Programación

Tema 4

Programación Orientada a Objetos

Tema 4 POO

Tema 4

1.- Definición de POO
2.- Clases Objetos, instancias
3.- Conceptos básicos
4.- Atributos
5.- Métodos
6.- Constructores
7.- Paso de objetos como parámetro
8.- final
9.- Herencia

1. Definición POO

- La programación orientada a objetos es un modelo de programación en el que el diseño de software se organiza alrededor de objetos, en vez de usar funciones y lógica.

- Es un **paradigma de programación**, esto es, un **modelo** o un **estilo** de programación que proporciona unas guías acerca de cómo trabajar con él y que está basado en el concepto de clases y objetos.

Principios básicos de POO

1.- La encapsulación

- La encapsulación presenta toda la información importante de un objeto dentro del mismo y solo expone la información elegida 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 tienen un acceso privado, y los comportamientos o métodos que cuenta con un acceso público.

2.- La abstracción

- La abstracción, que se produce cuando el usuario interactúa solo con los atributos y métodos seleccionados de un objeto, usando herramientas simplificadas de alto nivel para acceder a un objeto complejo.

Principios básicos de POO

3.- La herencia

- La herencia define relaciones jerárquicas entre clases, de modo que atributos y métodos comunes puedan ser reutilizados. Las clases principales extienden atributos y comportamientos a las clases secundarias.

4.- El polimorfismo

- El polimorfismo reside en diseñar objetos para compartir comportamientos, lo que permite procesar objetos de distintos modos.
- Al usar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute distintos comportamientos de dos modos: anulación de método y sobrecarga de método.

BENEFICIOS DE POO

- Reutilización del código.
- Convierte cosas complejas en estructuras simples reproducibles.
- Evita la duplicación de código.
- Permite trabajar en equipo gracias al encapsulamiento, puesto que minimiza la posibilidad de duplicar funciones cuando distintas personas trabajan sobre un mismo objeto al mismo tiempo.
- Al estar la clase bien estructurada permite la corrección de errores en diversos lugares del código.
- Protege la información mediante la encapsulación, pues solo se puede accederá los datos del objeto mediante propiedades y métodos privados.
- La abstracción nos permite construir sistemas más complejos y de un modo más sencillo y organizado.

2.- Clases Objetos e Instancias

Para crear POO se hacen **clases** y luego se crean **objetos** a partir de dichas clases, que constituyen el modelo a partir del que se estructuran los datos y los comportamientos.

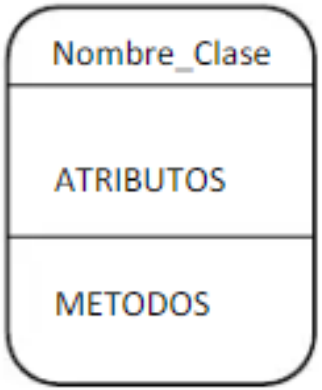
Una clase es una plantilla, que define de modo genérico cómo serán los objetos de un determinado tipo.

Disponer de una serie de atributos y una serie con los comportamientos que son los métodos de la clase.

Con la clase pueden crearse instancias de un objeto, cada uno de ellos con sus atributos definidos independientemente.

2.- Clases Objetos e Instancias

- Representación de una clase UML



3.- Conceptos Básicos

- Una clase es una agrupación de **datos**(variables o campos) y **funciones**(métodos) que oepran sobre esos datos

```
[public] Class className{
        // definición de variables
        // definición de métodos
}
```

- La palabra **public** es opcional: si no se pone, la clase solo es visible para las demás clases del **package**. Todos los métodos y variables deben ser definidos dentro del **bloque** {..} de la clase.
- Un **objeto** (instancia) es un ejemplar concreto de la clase. Las **clases** son como "tipos de variables", mientras que los **objetos** son como variables concretas de un tipo determinado.
- Una clase proporciona una definición para un tipo particular de objeto: define sus datos y la funcionalidad sobre dichos datos.

3.- Conceptos Básicos

- Una clase es una plantilla
- Hay una copia de esa clase por programa, pero muchos objetos de esa clase en el mismo programa.
- Los métodos definen las operaciones que se pueden realizar , que es lo que hace la clase, no pueden existir fuera de la clase.
- Todas las variables y funciones de Java deben pertenecer a una clase.
- En Java no existe el concepto de variables y funciones globales
- En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene la extensión *.java. Por lo general, lo habitual es escribir una sola clase por fichero.

3.- Conceptos Básicos

```
3  public class Punto {
4      double x;
5      double y;
6
7      public Punto(int a, int b) {
8          x=a;
9          y=b;
10     }
11     public double calculadistancia() {
12         double z=Math.sqrt((x*x)+(y*y));
13         return z;
14     }
15 }
```

Control de acceso a los miembros de una clase

- Los modificadores más importantes desde el punto de vista del diseño de clases y objetos, son los que permiten controlar la visibilidad y acceso a los métodos y variables que están dentro de una clase.
  - Uno de los beneficios de las clases, es que pueden proteger a sus variables y métodos (tanto de instancia como de clase) del acceso desde otras clases.
- Java soporta cuatro niveles de acceso a variables y métodos. En orden, del más público al menos público son: público (**public**), protegido (**protected**), sin modificador (también conocido como **package**) y privado (**private**).

Control de acceso a los miembros de una clase

	public	protected	(sin modificador)	private
Clase	SI	SI	SI	SI
Subclase en el mismo paquete	SI	SI	SI	NO
No-Subclase en el mismo paquete	SI	SI	SI	NO
Subclase en diferente paquete	SI	SI/NO **(*)**	NO	NO
No-Subclase en diferente paquete (Universo)	SI	NO	NO	NO

(*) Los miembros (variables y metodos) de clase (static) si son visibles. Los miembros de instancia no son visibles.

4.- Atributos

- Los atributos definen la estructura de los objetos que instancien una clase.
- Cada objeto que se crea de una clase tiene su propia copia de los atributos y de los métodos.
- Por ejemplo, cada objeto de la clase Punto tiene sus propias coordenadas x e y.
- Para acceder al valor de un atributo se escribe:

  Objeto.atributo;

- Para aplicar un método a un objeto concreto

  Objeto.metodo;

5.- Métodos

- Los **métodos** son funciones definidas dentro de una clase. Salvo los métodos **static** o de clase, se aplican siempre a un objeto de la clase por medio del **operador punto** (.).
- Dicho objeto es su **argumento implícito**. Esto implica que se puede hacer referencia a sus atributos dentro del método. De todas formas, también se puede acceder a ellas mediante la referencia **this**.
- Los métodos pueden además tener otros **argumentos** explícitos que van entre paréntesis, a continuación del nombre del método.
- La primera línea de la definición de un método se llama **declaración**.
- El **valor de retorno** puede ser un valor de un **tipo primitivo o una referencia**

5.- Métodos

- No puede haber más que un único valor de retorno .
- Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.
- Los métodos pueden definir variables locales. Su visibilidad estará limitada al propio método. Las variables locales no se inicializan por defecto

```
6⊖    public Punto elPuntoMasLejano(Punto c) {
7         int a;
8         if(this.x >= c.x)
9             return this;
10        else
11            a++; // error a no inicializada
12        return c;
13    }
14 }
```

6.- Constructores

- Un **constructor** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del **constructor** es reservar memoria e inicializar los atributos de la clase.
- Sirven para la inicialización correcta de objetos.
- Los constructores no tienen valor de retorno (ni siquiera void) y se tienen que llamar igual que la clase.
- Una clase puede tener varios constructores, que se diferencian por el tipo y número de sus argumentos (**sobrecargados**).
- Se llama **constructor por defecto** al constructor que no tiene argumentos. Cuando no hay constructor, el compilador crea uno por defecto.
- Los constructores sólo pueden ser llamados por otros constructores o por métodos static. No pueden ser llamados por los métodos de objeto de la clase.

6.- Constructores

- Sobrecarga de constructores

```
3  public class Caja {
4      int alto;
5      int ancho;
6      int largo;
7      public Caja(int alto, int ancho, int largo) {
8          this.alto = alto;
9          this.ancho = ancho;
0          this.largo = largo;
1      }
2      public Caja() {
3          this.alto = -1;
4          this.ancho = -1;
5          this.largo = -1;
6      }
7
8      public Caja(int valor) {
9          this.alto = valor;
0          this.ancho = valor;
1          this.largo = valor;
2      }
3  }
```

7.- Paso de objetos como parámetro

- Los objetos también se pueden pasar a un método como parámetro, ya que son variables del tipo definido por su clase

```
3  public class Caja {
4      int alto;
5      int ancho;
6      int largo;
7      public Caja(int alto, int ancho, int largo) {
8          this.alto = alto;
9          this.ancho = ancho;
0          this.largo = largo;
1      }
2      public Caja() {
3          this.alto = -1;
4          this.ancho = -1;
5          this.largo = -1;
6      }
7
8      public Caja(Caja c) {
9          this.alto = c.alto;
0          this.ancho = c.ancho;
1          this.largo = c.largo;
2      }
3  }
```

8.- final

- Se utiliza para declarar variables de manera que su contenido no pueda ser modificado.
- Una variable **final** tiene que ser inicializada en su declaración.
- Una variable **final** se convierte en una constante.
- Por convenio, el identificador de una variable **final**, debe escribirse en mayúsculas.

```
final int CONTADOR=3;
```

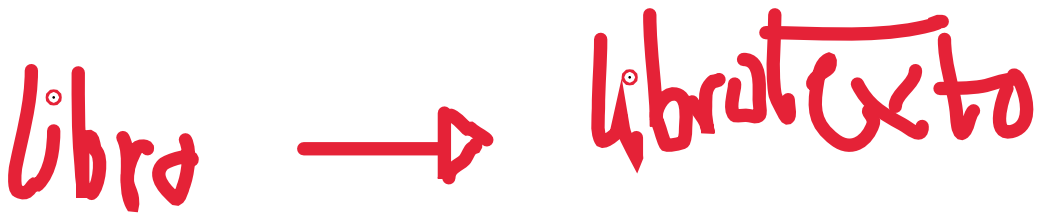
9.- Herencia

- La herencia es una de las características básicas de la Programación Orientada a Objetos (POO)
- Permite la reutilización del código, especialización o evolución de los sistemas
  - Sin embargo, también conduce a sistemas que son más complicados de entender y mantener.
- La Herencia es el mecanismo por el cual una clase extiende las propiedades y comportamientos (datos/atributos y métodos) de otra clase, que se denomina **clase base** o **clase Padre.**
- La clase 'heredera' se denomina **clase extendida** o **derivada**.
- Una clase **derivada** tiene sus propias propiedades y comportamientos (los que están especificados en su código), pero además tiene todas las propiedades y comportamientos de la clase base.

Sintaxis en la herencia

- La palabra clave **extends** indica que está creando una nueva clase que se deriva de una clase existente. El significado de «extends» es aumentar la funcionalidad.
- En la terminología de Java, una clase que se hereda se llama clase padre o superclase, y la nueva clase se llama clase hija o subclase

```
1  class ClaseHija extends ClasePadre {
2      //Metodos y campos
3  }
```



Ejemplo de herencia de Java

```
1   class Vehiculo {
2           private int numeroRuedas;
3
4           public Vehiculo(int numeroRuedas) {
5               this.numeroRuedas = numeroRuedas;
6           }
7       }
8
9       class Coche extends Vehiculo {
10          private String marca;
11
12          public Coche(String marca, int numeroRuedas) {
13              super(numeroRuedas);
14              this.marca = marca;
15          }
16      }
```

INICIALIZACIÓN DE CLASES DERIVADAS

- Cuando se crea un objeto de una clase derivada se crea implícitamente un objeto de la clase base que se inicializa con su constructor correspondiente.
- Para invocar la llamada al constructor de la clase base utilizamos la palabra **super.**
- Si en la creación del objeto se usa el constructor sin parámetros, entonces se produce una llamada implícita al constructor sin parámetros para la clase base.
- Pero si se usan otros constructores es necesario invocarlos explícitamente.

INICIALIZACIÓN DE CLASES DERIVADAS

Ejemplo de herencia de Java

```
1   class Vehiculo {
2           private int numeroRuedas;
3
4           public Vehiculo(int numeroRuedas) {
5               this.numeroRuedas = numeroRuedas;
6           }
7       }
8
9       class Coche extends Vehiculo {
10          private String marca;
11
12          public Coche(String marca, int numeroRuedas) {
13              super(numeroRuedas);
14              this.marca = marca;
15          }
16      }
```

Tipos de herencia

Herencia única

- Cuando una clase hereda otra clase, se conoce como herencia única

```java
class Vehiculo {
    private int numeroRuedas;

    public Vehiculo(int numeroRuedas) {
        this.numeroRuedas = numeroRuedas;
    }

    public void conducir() {
        System.out.println("Conduciendo...");
    }
}

class Coche extends Vehiculo {
    private String marca;

    public Coche(String marca, int numeroRuedas) {
        super(numeroRuedas);
        this.marca = marca;
    }

    public void abrirPuerta(){
        System.out.println("Abriendo puerta...");
    }
}
```

```java
    public static void main(String[] args) {
        Coche coche = new Coche("Mercedes", 4);
        coche.abrirPuerta();
        coche.conducir();
    }
```

Tipos de herencia

Herencia multinivel

- Cuando existe una cadena de herencia, se le conoce como herencia multinivel.

```java
class Vehiculo {
    private int numeroRuedas;

    public Vehiculo(int numeroRuedas) {
        this.numeroRuedas = numeroRuedas;
    }

    public void conducir() {
        System.out.println("Conduciendo...");
    }
}

class Coche extends Vehiculo {
    private String marca;

    public Coche(String marca, int numeroRuedas) {
        super(numeroRuedas);
        this.marca = marca;
    }

    public void abrirPuerta() {
        System.out.println("Abriendo puerta...");
    }
}
```

```java
class CuatroXCuatro extends Coche {

    public CuatroXCuatro(String marca, int numeroRuedas) {
        super(marca, numeroRuedas);
    }

    public void activarCuatroPorCuatro() {
        System.out.println("Activando 4x4");
    }
}

public class Main {

    public static void main(String[] args) {
        CuatroXCuatro coche = new CuatroXCuatro("Mercedes", 4);
        coche.abrirPuerta();
        coche.conducir();
        coche.activarCuatroPorCuatro();
    }

}
```

Tipos de herencia

Herencia jerárquica

- Cuando dos o más clases heredan una sola clase, se conoce como herencia jerárquica

```java
class Vehiculo {
    private int numeroRuedas;

    public Vehiculo(int numeroRuedas) {
        this.numeroRuedas = numeroRuedas;
    }

    public void conducir() {
        System.out.println("Conduciendo...");
    }
}

class Coche extends Vehiculo {
    private String marca;

    public Coche(String marca, int numeroRuedas) {
        super(numeroRuedas);
        this.marca = marca;
    }

    public void abrirPuerta() {
        System.out.println("Abriendo puerta...");
    }
}
```

```java
class Moto extends Vehiculo {

    private int centimetrosCubicos;

    public Moto(int centimetrosCubicos, int numeroRuedas) {
        super(numeroRuedas);
        this.centimetrosCubicos = centimetrosCubicos;
    }

    public void subirMarchaAPedal() {
        System.out.println("Subiendo marcha a pedal...");
    }
}

public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche("Mercedes", 4);
        coche.abrirPuerta();
        coche.conducir();
        // coche.subirMarchaAPedal(); este no es posible
    }

}
```