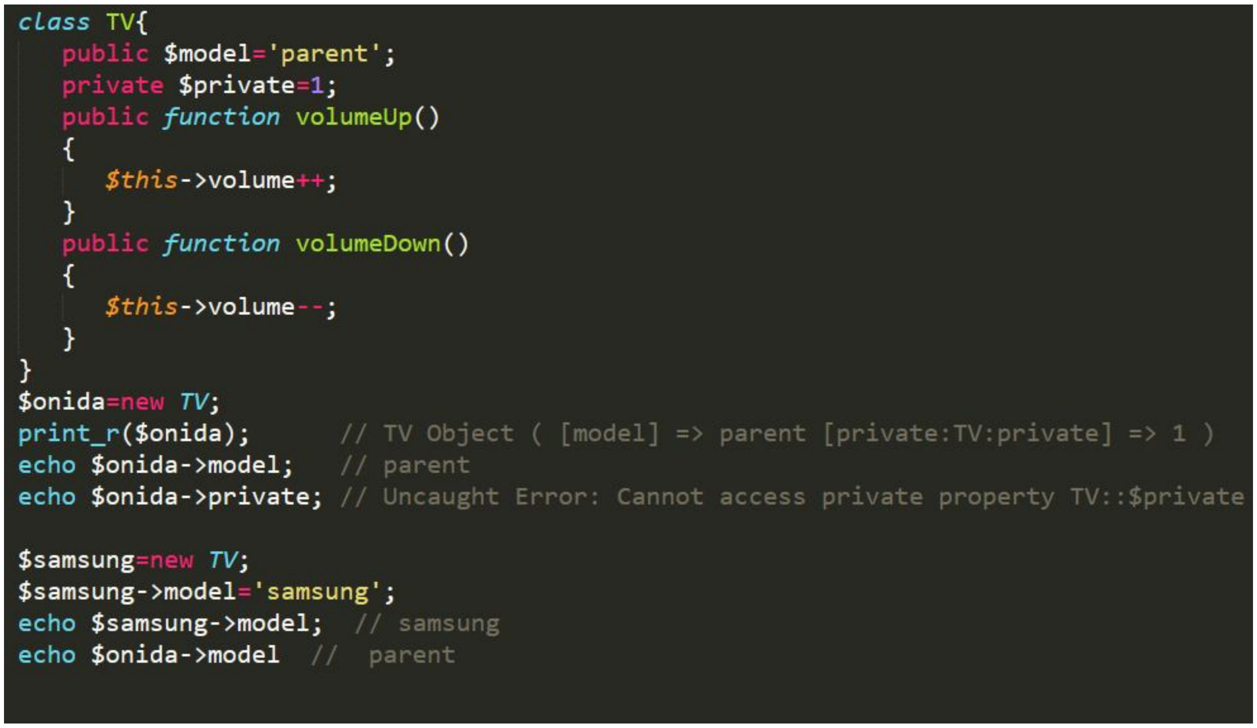
PHP-OOPS-CODEIGNITER-LARAVEL

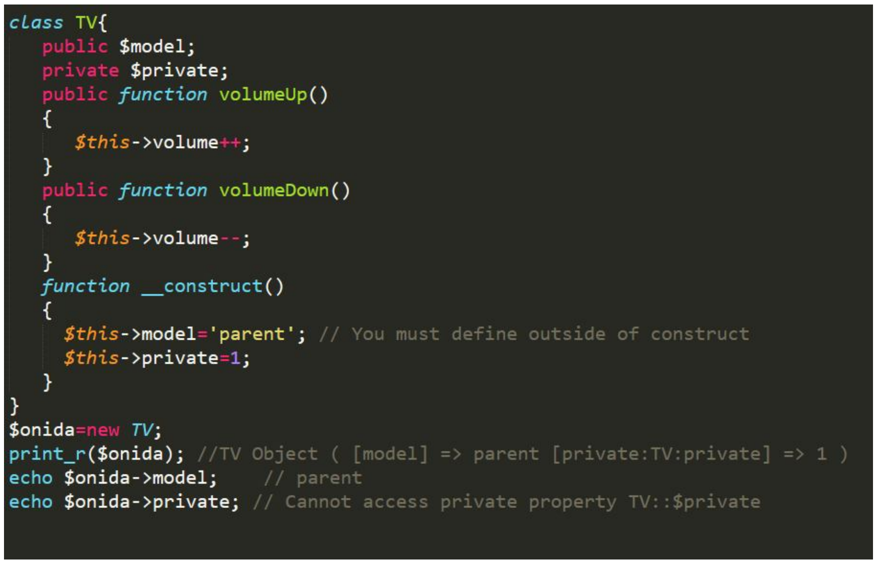
OOPS: OOPS is a type of programming principle that helps in building complex, reusable applications.

Class: This is a programmer-defined data type, which includes member functions as well as data member. **Object**: Object is a specific instance of a class.



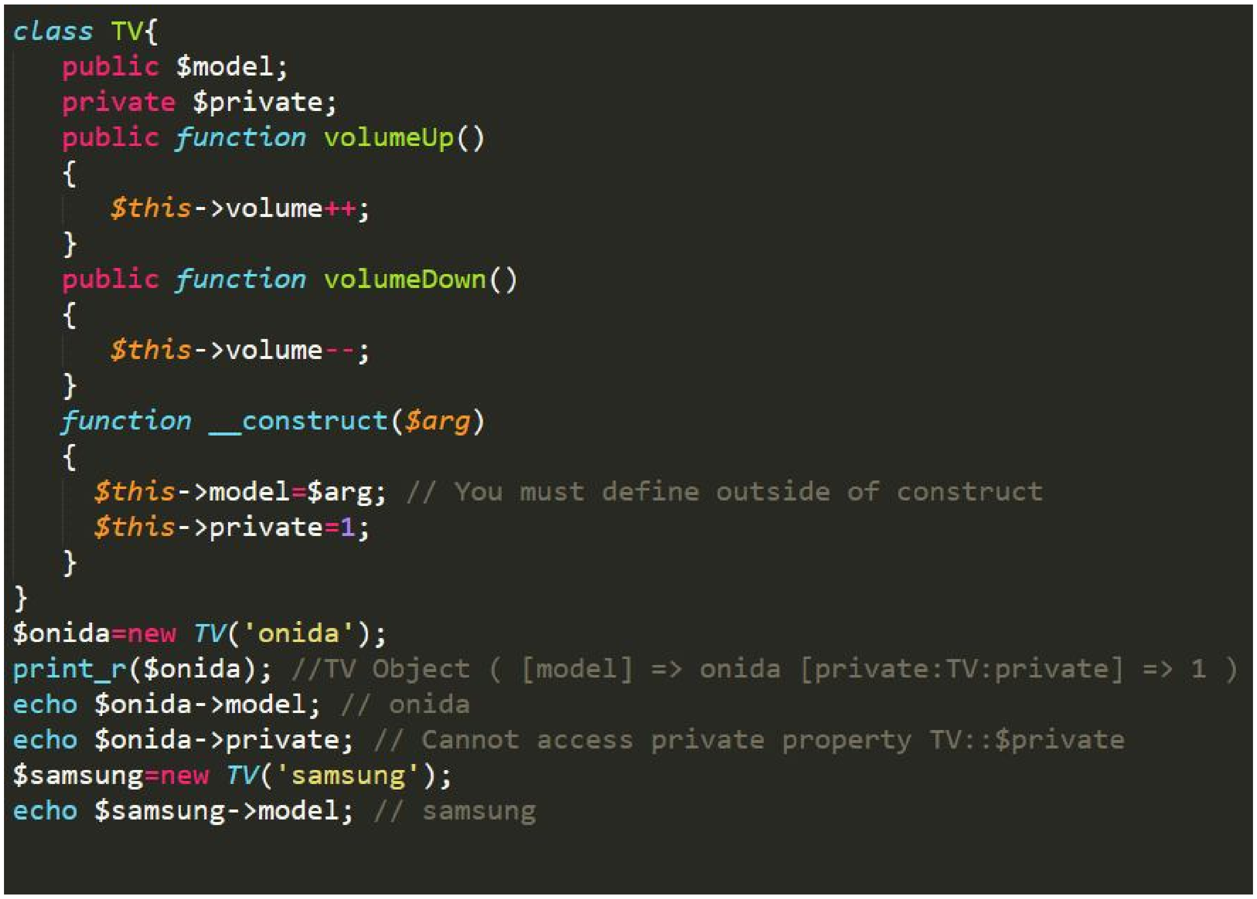
```php
class TV{
    public $model='parent';
    private $private=1;
    public function volumeUp()
    {
        $this->volume++;
    }
    public function volumeDown()
    {
        $this->volume--;
    }
}
$onida=new TV;
print_r($onida);      // TV Object ( [model] => parent [private:TV:private] => 1 )
echo $onida->model;   // parent
echo $onida->private; // Uncaught Error: Cannot access private property TV::$private

$samsung=new TV;
$samsung->model='samsung';
echo $samsung->model;  // samsung
echo $onida->model  //  parent
```

Constructor: Constructor is a special function and it automatically called when any object of a class is instantiated. Constructor is also called magic function because in PHP, magic method is start usually with two underscore.



```php
class TV{
   public $model;
   private $private;
   public function volumeUp()
   {
      $this->volume++;
   }
   public function volumeDown()
   {
      $this->volume--;
   }
   function __construct()
   {
     $this->model='parent'; // You must define outside of construct
     $this->private=1;
   }
}
$onida=new TV;
print_r($onida); //TV Object ( [model] => parent [private:TV:private] => 1 )
echo $onida->model;    // parent
echo $onida->private; // Cannot access private property TV::$private
```

You can also pass **argument** in **constructer** function.



```
class TV{
    public $model;
    private $private;
    public function volumeUp()
    {
        $this->volume++;
    }
    public function volumeDown()
    {
        $this->volume--;
    }
    function __construct($arg)
    {
        $this->model=$arg; // You must define outside of construct
        $this->private=1;
    }
}
$onida=new TV('onida');
print_r($onida); //TV Object ( [model] => onida [private:TV:private] => 1 )
echo $onida->model; // onida
echo $onida->private; // Cannot access private property TV::$private
$samsung=new TV('samsung');
echo $samsung->model; // samsung
```

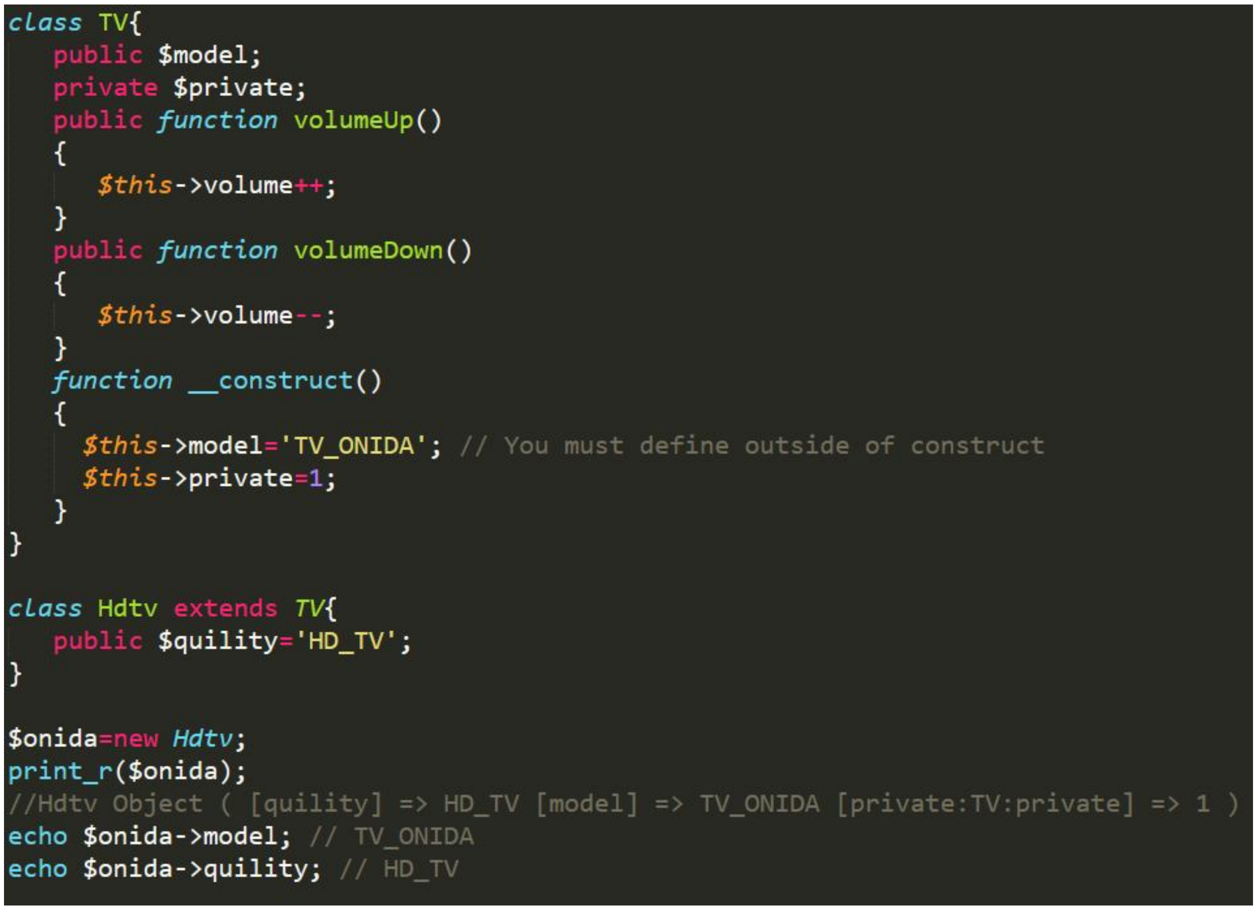
Inheritance:

Inheritance is a mechanism wherein a new class is derived from an existing class.

Advances of inheritance:

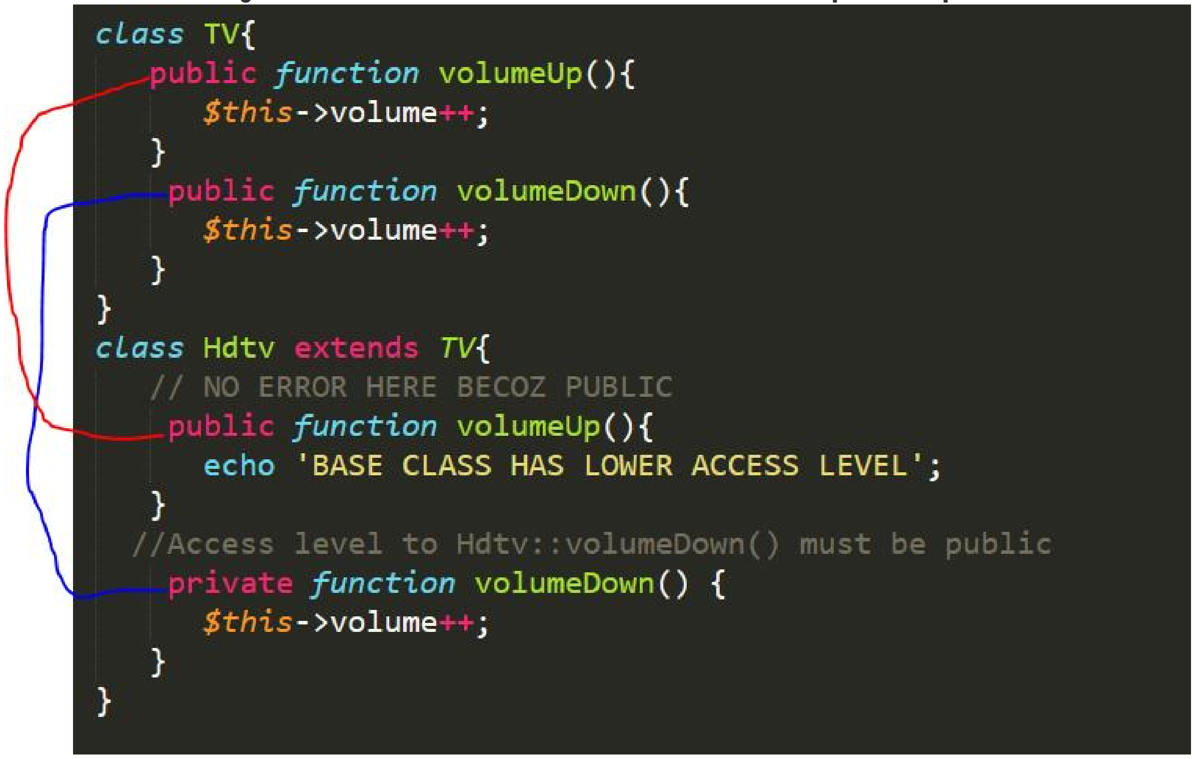
- **Reusability** - facility to use public methods of base class without rewriting the same.
- **Extensibility** - extending the base class logic as per business logic of the derived class.
- **Data hiding** - base class can decide to keep some data private so that it cannot be altered by the derived class
- **Overriding** -With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

❖ Child class has more power than parent class member and function. Parent class get **override.**



```
class TV{
    public $model;
    private $private;
    public function volumeUp()
    {
        $this->volume++;
    }
    public function volumeDown()
    {
        $this->volume--;
    }
    function __construct()
    {
       $this->model='TV_ONIDA'; // You must define outside of construct
       $this->private=1;
    }
}

class Hdtv extends TV{
    public $quility='HD_TV';
}

$onida=new Hdtv;
print_r($onida);
//Hdtv Object ( [quility] => HD_TV [model] => TV_ONIDA [private:TV:private] => 1 )
echo $onida->model; // TV_ONIDA
echo $onida->quility; // HD_TV
```

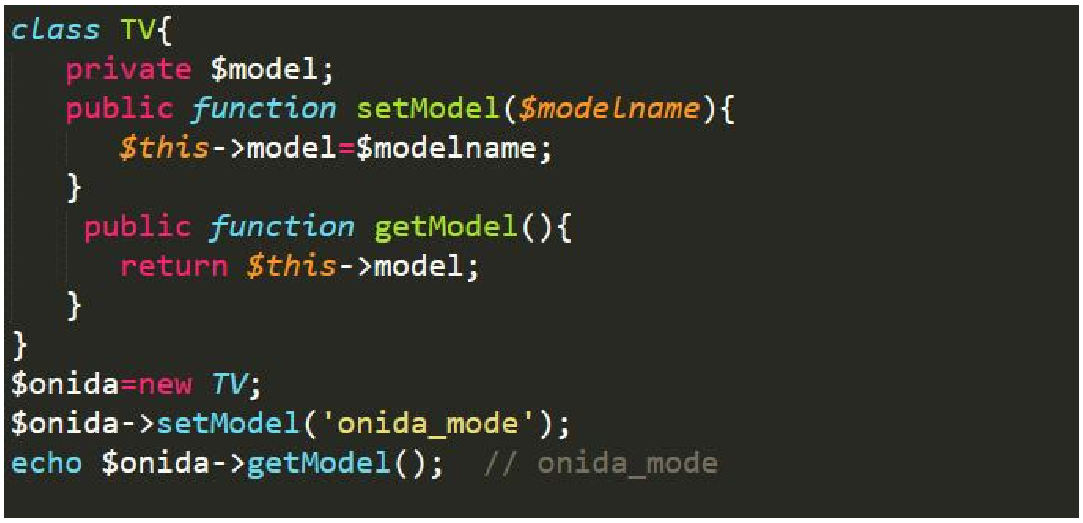
- ❖ **When overriding child class must have lower access level as compared to parent.**



```
class TV{
    public function volumeUp(){
        $this->volume++;
    }
     public function volumeDown(){
        $this->volume++;
    }
}
class Hdtv extends TV{
    // NO ERROR HERE BECOZ PUBLIC
     public function volumeUp(){
        echo 'BASE CLASS HAS LOWER ACCESS LEVEL';
    }
   //Access level to Hdtv::volumeDown() must be public
     private function volumeDown() {
        $this->volume++;
    }
}
```

Encapsulation

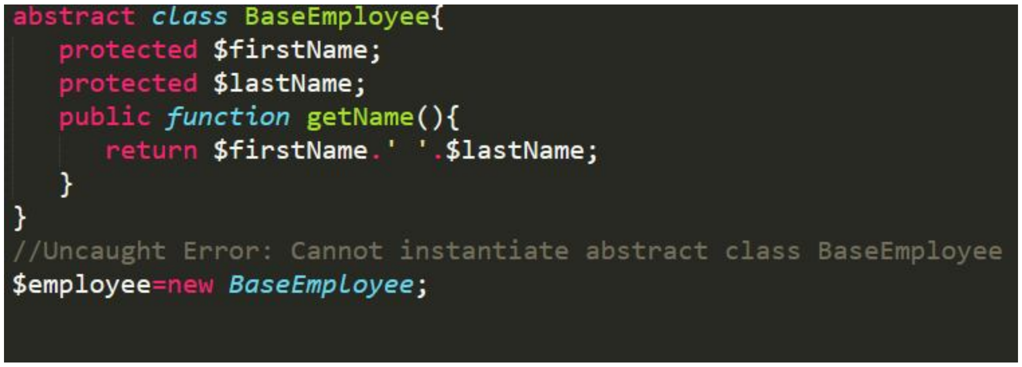
Wrapping some data in single unit is called Encapsulation. Encapsulation is used to safe data or information in an object from other it means encapsulation is mainly used for protection purpose. Second advantage of encapsulation is you can make the class read only or write only by providing setter or getter method.



```
class TV{
    private $model;
    public function setModel($modelname){
        $this->model=$modelname;
    }
     public function getModel(){
        return $this->model;
    }
}
$onida=new TV;
$onida->setModel('onida_mode');
echo $onida->getModel();  // onida_mode
```

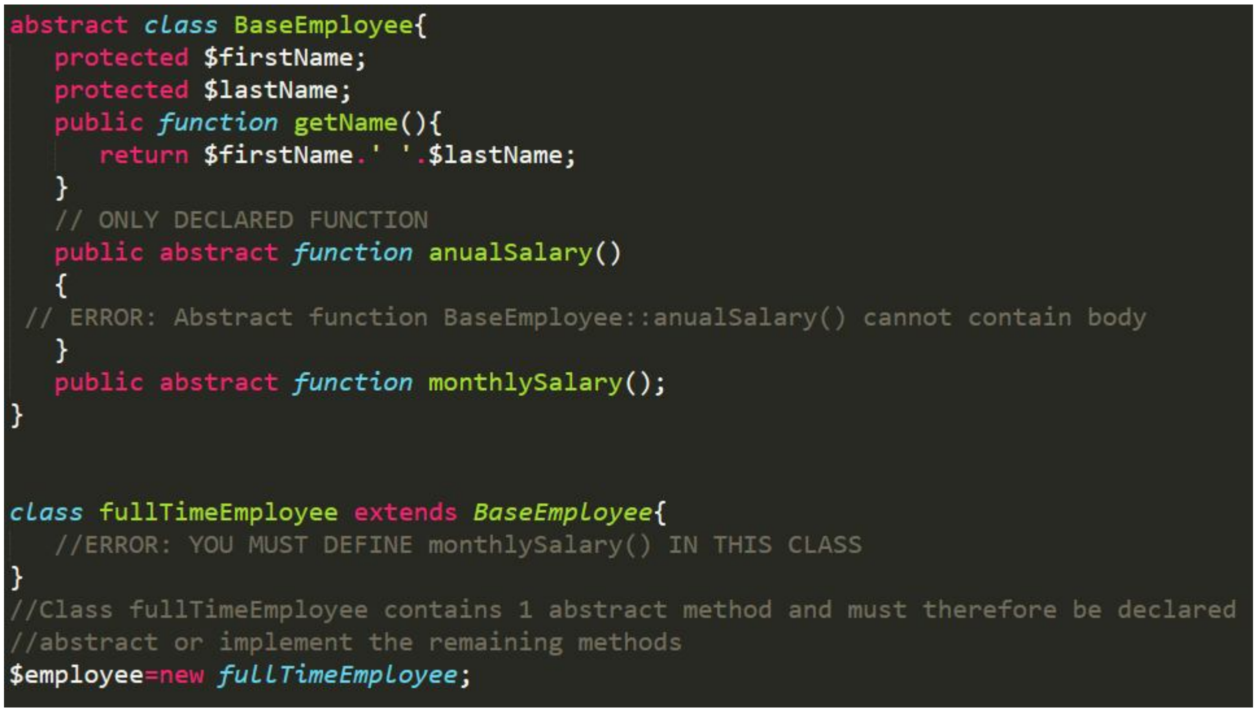
Class Abstraction

PHP 5 introduces abstract classes and methods. Classes defined as abstract may not be instantiated, and any class that contains at least one abstract method must also be abstract. *Methods defined as abstract simply declare the method's signature - they cannot define the implementation*. 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 (***or a less restricted***) visibility. For example, if the abstract method is defined as **protected**, the function implementation must be defined as either **protected or public, but not private**. Furthermore the signatures of the methods must match, i.e. the type hints and the number of required arguments must be the same. For example, if the child class defines an optional argument, where the abstract method's signature does not, there is no conflict in the signature.



```
abstract class BaseEmployee{
    protected $firstName;
    protected $lastName;
    public function getName(){
        return $firstName.' '.$lastName;
    }
}
//Uncaught Error: Cannot instantiate abstract class BaseEmployee
$employee=new BaseEmployee;
```

Abstract class uses for Enforcement. That means if we **declared a method** and **not define** that method then the class that drive this class must define that methods. So every class that extent a class having a abstract methods must define that methods in their own body.

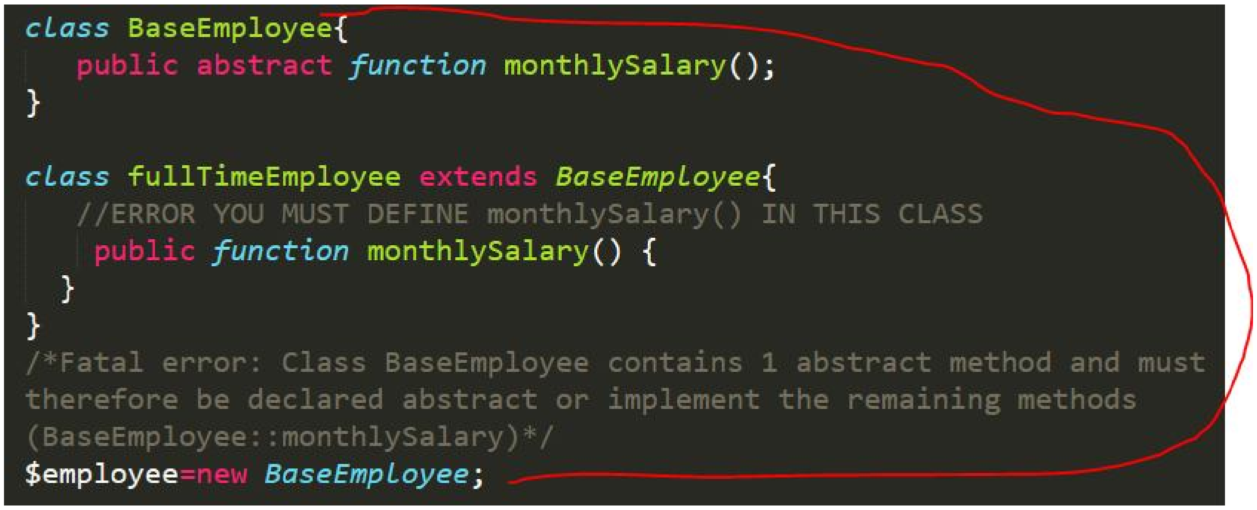


```
abstract class BaseEmployee{
   protected $firstName;
   protected $lastName;
   public function getName(){
      return $firstName.' '.$lastName;
   }
   // ONLY DECLARED FUNCTION
   public abstract function anualSalary()
   {
 // ERROR: Abstract function BaseEmployee::anualSalary() cannot contain body
   }
   public abstract function monthlySalary();
}


class fullTimeEmployee extends BaseEmployee{
   //ERROR: YOU MUST DEFINE monthlySalary() IN THIS CLASS
}
//Class fullTimeEmployee contains 1 abstract method and must therefore be declared
//abstract or implement the remaining methods
$employee=new fullTimeEmployee;
```

-If any class that has at least one abstract method becomes abstract class no need to use **abstract class.**

-Can declared and also define the function in abstract class. See getName is define in abstract class.

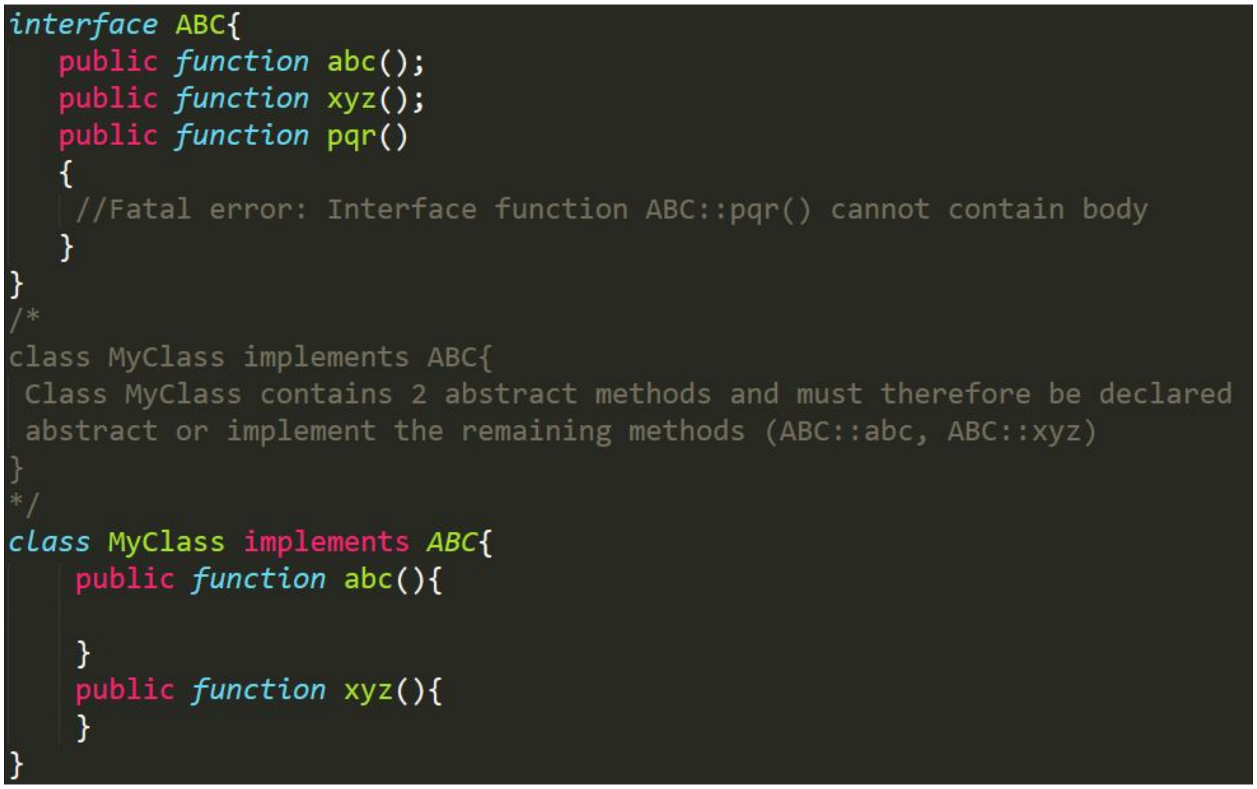
-If abstract removed from class name and define a abstract method in that class then also you can't create direct object of that base class.



```
class BaseEmployee{
   public abstract function monthlySalary();
}

class fullTimeEmployee extends BaseEmployee{
   //ERROR YOU MUST DEFINE monthlySalary() IN THIS CLASS
    public function monthlySalary() {
   }
}
/*Fatal error: Class BaseEmployee contains 1 abstract method and must
therefore be declared abstract or implement the remaining methods
(BaseEmployee::monthlySalary)*/
$employee=new BaseEmployee;
```

Interface

An Interface allows the users to create programs, specifying the **only public methods** that a class must implement, without involving the complexities and details of how the particular methods are implemented. It is generally referred to as the next level of abstraction. It resembles the abstract methods, resembling the abstract classes. An Interface is defined just like a class is defined but with the class keyword replaced by the interface keyword and just the function prototypes. The interface **contains no Data Variables/Data Member**. The interface is helpful in a way that it ensures to maintain a sort of metadata for all the methods a programmer wishes to work on. **Not contain construct function**.



```
interface ABC{
   public function abc();
   public function xyz();
   public function pqr()
   {
    //Fatal error: Interface function ABC::pqr() cannot contain body
   }
}
/*
class MyClass implements ABC{
 Class MyClass contains 2 abstract methods and must therefore be declared
 abstract or implement the remaining methods (ABC::abc, ABC::xyz)
}
*/
class MyClass implements ABC{
    public function abc(){

    }
    public function xyz(){
    }
}
```

Few characteristics of an Interface are:

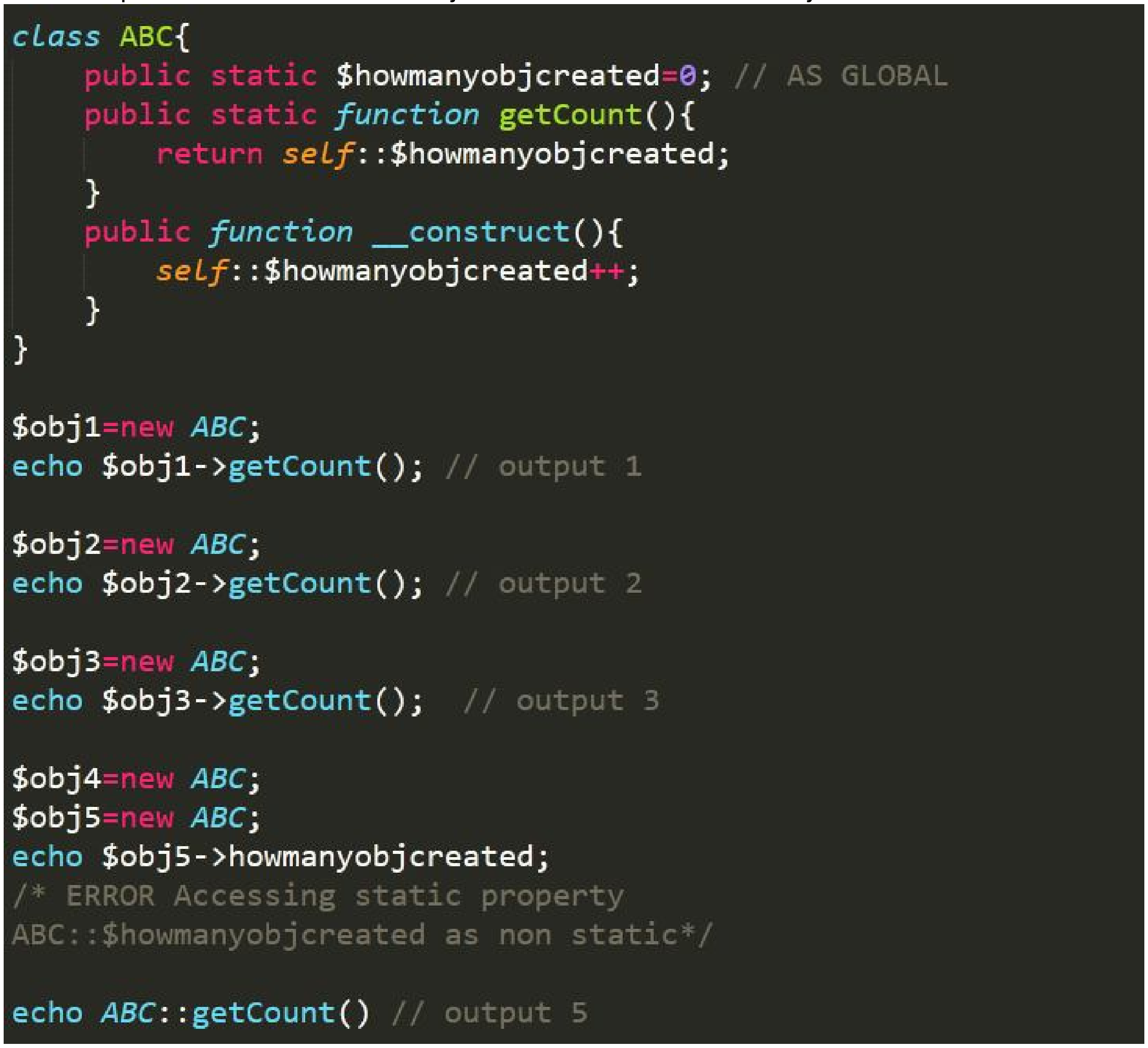
- An interface consists of methods that have no implementations, which means the interface methods are abstract methods.
- All the methods in interfaces must have public visibility scope.
- Interfaces are different from classes as the class can inherit from one class only whereas the class can implement one or more interfaces.

Advantages of PHP Interface

- An interface allows unrelated classes to implement the same set of methods, regardless of their positions in the class inheritance hierarchy.
- An **interface can model multiple inheritances** because a class can implement more than one interface whereas it can extend only one class.
- The implementation of an inheritance will save the caller from full implementation of object methods and focus on just he objects interface, therefore, the caller interface remains unaffected.

STATIC KEYWORDS IN OPPS

Static in oops refers to the class not to object. Static is not in context of object.

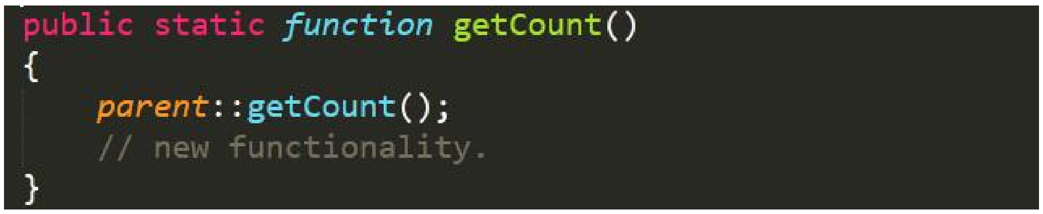


```
class ABC{
    public static $howmanyobjcreated=0; // AS GLOBAL
    public static function getCount(){
        return self::$howmanyobjcreated;
    }
    public function __construct(){
        self::$howmanyobjcreated++;
    }
}

$obj1=new ABC;
echo $obj1->getCount(); // output 1

$obj2=new ABC;
echo $obj2->getCount(); // output 2

$obj3=new ABC;
echo $obj3->getCount();  // output 3

$obj4=new ABC;
$obj5=new ABC;
echo $obj5->howmanyobjcreated;
/* ERROR Accessing static property
ABC::$howmanyobjcreated as non static*/

echo ABC::getCount() // output 5
```

$this	**self**
Represents an instance of the class or object	Represents a class
Always begin with dollar ($) sign	Never begin with dollar($) sign
Is followed by the -> operator	Is followed by the :: operator
The property name after the -> operator does not take dollar ($) sign, e.g., $this->property.	The property name after the :: operator always take the dollar ($) sign.

When we override the function and want a parent functionalities also in it then we use

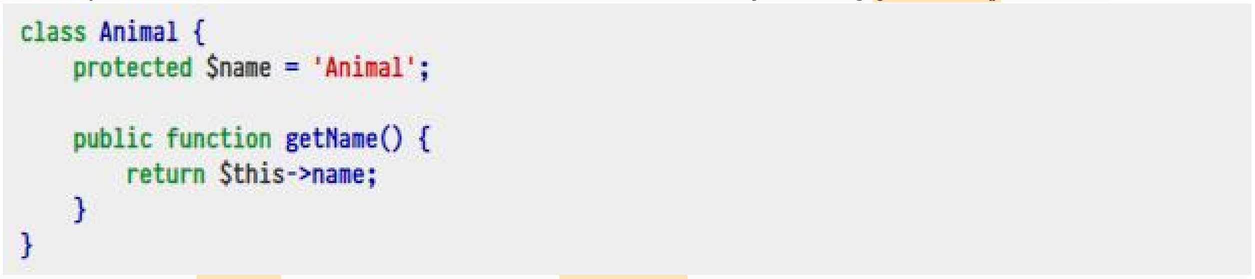
parent::functionName();



```
public static function getCount()
{
    parent::getCount();
    // new functionality.
}
```

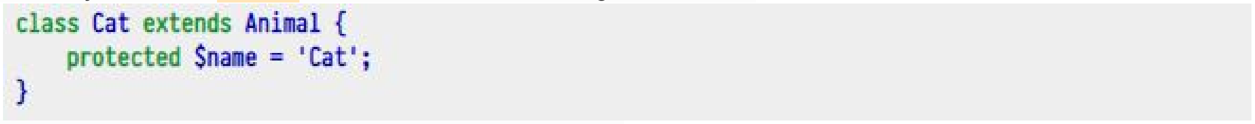
Any class in which all methods are statics are called statics class. Class does not have any special keywords like statics.

Late Statics Binding: Let's jump into an example up front to understand the concept. Let's create two classes parent and child which would tell us the name of the object using getName() method:



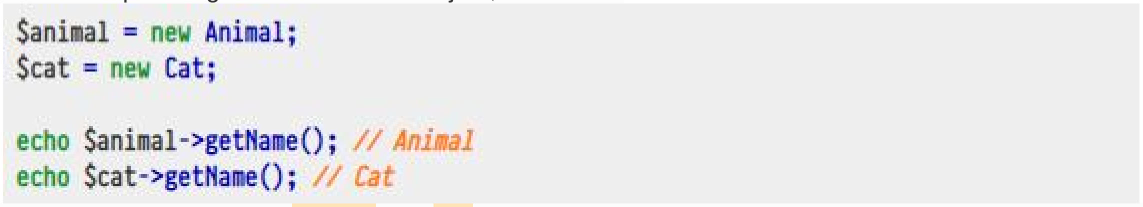
```
class Animal {
    protected $name = 'Animal';

    public function getName() {
        return $this->name;
    }
}
```

Let's extend the Animal class so we can use its getName() method without repeating it in child class. We will only need the $name variable in child class to get it's name:



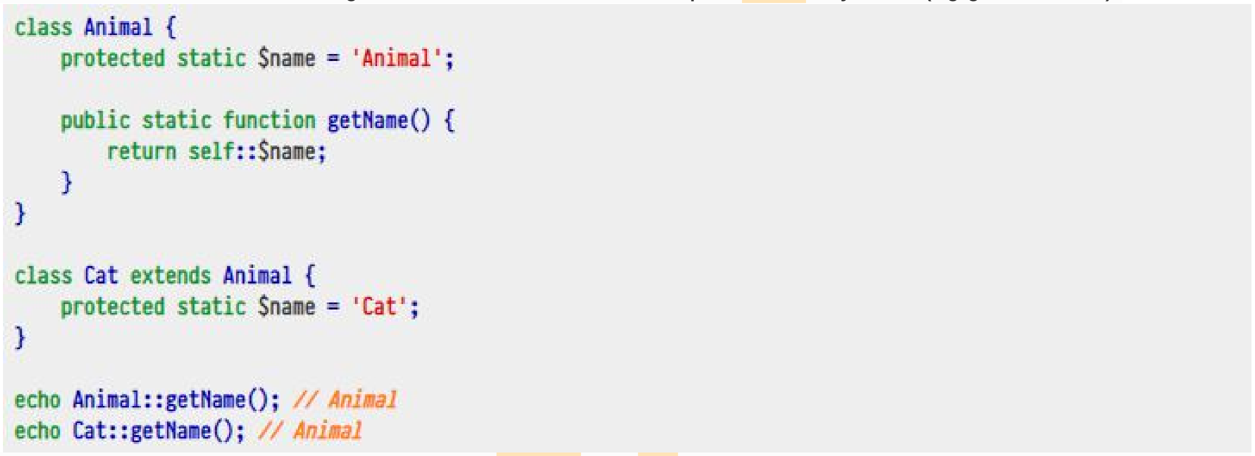
```
class Cat extends Animal {
    protected $name = 'Cat';
}
```

Now we expect to get names of each object, let's do so:



```
$animal = new Animal;
$cat = new Cat;

echo $animal->getName(); // Animal
echo $cat->getName(); // Cat
```

And we successfully get Animal and Cat echoed out. But now let's modify code a bit so that we can use those classes without creating their instances with the help of static keyword (eg global state):



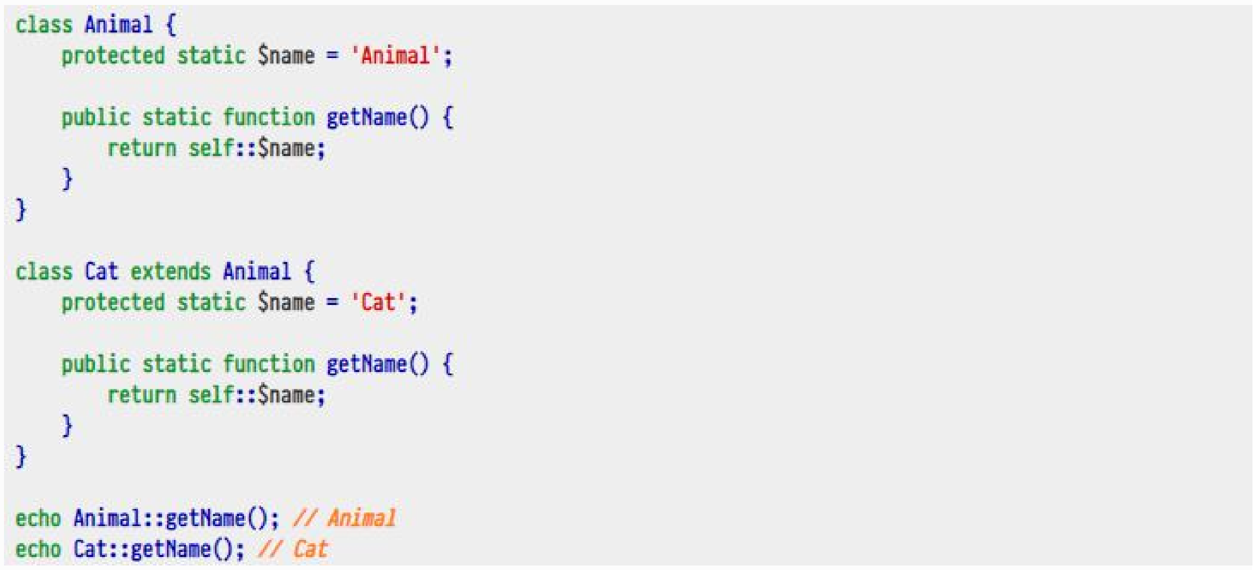
```
class Animal {
    protected static $name = 'Animal';

    public static function getName() {
        return self::$name;
    }
}

class Cat extends Animal {
    protected static $name = 'Cat';
}

echo Animal::getName(); // Animal
echo Cat::getName(); // Animal
```

Noticed the problem ? We wanted to see Animal and Cat to be echoed out like previous example but in both cases it said Animal not Cat.

The reason why in previous example things worked the way expected is because we explicitly created new instances of both classes using new keyword and PHP knew which class and method to use. This is not the case in second static example. In the second example, we are using the self keyword which *always resolves to current class* where it is called. This is reason why the name for the Cat class wasn't echoed out.

So how do we get the name of cat ? Here are few ways.

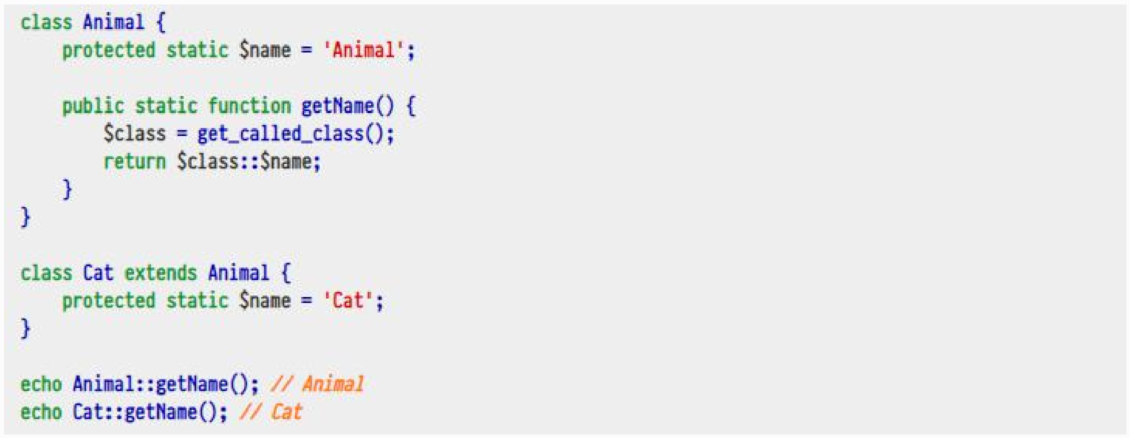
By Repeating Same Code In Child Class



```php
class Animal {
    protected static $name = 'Animal';

    public static function getName() {
        return self::$name;
    }
}

class Cat extends Animal {
    protected static $name = 'Cat';

    public static function getName() {
        return self::$name;
    }
}

echo Animal::getName(); // Animal
echo Cat::getName(); // Cat
```

This works but it defeats the purpose of inheritance. What is the point of extending Animal class when we need to repeat same code in child class ? This isn't ideal.

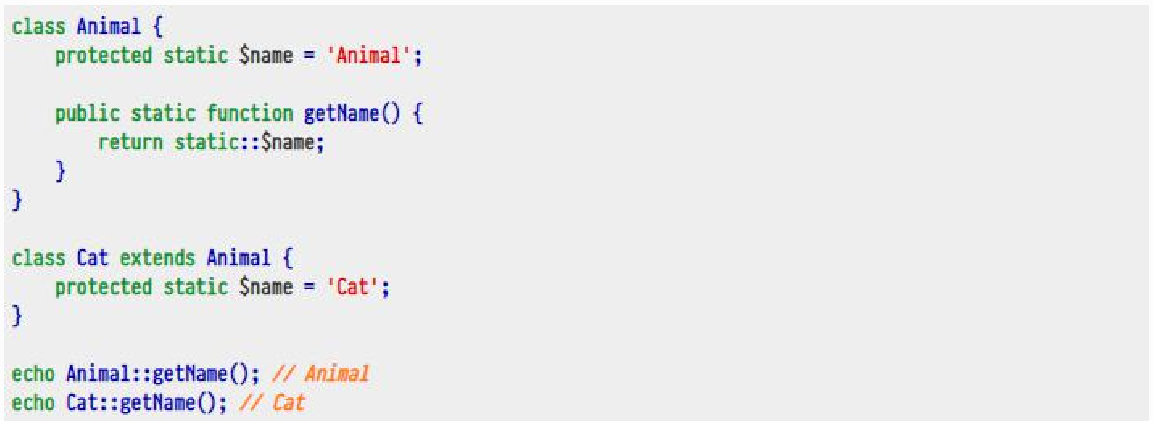
By Using `get_called_class()`



```php
class Animal {
    protected static $name = 'Animal';

    public static function getName() {
        $class = get_called_class();
        return $class::$name;
    }
}

class Cat extends Animal {
    protected static $name = 'Cat';
}

echo Animal::getName(); // Animal
echo Cat::getName(); // Cat
```

This is better and works for our purpose.

By Using `static` keyword:



```php
class Animal {
    protected static $name = 'Animal';

    public static function getName() {
        return static::$name;
    }
}

class Cat extends Animal {
    protected static $name = 'Cat';
}

echo Animal::getName(); // Animal
echo Cat::getName(); // Cat
```

PHP 5.3 introduced the `static` keyword to help deal with this issue. Before that, `get_called_class()`was what was used. Let's get the expected result using `static` keyword:

And this works fine too. These days using `static` keyword seems to be common practice instead of `get_called_class()` to deal with this issue though `get_called_class()` has many other uses too.

So in simple words, **late static binding** is something that helps us correctly resolve to **static** classes at run time. So when we use `self` keyword, PHP checks it at compile time which class to **bind** the method call to but when we use `static` keyword, PHP would check it **late** eg it would determine which class to use and bind method call to at runtime. Doing it at runtime is what helps PHP determine which class was meant.

DEPENDENCY INJECTION

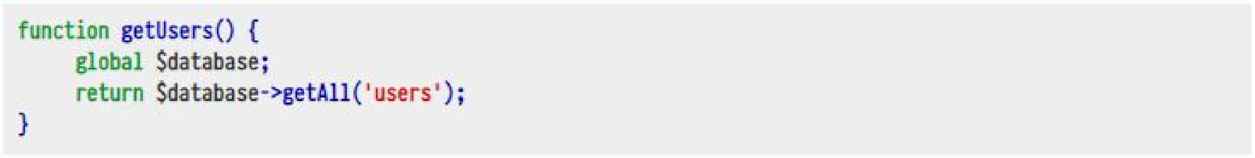
https://code.tutsplus.com/tutorials/dependency-injection-in-php--net-28146

In simple terms, Dependency Injection is a design pattern that helps avoid hard-coded dependencies for some piece of code or software.

The dependencies can be changed at run time as well as compile time. We can use Dependency Injection to write modular, testable and maintainable code.

- **Modular**: The Dependency Injection helps create completely self-sufficient classes or modules
- **Testable**: It helps write testable code easily eg unit tests for example
- **Maintainable**: Since each class becomes modular, it becomes easier to manage it

The Problem: We have dependencies almost always in our code. Consider the following procedural example which is pretty common:



```
function getUsers() {
    global $database;
    return $database->getAll('users');
}
```

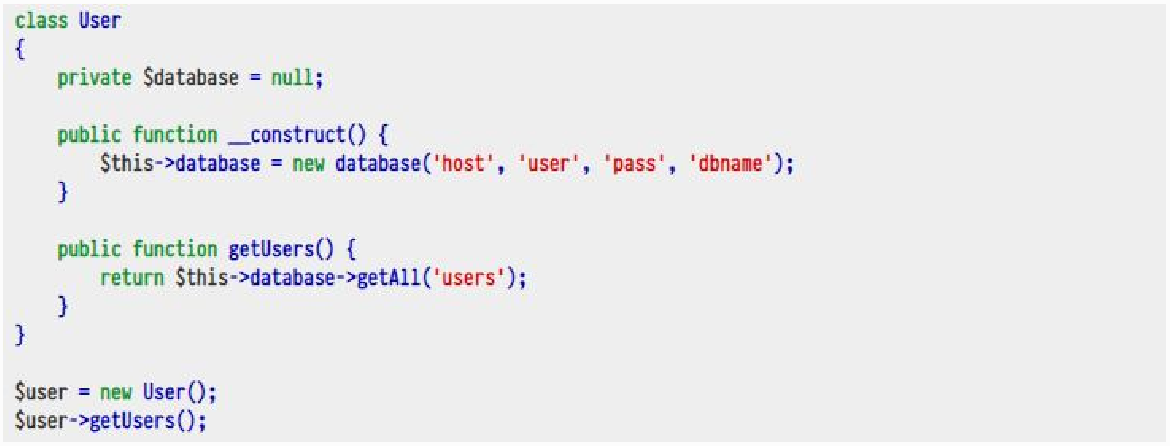
Here the function **`getUsers`** has dependency on the **`$database`** variable (tight coupling). It has some of these problems:

- The function **`getUsers`** needs a **working** connection to *some **database*** . Whether there is successful connection to database or not is the fate of **`getUsers`** function
- The **`$database`** comes from outer scope so chances are it might be overwritten by some other library or code in the same scope in which case function may fail

Of course you could have used the **`try-catch`** constructs but it still doesn't solve the second problem.

Let's consider another example for a class:

Let's consider another example for a class:

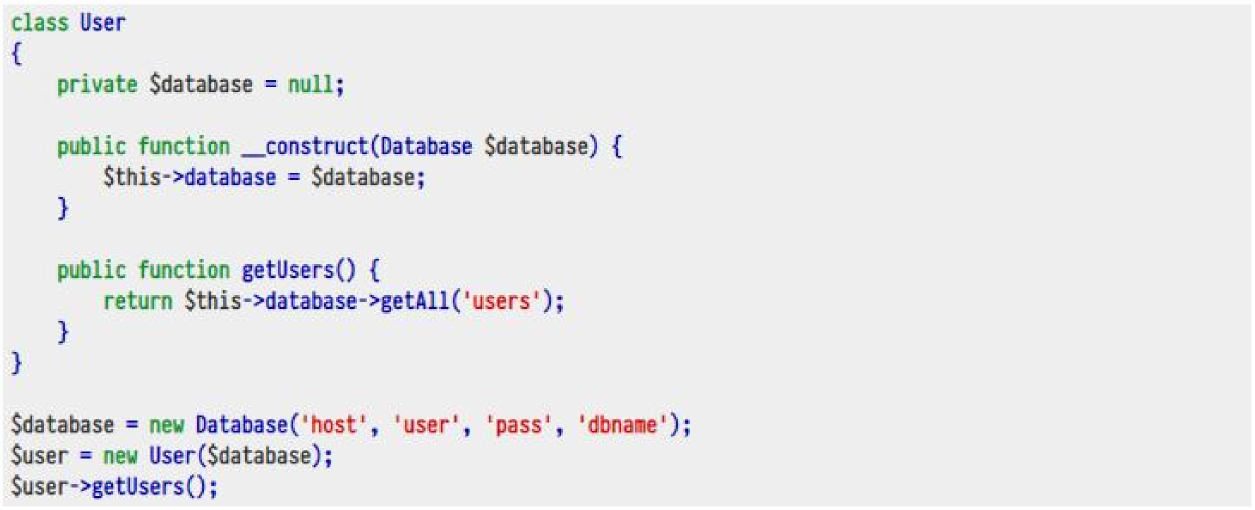


```
class User
{
    private $database = null;

    public function __construct() {
        $this->database = new database('host', 'user', 'pass', 'dbname');
    }

    public function getUsers() {
        return $this->database->getAll('users');
    }
}

$user = new User();
$user->getUsers();
```

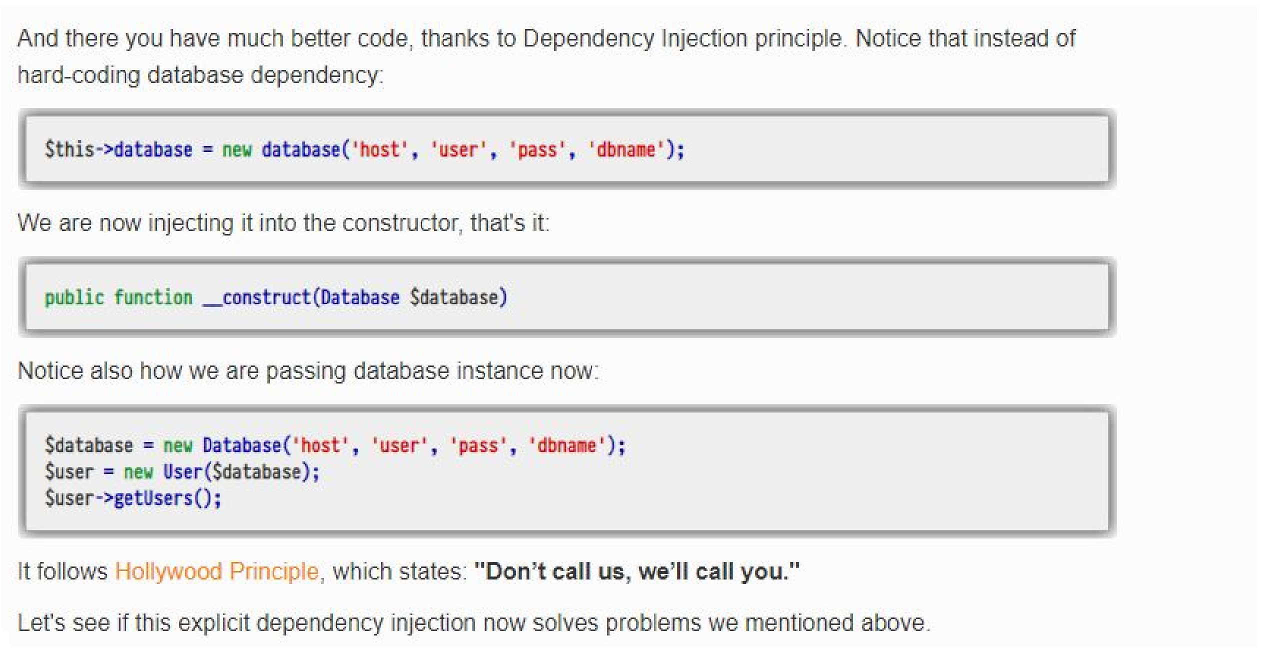
This code again has these problems:

- The class `User` has implicit dependency on the specific `database`. All dependencies should always be **explicit** not implicit. This defeats Dependency inversion principle
- If we wanted to change database credentials, we need to edit the `User` class which is not good; every class should be completely **modular** or black box. If we need to operate further on it, we should actually use its public properties and methods instead of editing it again and again. This defeats Open/closed principle
- Let's assume right now class is using MySQL as database. What if we wanted to use some other type of database ? You will have to modify it.
- The `User` class does not necessarily need to know about database connection, it should be confined to its own functionality only. So writing database connection code in `User` class doesn't make it modular. This defeats the Single responsibility principle. Think of this analogy: A cat knows how to `meow` and a dog knows how to `woof`; you cannot mix them or expect dog to say `meow`. Just like real world, each object of a class should be responsible for its own specific task.
- It would become harder to write unit tests for the `User` class because we are instantiating the database class inside its constructor so it would be impossible to write unit tests for the `User` class without also testing the database class.

Enter Dependency Injection!: Let's see how we can easily take care of above issues by using Dependency Injection. The Dependency Injection is nothing but **injecting a dependency explicitly**. Let's re-write above class:



```
class User
{
    private $database = null;

    public function __construct(Database $database) {
        $this->database = $database;
    }

    public function getUsers() {
        return $this->database->getAll('users');
    }
}

$database = new Database('host', 'user', 'pass', 'dbname');
$user = new User($database);
$user->getUsers();
```


And there you have much better code, thanks to Dependency Injection principle. Notice that instead of hard-coding database dependency:

```php
$this->database = new database('host', 'user', 'pass', 'dbname');
```

We are now injecting it into the constructor, that's it:

```php
public function __construct(Database $database)
```

Notice also how we are passing database instance now:

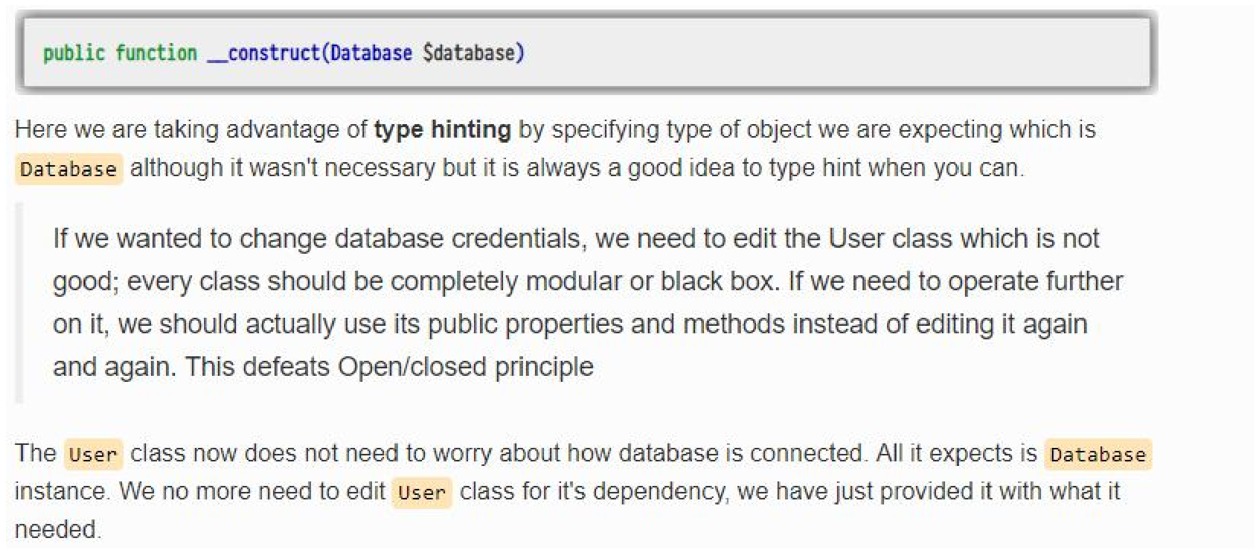
```php
$database = new Database('host', 'user', 'pass', 'dbname');
$user = new User($database);
$user->getUsers();
```

It follows Hollywood Principle, which states: **"Don't call us, we'll call you."**

Let's see if this explicit dependency injection now solves problems we mentioned above.

The class User has implicit dependency on the specific database . All dependencies should always be explicit not implicit. This defeats Dependency inversion principle.

We have already made database dependency explicit by requiring it into the constructor of the `User` class:


```php
public function __construct(Database $database)
```

Here we are taking advantage of **type hinting** by specifying type of object we are expecting which is `Database` although it wasn't necessary but it is always a good idea to type hint when you can.

> If we wanted to change database credentials, we need to edit the User class which is not good; every class should be completely modular or black box. If we need to operate further on it, we should actually use its public properties and methods instead of editing it again and again. This defeats Open/closed principle

The `User` class now does not need to worry about how database is connected. All it expects is `Database` instance. We no more need to edit `User` class for it's dependency, we have just provided it with what it needed.

Let's assume right now class is using MySQL as database. What if we wanted to use some other type of database ? You will have to modify it.

Again, the `User` class doesn't need to know which type of database is used. For the `Database`, we could now create different adapters for different types of database and pass to `User` class. For example, we could create an `interface` that would enforce common methods for all different types of database classes that must be implement by them. For our example, we pretend that interface would enforce to have a `getUser()` method requirement in different types of database classes.

The User class does not necessarily need to know about database connection, it should be confined to its own functionality only. So writing database connection code in User class doesn't make it modular. This defeats the Single responsibility principle. Of course `User` class now doesn't know how database was connected. It just needs a valid connected `Database` instance.

It would become harder to write unit tests for the User class because we are instantiating the database class inside its constructor so it would be impossible to write unit tests for the User class without also testing the database class. If you have wrote unit tests, you know now it will be a breeze to write tests for the `User` class using something like Mockery or similar to create mock object for the `Database`.

Different Ways of Injecting Dependencies

Now that we have seen how useful Dependency Injection is, let's see different ways of injecting dependencies. There are three ways you can inject dependencies:

- Constructor Injection
- Setter Injection
- Interface Injection

Constructor Injection

We have already seen example of **Constructor Injection** in above example. Constructor injection is useful when:

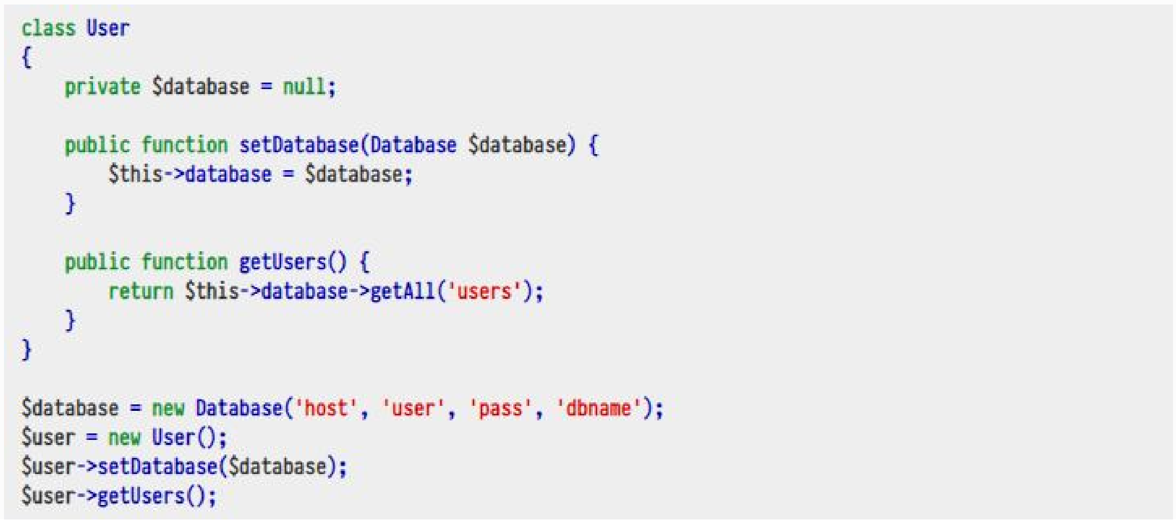
- A dependency is **required** and class can't work without it. By using constructor injection. we make sure all its required dependencies are passed.
- Since constructor is called only at the time of instantiating a class, we can make sure that its dependencies cant be changed during the life time of the object.

Constructor injection suffer from one problem though:

- Since constructor has dependencies, it becomes rather difficult to extend/override it in child classes.

Setter Injection

Unlike Constructor injection which makes it **required** to have its dependencies passed, setter injection can be used to have **optional dependencies**. Let's pretend that our `User` class doesn't require `Database` instance but uses optionally for certain tasks. In this case, you would use a setter method to inject the `Database` into the `User` class something like:



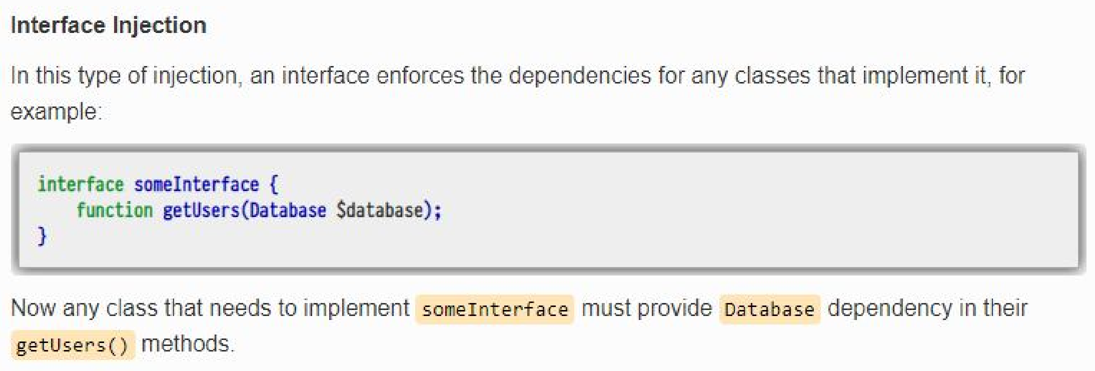
```
class User
{
    private $database = null;

    public function setDatabase(Database $database) {
        $this->database = $database;
    }

    public function getUsers() {
        return $this->database->getAll('users');
    }
}

$database = new Database('host', 'user', 'pass', 'dbname');
$user = new User();
$user->setDatabase($database);
$user->getUsers();
```

As you can see, here we have used `setDatabase()` setter function to inject `Database` dependency into the `User` class. If we needed some other dependency, we could have created one more setter method and injected in the similar fashion.

So Setter Injection is useful when:

- A class needs optional dependencies so it can set itself up with default values or add additional functionality it needs.

Notice that you could also inject dependency via **public property** for a class. So instead of using setter function `$user->setDatabase($database);`, you could also do `$user->database = new Database(...);`



Interface Injection

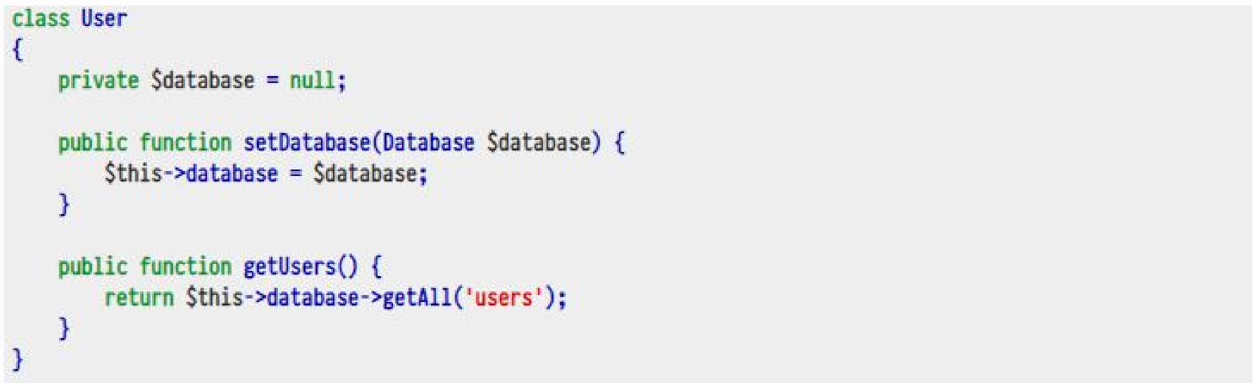
In this type of injection, an interface enforces the dependencies for any classes that implement it, for example:

```
interface someInterface {
    function getUsers(Database $database);
}
```

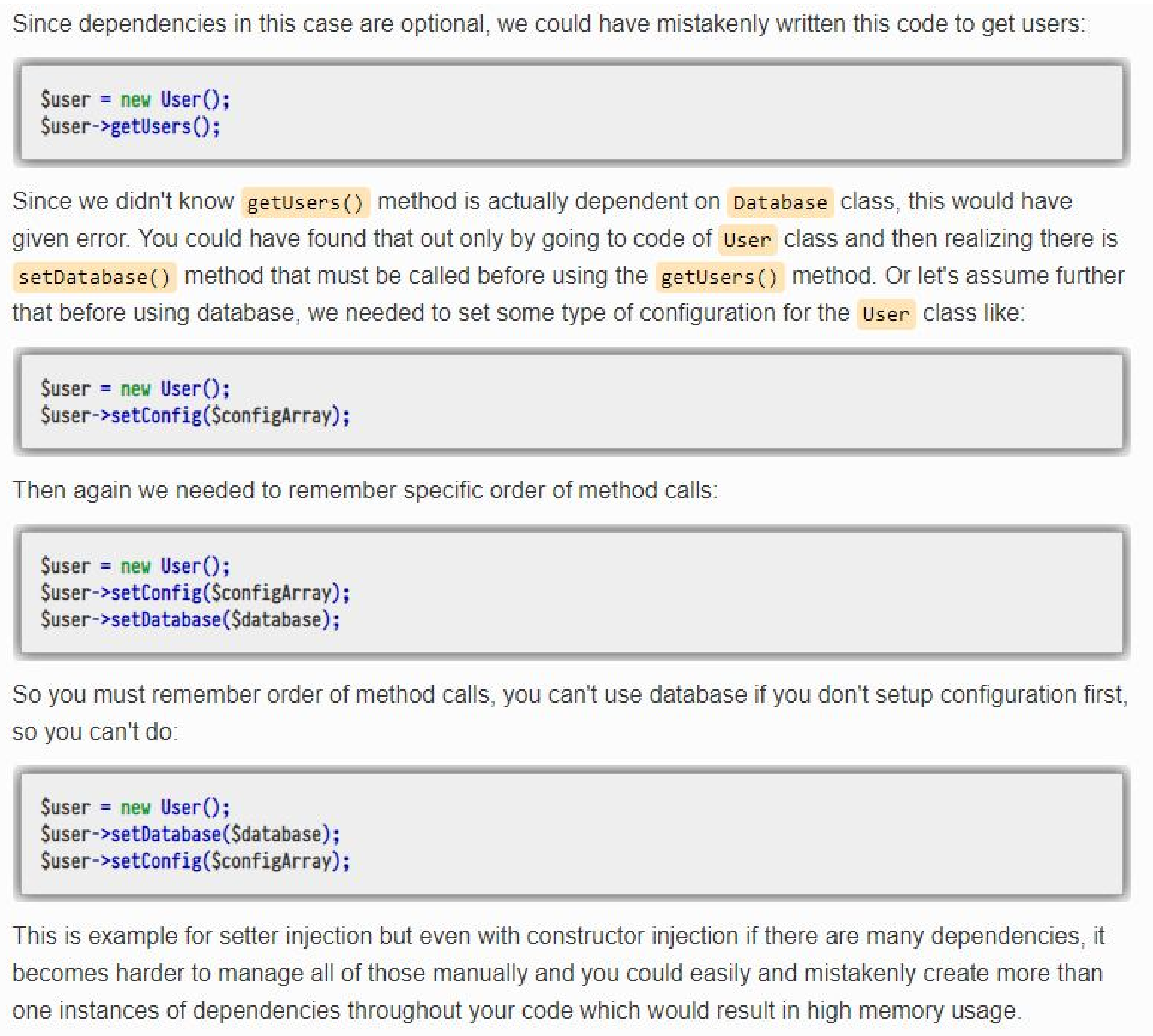
Now any class that needs to implement `someInterface` must provide `Database` dependency in their `getUsers()` methods.

The Problem Again

So for we have seen very contrived example of injecting dependency into a simple class but in real world applications, a class might have many dependencies. It isn't all that easy to manage all those dependencies because you need to KNOW which dependencies are required by a certain class and HOW they need to be instantiated. Let's take example of setter injection:



```
class User
{
    private $database = null;

    public function setDatabase(Database $database) {
        $this->database = $database;
    }

    public function getUsers() {
        return $this->database->getAll('users');
    }
}
```



Since dependencies in this case are optional, we could have mistakenly written this code to get users:

```php
$user = new User();
$user->getUsers();
```

Since we didn't know `getUsers()` method is actually dependent on `Database` class, this would have given error. You could have found that out only by going to code of `User` class and then realizing there is `setDatabase()` method that must be called before using the `getUsers()` method. Or let's assume further that before using database, we needed to set some type of configuration for the `User` class like:

```php
$user = new User();
$user->setConfig($configArray);
```

Then again we needed to remember specific order of method calls:

```php
$user = new User();
$user->setConfig($configArray);
$user->setDatabase($database);
```

So you must remember order of method calls, you can't use database if you don't setup configuration first, so you can't do:

```php
$user = new User();
$user->setDatabase($database);
$user->setConfig($configArray);
```

This is example for setter injection but even with constructor injection if there are many dependencies, it becomes harder to manage all of those manually and you could easily and mistakenly create more than one instances of dependencies throughout your code which would result in high memory usage.

Solution - Dependency Injection Container

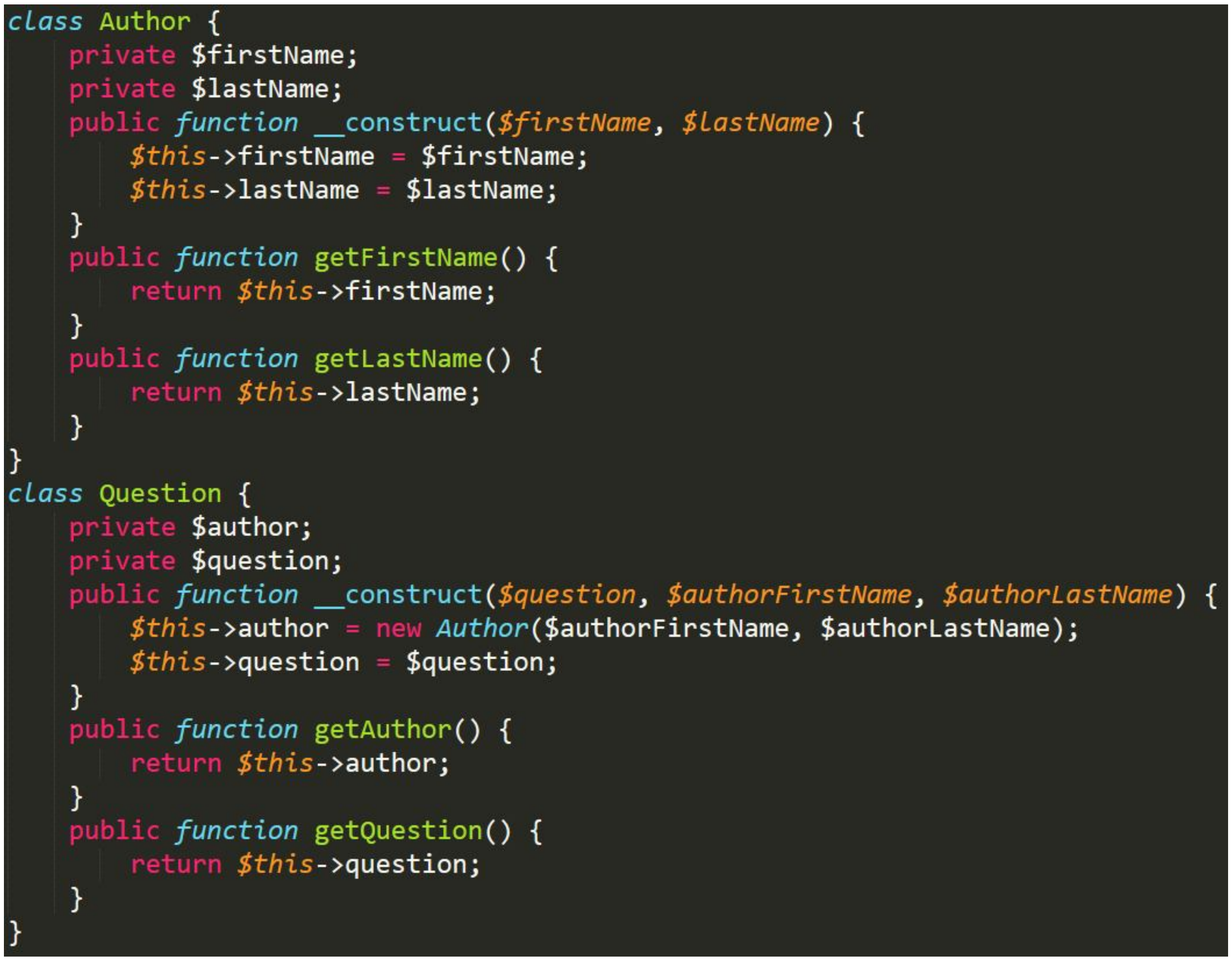
Of course it would be difficult to manage dependencies manually; this is why you need a Dependency Injection Container. A Dependency Injection Container is something that handles dependencies for your class(es) automatically. If you have worked with Laravel or Symfony, you know that their components have dependencies on on other classes. How do they manage all of those dependencies ? Yes they use some sort of Dependency Injection Container.

There are quite some dependency injection containers out there for PHP that can be used for this purpose or you can also write your own. Each container might have bit of different syntax but they perform the same thing under the hood.

So in conclusion, you must always remove hard-coded dependencies from your code and inject them using Dependency Injection instead for its benefits and then have all the injected dependencies managed automatically for you by using some dependency injection container.

Consider Another Example:

Before digging into the subject, let's precisely define what dependency injection is. Let's imagine that you currently work on a "Question and Answers" website, similar to Stack Overflow. You would more than likely create a class, called `Question`, which would contain a member of type `Author`. In 'ye olden days, programmers would have created the `Author` object directly in the `Question` constructor, like this:

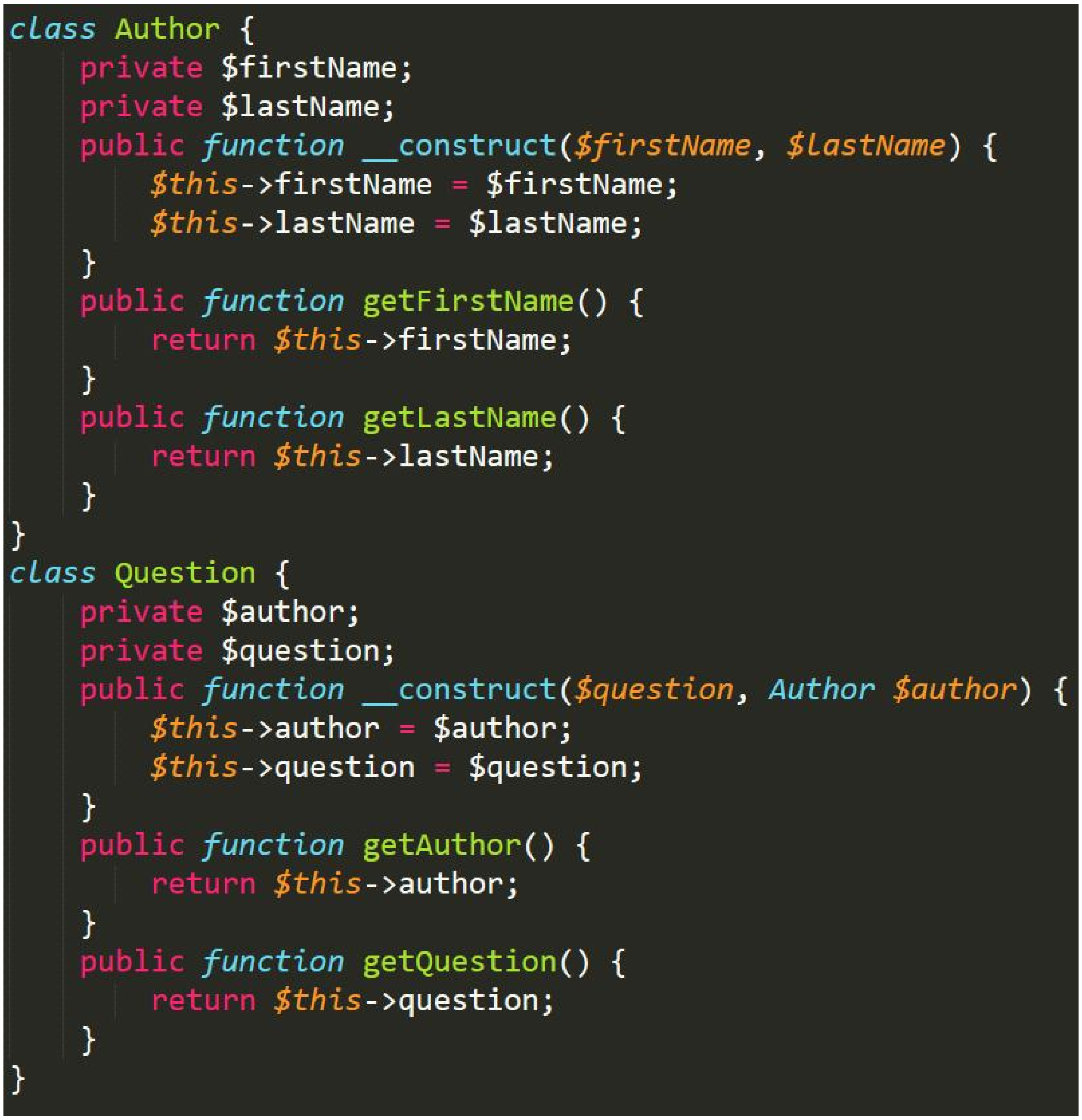


```php
class Author {
    private $firstName;
    private $lastName;
    public function __construct($firstName, $lastName) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }
    public function getFirstName() {
        return $this->firstName;
    }
    public function getLastName() {
        return $this->lastName;
    }
}
class Question {
    private $author;
    private $question;
    public function __construct($question, $authorFirstName, $authorLastName) {
        $this->author = new Author($authorFirstName, $authorLastName);
        $this->question = $question;
    }
    public function getAuthor() {
        return $this->author;
    }
    public function getQuestion() {
        return $this->question;
    }
}
```

While many programmers might call this good code, there are in fact many problems with it:

- The author's information passed to the `Question` constructor has nothing to do inside `Question`'s scope. The name of an author should be inside the `Author` class because it has nothing to do with the question, itself.
- The `Author` class is tightly coupled with the `Question` class. If we add a new parameter to `Author`'s constructor, we then have to modify every class where we create an `Author` object - a tedious and long process, especially in large applications.
- Unit testing the `Question` class creates the unwanted behavior of having to test the `Author` class as well.

Dependency injection alleviates these issues by inserting the dependencies through the dependent class' constructor ("Constructor Injection"). The result is highly maintainable code, which might look like this:

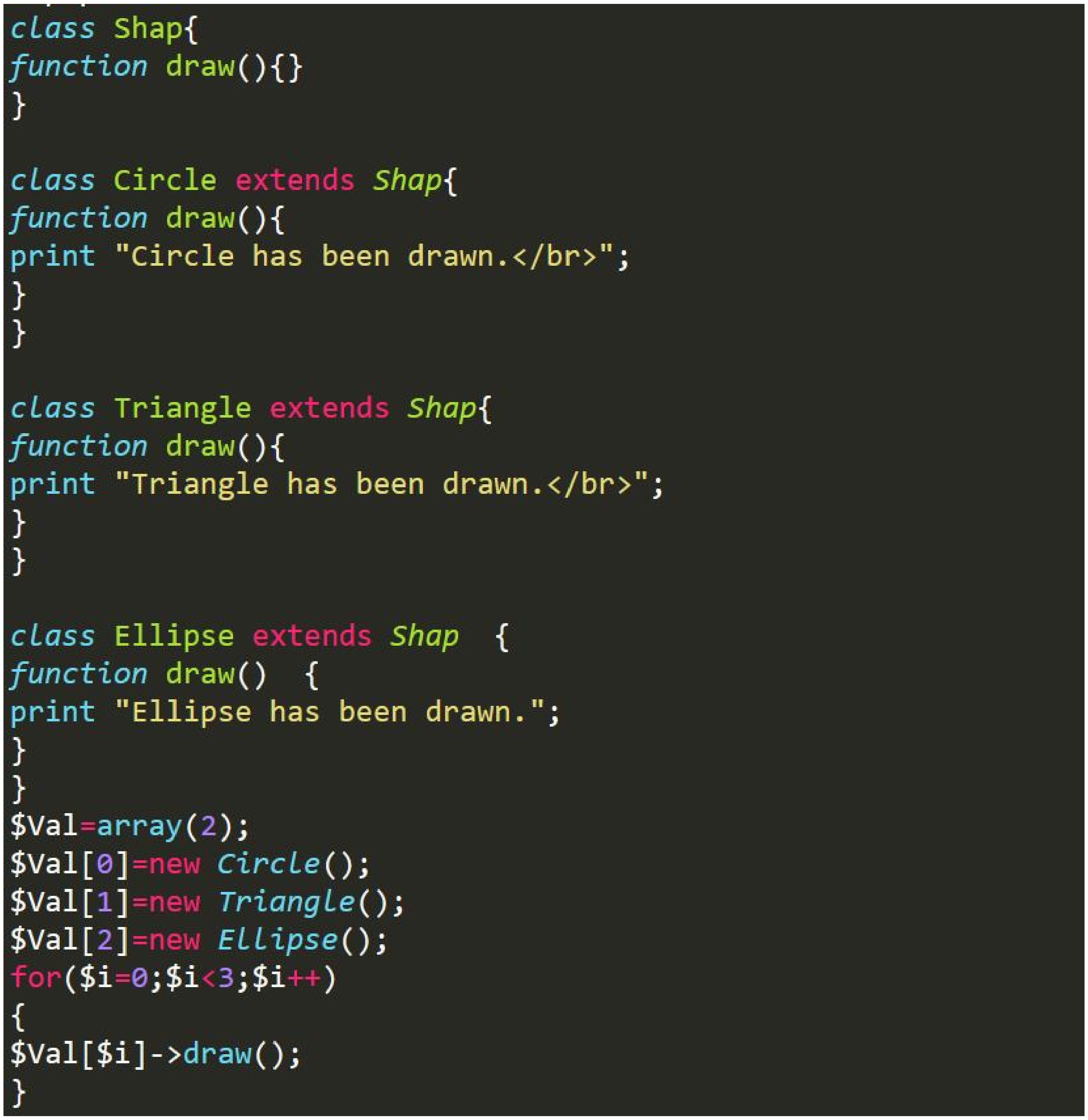


```php
class Author {
    private $firstName;
    private $lastName;
    public function __construct($firstName, $lastName) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }
    public function getFirstName() {
        return $this->firstName;
    }
    public function getLastName() {
        return $this->lastName;
    }
}
class Question {
    private $author;
    private $question;
    public function __construct($question, Author $author) {
        $this->author = $author;
        $this->question = $question;
    }
    public function getAuthor() {
        return $this->author;
    }
    public function getQuestion() {
        return $this->question;
    }
}
```

Polymorphism is derived from two Greek words. Poly (meaning many) and morph (meaning forms). Polymorphism is one of the PHP Object Oriented Programming (OOP) features. In general, polymorphism means the ability to have many forms. If we say it in other words, "**Polymorphism describes a pattern in Object Oriented Programming in which a class has varying functionality while sharing a common interfaces.**". There are two types of Polymorphism; they are:

1. Compile time (function overloading)
2. Run time (function overriding)

But PHP "does not support" compile time polymorphism, which means **function overloading** and **operator overloading**.



```
class Shap{
function draw(){}
}

class Circle extends Shap{
function draw(){
print "Circle has been drawn.</br>";
}
}

class Triangle extends Shap{
function draw(){
print "Triangle has been drawn.</br>";
}
}

class Ellipse extends Shap  {
function draw()  {
print "Ellipse has been drawn.";
}
}
$Val=array(2);
$Val[0]=new Circle();
$Val[1]=new Triangle();
$Val[2]=new Ellipse();
for($i=0;$i<3;$i++)
{
$Val[$i]->draw();
}
```

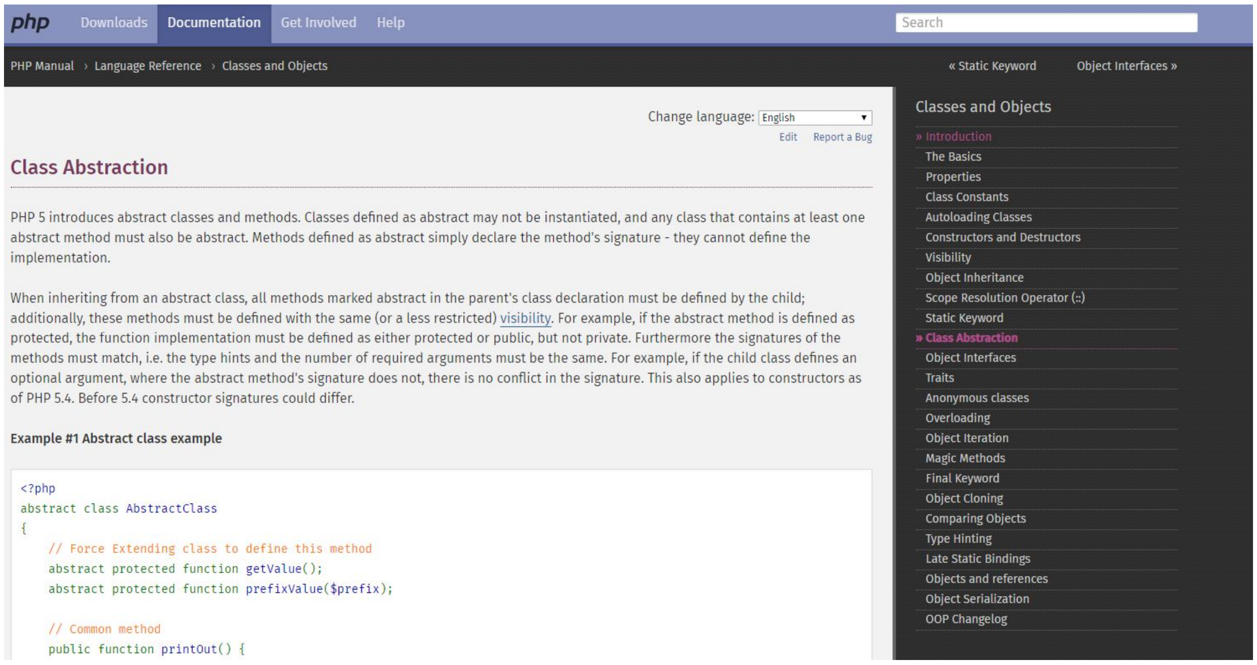
Runtime Polymorphism: The Runtime polymorphism means that a decision is made at runtime (not compile time) or we can say we can have multiple subtype implements for a super class, function overloading is an example of runtime polymorphism. I will first describe function overloading. When we create a function in a derived class with the same signature (in other words a function has the same name, the same number of arguments and the same type of arguments) as a function in its parent class then it is called method overriding.

Example of runtime polymorphism in PHP: In the example given **above** we created one base class called Shap. We have inherit the Shap class in three derived classes and the class names are Circle, Triangle and Ellipse. Each class includes function draw to do runtime polymorphism. In the calling process, we have created an array of length 2 and each index of the array is used to create an object of one class. After that we use a loop that is executed for the length of the array and each value of $i is passed to the object array variable called $Val[$i]. So it is executed three times and it will call the draw() method of every class (but those classes have a draw method, whose object is previously created).

Very USEFULL LINKS

https://codeinphp.github.io/post/late-static-binding-in-php/

https://www.youtube.com/watch?v=k9ak_rv9X0Y&list=PLfdtiItiRHWGXSggf05W-pJbD47-_d8bJ



Class Abstraction

PHP 5 introduces abstract classes and methods. Classes defined as abstract may not be instantiated, and any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature - they cannot define the implementation.

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 (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private. Furthermore the signatures of the methods must match, i.e. the type hints and the number of required arguments must be the same. For example, if the child class defines an optional argument, where the abstract method's signature does not, there is no conflict in the signature. This also applies to constructors as of PHP 5.4. Before 5.4 constructor signatures could differ.

Example #1 Abstract class example

```php
<?php
abstract class AbstractClass
{
    // Force Extending class to define this method
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    // Common method
    public function printOut() {
```

PHP self Vs this208

https://phppot.com/php/php-self-vs-this/

https://www.youtube.com/channel/UCIlPGCkXqjTDlmFPS6oMU9g/playlists cache tutorial

What are the must learn OOPS concepts in PHP?

overriding

- **Class** – This is a programmer-defined data type, 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.

- **Destructor** – refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.