type of class that can't be instantiated — in other words, you can't create objects from it. Instead, you create child classes from the abstract class, and create objects from those child classes instead. An abstract class is designed to be used as a template for creating classes.

An abstract class contains one or more ***abstract methods***. When you add an abstract method to an abstract class, you don't include any code inside the method. Instead, you leave the implementation of the method to any child classes that inherit from the abstract class.

** The moment you add one or more abstract methods to a class, you must declare that class to be abstract.

When a child class extends an abstract class, the child class must implement all of the abstract methods in the abstract class. (If it doesn't then PHP generates an error.) In this way, the abstract class lays down the rules as to how its children should behave. Any code that uses a child class of the abstract class knows that the child class will implement a given set of methods.

** You can also add regular, non-abstract methods to your abstract class. These methods are then inherited by child classes as normal.

```
abstract class Person {

  private $firstName = "";
  private $lastName = "";

  public function setName( $firstName, $lastName ) {
    $this->firstName = $firstName;
    $this->lastName = $lastName;
  }

  public function getName() {
```

```
   return "$this->firstName $this->lastName";
  }

  abstract public function showWelcomeMessage();
}
```

Now extends abstract class

```
class Member extends Person {

  public function showWelcomeMessage() {
    echo "Hi " . $this->getName() . ", welcome to the forums!<br>";
  }

  public function newTopic( $subject ) {
    echo "Creating new topic: $subject<br>";
  }
}

class Shopper extends Person {

  public function showWelcomeMessage() {
    echo "Hi " . $this->getName() . ", welcome to our online store!<br>";
  }

  public function addToCart( $item ) {
    echo "Adding $item to cart<br>";
  }
}
```

As you can see, each class implements the abstract showWelcomeMessage()method from the Person class. They implement the method differently — Memberdisplays a "welcome to the forums" message, while Shopper displays "welcome to our online store" — but that's OK. The point is that they've fulfilled the obligation laid down by the abstract class to implement the showWelcomeMessage() method.
If one class — say, Shopper — fails to implement showWelcomeMessage()then PHP raises an error:

As well as implementing the abstract method, each class also contains a class-specific method. Member contains a newTopic() method to create new forum topics, while Shopper contains addToCart() for adding items to a shopping cart.

We can now create Member and Shopper objects in our website. As well as calling newTopic() and addToCart() on these objects, we can also callgetName() and setName(), since these methods are inherited from the abstractPerson class.

More importantly, since we know that our Member and Shopper classes are derived from Person, we can happily call the showWelcomeMessage() method on Member and Shopper objects, safe in the knowledge that both classes implement this method. We know that Member and Shopper must implement it because it was declared abstract inside Person.

```
$aMember = new Member();
$aMember->setName( "John", "Smith" );
$aMember->showWelcomeMessage();
$aMember->newTopic( "Teddy bears are great" );

$aShopper = new Shopper();
$aShopper->setName( "Mary", "Jones" );
$aShopper->showWelcomeMessage();
$aShopper->addToCart( "Ornate Table Lamp" );
```

Interfaces:

Interfaces are, in many ways, similar to abstract classes. An interface is a template that defines how one or more classes should behave.
There are several key differences between an abstract class and an interface:

•No methods in an interface can be implemented within the interface. They are all "abstract". (In an abstract class, you can mix abstract and non-abstract methods.)

•An interface can't contain properties, only methods.

•A class implements an interface, whereas a class extends or inherits from an abstract class.

•A class can implement more than one interface at the same time. (That same class can also extend a parent class.) In contrast, a class can only be derived from one parent class (abstract or otherwise).

As with an abstract class, an interface declares one or more methods that must be implemented by any class that implements the interface. The syntax looks like this:

```
interface MyInterface {
      public function aMethod();
      public function anotherMethod();
    }
```

To create a class that implements an interface

```
class MyClass implements MyInterface {

        public function aMethod() {
          // (code to implement the method)
        }

        public function anotherMethod() {
          // (code to implement the method)
        }

      }
```

Interfaces are useful when you want to create several otherwise unrelated classes that need to implement a common set of features.
For example, a web forum might well contain a Member class for forum members, and a Topic class to store topics that members create in the forum. Inheritance-wise, these classes will likely be unrelated, since they perform quite different functions.
However, let's say that we want to be able to save and retrieve both Member andTopic objects to and from a MySQL database. To achieve this, we can create an interface called Persistable that specifies the methods required for objects to persist in a database:

```
interface Persistable {

  public function save();
  public function load();
  public function delete();

}
```

Now let's create our Member class, and make it implement the Persistableinterface. This means that the class must provide implementations for save(),load() and delete():

```
class Member implements Persistable {

  private $username;
  private $location;
```

```
  private $homepage;

  public function __construct( $username, $location, $homepage ) {
    $this->username = $username;
    $this->location = $location;
    $this->homepage = $homepage;
  }

  public function getUsername() {
    return $this->username;
  }

  public function getLocation() {
    return $this->location;
  }

  public function getHomepage() {
    return $this->homepage;
  }

  public function save() {
    echo "Saving member to database<br>";
  }

  public function load() {
    echo "Loading member from database<br>";
  }

  public function delete () {
    echo "Deleting member from database<br>";
  }


}
```

Similarly, our Topic class also implements Persistable, so it needs to includesave(), load() and delete() methods too:

```
class Topic implements Persistable {

  private $subject;
  private $author;
  private $createdTime;

  public function __construct( $subject, $author ) {
    $this->subject = $subject;
```

```
    $this->author = $author;
    $this->createdTime = time();
  }

  public function showHeader() {
    $createdTimeString = date( 'l jS M Y h:i:s A', $this->createdTime );
    $authorName = $this->author->getUsername();
    echo "$this->subject (created on $createdTimeString by $authorName)<br>";
  }

  public function save() {
    echo "Saving topic to database<br>";
  }

  public function load() {
    echo "Loading topic from database<br>";
  }

  public function delete () {
    echo "Deleting topic from database<br>";
  }

}
```

We've also included some class-specific methods, such asMember::getUsername() to return the member's username, andTopic::showHeader() to display the topic's subject, author and creation time.

Now we can create Member and Topic objects and call their getUsername()and showHeader() methods respectively. What's more, since we know that both classes implement the Persistable interface, we can also call methods such assave(), load() and delete() on the objects, knowing that the objects must implement those methods:

```
$aMember = new Member( "fred", "Chicago", "http://example.com/" );
echo $aMember->getUsername() . " lives in " . $aMember->getLocation() ."<br>";
$aMember->save();

$aTopic = new Topic( "Teddy Bears are Great", $aMember );
$aTopic->showHeader();
$aTopic->save();
```

As you can see, an interface lets you bring together otherwise unrelated classes in order to

use them for a specific purpose, such as storing the objects in a database. Remember also that one class can implement more than one interface:

```
class MyClass implements anInterface, anotherInterface {
    ...
}
```

Static method and class :

- Static methods and properties in php is very useful feature. Static methods and properties in php can directly accessible without creating object of class.

- Your php class will be static class if your all methods and properties of the class is static. Static Methods and Properties in PHP will be treated as public if no visibility is defined.

- Static properties of class is a property which is directly accessible from class with the help of ::(scope resolution operator).

- You can declare static property using static keyword. In other word you can make any property static by using static keyword.

  following is the basic example of static variable in php class:

```
class test
{
private static $no_of_call = 0;
public function __construct()
{
self::$no_of_call = self::$no_of_call + 1;
echo "No of time object of the class created is: ". self::$no_of_call;
}
}
$objTest = new test(); // Prints No of time object of the class created is 1
```

```
$objTtest2 = new test(); //Prints No of time object of the class created is 2
```

Output:

No of time object of the class created is: 1
No of time object of the class created is: 2

- So creating static variable or property is very useful if you want to share some data between the different object of the same class.

- using static keyword. You can access all visible static methods for you using :: like in static variables.

```
class test
{
static function abc($param1 , $param2)
{
echo "$param1 , $param2";
}
}
test::abc("ankur" , "techflirt");
```

- If you will use regular or normal method statically then you will get E_STRICT warning. In case of variable or property it was throwing fatal. Let us take above example

```
class test
{
function abc($param1 , $param2)
{
echo "$param1 , $param2";
}
}
test::abc("ankur" , "techflirt"); //will work fine with warning.
```

Since static methods is called direct $this variable will not available in the method.

```
<?php
class User {
  public $name;
    public $age;

    public static $minimumPasswordLength = 6;

        public function Describe() {
  return $this->name . " is " . $this->age . " years old";

    }

    public static function ValidatePassword($password) {

          // if(strlen($password) >= $this->$minimumPasswordLength)

        if(strlen($password) >= self::$minimumPasswordLength)
   return true;
   else
  return false;
 }
}
$password = "testhello";
if(User::ValidatePassword($password))
   echo "Password is valid!";
else
echo "Password is NOT valid!";

?>
```

What we have done to the class, is adding a single static variable, the $minimumPasswordLength which we set to 6, and then we have added a static function to validate whether a given password is valid. I admit that the validation being performed here is very limited, but obviously it can be expanded. Now, couldn't we just do this as a regular variable and function on the class? Sure we could, but it simply makes more sense to do this statically, since we don't use information specific to one user - the functionality is general, so there's no need to have to instantiate the class to use it.

As you can see, to access our static variable from our static method, we prefix it with the self keyword, which is like this but for accessing static members and constants. Obviously it only works inside the class, so to call the ValidatePassword() function from outside the class, we use the name of the class. You will also notice that accessing static members require the double-colon operator instead of the -> operator, but other than that, it's basically the same.

Polymorphism :

- Why method polymorphism cannot be achieved?

The reason why polymorphism for methods is not possible in PHP is because you can have a method that accepts two parameters and call it by passing three parameters. This is because PHP is not strict and contains methods like func_num_args() and func_get_arg() to find the number of arguments passed and get a particular parameter.

Because PHP is not type strict and allows variable arguments, this is why method polymorphism is not possible.

Since PHP 5 introduces the concept of Type Hinting, polymorphism is possible with class methods. The basis of polymorphism is Inheritance and overridden methods.

```
class BaseClass {

  public function myMethod() {
    echo "BaseClass method called";
  }
}
```

```
class DerivedClass extends BaseClass {
  public function myMethod() {
    echo "DerivedClass method called";
  }
}

function processClass(BaseClass $c) {
  $c->myMethod();
}
$c = new DerivedClass();
processClass($c);
```

Another Example

```
class lineitem {
        var $amount,$what;
        function lineitem ($amount,$desc) {
                $this->amount = $amount;
                $this->what = $desc;
                }

        function getname () {
                return $this->what;
                }

        function getval () {
                return $this->amount;
                }

        }

class bill extends lineitem {
```

```
        function bill ($amount,$desc) {
                $this->amount = -$amount;
                $this->what = $desc;
                }

        }

class income extends lineitem {

        }

$acc[0] = new bill(15.40,"Break Rolls");
$acc[1] = new bill(17.75,"Carpet leaner");
$acc[2] = new income(37.84,"options added by Berks Folks");

$tot = 0;
$format = "%-35s ... %8.2f\n";

foreach ($acc as $item) {
        $amount = $item->getval();
        $whom = $item->getname();
        printf($format, $whom,$amount);
        $tot += $amount;
        }

printf($format, "TOTAL ...",$tot);
```

```
interface Human {
 public function getName();
 public function setName($name);
}

abstract class Military {
 private $rank;

 public function __construct($rank) {
 $this->rank = $rank;
 }
 public function setRank($rank) {
 $this->rank = $rank;
 }
 public function getRank() {
 return $this->rank;
 }
}
```

```
class Soldier extends Military implements Human {
 private $name;

 public function __construct($name, $rank) {
 $this->name = $name;
 parent::__construct($rank); # parent::setName($rank);
 }
 public function setName($name) {
 $this->name = $name;
 }
 public function getName() {
 return "My name is: " . $this->name . "<br />n";
 }
 public function getRank() {
 return "My rank is: " . parent::getRank() . "<br />n";;
 }
 public function getFull() {
 return "I am " . parent::getRank() . " {$this->name}<br />n";
 }
}
```

```
$goodSoldier = New Soldier('Thomas', 'Officer');


echo $goodSoldier->getName();

echo $goodSoldier->getRank();

echo $goodSoldier->getFull();

echo "<br />n";

$goodSoldier->setRank('Colonel');

$goodSoldier->setName('Mustard');


echo $goodSoldier->getName();

echo $goodSoldier->getRank();

echo $goodSoldier->getFull();
```

PHP Magic methods

__autoload()

```
// code
require_once('lib/mylib/myclass.php');
require_once('lib/mylib/myclass2.php');
require_once('lib/anotherlib/someclass.php');

$myclass = new MyClass;
$myclass2 = new MyClass2;
$class = new SomeClass('value');

// more code
```

While this code will work, it can be somewhat cumbersome to load all those files using 'require_once' (or any other PHP include function). These includes can easily be excluded from your code when using class autoloading.

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 function which is automatically called in case you are trying to use a class/interface which hasn't been defined yet.

This is how it works in action. We will create two classes. So create Image.php file and paste this in:

```
<?php
  class Image {

     function __construct() {
        echo 'Class Image loaded successfully <br />';
     }

  }
?>
```

```
<?php
  class Test {

     function __construct() {
        echo 'Class Test working <br />';
     }

  }
?>
```

```
<?php
  function __autoload($class_name) {
     require_once $class_name . '.php';
  }

  $a = new Test();
  $b = new Image();
?>
```

__destruct()

As the name implies, the `__destruct()` method is called when the object is destroyed. It accepts no arguments and is commonly used to perform any cleanup operations such as closing a database connection. In our case, we'll use it to disconnect from the network.

```
class Device {
   //...
   public function  __destruct() {
      // disconnect from the network
      $this->disconnect();
      echo $this->name . ' was destroyed' . PHP_EOL;
   }
   //...
}
```

With the above destructor in place, here is what happens when a Device object is destroyed:

```
$device = new Device(new Battery(), 'iMagic');
// iMagic connected
unset($device);
```

// iMagic disconnected
// iMagic was destroyed

Before it is destroyed, the destructor calls the `disconnect()` method and prints a message,

Destroying Objects:

So far our objects have been automatically destroyed at the end of the script they were created in, thanks to PHP's automatic garbage collection. However, you will almost certainly want to arbitrarily delete objects at some point in time, and this is accomplished using unset() in the same way as you would delete an ordinary variable.

Calling unset() on an object will call its destructor before deleting the object, as you would expect.

Best Practice of Classes and Objects

Following are some best practice of using classes and objects in your application.

1.Instead of assigning variable of the classes after creating object it is good if you use constructor.

2.Use visibility as required. Do not make your variable and method either more secure or completely open. Over security will effect your flexibility, under security will distrust your structure.

3.Follow some convention in your classes and objects. Like start all public method with camel case, all protected method and varaible prefix with _ etc. It will give you better visibility.

4.Do not try to do every thing in single class. Create class very specific to your requirement. It will same your time and execution.

5.Always try to create every class in separate file and follow some naming convention.