Name : Mayur Purushvani

Object-oriented programming is a programming style in which it is customary to group all of the variables and functions of a particular topic into a single class.

For the example given below, we are going to create a Car class into which we will group all of the code which has something to do with cars.

```
class Car {
  // The code
}
```

Properties :

```
Class Car {

        Public $comp;

        Public $color;

}
```

Objects :

```
$bmw = new Car();
```

Usage of object :

We can access the propery and method of class with objects.

```
$bmw = color = 'blue';
```

Methods :

The classes most often contain functions. A function inside a class is called a method. Here we add the method hello() to the class with the prefix public.

```
class Car {

  public $comp;
  public $color = 'beige';
  public $hasSunRoof = true;

  public function hello()
  {
    return "beep";
  }
}


$bmw = new Car();

$bmw->hello();
```

$this keyword :

The $this keyword indicates that we use the class's own methods and properties, and allows us to have access to them within the class's scope.

```
$this->propertyname;

$this->methodname;
```

Example :

```
class Car {

    public $comp;

    public $color = 'beige';

    public $hasSunRoof = true;

    public function hello() {

    return "Beep I am a <i>" . $this -> comp . "</i>, and I am <i>" .$this -> color;

    }

}
```

Chaining methods and properties :

when a class's methods return the $this keyword, they can be chained together to create much more streaming code.

Example :

The fill() method adds gallons of fuel to our car's tank.
The ride() method calculates how much fuel we consume when we ride a certain distance, and then subtracts it from the tank. In our example, we assume that the car consumes 1 gallon of fuel every 50 miles.

```
class Car {
  public $tank;
  public function fill($float)
  {
    $this-> tank += $float;
    return $this;
  }
    public function ride($float)
  {
    $miles = $float;
    $gallons = $miles/50;
    $this-> tank -= ($gallons);
    return $this;
  }
}
$bmw = new Car();
$tank = $bmw -> fill(10) -> ride(40) -> tank;
echo "The number of gallons left in the tank: " . $tank . " gal.";
```

Access Modifiers :

Public :

```
<?php

class Car {
  public $model;
  public function getModel()
  {
    return "The car model is " . $this -> model;
  }
}

$mercedes = new Car();
$mercedes -> model = "Mercedes";
echo $mercedes -> getModel();
```

Private :

It gives an fatel error.

```
<?php

class Car {
  private $model;
  public function getModel()
  {
    return "The car model is " . $this -> model;
  }
}
$mercedes = new Car();
$mercedes -> model = "Mercedes benz";
echo $mercedes -> getModel();
?>
```

How to access a private properties of class :

Setters that set the values of the private properties.

Getters that get the values of the private properties.

```
<?php
class Car {
private $model;
public function setModel($model)
  {
    $this -> model = $model;
  }
  public function getModel()
  {
    return "The car model is  " . $this -> model;
  }
}
$mercedes = new Car();
$mercedes -> setModel("Mercedes benz");
echo $mercedes -> getModel();
?>
```

Magic methods :

The __construct() magic method :

Example :

```
class Car {

  private $model = '';

  public function __construct($model = null)

  {
    if($model)
    {
      $this -> model = $model;
    }
  }
  public function getCarModel()
  {
    return ' The car model is: ' . $this -> model;
  }
}
$car1 = new Car('Mercedes');
echo $car1 -> getCarModel();
```

Magic constraints :

__CLASS__ (magic constants are written in uppercase letters and prefixed and suffixed with two underlines).
__LINE__ to get the line number in which the constant is used.
__FILE__ to get the full path or the filename in which the constant is used.
__METHOD__ to get the name of the method in which the constant is used.

Example :

```
class Car {
  private $model = '';
  public function __construct($model = null)
  {
    if($model)
    {
      $this -> model = $model;
    }
  }
  public function getCarModel()
  {
    return " The <b>" . __CLASS__ . "</b> model is: " . $this -> model;
  }
}
$car1 = new Car('Mercedes');
echo $car1 -> getCarModel();
```

Inheritance :

Inheritance allows us to write the code only once in the parent, and then use the code in both the parent and the child classes.

Example :

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

  public function getModel()
  {
    return $this -> model;
  }
}
class SportsCar extends Car{

  private $style = 'fast and furious';

  public function driveItWithStyle()
  {
    return 'Drive a '  . $this -> getModel() . ' <i>' . $this -> style . '</i>';
  }
}
$sportsCar1 = new SportsCar();
$sportsCar1 -> setModel('Ferrari');
echo $sportsCar1 -> driveItWithStyle();
```

Abstract classes and methods :

We use abstract classes and methods when we need to commit the child classes to certain methods that they inherit from the parent class but we cannot commit about the code that should be written inside the methods.

We can use ‘abstract’ keyword.
We can also have a non-abstract methods inside the abstract class.

Example :

```
abstract class Car {
  protected $tankVolume;
  public function setTankVolume($volume)
  {
    $this -> tankVolume = $volume;
  }
  abstract public function calcNumMilesOnFullTank();
}
```

Interface :

An interface commits its child classes to abstract methods that they should implement.
To implement an interface in our class, use 'implements' keyword.
We can also implements a number of instances in a single class.

```
interface interfaceName {
  // abstract methods
}

class Child implements interfaceName {
  // defines the interface methods and may have its own code
}
```

The difference between abstract class and interface :

	interface	abstract class
the code	- abstract methods - constants	- abstract methods - constants - concrete methods - concrete variables
access modifiers	- public	- public - protected
number of parents	The same class can implement more than 1 interface	The child class can inherit only from 1 abstract class

Polymorphism :

According to the Polymorphism principle, methods in different classes that do similar things should have the same name.

For example, we can call the method that calculates the area calcArea() and decide that we put, in each class that represents a shape, a method with this name that calculates the area according to the shape. Now, whenever we would want to calculate the area for the different shapes, we would call a method with the name of calcArea() without having to pay too much attention to the technicalities of how to actually calculate the area for the different shapes. The only thing that we would need to know is the name of the method that calculates the area.
In order to implement the polymorphism principle, we can choose between abstract classes and interfaces.

Example:

```
interface Shape {
  public function calcArea();
}
class Circle implements Shape {
  private $radius;

  public function __construct($radius)
  {
    $this -> radius = $radius;
  }
  public function calcArea()
  {
    return $this -> radius * $this -> radius * pi();
  }
}
```