

¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

A lo largo de la historia, han ido apareciendo diferentes paradigmas de programación. Lenguajes secuenciales como COBOL o procedimentales como Basic o C, se centraban más en la lógica que en los datos. Otros más modernos como Java, C# y Python, utilizan paradigmas para definir los programas, siendo la Programación Orientada a Objetos la más popular.

Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

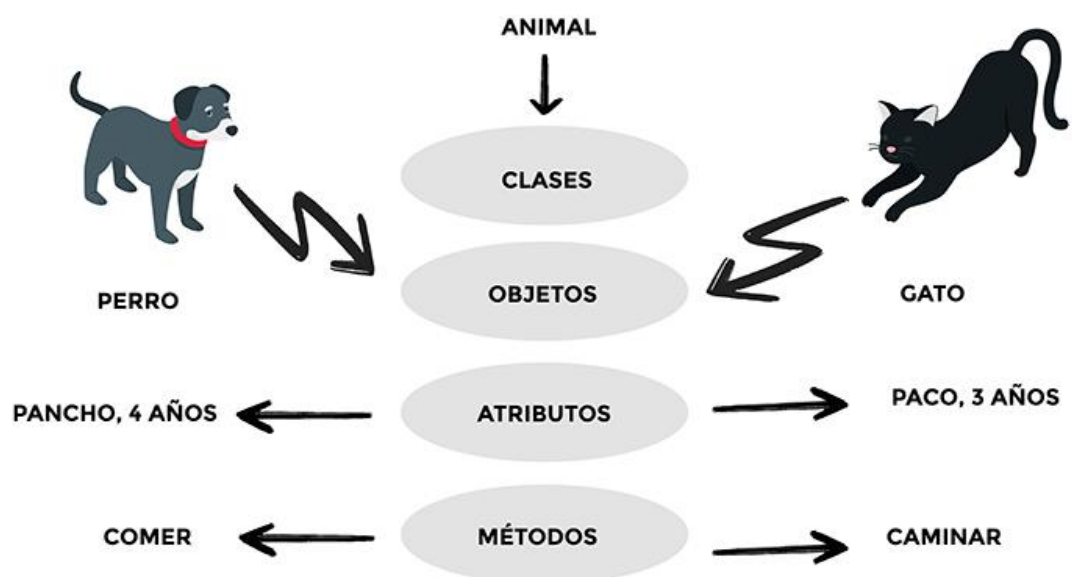
Un programador diseña un programa de software organizando piezas de información y comportamientos relacionados en una plantilla llamada clase. Luego, se crean objetos individuales a partir de la plantilla de clase. Todo el programa de software se ejecuta haciendo que varios objetos interactúen entre sí para crear un programa más grande.

Clases, objetos e instancias

¿Cómo se crean los programas orientados a objetos? Resumiendo mucho, consistiría en hacer clases y crear objetos a partir de estas clases. Las clases forman el modelo a partir del que se estructuran los datos y los comportamientos. El primer y más importante concepto de la POO es la distinción entre clase y objeto.

Una clase es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse 'animal' y tener una serie de atributos, como 'nombre' o 'edad' (que normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

Un ejemplo sencillo de un objeto, como decíamos antes, podría ser un animal. Un animal tiene una edad, por lo que creamos un nuevo atributo de 'edad' y, además, puede envejecer, por lo que definimos un nuevo método. Datos y lógica. Esto es lo que se define en muchos programas como la definición de una clase, que es la definición global y genérica de muchos objetos.



Con la clase se pueden crear instancias de un objeto, cada uno de ellos con sus atributos definidos de forma independiente. Con esto podríamos crear un gato llamado Paco, con 3 años de edad, y otro animal, este tipo perro y llamado Pancho, con una de edad de 4 años. Los dos están definidos por la clase animal, pero son dos instancias distintas. Por lo tanto, llamar a sus métodos puede tener resultados diferentes. Los dos comparten la lógica, pero cada uno tiene su estado de forma independiente.

Todo esto, junto con los principios que vamos a ver a continuación, son herramientas que nos pueden ayudar a escribir un código mejor, más limpio y reutilizable.

Principios de la Programación Orientada a Objetos

- **La encapsulación**

La encapsulación contiene **toda la información importante de un objeto dentro del mismo** y solo expone la información seleccionada al mundo exterior.

Esta propiedad permite asegurar que la información de un objeto esté oculta para el mundo exterior, agrupando en una Clase las características o atributos que cuentan con un acceso privado, y los comportamientos o métodos que presentan un acceso público.

La encapsulación de cada objeto es responsable de su propia información y de su propio estado. La única forma en la que este se puede modificar es mediante los propios métodos del objeto. Por lo tanto, los atributos internos de un objeto deberían ser **inaccesibles desde fuera**, pudiéndose modificar sólo llamando a las funciones correspondientes. Con esto conseguimos mantener el estado a salvo de usos indebidos o que puedan resultar inesperados.

Usamos de ejemplo un coche para explicar la encapsulación. El coche comparte información pública a través de las luces de freno o intermitentes para indicar los giros (interfaz pública). Por el contrario, tenemos la interfaz interna, que sería el mecanismo propulsor del coche, que está oculto bajo el capó. Cuando se conduce un automóvil es necesario indicar a otros conductores tus movimientos, pero no exponer datos privados sobre el tipo de carburante o la temperatura del motor, ya que son muchos datos, lo que confundiría al resto de conductores.

- Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior.
- El encapsulamiento consiste en agrupar en una Clase las características(atributos) con un acceso privado y los comportamientos (métodos) con un acceso público.
- Acceder o modificar los miembros de una clase a través de sus métodos.



Ejemplo:

Cuando no hay encapsulamiento se pueden presentar problemas:

```

public class UniversityStudent {
    int id;
    String name;
    String gender;
    String university;
    String career;
    int numSubjects;
    public UniversityStudent(int id, String name, String gender,
        String university, String career, int numSubjects) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.university = university;
        this.career = career;
        this.numSubjects = numSubjects;
    }
    void inscribeSubjects() {
        // TODO: implement
    }
    void cancelSubjects() {
        // TODO: implement
    }
    void consultRatings() {
        // TODO: implement
    }
}

```

```

class Main{
    public static void main(String []args){
        UniversityStudent student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
        student.numSubjects = -15; //se accede a un atributo de forma directa (rompemos el principio de encapsulamiento)
        System.out.println("Numero de materias : "+student.numSubjects);
    }
}

```

```

run:
Numero de materias : -15
BUILD SUCCESSFUL (total time: 0 seconds)

```

Podemos ver que cuando no hay encapsulamiento los atributos pueden tomar valores inconsistentes, lo cual seria fatal para cualquier sistema.

Aplicando el encapsulamiento tenemos:

```

public class UniversityStudent {
    private int id;
    private String name;
    private String gender;
    private String university;
    private String career;
    private int numSubjects;
    public UniversityStudent(int id, String name, String gender,
        String university, String career, int numSubjects) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.university = university;
        this.career = career;
        this.numSubjects = numSubjects;
    }

    public void inscribeSubjects() {
        // TODO: implement
    }
    public void cancelSubjects() {
        // TODO: implement
    }
    public void consultRatings() {
        // TODO: implement
    }
}

```

```

    public void setNumSubjects(int numSubjects){
        if( numSubjects < 0 || numSubjects > 10 )
            System.out.println("Numero invalido de materias");
        else
            this.numSubjects = numSubjects;
    }
    public int getNumSubjects(){
        return numSubjects;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public String getGender() {
        return gender;
    }
    public String getUniversity() {
        return university;
    }
    public String getCareer() {
        return career;
    }
}

```

```

class Main{
    public static void main(String []args){
        UniversityStudent student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
        student.setNumSubjects(-15);
        System.out.println("Numero de materias "+student.getNumSubjects());
        student.setNumSubjects(6);
        System.out.println("Numero de materias "+student.getNumSubjects());
    }
}

```

```

run:
Numero invalido de materias
Numero de materias 8
Numero de materias 6
BUILD SUCCESSFUL (total time: 0 seconds)

```

Con el resultado anterior vemos que el encapsulamiento nos ayuda a proteger la integridad de los datos y nos asegura que los atributos de nuestra clase solo podran ser accedidos a traves de los metodos definidos en dicha clase.

Ejemplo de encapsulamiento en C++:

```

class UniversityStudent {
private:
    int id;
    string name;
    string gender;
    string university;
    string career;
    int numSubjects;

public:
    UniversityStudent(int _id,string _name,string _gender,
        string _university,string _career,int _numSubjects){
        id = _id;
        name = _name;
        gender = _gender;
        university = _university;
        career = _career;
        numSubjects = _numSubjects;
    };
    void inscribeSubjects() {
        // TODO: implement
    }
    void cancelSubjects() {
        // TODO: implement
    }
    void consultRatings() {
        // TODO: implement
    }
}

```

```

void setNumSubjects(int _numSubjects){
    if(_numSubjects < 0 || _numSubjects > 10)
        cout<<"Numero invalido de materias";
    else
        numSubjects = _numSubjects;
}

int getNumSubjects(){
    return numSubjects;
}

int getId() {
    return id;
}

string getName() {
    return name;
}

string getGender() {
    return gender;
}

string getUniversity() {
    return university;
}

string getCareer() {
    return career;
}
};

```

Ejemplo de encapsulamiento en Python:

```

class UniversityStudent:

    def __init__(self,id,name,gender,university,career,numsubjects):
        self.__id = id
        self.__name = name
        self.__gender = gender
        self.__university = university
        self.__career = career
        self.__numsubjects = numsubjects

    def inscribeSubjects(self):
        pass

    def cancelSubjects(self):
        pass

    def consultRatings(self):
        pass

```

```

def setNumSubjects(self,numSubjects):
    if numSubjects < 0 or numSubjects > 10:
        print "Numero invalido de materias"
    else:
        self.__numsubjects = numSubjects

def getNumSubjects(self):
    return self.__numsubjects

def getId(self):
    return self.__id

def getName(self):
    return self.__name

def getGender(self):
    return self.__gender

def getUniversity(self):
    return self.__university;

def getCareer(self):
    return self.__career

```

Mensaje

Un mensaje es una comunicación dirigida desde un objeto A ordenando a otro objeto B que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

Los mensaje son los que permiten la comunicacion entre objetos.

Ejemplo:



Constructor

Cuando se crea un objeto es necesario inicializar sus variables con valores coherentes. Esto se hace por medio de un constructor que tiene las siguientes características: Se llama igual que la clase. Retorna una instancia de una clase Pueden existir varios, pero siguiendo las reglas de la sobrecarga de funciones. De entre los que existan, tan sólo uno se ejecutará al crear un objeto de la clase.

```

1
2 class Point:
3     """ Point class represents and manipulates x,y coords. """
4
5     def __init__(self, x=0, y=0):
6         """ Create a new point at x, y """
7         self.x = x
8         self.y = y
9
10    # Other statements outside the class continue below here.
11
12    q = Point(

```

x=0, y=0
Create a new point at x, y

```

1 <?php
2 class ClaseEjemplo {
3     function __construct() {
4         print "Constructor Ejemplo\n";
5     }
6 }
7
8 class SubClaseEjemplo extends ClaseEjemplo {
9     function __construct() {
10         parent::__construct();
11         print " Constructor Sub Clase \n";
12     }
13 }
14

```

Constructor en Python y Constructor en PHP

Destructor

Es un método de una clase cuyo fin es eliminar un objeto de una clase.


```

class Estudiante {
private:
    static int numero_estudiantes = 0;
    char* nombre;
    int edad;
    char* direccion;
    char* carrera;
public:
    Estudiante();
    void estudiar();
    void almorzar();
    ~Estudiante();
};

Estudiante::Estudiante() {
    numero_estudiantes++;
}

Estudiante::~~Estudiante() {
    numero_estudiantes--;
}

```

Constructor en C++ y su respectivo destructor en C++

Métodos de acceso

Son los métodos o funciones que permiten obtener o modificar los atributos de un objeto. Además, estos métodos tienen la limitación de proveer información acerca del estado de un objeto específicamente sus propiedades.

```

public class CuentaBancaria {
    //Declaración de Atributos
    private String numeroCuenta;
    private String nombreCliente;
    private float interesAnual;
    private float saldo;

    /**...*/
    → public String getNumeroCuenta() {
        return numeroCuenta;
    }













    /**...*/
    → public void setNumeroCuenta(String numeroCuenta) {
        this.numeroCuenta = numeroCuenta;
    }
}

```

En el ejemplo anterior se pueden visualizar dos métodos de acceso de la clase CuentaBancaria. El método getNumeroCuenta() retorna el atributo numeroCuenta y no tiene ninguna otra especialidad. El método setNumeroCuenta(String numeroCuenta) actualiza el atributo numeroCuenta de acuerdo al valor pasado por parámetro.

Modificadores de acceso

Son palabras claves de los lenguajes de programación orientado a objetos para configurar la accesibilidad a las clases, métodos o propiedades.

Visibilidad	Public	Private	Protected	Default*
Desde la misma clase				
Desde una subclase				
Desde otra clase (no subclase)				

**Con default, me refiero a omitir el modificador de acceso.*