Programación Orientada a OBJETOS. OBJETOS Y CLASES

- Objetos
- Clases
- Atributos - Estado
- Métodos – Comportamiento

La programación orientada a objetos es un paradigma de programación que se basa en la utilización de **objetos** para modelar los datos del mundo real y sus comportamientos. Se trata de poder llevar los datos del mundo real a las aplicaciones informáticas de la forma más adecuada y segura para su tratamiento.

Clientes, cuentas bancarias, alumnos/as, socios/as de un club, libros, parcelas de un camping, habitaciones de un hotel, reservas, artículos de un catálogo … todos serán **objetos** a la hora de procesarlos en una aplicación informática.

- **En POO,** no tenemos que pensar en un **objeto** como en una **"cosa"** cualquiera. Consideramos un **objeto** a cada una de las **entidades de negocio** que nuestra aplicación va a procesar y gestionar.

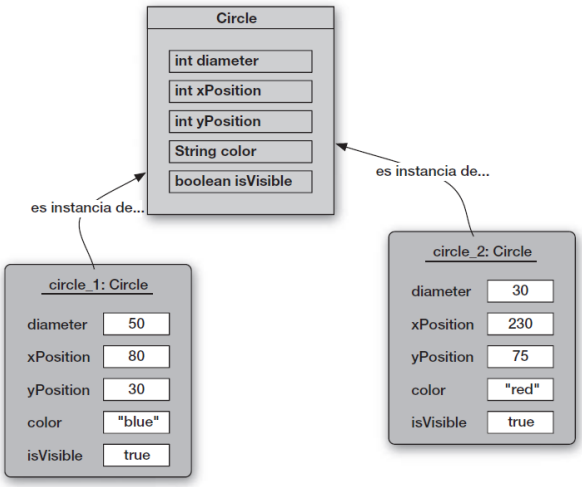
- **En POO** los objetos reinventan los tipos de datos que se utilizaban en la programación estructurada clásica, y sobre todo fusionan la separación tradicional que existía entre **datos y procesos**. En POO una buena parte de los procesos forman parte de los datos en sí mismos, y definen su **comportamiento**.

Las **Clases** son la forma que tenemos en POO de modelar los objetos que nuestras aplicaciones van a manejar. Son como un **patrón,** o un **molde** a partir del cual se van a generar (**instanciar** es el término correcto) **objetos**. Cada objeto tendrá unas características (**atributos**) propias cuyos valores concretos en un momento dado definen su **estado**, y lo diferencian de los demás objetos de su Clase. Todos los objetos de una Clase compartirán un determinado **comportamiento** (**métodos**), que es en definitiva todo lo que vamos a poder hacer con esos objetos a nivel de aplicación informática.

- **Una clase describe/define un tipo de objeto. Los objetos son instancias de una clase.**

- El conjunto de valores de todos los atributos que definen un objeto (como la posición *x,* la posición *y,* el color, el diámetro y el estado de visibilidad de un círculo) se denomina **estado** del objeto.

- Las operaciones que podemos invocar para manipular, manejar, o cambiar en cualquier sentido el estado de un objeto se denominan **métodos** y definen el **comportamiento** de un objeto y lo que puede realmente hacer con él.


Circle
int diameter
int xPosition
int yPosition
String color
boolean isVisible
es instancia de...
es instancia de...
circle_1: Circle
diameter 50
xPosition 80
yPosition 30
color "blue"
isVisible true
circle_2: Circle
diameter 30
xPosition 230
yPosition 75
color "red"
isVisible true

ESTRUCTURA DE UNA CLASE EN JAVA:

En Java. las clases que sirven para **INSTANCIAR OBJETOS** **definen un nuevo tipo de dato.** y tienen una estructura común y bien definida que vamos a ver a continuación. **No hay nada que improvisar**. En muchos proyectos se suelen empaquetar en un paquete aparte, llamado **Entidades o Entities.**

Hay otro tipo de clases que **ya conocemos y hemos trabajado con ellas durante el curso** (main-class y helper-classes). Su estructura no está orientada a modelar objetos, sino a ofrecer métodos de utilidad. Las clases "helper" en muchos proyectos se suelen empaquetar en un paquete llamado **Utilidades o Utilities.**

public class NombreClase //Estructura de una clase "entidad" (Libro, Cliente …)
{
Declaración de Atributos (Estado) – generalmente **private**
Definición de Métodos – generalmente **public**
- Constructores
- Selectores y mutadores (Setters y Getters)
- **Métodos** que definen el **comportamiento** del objeto

}

- Los atributos almacenan datos dentro de un objeto. Representan el estado actual del objeto.
- Los **constructores** son responsables de garantizar que un objeto se configure apropiadamente en el momento de crearlo(instanciarlo) por primera vez.
- Los **métodos** implementan el **comportamiento** de un objeto, proporcionan su **funcionalidad**.

CONSTRUCTORES – SELECTORES (GET) – MUTADORES (SET)

Los **constructores** tienen un papel esencial que cumplir. Son los métodos que ejecutamos al hacer **new**, y se encargan de "crear/construir" cada objeto de una **clase**, de forma que el objeto queda listo para ser utilizado de inmediato. Al proceso de "construcción" de un objeto se le denomina **instanciación**, pues da lugar a una nueva **instancia** (objeto) del tipo de dato definido por esa clase (Libro-Artículo-Cliente-etc).

- **El método constructor se llama igual que la clase.** Por eso lleva la **1ª letra en mayúsculas**, en contra de las normas de Java en cuanto a los nombres de método (minúsculas). Tampoco lleva valor de retorno, ni siquiera **void**, pues un **constructor instancia objetos.** Es a lo que se dedica.

- **Si no creamos un constructor**, **toda clase** tendrá su constructor por defecto, proporcionado por la **JVM**, que permitirá crear **objetos** con sus atributos inicializados al valor por defecto de su tipo.

  Entonces, ¿Para qué tomarnos las molestias de crear un constructor, o varios **sobrecargados**?

  El constructor por defecto inicializa los valores de los atributos a sus valores de tipo por defecto (numéricos a 0) … y esto no suele ser lo deseable. Por eso lo habitual es codificar nuestros propios constructores para crear objetos desde el primer momento con unos valores determinados.

- En muchos casos incluso se sobrecargará el constructor de clase, para poder instanciar objetos de diversas formas. Veremos ejemplos de **sobrecarga (overload)** de métodos, también constructores.

- Los constructores **siempre** llevarán parámetros en su firma, y como se ha comentado **NUNCA** se les pone **void** pues, aunque aparentemente no devuelven "nada"**,** crean un objeto con los argumentos enviados en la llamada al constructor.

Selectores y Mutadores (setters|getters):

Son los métodos que nos permitirán implementar la **encapsulación** de los atributos. Es exigible que **sólo se pueda acceder a los atributos de un objeto** a través de los **setters|getters**. De esta forma se puede centralizar en ellos también la gestión de excepciones y se protege a los atributos de cualquier otro acceso no deseado.

- La llamada a un **selector ("getter")** es una consulta para conocer el valor actual de un atributo. Los **selectores ("getter**") siempre devuelven un valor, resultado precisamente de obtener (get) el valor del atributo que se consulta. **Nunca llevan parámetros en su firma.**

- La llamada a un **mutador ("setter")** es una solicitud para cambiar el valor de un atributo. Devuelven void y deben de llevar un parámetro del mismo tipo que el atributo al que modifican.

Para un atributo **private int edad;**
(representa la edad de una persona)

```
public int getEdad()
{
    return edad;
}
```

```
public void setEdad (int edad)
{
    this.edad=edad;
}
```

Dado que los métodos **set y get** en esencia son iguales para todos los atributos de una clase, la mayoría de **IDES (Eclipse, NetBeans, etc) incluyen opciones de refactorización** de código para generar automáticamente las versiones básicas de todos los métodos **set|get** para **todos los atributos** de cada clase

CONSTRUCTORES/SETERS/GETTERS son **TODOS** métodos **public**, porque si no sería imposible llamarlos desde fuera de la clase, y no se podría instanciar y manipular objetos

EJEMPLO: clase Cubo. Modela objetos imaginarios de tipo Cubo que representan a recipientes para contener líquidos (cubos) del mundo real.

```
public class Cubo {
   private int capacidad;       //ATRIBUTOS - Cada Objeto de tipo Cubo tendrá un valor para capacidad y contenido
   private int contenido;

   public Cubo (int c) {        //CONSTRUCTOR – Cuando ejecutamos Cubo c = new Cubo(5) estamos creando un nuevo
      this.capacidad =c;        //                    objeto cubo con una capacidad=5 y un contenido=0
   }

   public int getCapacidad() {        //GETTERS y SETTERS
      return this.capacidad;
   }
   public void setCapacidad(int litros) {
      this.capacidad = litros;
   }
   public int getContenido() {
      return this.contenido;
   }
   public void setContenido(int litros) {
      this.contenido = litros;
   }
```

```
                                        //MÉTODOS que definen el COMPORTAMIENTO DEL OBJETO
   public void vacia() {
      this.setContenido(0);
   }
   public void vacia(int litros) {       //METODO SOBRECARGADO. Esta 2ª versión permite vaciar un nº de litros concreto
      if (this.contenido >= litros){
            this.setContenido(this.getContenido()-litros);
      }
   }

   public void llena() {
      this.setContenido(this.capacidad);
   }

  public void pinta() {
      for (int nivel = this.getCapacidad(); nivel > 0; nivel--) {
         if (this.getContenido() >= nivel) {
            System.out.println("|~~~~~|");
         } else  System.out.println("|     |");
      }
      System.out.println("=======\n");
   }
}
```

Ejemplo instanciación objetos y uso de la clase Cubo.

```
public class PruebaCubos {
    public static void main(String[] args) {
        //llamadas al constructor de clase para crear 2 cubos de distinta capacidad
        Cubo cuboPeq = new Cubo(3);
        Cubo cuboGran = new Cubo(5);

        //cada cubo creado llama al método de clase pinta() para mostrar su representación en consola
        cuboPeq.pinta();
        cuboGran.pinta();

        //cada cubo llama al método de clase llena() y después al método pinta() de nuevo
        System.out.println("\nLlenamos los cubos\n ");
        cuboPeq.llena(); cuboGran.llena();
        cuboPeq.pinta(); cuboGran.pinta();

        //cada cubo llama al método de clase vacía(). El pequeño por completo y el grande en parte (le quitamos 3)
        System.out.println("\nVaciamos cubos\n ");
        cuboPeq.vacia();
        cuboPeq.pinta();
        cuboGran.vacia(3);
        cuboGran.pinta();
   }
}
```

Cuando programamos en Java, lo que hacemos esencialmente es:

- **Instanciar** objetos.
- **Invocar métodos** sobre dichos objetos, para gestionar su comportamiento e interrelación.

Ámbito/visibilidad de los elementos de una clase
public, protected y private

Al definir los elementos de una clase **(atributos y métodos)**, se deben especificar sus ámbitos de visibilidad con las palabras reservadas **public** (público), **protected** (protegido) y **private** (privado). En la siguiente tabla se muestra desde dónde es visible/accesible un atributo o método según el modificador de ámbito que lleve

	En la misma clase	En el mismo paquete	En una subclase	Fuera del paquete
private	✔	✖	✖	✖
protected	✔	✔	✔	✖
public	✔	✔	✔	✔
sin especificar	✔	✔	✖	✖

Como hemos comentado, por el momento y en general, en nuestros primeros proyectos, todos los **atributos** serán **private**, y todos los **métodos** serán **public**.

CAMPOS (ATRIBUTOS) de un OBJETO VS Variables Locales VS Parámetros formales MÉTODOS.

Tenemos que aprender a distinguir con claridad entre estos 3 diferentes "contenedores" para los datos de nuestros programas

ATRIBUTOS DE UN OBJETO.

Los **atributos** son las variables "de instancia" que definen el estado de un objeto. Se declaran al principio de una clase y lo lógico es que su ámbito sea **private** para que sólo desde esa clase sean accesibles y no se pueda modificar el estado de un objeto sin control. **Sólo debería de permitirse modificar el valor de un atributo a través del método setAtributo() correspondiente.**

```
public class Articulo
{
        private int precio;
        private int existencias;
        private String descripcion;
        …
}
```

- Los **atributos** (sus valores) "viven" en el **HEAP** (Montón), que es la zona de memoria dónde residen los objetos que vamos creamos en cada aplicación.
- Los **atributos** almacenan datos que persisten durante la vida del objeto. Mantienen el **estado actual** del objeto. Su ciclo de vida coincide con la duración del objeto al que pertenecen.

NO ES OBLIGATORIO inicializar los **atributos.** Cuando se declaran al principio del código de cada clase lo que se está definiendo es un "molde" para instanciar objetos a partir de él. **Lo lógico es inicializar los atributos de cada objeto en la llamada al constructor**. Si no hay, Java ejecutará un **constructor por defecto**, e inicializará los valores de los atributos a los valores por defecto para su tipo. Esta es una diferencia clara y lógica con respecto a las **variables locales** en Java, que siempre han de **inicializarse** antes de poder ser utilizadas.

VARIABLES LOCALES.

Las **variables locales** son "locales" al método en que se declaran. Por ello **su ámbito no se define** (public/private/protected). "Viven" en la **PILA** (Stack), y su ámbito está limitado al método al que pertenecen. Su ciclo de vida coincide con el tiempo durante el cual se está ejecutando el método. Se crean cuando se invoca el método y se destruyen cuando el método termina. **Incluso hay casos especiales cómo las variables de control en los bucles FOR cuyo ciclo de vida se limita al bucle en el que han sido declaradas.**

Las variables locales deben de ser inicializadas OBLIGATORIAMENTE antes de utilizarlas por 1ª vez, o tendremos un error de compilación.

Las variables locales de los **tipos primitivos** (byte, short, int, long, float, double, char, boolean) almacenan sus valores directamente en la PILA (STACK).

Sin embargo, las variables locales **de tipo Objeto NO ALMACENAN UN OBJETO en el STACK.** Los objetos "viven" en el **HEAP**. Las variables de tipo objeto almacenan en el STACK la dirección de memoria del objeto al que están asociadas, **y constituyen la referencia permanente** que nos va a permitir manejar y acceder a cada OBJETO de forma "indirecta", pero completa.

PARÁMETROS FORMALES.

Las variables que se declaran en la firma de los métodos son los **parámetros FORMALES** que recibe un método para realizar su función. Tampoco se define su ámbito, pues obviamente se limita al método en cuya firma (cabecera) están declaradas. Su misión es la de recibir los datos que el método necesita.

Cada vez que un método es llamado, se le envían datos concretos para que trabaje **(argumentos)**. **Una copia exacta de los argumentos** de llamada se "depositan" en los **parámetros formales,** y así el método inicia su ejecución.

Es una práctica habitual, y correcta, llamar a los **parámetros formales** en constructores/setters igual que a los **campos** sobre los que actúan. Sólo hay que saber usar la palabra clave **this**. (referencia al objeto actual). Si se entiende bien el papel del **this,** llamar igual a parámetros formales y atributos. incluso facilita la legibilidad del código. Si hemos encontrado un buen nombre descriptivo para un campo/parámetro que además están directamente relacionados … **¿para que buscar otro?**

NO ES NECESARIO USAR NOMBRES DIFERENTES - MEJOR Y MÁS CLARO USAR NOMBRES IGUALES Y this.

```
public void setCapacidad(int litros) {
    this.capacidad = litros;
}
public void setContenido(int litros) {
    this.contenido = litros;
}
```

```
public void setCapacidad(int capacidad)
{
    this.capacidad = capacidad;
}
public void setContenido(int contenido)
{
    this.contenido = contenido;
}
```

* Métodos Set de la clase **Cubo** tal y como los escribimos EN EL EJEMPLO, y a la derecha la alternativa más lógica que emplearemos **de ahora en adelante.**

EJEMPLO 2: CUENTAS BANCARIAS.

```
public class Cuenta
{
   private String titular;
   private double saldo;

   public Cuenta(String titular) {                    //CONSTRUCTOR SÓLO CON TITULAR
      this(titular, 0);
   }

   public Cuenta(String titular, double saldo) {      //CONSTRUCTOR CON TITULAR E INGRESO INICIAL
      this.titular = titular;
      this.saldo = saldo;
   }
                                                      //SETTERS Y GETTERS
   public String getTitular() {
      return titular;
   }
   public void setTitular(String titular) {
      this.titular = titular;
   }

   public double getSaldo() {
      return saldo;
   }
   public void setSaldo(double saldo) {
      this.saldo = saldo;
   }
                                                        //MÉTODOS DE COMPORTAMIENTO
   public void ingresar(double cantidad) {
      if(cantidad > 0){
         setSaldo(getSaldo() + cantidad);
      }
   }

   public void reintegrar(double cantidad) {
      if (getSaldo() - cantidad < 0) {
         setSaldo(0);
      } else {
         setSaldo(getSaldo()- cantidad);
      }
   }

   public String toString() {
      return getTitular() + " tiene " + getSaldo() + " € en cuenta";
   }
}
```

EJEMPLO DE USO DE LA CLASE CUENTA

```
public class AppCuentas
{
   public static void main(String[] args) {

      //INSTANCIAMOS 3 OBJETOS DE TIPO CUENTA – Usamos la SOBRECARGA DE CONSTRUCTORES
      Cuenta c01 = new Cuenta("Ana");
      Cuenta c02 = new Cuenta("Luisa", 1000);
      Cuenta c03 = new Cuenta("Víctor", 2000);

     //MOSTRAMOS LOS OBJETOS – Podemos hacerlo así, directamente, al haber codificado el método
   toString
      System.out.println("\nDespués de instanciar los Objetos:\n");
      System.out.println(c01);
      System.out.println(c02);
      System.out.println(c03);

      //INGRESOS
      c01.ingresar(500);
      c02.ingresar(1000);
      c03.ingresar(2000);

     //MOSTRAMOS LOS OBJETOS
      System.out.println("\nDespués de hacer los Ingresos:\n");
      System.out.println(c01);
      System.out.println(c02);
      System.out.println(c03);

     //REINTEGROS
      c01.reintegrar(200);
      c02.reintegrar(500);
      c03.reintegrar(800);

      //MOSTRAMOS LOS OBJETOS
      System.out.println("\nDespués de hacer los Reintegros:\n");
      System.out.println(c01);
      System.out.println(c02);
      System.out.println(c03);
   }
}
```