PHP | Day 4 | Pre-Work

Day 4 | Php

Table of contents:

0:00 / 16:30

To view this video please enable JavaScript, and consider upgrading to a newer web browser

Object-oriented vs procedural PHP

Initially, PHP was not designed as an object-oriented language. Extensive object-oriented features has been added over time.
Object-oriented programming traces its roots back to the 1960s, when computer programmers realized that increasingly complex programs are hard to maintain.

Programs sent a series of instructions to the computer to be processed **sequentially** (one command after another). Occasionally, a programmer will group parts of the code in functions and call that function several times (and perhaps with different arguments).

That resembles the way PHP is usually written. This approach - known as **procedural programming** - works well for short, simple scripts. But once you expand your code to more than just a few thousands lines of code, it becomes increasingly difficult to spot mistakes. If you make a change to the part of the program’s logic, you need to ensure that you do not encounter any unwanted side-effects.

The answer was to **break up long, procedural code into discrete units that are logically separated.** In many ways, this process is similar to the creation of custom functions. However, OOP takes things a step further by removing all functions from the main script, and grouping them in specialized units called **classes**.

The classes are often used to hide the implementation details (like for instance how you are actually saving data into a database), and presents to the rest of your code just the data and functions that are needed (like for instance a method $user1->save_to_database();

Once you are sure that save_to_database() is working (through testing), you can tell your coworkers to use that function/method on user objects, and they do not need to worry about implementation details (like creating the actual SQL statement).

The approach taken by OOP has two distinct advantages, namely:

1. **Code reusability**: Breaking down complex tasks into generic modules makes it much easier to reuse code. Class files are normally separated from the main script in their own files, so they can be quickly deployed in different projects.

2. **Easier maintenance and reliability:** Concentrating on generic tasks means each method defined in a class normally handles a single task. This makes it easier to identify and eliminate errors. The modular nature of code stored outside the main script means that, if a problem does arise, you fix it in just one place. Once a class has been thoroughly tried and tested, you can treat it like a black box, and rely on it to produce consistent results. This makes developing complex projects in teams a lot easier. Individual developers don't need to concern themselves with internal affairs of a particular unit; all that matters is that it produces the expected result.

Key Terminology

1. **Functions** defined in a class are called **methods**
2. **Variables** defined within a class are called **properties**
3. An **object is instantiated,** that is the action of **creating an object from a class**.
4. **Class defines what is contained in an object** (methods and properties)
5. A Class can **inherit properties and methods** from another class (also known as **parent** or **super-class**)
6. An **abstract class** can be inherited by another class, but not instantiated (aka you can not create an object directly from an abstract class)

7. An **interface** can contain constants and method definitions that must be implemented by a class (in the case we declare the class that way)

Control Access to properties and methods

Contrary to plain JavaScript, when you define the methods and the properties of a class, you can control the access to those methods and properties through the **scope:**

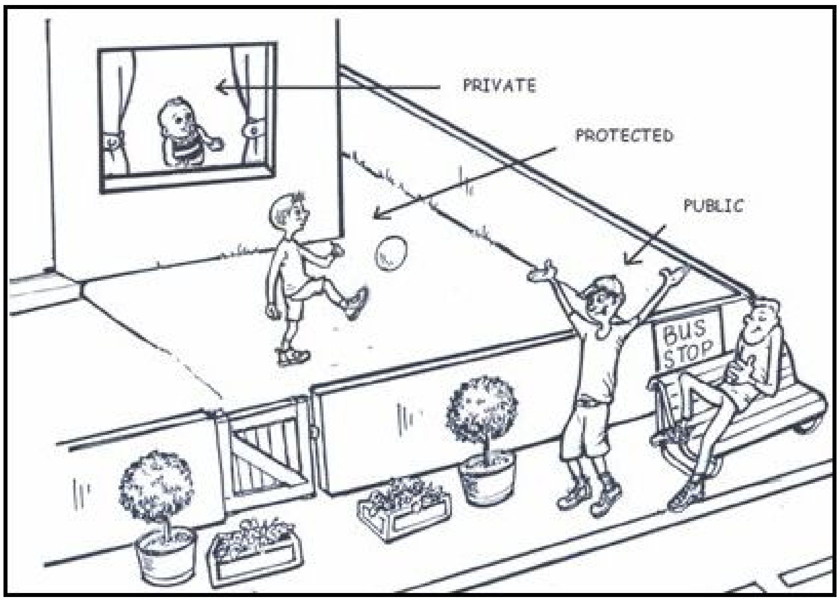


Image taken from mvark.blogspot.in

PHP has different access control levels, defined through keywords **private**, **public**, and **protected** (among others):

- **private scope** when you want your variable or method to be visible in its own class only.
- **public scope** to make that variable or method available from anywhere, other classes and instances of the object.
- **protected scope** when you want to make your variable or method visible in all classes that extend current class including the parent class.

In real-life practice, **private** and **public** are mostly used. If you want to dive further into the subject, start here: http://php.net/manual/en/language.oop5.visibility.php https://stackoverflow.com/questions/4361553/what-is-the-difference-between-public-private-and-protected

Procedural vs. Object Oriented Programming

This example shows the normal procedural way of programming. We will focus just on code within the php tags (<?php ?>) because you are already quite familiar with HTML.

First we store the string Hello World in a variable called $myMsg. Afterwards, we are printing the variable value using the built in php function print() on a new line.

File: HelloWorld-NOOP.php

```
<!DOCTYPE html>
<html>
<head>
      <title>HelloWorld - NOOP</title>
</head>

<body>
       <h1>HelloWorld - NOOP (Not OOP)</h1>

       <?php
                   $myMsg = "Hello World";
                   print "<br>" . $myMsg;
       ?>

</body>

</html>
```

Let’s now recreate the same program in an object oriented way. We will start with a class called **HelloWorldOOP**. The basic syntax for creating a class is by using the keyword “class” and after that writing the name of the class. The class content is within opening and closing curly braces.

HelloWorldOOP.php

```
<!DOCTYPE html>
<html>
<head>
	<title>HelloWorld OOP</title>
</head>
<body>
<h1>HelloWorld - OOP</h1>

<?php
class HelloWorldOOP
{
	public function displayMessage()
	{
		$myMsg = "Hello World - OOP";
		print $myMsg;
	}
}

$myHelloWorldOOP = new HelloWorldOOP();
$myHelloWorldOOP->displayMessage();

?>
</body>
</html>
```

Within the class we can define various properties and methods. In this example we don't create any property, just a method called **displayMessage()**. The keyword **public** before the method's name describes the method's scope. The code inside the method is pretty much the same as we had before.

If we run the code nothing is going to happen yet. In OOP we generally don't run the things directly from the class, instead we make an **instantiation of that class and put it into an object**.

That is essentially what **an object** is. It **is an instantiation or a copy of the class definition, instantiated into a variable reference**. In this example we create (instantiate) a new object called **$myHelloWorldOOP** which is an **instance** of the HelloWorldOOP() class. To create new objects we use the **new keyword**. The newly created object has access to all properties and methods defined in the Class. In php we use the syntax called a **little arrow (->) for accessing the properties and methods from the class**. In this example the object $myHelloWorldOOP points to the displayMessage() methods. If we run the code now we will get the same results as in the procedural example above.

Static properties and methods

When we have the keyword **static** in the definition of the properties and methods, means that we can access them directly from the class, **without having to make an instance of the class**. In the example below we can see that this time the function is defined as static, which means **we can access it directly, using the class name with double colons (::),** instead of the little arrow (**->**) and call the function **displayValue()** directly.

HelloWorldOOP-static.php

```
<!DOCTYPE html>
<html>
<head>
      <title>HelloWorld OOP - Static</title>
</head>

<body>
       <h1>HelloWorld - OOP - Static</h1>

<?php
class HelloWorldOOPStatic
{
      public static function displayMessage()
      {
              $myMsg = "Hello World - OOP Static";
              print $myMsg;
      }
}
HelloWorldOOPStatic::displayMessage();
?>

</body>
</html>
```

Once the properties or methods are defined as **static**, they are accessible **without having to instantiate the class.**

“this” keyword

A class is a blueprint for an object: it specifies how the object looks like (properties) and what it can do (methods). When you instantiate a class, you create an object. If you create the class, you can use **$this** to refer to the object itself. This is why you can't set the **$this**, because it's related to the object. It's a special, read-only variable that can be used to refer to the object itself.

Let’s consider the following example:

```
<!DOCTYPE html>
<html>
<head>
      <title>Simple Class</title>
</head>
<body>
<h1>Simple Class</h1>
<?php
//Define the Class
class SimpleClass
{
  public $myMsg = "My Simple Class";
  public function displayMsg()
      {
              print $this->myMsg;
      }
}
//Execute Code Using the Class
$mySimpleClass = new SimpleClass();
$mySimpleClass->displayMsg();
?>
</body>
</html>
```

If we take a look at the **displayMsg()** method, it is a little bit different than what we have seen before. Instead of **print $myMsg** we use print **$this->myMsg**. This means that this refers to the class property called **$myMsg**.

Note that when we access the myMsg property using the **little arrow** (**->**) we **don’t use dollar ($) sign** before it.

Return

To let a class method return a value, we use the return statement. PHP **return statement immediately terminates the execution of a method** when it is called. Please note that built-in PHP methods such as **print** and **echo do not return a value**, instead they are printing some results on the screen.

Let’s see the following example which returns a value from a method:

```
<!DOCTYPE html>
<html>
<head>
      <title>Simple Class with Return</title>
</head>
<body>

<h1>Simple Class with Return</h1>
<?php
//Define the Class

class SimpleClass2
{
  public $myMsg = "My Simple Class with Return<br><br>";

   public function displayMsg2()
      {
         return $this->myMsg;  //return message
      }
}

//Execute Code Using the Class
$mySimpleClass2 = new SimpleClass2();
$rtnVal = $mySimpleClass2->displayMsg2();
print $rtnVal;


?>
</body>
</html>
```

We can see that the example above prints the result of the method a little bit differently. This time instead of printing the method, it is returning a value. **Outside the class we are creating a new object**. The new object executes the **displayMsg()** method, but this time we store the information returned from the method in the $rtnVal variable. And then in order to print the returned value we print the variable.

Method Arguments

Information can be passed to functions through arguments. An **argument** is just like a variable.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Let’s consider the following example:

```
<!DOCTYPE html>
<html>
<head>
      <title>SleptYearsPassData</title>
</head>
<body>
<h1>SleptYearsPassData</h1>
<?php
//Define the Class
class SleptYearsPassData
{
  public $hoursSleptperNight = 8;
  public $myAge = 54;
  public function calcSleptYears($a_hoursSleptperNight,
$a_myAge)
      {
              $sleptYears = ($a_myAge *
$a_hoursSleptperNight)/24;
              return $sleptYears;
      }
}
//Execute Code Using the Class
$mySleptYearsPassData = new SleptYearsPassData();
$rtnVal
= $mySleptYearsPassData-
>calcSleptYears($mySleptYearsPassData->myAge,
$mySleptYearsPassData->hoursSleptperNight);
print "You have slept $rtnVal years of your life away!";
?>
</body>
</html>
```

In the example above in the calcSleptYears() method accepts two arguments, **$a_hoursSleptperNight** and **$a_myAge**. So the calculation within the method is based on the passed arguments. Notice that in the last line of the php script, using print we are printing strings that contain the variable $rtnVal inside of it.

Please note that the prefix “**a_**” was randomly selected - just to show that the variable names used in a method definition

calcSleptYears($a_hoursSleptperNight, $a_myAge)

must not be the same as the variables used in the actual method call:

$mySleptYearsPassData->calcSleptYears($mySleptYearsPassData->myAge, $mySleptYearsPassData->hoursSleptperNight);

are not necessarily the same as the variable names actually used.

We could also use calls like this one:

```
$rtnVal = $mySleptYearsPassData->calcSleptYears(54,8);
```

or

```
$hoursPerNight = 8;
$theAge = 54;
$rtnVal = $mySleptYearsPassData-
>calcSleptYears($hoursPerNight, $theAge);
```

This example was simply constructed to demonstrate the possibility of sending parameters to the method. There is, of course, more elegant way to do the same work:

```
<!DOCTYPE html>
<html>
<head>
      <title>SleptYearsPassDataElegant</title>
</head>
<body>
<h1>SleptYearsPassDataElegant</h1>
<?php
//Define the Class
class SleptYearsPassData2
{
  public $hoursSleptperNight = 8;
  public $myAge = 54;
  public function calcSleptYears2()
      {
              $sleptYears = ($this->myAge * $this-
>hoursSleptperNight)/24;
              return $sleptYears;
      }
}
//Execute Code Using the Class
$mySleptYearsPassData = new SleptYearsPassData2();
$rtnVal
= $mySleptYearsPassData->calcSleptYears2();
print "You have slept elegantly $rtnVal years of your life
away !";
?>
</body>
</html>
```

Constructor

Classes can have a **special built-in method called a constructor.** Constructors allow you to initialise your object's properties when you instantiate an object.

So once you define a constructor within the class definition and assign default values to it, each new created objects will get those properties by default.

Let's consider the following example:

```
<h1>FruitConstructor</h1>

<?php
class FruitConstructor
{
  public $apples;
  public $oranges;
  public $bananas;

  function __construct($apple_arg, $orange_arg, 
$banana_arg)
  {
      $this->apples   = $apple_arg;
      $this->oranges  = $orange_arg;
      $this->bananas  = $banana_arg;
  }
  public function addFruit()
  {
      $totalFruit = $this->apples + $this->oranges + 
$this->bananas;
      return $totalFruit;
  }
}
$myFruit = new FruitConstructor(3, 7, 5);
$rtnVal = $myFruit->addFruit();
print "You have $rtnVal pieces of fruit";
?>
```

From the example above we can understand the constructor definition:

```
function __construct(/* some variables here are optional 
*/ ) {
/*
* some code here
*/
}
```

All the constructor is doing here is that it takes three arguments and stores them as object properties, while we are creating a new instance from the class, as follows:

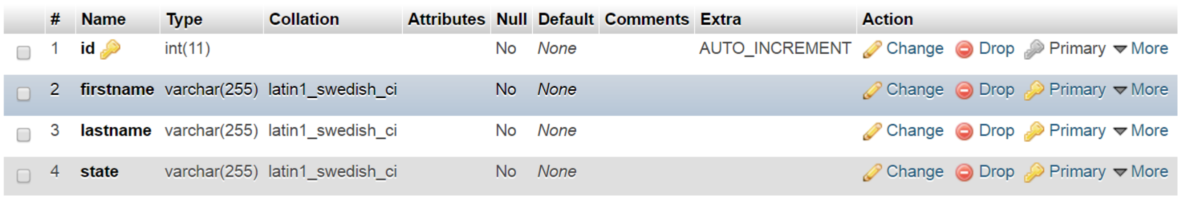
```
$myFruit = new FruitConstructor(3, 7, 5);
```

Prio1: MySQLi Object interface

MySQLi is an extension for accessing MySQL databases.
MySQLi can be accessed through procedural functions, as well as through an object oriented-interface.

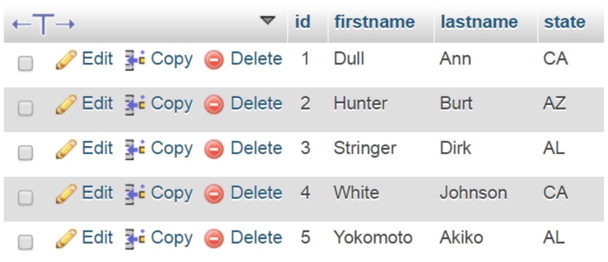
We will briefly discuss most common object-oriented methods of MySQLi. For more information on the issue start here: http://php.net/manual/en/mysqli.quickstart.dual-interface.php

Before establishing connection with the database, create one called **test** and define the table **author** it as follows:



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	*None*		AUTO_INCREMENT	Change Drop Primary More
2	firstname	varchar(255)	latin1_swedish_ci		No	*None*			Change Drop Primary More
3	lastname	varchar(255)	latin1_swedish_ci		No	*None*			Change Drop Primary More
4	state	varchar(255)	latin1_swedish_ci		No	*None*			Change Drop Primary More

And add some test data:



←T→	id	firstname	lastname	state
Edit Copy Delete	1	Dull	Ann	CA
Edit Copy Delete	2	Hunter	Burt	AZ
Edit Copy Delete	3	Stringer	Dirk	AL
Edit Copy Delete	4	White	Johnson	CA
Edit Copy Delete	5	Yokomoto	Akiko	AL

We establish a connection to the MySQL database through creating a new object of the class mysqli, like this:

```
$mysqli = new mysqli('localhost','root','', 'test');
```

It accepts four parameters :

- 1st parameter is the **localhost**
- 2nd is the **username** of the User
- 3rd is the **password** of the User (in this case we haven't set a password so we keep this parameter empty).

- 4th is the **database name**

Then we can check whether the object was created correctly and establish connection with the “test” database using the following condition:

```
if ($mysqli->connect_errno) {
   echo "Failed to connect to MySQL: " . $mysqli-
>connect_error;
}
```

Once we have established a connection, we create our SQL query as follows:

```
$sql_statement  = "SELECT lastname, firstname, state ";
$sql_statement .= "FROM author ";
$sql_statement .= "WHERE state = 'CA' ";
$sql_statement .= "ORDER BY lastname, firstname ";
```

You can notice that using the “**.=**” syntax we are concatenating new strings to the existent one. This is one of the more convenient ways of creating long strings instead of writing like this:

```
$sql_statement  = "SELECT lastname, firstname, state
FROM author WHERE state = 'CA'  ORDER BY lastname,
firstname ";
```

Next we run a **query()** method as part of **$mysqli** object that was created, and pass to it the **query(sql_statement)**. The results from the query are stored in the result variable:

```
$result = $mysqli->query($sql_statement);
```

Then we will create a validation to check whether SQL query was successfully executed:

```
if (!$result) {
      $outputDisplay .= "<p>MySQL No: " . $mysqli->errno .
"</p>";
      $outputDisplay .= "<p>MySQL Error: " . $mysqli-
>error . "</p>";
      $outputDisplay .= "<p>SQL Statement: " .
$sql_statement . "</p>";
      $outputDisplay .= "<p>MySQL Affected Rows: " .
$mysqli->affected_rows . "</p>";
}
```

If it runs successfully, without any errors we can extend the condition using the else block.

We have two options for fetching the data from the Database, individual row and multiple rows.

Individual row

To get an individual row from the returned result set we are using the built-in method **fetch_assoc()** and stored the returned row, into the $row variable, like this:

```
$row = $result->fetch_assoc();
```

Then we pass the column name and store it into a new variable, like this:

```
$firstname = $row['firstname'];
$lastname = $row['lastname'];
$state = $row['state'];
```

Now we are able print the data from the Database on the screen like this:

```
print "Author name: " . $firstname;
print "<br>";
print "Author lastname: " . $lastname;
print "<br>";
print "Author state: " . $state;
```

This way we will get just the first author returned from the query. This way of fetching data from the DB is quite rare.

Multiple rows

Most common way of fetching data from the Database is using a **foreach() loop**. It returns all the records returned from the SQL query.

The foreach loop works only on arrays, and is used to loop through each key/value pair in an array. For every loop iteration, the value of the current array element is assigned to $value and the array pointer is moved by one, until it reaches the last array element.

Within the else block (delete or comment the previous code), make sure it is clear we use the **fetch_all()** built-in method instead of fetch_assoc() as in individual row and specifies what type of array that should be produced: MYSQLI_ASSOC, MYSQLI_NUM or MYSQLI_BOTH.

Can be one of the following values:

```
$rows = $result->fetch_all(MYSQLI_ASSOC);
```

All the rows returned from the SQL query are stored as an associative array within the $rows array. Most suitable way for looping over an associative array is using the foreach loop as follows:

```
foreach ($rows as $row) {
  echo "<hr>";
  echo "<br>";
  echo $row['firstname'];
  echo "<br>";
  echo $row['lastname'];
  echo "<br>";
  echo $row['state'];
}
```

Using the foreach loop we get all returned results from the query, print each column name and separate the rows (records) using the HTML tag <hr>. This is the most common and suitable way of fetching data from a Database.

Prio2: Commenting with DocBlocks

The DocBlock commenting style is a widely accepted method of documenting classes. A DocBlock is defined by using a block comment that starts with an additional asterisk:

```
/**
* This is a very basic DocBlock
*/
```

The real power of DocBlocks comes with the ability to use tags, which start with an at symbol (@) immediately followed by the tag name and the value of the tag. DocBlock tags allow developers to define authors of a file, the license for a class, the property or method information, and other useful information.

The most common tags used follow:

1. **@author:** The author of the current element (which might be a class, file, method, or any bit of code) are listed using this tag. Multiple author tags can be used in the same DocBlock if more than one author is credited. The format for the author name is John Doe <john.doe@email.com>.
2. **@copyright:** This signifies the copyright year and name of the copyright holder for the current element. The format is 2010 Copyright Holder.
3. **@license:** This links to the license for the current element. The format for the license information is *http://www.example.com/path/to/license.txt License Name*.
4. **@var:** This holds the type and description of a variable or class property. The format is type element description.
5. **@param:** This tag shows the type and description of a function or method parameter. The format is type $element_name element description.
6. **@return:** The type and description of the return value of a function or method are provided in this tag. The format is type return element description

A sample class commented with DocBlocks might look like this:

```
/**
* A simple class
*
* This is the long description for this class,
* which can span as many lines as needed. It is
* not required, whereas the short description is
* necessary.
*
* It can also span multiple paragraphs if the
* description merits that much verbiage.
*
* @author Jason Lengstorf <jason.lengstorf@ennuidesign.com>
* @copyright 2010 Ennui Design
* @license http://www.php.net/license/3_01.txt PHP License 3.01
*/
class SimpleClass
{
 /**
  * A public variable
  *
  * @var string stores data for the class
```

```php
    */
    public $foo;

    /**
     * Sets $foo to a new value upon class instantiation
     *
     * @param string $val a value required for the class
     * @return void
     */
    public function __construct($val)
    {
        $this->foo = $val;
    }

    /**
     * Multiplies two integers
     *
     * Accepts a pair of integers and returns the
     * product of the two.
     *
     * @param int $bat a number to be multiplied
     * @param int $baz a number to be multiplied
     * @return int the product of the two parameters
     */
    public function bar($bat, $baz)
    {
        return $bat * $baz;
    }
}

?>
```

Once you scan the preceding class, the benefits of DocBlock are apparent: everything is clearly defined so that the next developer can pick up the code and never have to wonder what a snippet of code does or what it should contain.

Prio3: Advanced OO concepts

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 that inherits features of another class are known as a child class. The class which is being inherited is known as the parent class. The concept of the inheritance in oop is the same as inheritance in the real world. For example, a child inherits characteristics of their parent. Same happens in oop. One class is inheriting characteristics of another class.

Let’s consider one simple example:

```
<?php
Class Employee {
public $salary = "3500";
}
Class Driver extends Employee {
public $name = "John";
public $position = "driver";
public function showDetails() {
  return
    'My name is ' . $this->name .
    ' and my position in the company is ' . $this-
>position .
    ', my monthly salary is ' . $this->salary . ' Euros';
}
}
$driverObj = new Driver();
$result = $driverObj->showDetails();
echo $result;
?>
```

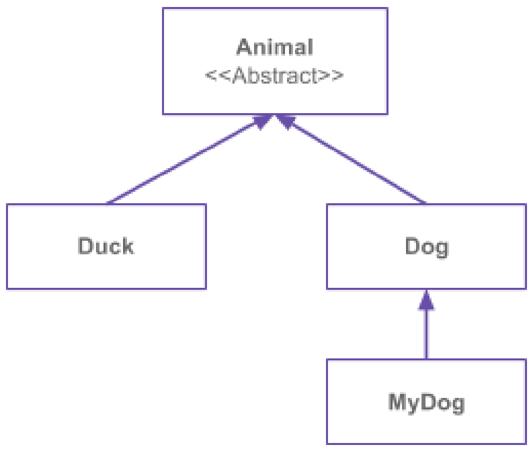
Here we can see how we can access the property $salary from the Employee class and print the whole statement.

Let’s consider a more advanced example. For instance, you can create a class Car and then extend it in a child class **SportsCar**, and then add the additional method **driveItWithStyle**:

```
// The parent class has its properties and methods
class Car {

//A private property or method can be used only by the
parent.
private $model;

// Public methods and properties can be used
 // by both the parent and the child classes.
public function setModel($model)
{
  $this -> model = $model;
}

public function getModel()
{
  return $this -> model;
}
}


// The child class can use the code it
// inherited from the parent class,
// and it can also have its own code
class SportsCar extends Car {

private $style = 'fast and furious';

public function driveItWithStyle()
{
  return 'Drive a ' . $this -> getModel() . ' <i>' .
$this -> style . '</i>';
}
}


//create an instance from the child class
$sportsCar1 = new SportsCar();

// Use a method that the child class inherited from the
parent class
$sportsCar1->setModel('Ferrari');

// Use a method that was added to the child class
echo $sportsCar1->driveItWithStyle();
```

Abstract Class

The idea behind the Abstract Class is to protect you (or any other developer working on the same project) from trying to instantiate class that should only be used for inheritance, but not instantiate itself.



For example, we have Animal Class and derive from it a Duck Class, Dog Class and derived from that is a specific MyDog Class.
Inside of the **Animal** class, we can define methods and properties that are common for all animals.

That way we get the benefit from inheritance but not accidental instantiation of something that doesn't make sense to live on its own.

Let's consider one simple example using an Abstract Class:

```php
<?php
abstract class Animal
{
  public $name;
  public $age;

  public function Describe()
  {
      return $this->name . ", " . $this->age . " years
old";
  }
}
class Dog extends Animal
{
  public function Greet()
  {
      return "Woof!";
  }

  public function Describe()
  {
      return parent::Describe() . ", and I'm a dog !";
  }
}
// $animal = new Animal(); will throw an error, we can not
create instance from an abstract class
$animal = new Dog();
$animal->name = "Bob";
$animal->age = 7;
echo $animal->Describe();
echo "<br>";
echo $animal->Greet();
?>
```

You may have noticed that we have the same function called Describe() in both classes. This is called **method overriding** and it just other legal oop concept. It allows you to **extend the functionality of the method**. We use the parent keyword to reference the Animal class, then we call Describe() function on it and extend it further.

Outside the class we instantiate the Dog class, set the two properties and then call the two methods defined in it. If you test this code, you will see that the Describe() method is now a combination of the Animal and the Dog version, as expected.

Interface

Interfaces have two main purposes. One it to define constants (constants are like variables except that once they are defined they cannot be changed or undefined) that can be used with another class (many classes). Each class can implement as many Interfaces as it needs.

Instead of inheriting, we are implementing the interfaces. The main rules is that once you implement any interface. It acts as an enforcing mechanism to enforce consistency across the App. Imagine you are working in a team with many developers on the same app, you can define an interface which enforces some rules. For example the method for getting the Job title must be always getJobTitle(); so you define this method inside the interface and this way the other developers are enforced to use this method for getting the Job title instead of creating their own such as GetJobTitle(), listJobTitle(), getherJobTitle ect.

Methods within the Interface doesn’t contain any functionality, but just blank method definition, like this:

```
interface StandardFunction {
  public function getJobTitle();
}
```

This way you have consistent names. Whenever you implement an Interface that has defined methods within it, you have made contact with the Interface, that **you must override that method.** You should **implement all the methods that the Interface contains**. It is totally legal if you override the interface method and not extend it, like this:

```
function getJobTitle() {
}
```

This meets the minimal requirement for the OO PHP compiler.

Final

The opposite of **abstract**, **final** prevents child classes from overriding a method by prefixing its definition with final, like this:

```
final public function getJobTItle() {
}
```

If the Class itself is being defined as final, then it cannot be extended, like this:

```
final class Animal {
}
```

Last modified: Tuesday, 19 June 2018, 11:14 AM