Programación orientada a objetos

¿Qué es?

La programación orientada a objetos se basa en crear objetos que contienen una estructura basada en propiedades(datos) y comportamientos(funciones)

Ventajas

- Es más rápido
- Proporciona una mejor estructura
- Más fácil de mantener y modificar
- Es más fácil reutilizar código

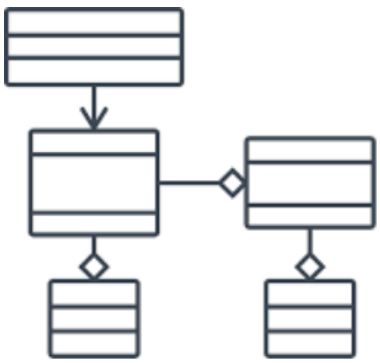
¡Tip!

Si les preguntan cuáles son los pilares de la programación orientada a objetos ustedes responden:

"Los cuatro pilares fundamentales son abstracción, encapsulamiento, polimorfismo y herencia."

¿Qué es UML?

Una forma de representar cada una de las cosas de programación orientada a objetos es usar el lenguaje unificado de modelado (UML).



Profesor Ignacio Quiroga

¿Qué es una clase?

Una clase es una plantilla que podemos usar para representar cosas las cuales tienes propiedades y comportamientos

Téngase en cuenta que una clase siempre se declara con una letra mayúscula al principio.

¿Qué es un objeto?

Un objeto sería una instancia de una de clase.

¡Tip!

Se dice instanciar a la creación de un objeto.

Ejemplo práctico

Vamos entonces a poner un ejemplo práctico para entender esto, el cual lo utilizaremos a lo largo de esta guía.

Nota: Para la guía se usará el lenguaje PHP.

Vamos a suponer que tenemos una persona que va a ser nuestro cliente.

Este cliente tiene sus propiedades (información) y sus comportamientos (funcionalidades) que hará en nuestro sistema

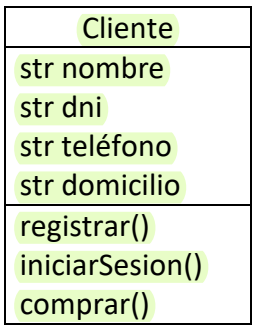
Entonces digamos por ejemplo que algunas de las propiedades de este cliente podrían ser las siguientes:

Nombre, DNI, Teléfono, Domicilio

Y algunas de las funcionalidades o cosas que hará en nuestro sistema será lo siguiente:

Registrarse, Iniciar Sesión, Comprar

Visto en UML tendríamos lo siguiente:



En código se vería algo así:

```php
<?php
class Cliente{
    function __construct() {
        $nombre;
        $dni;
        $telefono;
        $domicilio;
    }
    function registrar_nuevo_cliente(){

    }
    function iniciar_sesion_cliente(){

    }
    function comprar(){

    }
}
?>
```

Note que hemos agregado una función __construct que veremos a continuación.

Constructor

Un constructor es una función que se ejecuta inmediatamente al instanciar una clase.

Encapsulamiento

El encapsulamiento es cuando limitamos el acceso o damos un acceso restringido de una propiedad.

- ✓ public: Se puede acceder a la propiedad o método desde cualquier lugar.
- ✓ protected: Se puede acceder a la propiedad o método dentro de la clase y por clases derivadas de esa clase.
- ✓ private: Sólo se puede acceder a la propiedad o método dentro de la clase.

Siguiendo con el ejemplo:

```php
<?php
class Cliente{
    public $nombre;
    public $dni;
    public $telefono;
    public $domicilio;
    function __construct() {

    }
    private function registrar_nuevo_cliente(){

    }
    private function iniciar_sesion_cliente(){

    }
    private function comprar(){

    }
}
?>
```

Note que la función __construct no tiene modificador, si no se pone nada por defecto es public.

Otra cosa es que la función __construct tiene que ser public sino dará error a la hora de instanciar

Probemos ahora instanciar esta clase y crear un nuevo cliente.

Instanciar no es nada más que crear un objeto de esa clase, la palabra clave para instanciar es new

Para crear una instancia nuevoCliente de esa clase se usa la siguiente forma:

```
$nuevoCliente = new Cliente();
```

```php
<?php
class Cliente{
    public $nombre;
    public $dni;
    public $telefono;
    public $domicilio;
    public function __construct() {

    }
    private function registrar_nuevo_cliente(){

    }
    private function iniciar_sesion_cliente(){

    }
    private function comprar(){

    }
}
$nuevoCliente = new Cliente();
?>
```

Lo siguiente sería pasarle los datos de nuestro cliente:

nombre = “Prueba”

dni = “11.111.111”

teléfono = “11-1111”

domicilio = “Calle 111”

Por ahora le diremos al constructor que les ponga estos datos a las propiedades para ello veremos como acceder a las propiedades tanto dentro de la clase como fuera de la clase

En PHP para acceder a una propiedad fuera de la clase se usa el símbolo “->”

Por ejemplo, para acceder a la propiedad nombre del objeto nuevoCliente sería de la siguiente forma

$nuevoCliente->nombre

Ahora dentro de la clase se usa otra forma de acceder a las propiedades o métodos

Por ejemplo, dentro del constructor para acceder a la propiedad nombre sería de la siguiente forma

$this->nombre

¡Tip!

El this simboliza esta clase es decir que estamos accediendo a algo de la misma clase

Ahora si digámosle al constructor cuál es la información con la que trabajaremos

```php
<?php
class Cliente{
    public $nombre;
    public $dni;
    public $telefono;
    public $domicilio;
    public function __construct() {
        $this->nombre = "Prueba";
        $this->dni = "11.111.111";
        $this->telefono = "11-1111";
        $this->domicilio = "Calle 111";
    }
    private function registrar_nuevo_cliente(){
    }
    private function iniciar_sesion_cliente(){
    }
    private function comprar(){
    }
}
$nuevoCliente = new Cliente();
echo $nuevoCliente->nombre;
?>
```

Como último vamos a hacerlo mejor pasando los datos por parámetros a la clase y que el constructor se encargue de poner los datos a cada propiedad, esto nos va a permitir pasarle cualquier tipo de dato y que genere un objeto con información que se le otorgue.

```php
<?php
class Cliente{
    public $nombre;
    public $dni;
    public $telefono;
    public $domicilio;
    public function __construct($nombre,$dni,$telefono,$domicilio) {
        $this->nombre = $nombre;
        $this->dni = $dni;
        $this->telefono = $telefono;
        $this->domicilio = $domicilio;
    }
    private function registrar_nuevo_cliente(){
    }
    private function iniciar_sesion_cliente(){
    }
    private function comprar(){
    }
}
$nuevoCliente = new Cliente("Prueba","11.111.111","11-1111","Calle 111");
echo $nuevoCliente->nombre;
?>
```

¿Qué pasaría si trato de acceder por fuera de la clase a la función comprar()?

Si trato yo de acceder por medio de $nuevoCliente->comprar() nos diría lo siguiente

Fatal error: Uncaught Error: Call to private

Esto pasa porque nosotros hemos dejado declarado que la función es privada por lo cual sólo podrá ser accedida dentro de la clase.

Herencia

Es cuando una clase deriva de otra clase.

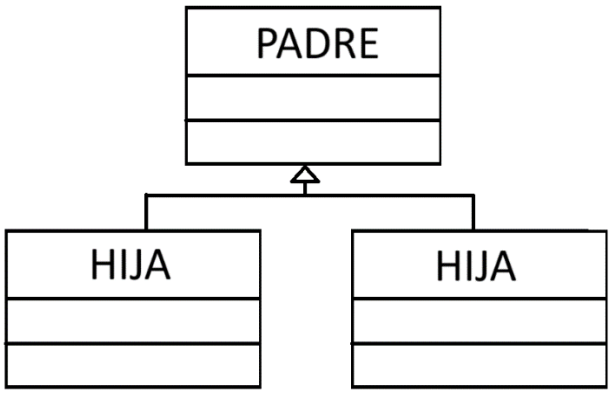
La clase secundaria heredará todas las propiedades y métodos públicos y protegidos de la clase principal. Además, puede tener sus propias propiedades y métodos.

Clases padre

Son las clases que luego pasaran sus propiedades y comportamientos a las clases hijas

Clases hijas

Son las que heredan propiedades y comportamientos de la clase padre



Para que una clase hija herede de una clase en PHP se usa la palabra clave ***extends***

Siguiendo con el ejemplo:

Ahora supongamos que también tenemos trabajadores los cuales utilizan las mismas propiedades

Nombre, DNI, Teléfono, Email, Domicilio

Pero tiene otras funcionalidades iguales pero otras distintas

Registrarse, Iniciar Sesión, Vender

Para aprovechar lo que nos ofrece la programación orientada a objetos lo ideal sería utilizar la herencia produciendo por ejemplo una clase padre la cual tenga las propiedades y funciones que se utilizan en las clases y hereden de esta

Si queremos que la clase Cliente herede de la clase Persona se usaría lo siguiente:

```
class Cliente extends Persona
```

Veamos entonces como sería una forma de manejar esta situación

Propondremos una solución: “Crear una clase padre llamada Personas y dos clases hijas (Cliente y Empleado) “.

```php
<?php
class Persona{
    public $nombre;
    public $dni;
    public $telefono;
    public $domicilio;
    protected function registrar(){
    }
    protected function iniciar(){
    }
}
class Cliente extends Persona{
    public function __construct($nombre,$dni,$telefono,$domicilio) {
        $this->nombre = $nombre;
        $this->dni = $dni;
        $this->telefono = $telefono;
        $this->domicilio = $domicilio;
    }
    private function comprar(){
    }
}
class Empleado extends Persona{
    public function __construct($nombre,$dni,$telefono,$domicilio) {
        $this->nombre = $nombre;
        $this->dni = $dni;
        $this->telefono = $telefono;
        $this->domicilio = $domicilio;
    }
    private function vender(){
    }
}
$nuevoCliente = new Cliente("Prueba","11.111.111","11-1111","Calle 111");
$nuevoEmpleado = new Empleado("Empleado","22.222.222","22-2222","Calle 222");
echo $nuevoCliente->nombre;
?>
```

Note lo siguiente:

- ✓ Las propiedades las pusimos public, entonces por fuera de la clase podemos acceder a ella
- ✓ La función registrar e iniciar son protected por lo cual sólo la podrá acceder o la clase persona o las clases que hereden de ésta
- ✓ Las funciones comprar y vender son privadas, comprar() sólo podrá ser accedida en Cliente y vender() sólo podrá ser accedida en Empleado

Polimorfismo

Es la capacidad de un objeto de comportarse de forma diferente según el contexto que se encuentre.

Hay varios tipos de polimorfismo veremos algunos casos para explicarlo.

Primero supongamos que una clase hereda de otra y quiere sobrescribir un método.

```php
<?php
class Persona{
    function registro(){
        echo "Nuevo registro";
    }
}
class Cliente extends Persona{
    #[Override]
    function registro(){
        echo "Registro de Cliente";
    }
}
class Empleado extends Persona{
    #[Override]
    function registro(){
        echo "Registro de Empleado";
    }
}

$nuevoCliente = new Cliente();
$nuevoCliente->registro("Prueba");
$nuevoEmpleado = new Empleado();
$nuevoEmpleado->registro("Prueba");
?>
```

Otra forma, implica que varias clases con métodos con el mismo nombre, otra clase podrá decidir que hace según el parámetro pasado

```php
<?php

class Cliente{
    function registro(){
        echo "Registro de Cliente";
    }
}
class Empleado{
    function registro(){
        echo "Registro de Empleado";
    }
}

class Nuevo{
    function registro($object){
        $object->registro();
    }
}
$nuevoCliente = new Cliente();
$nuevo = new Nuevo();
$nuevo->registro($nuevoCliente);
?>
```

Hay un tipo de polimorfismo que se llama sobrecarga, trata de crear varios métodos con el mismo nombre, pero con diferente cantidad de parámetros.

En PHP no se puede esto, pero dejo un ejemplo de cómo sería si existiese

```php
class Cliente{
    function registro(){
        echo "Registro de Cliente";
    }
    function registro($nombre){
        echo "Registro ".$nombre;
    }
    function registro($nombre,$tipo){
        echo "Registro de ".$tipo.": ".$nombre;
    }
}
$nuevoCliente = new Cliente();
$nuevoCliente->registro();
$nuevoCliente->registro("Prueba");
$nuevoCliente->registro("Prueba","admin");
```

La alternativa que existe a esto sería lo siguiente pero no es polimorfismo:

```php
class Cliente{
    function registro(){
        $args = func_get_args();
        switch(count($args)){
            case 0:
                echo "Registro de Cliente";
                break;
            case 1:
                echo "Registro ".$args[0];
                break;
            case 2:
                echo "Registro de ".$args[1].": ".$args[0];
                break;
        }
    }
}
$nuevoCliente = new Cliente();
$nuevoCliente->registro();
$nuevoCliente->registro("Prueba");
$nuevoCliente->registro("Prueba","admin");
```

Métodos y propiedades estáticas

Sirven para poder hacer llamados sin tener que instanciar la clase

```php
<?php

class Persona{
    public static $pagina = "PAGINA POO";

    public static function saludar($nombre){
        echo "Bienvenido ".$nombre;
    }
}
echo Persona::$pagina."<br>";
Persona::saludar("Prueba");
?>
```

Clases abstractas

Son clases padres que definen métodos, pero su funcionalidad la dará la clase hija

Si varias clases heredan de la clase abstracta estamos usando Polimorfismo.

```php
<?php
abstract class Persona{
    abstract function registrar();
}
class Cliente extends Persona{
    function registrar(){
        echo "Registro Cliente";
    }
}
class Empleado extends Persona{
    function registrar(){
        echo "Registro Expleado";
    }
}
?>
```

¡Todos los métodos tienen que ser implementados por las clases hijas!

Interfaces

Las interfaces permiten especificar cuales tienen que ser los métodos que tienen que ser implementados.

Si varias clases usan la misma interfaz estamos usando polimorfismo.

```php
<?php
interface Persona{
    public function registro();
}
class Cliente implements Persona{
    public function registro(){
        echo "Registro Cliente";
    }
}
class Empleado implements Persona{
    public function registro(){
        echo "Registro Empleado";
    }
}
?>
```

¡Todos los métodos tienen que ser implementados por la clase que usa la interfaz!

Ahora la pregunta es:

¿Pero cuando usamos interfaces y cuándo clases abstractas?

Bueno tomemos en cuenta lo siguiente:

Interfaz

- ⊗ No se pueden declarar propiedades
- ⊗ No podemos definir funcionalidades
- ✔ Podemos implementar múltiples interfaces

```php
<?php
interface Persona{
    function registro();
    function saludar();
}
class Cliente implements Persona{
    private $nombre;
    function __construct($nombre){
        $this->nombre = $nombre
    }
    public function saludar(){
        echo "Bienvenido ".$this->nombre."<br>"
    }
    public function Registro(){
        echo "Registro Cliente";
    }
}
class Empleado implements Persona{
    private $nombre;
    function __construct($nombre){
        $this->nombre = $nombre
    }
    public function saludar(){
        echo "Bienvenido ".$this->nombre."<br>"
    }
    public function registro(){
        echo "Registro Empleado";
    }
}
?>
```

```php
<?php
interface Persona{
    function registro();
    function saludar();
}
interface Admin{
    function gestionar();
}
class Empleado implements Persona,Admin{
    private $nombre;
    function __construct($nombre){
        $this->nombre = $nombre;
    }
    public function saludar(){
        echo "Bienvenido ".$this->nombre."<br>";
    }
    public function registro(){
        echo "Registro Empleado";
    }
    public function gestionar(){
        echo "Empleado gestiona";
    }
}
?>
```

Abstractas

- ✔ Se pueden declarar propiedades
- ✔ Se pueden definir funcionalidades
- ⊗ No se pueden heredar múltiples clases abstractas

```php
abstract class Persona{
    protected $nombre;
    abstract function registrar();
    function __construct($nombre){
        $this->nombre = $nombre;
    }
    function saludar(){
        echo "Bienvenido ".$this->nombre."<br>";
    }
}
class Cliente extends Persona{
    function registrar(){
        echo "Registro Cliente";
    }
}
class Empleado extends Persona{
    function registrar(){
        echo "Registro Empleado";
    }
}
?>
```

Profesor Ignacio Quiroga

Traits

Teniendo en cuenta que PHP no nos deja heredar de varias clases hay una opción que podemos usar y son los traits.

Son pequeños códigos que funcionan como clases los cuáles los podemos usar dentro de otra clase y se pueden usar múltiples a la vez.

```php
<?php
interface Persona{
    public function comprar();
}
class Cliente implements Persona{
    use message1{mensaje1 as private bienvenida;}
    use message2;
    public function comprar(){
        $this->bienvenida();
        echo "Comprando";
    }
}
trait message1{
    public function mensaje1(){
        echo "POO INTERFCE<br>";
    }
}
trait message2{
    public function mensaje2(){
        echo "Bienvenido a las compras<br>";
    }
}
?>
```

Namespaces

Los espacios de nombres no permiten organizar mejor las clases y también utilizar el mismo nombre de clase múltiples veces

```php
<?php
namespace Cliente;
class Cliente{
    public $nombre = "Prueba";
    public $dni = "11.111.111";
    public $telefono = "11-1111";
    public $domicilio = "Prueba 111";
    private function registrar(){
    }
    private function iniciar(){
    }
}
?>
```

```php
<?php
namespace Cliente2;
class Cliente{
    public $nombre = "Prueba 2";
    public $dni = "22.222.22";
    public $telefono = "22-2222";
    public $domicilio = "Prueba 222";
    private function registrar(){
    }
    private function iniciar(){
    }
}
?>
```

```php
<?php
require_once("namespace.php");
require_once("namespace2.php");
use Cliente2\Cliente as miCliente;
$nuevoCliente = new Cliente\Cliente();
$nuevoCliente2 = new miCliente();
echo $nuevoCliente->nombre;
echo $nuevoCliente2->nombre;
?>
```

Autoloaders

Una cosa que caemos al estar usando clases es que cada vez que necesitemos instanciar una clase debemos incluir el archivo donde este definida la clase y esto nos puede llenar de líneas de código

Para ello se usan los autoloaders, su función es muy simple, cuando haya un problema al intentar instanciar una clase intentarán usar una función que nosotros hayamos creado para intentar solucionar el problema.

Entonces hay una táctica muy fácil que es una vez que tratemos de instanciar la clase incluir el archivo de la clase de esta manera

```php
<?php
spl_autoload_register('misClases');
function misClases($nombreClase){
    include_once $nombreClase.'.php';
}
?>
```

Y combinando namespaces y autoloaders podemos pensar lo siguiente:

- Poner de namespace la ruta al archivo
- Llamar al archivo del mismo nombre de la clase

¿Qué logramos con esto?

Bueno supongamos que tengo el Cliente en una carpeta Models\Cliente y dentro de esa carpeta está el archivo Cliente.php y tiene de namespace la ruta y la clase se llama igual:

```php
<?php
namespace Models\Cliente;
class Cliente{
    public $nombre = "Prueba";
    public $dni = "11.111.111";
    public $telefono = "11-1111";
    public $domicilio = "Prueba 111";
    private function registrar(){
    }
    private function iniciar(){
    }
}
?>
```

Bueno usando de esta forma el autoloader se encarga sólito de incluir cualquier archivo cuando se instancie una clase

```php
<?php
require_once("autoloader.php");
$nuevoCliente = new Models\Cliente\Cliente();
?>
```

¡Tip!

¡Para que funcione siempre el archivo se tiene que llamar igual que la clase!

Profesor Ignacio Quiroga