Object Oriented Programming (OOP)

What is OOP?

Object-oriented programming is a model / method of programming design organized around real-world objects.
OOP will allow us to program by manipulating objects.

In programming, an object have the same definition as an object of the real world.
For example, a car has several characteristics:

- The name of the model
- Its color
- The name of the constructor
- Etc ...

These characteristics, we will call them *properties* of the object.

This car can also do several actions:

- Accelerate
- Braking
- Etc. ..

These actions we will call them *methods* of the object.

Until now, you have learned to program in a 'procedural' way. That is to write everything in a file, the instructions one after the other ...

We will now see how to program in an object-oriented way.

The concepts of OOP

There are several concepts / principles that characterize OOP.
We will see two today:

Encapsulation:

This is to hide the details of the implementation and to expose only the methods.

The purpose of encapsulation is to:

- Reduce the complexity of development.
- Protecting the state of an object, access to variables is done via methods. This makes the class flexible and easy to manage.
- The code of an object is easily modifiable and that without worrying to break the code which uses this object.

Inheritance:

This is the relationship between objects. The relationship takes the form of a parent and a child. The child uses the methods defined in the parent.

The main purpose of inheritance is reusability. Indeed, a number of children can inherit from the same parent. This is very useful when you have to provide common features.

Creating a class in PHP

In programming, a class is what defines an object.

A class will therefore group all the properties and methods of our object.

We will take the animal class as an example (and yes, it may seem strange to you, but an animal can be considered as an object: it has properties and methods ...)

Attention a class is not an object!
Class is the concept, the definition, the object is the realization of this concept. In programming, we say that the object is an ***instance*** of the class.

Here is the syntax:

```
class Animal {

}
```

So I start with the keyword *class* followed by the name of my class.
Then everything happens inside the braces.
For now there is nothing but add properties.
For example, an animal is part of a family (mammal, reptile ...) and has a diet (omnivore, carnivore ...).

```
class Animal {

    // Properties
    public $ regime;
    public $ family;

}
```

I just added two properties to my class!
You will notice thekeyword *public* which means that one has access to the properties of the class directly from the outside (notion of scope).

Now that our class is created, we can create an object (an instance of our class) this way:

```
$ my_Animal = new Animal ();
```

To access a property of your pet, you access it in this way:

```
$ my_Animal-> regime;
```

The constructor

A constructor is a *method* that will allow to initialize certain properties when we create our object.

I add a constructor to my Animal class that will wait for two properties:

```
class Animal {

    // Properties
    public $ regime;
    public $ family;

    // Methods
    public function __construct ($ family, $ regime) {
    $ this-> family = $ family;
    $ this-> regime = $ regime
    }

}
```

You will notice the use of the keyword *$ this*.
$ this refers to the instance of the object itself.

That is, $ this-> family refers to the '*family*' *property* of the object. Whereas $ family refers to the argument of the function.

You will also notice that a method is a function in programming.

Now that we have defined a constructor, let's create a new Animal by giving it the values we want:

```
$ my_Animal = new Animal ('carnivore', 'mammal');
```

The toString method

Try to display your newly created object with:

```
echo $ my_Animal;
```

You will see an error! Indeed, an object is not a string of characters.
The echo function can not display the contents of the object.

You can use var_dump () to know the contents of your object, but there is another way to display relevant information about your object.

Thanks to the toString function, you will be able to display your object as a String.

To do this, add the toString () method to your class:

```
public function __toString () {
        return 'My animal is family'. $ this-> family
        . ' and is ' . $ this-> diet;
}
```

Now try again:

```
        echo $ my_Animal;
```

Encapsulation

Reminder: This is to hide the details of the implementation and to expose only the methods.

At first, we must hide the details of the implementation. This translates to 'hide' the properties of my class.
For that I use the keyword '*private*'.

```
class Animal {

        // Properties
        private $ regime;
        private $ family;

        // Methods
        ...
}
```

Then, you must be able to expose only the methods. For this, we will create two methods for each property: one to retrieve the value and another to modify the value.

We will call these methods '*getters*' *and* '*setters*'.

```
class Animal {

        // Properties
        private $ regime;

        // Methods
        public function getRegime () {
                return $ this-> regime;
        }

        public function setRegime ($ regime) {
                $ this-> diet = $ regime;
        }

}
```

We have implemented what is called encapsulation!

Inheritance

Reminder: This is the relationship between objects. The relationship takes the form of a parent and a child. The child inherits and uses the methods defined in the parent.

Let's take the example of our animal class.

A dog is a 'child' of animal because it has the same properties and methods. It even has its own properties and methods.

For example, a dog has a master and can bark.

Inheritance translates to thekeyword *extends.*

```
class Dog extends Animal

        // Properties
        private $ master

        //Methods
        Public__construct ($ family, $ diet, $ master) {
        parent :: __ construct ($ family, $ diet);
        $ this-> master = $ master;
        }
        public function getMaitre () {
                return $ this-> scheme;
        }

        public function setMaster ($ master) {
                $ this-> master = $ master;
        }

}
```

Note well the use of thekeyword *parent ::* which makes it possible to refer to the parent methods.

I can now create a Dog object:

```
$ my_Dog = new Dog ('mammal', 'carnivore', 'john');
```

Public, private, protected

Public indicates that the property or method will be accessible from anywhere.

Private indicates that the property or method will only be accessible within my class. The property or method will not be accessible from the outside.

Protected indicates that the property or method will be accessible within the class and inherited classes.

Override

When one of your classes has a method and child classes (inheritance) and you want to have the same method but with different behaviors, we will have to overwrite the parent method.
The two methods with the same name, the same parameters is called override.

The static method

Static properties and methods can be used without the need to instantiate the class. You can access them directly by using the class name

```
public static function getNumberPatte () {
   return 4;
}
```

If I define a method getNombrePatte () for the class Dog, it will be static because a dog will always have 4 legs whatever the dog that I take. We call a static method by doing:

```
Dog :: getNumberPatte ();
```